6

Deep Operating Manual

This chapter presents a concise operating manual for DEEP 2.0. The first section lists every menu option along with a short description of its purpose and the page number on which more details can be found if the short description is not sufficient.

Menu Options

File Menu Options

Read a database - Page 223

A text file in standard database format (such as Excel™ CSV) is read. The first line names the variables, and subsequent lines are the data, one case per line. Space, tab, and comma may be used as delimiters. Subsequent training will produce a predictive model by default, not a classifier.

Read a series (Simple) - Page 224

A univariate time series is read and a set of predictor and target variables are computed based on the values of the series, optionally differenced and/or log transformed. Predictive and classification targets are generated.

Read a series (Trend Path) - Page 228

A univariate time series is read and predictor and target variables are computed based on the path through time of short-term linear trends. Predictive and classification targets are generated.

Read a series (Fourier) - Page 233

A univariate time series is read and predictor and target variables are computed based on the Fourier coefficients of data in a moving window. Predictive and classification targets are generated.

Read a series (Morlet) - Page 237

A univariate time series is read and a set of predictor and target variables are computed based on Morlet wavelets in a moving window. Predictive and classification targets are generated.

Read XY points - Page 242

A set of points in a plane are read and their Fourier transform computed as predictors. Predictive and classification targets are generated.

Read MNIST image - Page 244

A standard MNIST-format image file is read. The corresponding MNIST label file must be read after the image file is read. Subsequent training will produce a model that is a classifier by default, not a predictive model.

***Read MNIST image* (Fourier)** - Page 244

A standard MNIST-format image file is read and its two-dimensional Fourier transform is computed to generate predictor variables. The corresponding MNIST label file must be read after the image file is read. Subsequent training will produce a model that is a classifier by default, not a predictive model.

Read MNIST labels - Page 245

A standard MNIST-format label file is read. The corresponding MNIST image file must be read before the label file is read.

Write activation file - Page 246

A text file containing the activation of a specified neuron for all training set cases is written.

Clear all data - Page 246

All training data is erased, but a trained model (if it exists) is retained. The purpose of this command is to allow reading a test dataset and evaluating the performance of a trained model on this new dataset.

Print

The currently selected display window (created under the *Display menu*) is printed. If no window is selected, *Print* is disabled.

Exit

The program is terminated.

Test Menu Options

Use CUDA (Toggle Yes/No)

This option is enabled only if a CUDA-capable device is present on the computer. If a check mark appears next to this option, the CUDA device will be used for compute-intensive operations. Click this option to toggle the check mark on and off.

Model Architecture - Page 247

The number of unsupervised and supervised layers is specified, as well as the number of neurons in each layer. If the data was read with the *Read a database* command or any of the *Series* commands, the model will be predictive by default, predicting numeric values of the target variable(s). If MNIST data was read, the model will be a classifier by default, employing a *SoftMax* output layer to classify according to the labels in the label file.

Database inputs and targets - Page 250

The user specifies one or more predictor variables and one or more target variables. If anything other than a database was read, the predictors and targets are predefined and need not be specified by the user. However, the user can still change them through this menu command if desired. During model training, predictors which are constant for all training cases are omitted from the model.

Advanced options

Options of an advanced nature and which would not normally be changed by the user can be set here. In DEEP 2.0 the only such option is the maximum number of threads allowed for non-CUDA threaded computation. The default should be excellent in all practical applications. It cannot be set to more than 64 due to limitations imposed by the Windows operating system.

RBM training params - Page 251

Parameters relevant to unsupervised RBM training can be set.

Supervised training params - Page 256

Parameters relevant to training the supervised layers can be set.

Autoencoding training params - Page 260

Parameters relevant to autoencoding training can be set.

Train - Page 262

The model is trained using the data currently present.

Test - Page 265

The trained model is tested with the data currently present.

Cross validate - Page 266

The model is evaluated by means of cross validation.

Analyze - Page 270

Two basic analyses of the trained model are performed. These are a comparison of the mean activation of inputs compared to those for the reconstructed data, and the mean activation of the final unsupervised layer. This is valid only if an RBM layer is present.

Display Menu Options

Receptive field - Page 271

A plot of the receptive fields (weights of the first/bottom layer) for one or more hidden neurons is displayed. This display may be printed with the *File* / *Print* command.

Generative sample - Page 272

A plot of one or more generative samples is displayed. This display may be printed with the *File* / *Print* command.

Read a Database

A text file in standard database format is read. In particular, standard-format Excel™ CSV files may be read, as well as databases produced by many common statistical and data analysis programs. The first line must specify the names of the variables in the database. The maximum length of each variable name is 15 characters. The name must start with a letter and may contain only letters, numbers, and the underscore (_) character.

Subsequent lines contain the data, one case per line. Missing data is not allowed.

Spaces, tabs, and commas may be used as delimiters for the first (variable name) and subsequent lines.

Here are the first few lines from a typical data file. Six variables are present, and three cases are shown.

```
RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
-0.82449359   0.25341070   0.30325535  -0.40908301  -0.10667177   0.73517430
-0.47731471  -0.13823473  -0.03947150   0.34984449   0.31303233   0.66533709
 0.12963752  -0.42903802   0.71724504   0.97796118  -0.23133837   0.81885117
```

Read a Series (Simple)

This is the most basic option for reading a time series and automatically generating predictor and target variables. The user specifies a window size, and this window is marched across the time extent of the series. With each placement, as many predictor variables as the window size are generated. Exactly three target variables are generated. The first (used for predictive models) is simply the next value of the predictors, the value one sample past the end of the window. The other two are binary class variables which reflect whether the first target is relatively large or small. The following parameters are specified by the user:

Column - The column of this variable in the database file.

Window - The size of the window placed on the data series. This many predictor variables are generated.

Shift - The number of samples to move the window to generate each case. The default value of one maximizes the number of cases generated. If the value is two, the window will be shifted by two sample points for each placement, thus cutting the number of cases generated roughly in half, et cetera.

Nature of variables - raw data - No transformation of any sort is applied to the time series. The predictors and the predicted target are just the values of the time series. This is appropriate when the values of the series itself are predictive and to be predicted, and the standard deviation of the series is at least approximately constant.

Nature of variables - log of raw data - The natural logarithm of each point in the time series is taken. The predictors and the predicted target are these log values. This is appropriate when the values of the series itself are predictive and to be predicted, but the data is multiplicative (the average standard deviation during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time.

Nature of variables - Difference - Each predictor and the predictive target is computed by taking the difference between adjacent values of the time series. This is appropriate when the changes in the series are predictive and to be predicted, and the variation of these differences is at least roughly constant.

Nature of variables - Difference of log - Each predictor and the predictive target is computed by taking the difference between adjacent values of the natural logarithm of the time series. This is appropriate when the changes in the series are predictive and to be predicted, but the values are multiplicative (the average standard deviation of the series during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The classic use of this variable is for equity prices: the actual price of an equity has no predictive power; rather, it is the changes that are predictive. Moreover, high-priced equities tend to have larger absolute variation than low-priced equities.

Trim tails - Many series have heavy tails; their values are occasionally far from their central tendency. Outliers, whether in predictors or targets, will usually cause severe problems in the training of models when the training algorithm attempts to accommodate the outliers to the detriment of the masses of central values. Setting this to a value greater than zero will cause the specified percent of the largest and the same percent of smallest values to be removed from the database. The input series itself is not affected; trimming applies only to the generated predictors and the predicted target.

Skip header record - If this box is checked, the first record in the series file being read is skipped.

Nature of model - Predict - The model type is set to *predictive* (although this can be overridden by the user in the *Supervised training params* menu). The target variable is set to the predictive target, which is just the next 'predictor' past the end of the window. The two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per sign - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per median - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the predictive target relative to its median, with *Lead_Pos* being the class if the predictive target exceeeds its median, and *Lead_Neg* being the class if the predictive target is less than or equal to its median. If no trained model exists at the time the series is read, this dataset will be used for training and hence the median of the predictive target will be computed. If a trained model exists at the time the series is read, this new dataset will (presumably) be used for testing, and the median already computed for the presumed training data will be used for defining class membership.

Target multiplier - The target variable is multiplied by this quantity.

As noted, the model type (predictive or classifier) is set according to the user-specified model type, although it can be reset using the *Supervised training params* menu. Similarly, the target variable is set to the predictive value or the two class variables according to the user-specified model type. These, too, can be changed with the *Database inputs and targets* menu. However, it is strongly recommended that the user not tamper with either of these settings. There is rarely any good reason for doing so, and other related default behaviors of the program may be impacted in ways that produce confusing results.

Note that if the model is a classifier and the predictors have almost no predictive power, the trained model will have a strong bias toward classifying cases into whichever class is more prevalent in the training set.

Also note that if the ***Nature of model - Classify per median*** option is chosen, the median of the generated target variable will be computed for defining the class membership of each case if and only if no trained model currently exists. If a trained model already exists, the median will not be recomputed, as the assumption is that the new dataset will serve as an independent test set. In particular, bear in mind that even if a new model is trained with newly read data, the median will *not* be recomputed, because at the time the series was read the program would have been unable to read your mind and know in advance that you will be using the new data to train a new model!

Read a Series (Trend Path)

This is a more advanced option for reading a time series and automatically generating predictor and target variables. The philosophy is described on Page 46. The user specifies a window size, and this window is marched across the time extent of the series. With each placement, the linear trend with a fixed lookback is computed for each observation in the window. Each of these generates a predictor variable. Optionally, the difference between these trend values is also computed to create predictor variables.

To be clear, suppose we specify a window size of 5 and a lookback of 20. Suppose we are at time 0. At a minimum, five predictors will be generated. These are the linear trend over the 20-observation time span ending at time 0, that ending at time -1, and those ending at times -2, -3, and -4. Optionally, five more predictors can be generated. The first would be the linear trend ending at time 0 minus that ending at time -1. The second would be that ending at time -1 minus that ending at -2, and so forth. One could consider these changes to be the instantaneous *velocity* of trend change as time passes.

Exactly three target variables are generated. The first (used for predictive models) is based on the observation (the raw input series) one sample past the end of the window. It can be the actual value, or the log of the value, or the change from the last observation in the window to the next observation after, or the difference of the logs of these quantities. The other two targets are binary class variables which reflect whether the first target is relatively large or small. In other words, these targets are exactly the same as the targets in the *Simple Series* described in the prior section.

The following parameters are specified by the user:

Column - The column of this variable in the database file.

Window - The size of the window placed on the data series. This many predictor variables are generated if the user chooses to compute values only. Twice this many predictors are generated if the user optionally chooses to include velocity as well.

Shift - The number of samples to move the window to generate each case. The default value of one maximizes the number of cases generated. If the value is two, the window will be shifted by two sample points for each placement, thus cutting the number of cases generated roughly in half, et cetera.

Nature of target - raw data - The predicted target is just the next value of the time series. This is appropriate when the values of the series itself are to be predicted, and the standard deviation of the series is at least approximately constant. The trends for predictors are based on the raw input series.

Nature of target - log of raw data - The predicted target is the log of the next value of the series. This is appropriate when the values of the series itself are to be predicted, but the data is multiplicative (the average standard deviation during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The trends for predictors are based on the log of the raw input series.

Nature of target - Difference - Each predictive target is computed by taking the difference between the next value of the time series just past the window, and the last value in the window. This is appropriate when the changes in the series are to be predicted, and the variation of these differences is at least roughly constant. The trends for predictors are based on the raw input series.

Nature of variables - Difference of log - Each predictive target is computed by taking the difference between the log of the next value of the time series just past the window, and the log of the last value in the window. This is appropriate when the changes in the series are to be predicted, but the values are multiplicative (the average standard deviation of the series during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The classic use of this variable is for equity prices: what we really want to predict is the change in price from today to tomorrow. Moreover, high-priced equities tend to have larger variation than low-priced equities. The trends for predictors are based on the log of the raw input series.

Trim tails - Many series have heavy tails; their values are occasionally far from their central tendency. Outliers, whether in predictors or targets, will usually cause severe problems in the training of models when the training algorithm attempts to accommodate the outliers to the detriment of the masses of central values. Setting this to a value greater than zero will cause the specified percent of the largest and the same percent of smallest values *of the target only* to be removed from the database. The input series itself is not affected, nor are the predictors; trimming applies only to the target.

Skip header record - If this box is checked, the first record in the series file being read is skipped.

Nature of model - Predict - The model type is set to *predictive* (although this can be overridden by the user in the *Supervised training params* menu). The target variable is set to the predictive target. The two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per sign - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per median - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the predictive target relative to its median, with *Lead_Pos* being the class if the predictive target exceeeds its median, and *Lead_Neg* being the class if the predictive target is less than or equal to its median. If no trained model exists at the time the series is read, this dataset will be used for training and hence the median of the predictive target will be computed. If a trained model exists at the time the series is read, this new dataset will (presumably) be used for testing, and the median already computed for the presumed training data will be used for defining class membership.

Lookback - The is the number of observations in the source series that will be used for computing linear trend.

Velocity - If this box is checked, then in addition to the *Window* predictors defined as the linear trends, the changes in the trends (the instantaneous velocity) will also be computed as predictors. Thus, there will be twice as many predictors as the window size. It is usually legitimate to treat trend/velocity pairs as real/imaginary pairs as inputs to a complex-domain model. Think of a sine wave and its derivative.

Target multiplier - The target variable is multiplied by this quantity.

As noted, the model type (predictive or classifier) is set according to the user-specified model type, although it can be reset using the *Supervised training params* menu. Similarly, the target variable is set to the predictive value or the two class variables according to the user-specified model type. These, too, can be changed with the *Database inputs and targets* menu. However, it is strongly recommended that the user not tamper with either of these settings. There is rarely any good reason for doing so, and other related default behaviors of the program may be impacted in ways that produce confusing results.

Note that if the model is a classifier and the predictors have almost no predictive power, the trained model will have a strong bias toward classifying cases into whichever class is more prevalent in the training set.

Also note that if the ***Nature of model - Classify per median*** option is chosen, the median of the generated target variable will be computed for defining the class membership of each case if and only if no trained model currently exists. If a trained model already exists, the median will not be recomputed, as the assumption is that the new dataset will serve as an independent test set. In particular, bear in mind that even if a new model is trained will newly read data, the median will *not* be recomputed, because at the time the series was read the program would have been unable to read your mind and know in advance that you will be using the new data to train a new model!

Read a Series (Fourier)

This is a very advanced option for reading a time series and automatically generating predictor and target variables. The philosophy is described on Page 54. The user specifies a window size, and this window is marched across the time extent of the series. With each placement, the Fourier coefficients are computed for the data that is in the window. These define a set of predictor variables.

Exactly three target variables are generated. The first (used for predictive models) is based on the observation (the raw input series) one sample past the end of the window. It can be the actual value, or the log of the value, or the change from the last observation in the window to the next observation after, or the difference of the logs of these quantities. The other two targets are binary class variables which reflect whether the first target is relatively large or small. In other words, these targets are exactly the same as the targets in the *Simple Series* described in an earlier section.

The following parameters are specified by the user:

Column - The column of this variable in the database file.

Window - The size of the window placed on the data series. Approximately this many predictor variables are generated.

Shift - The number of samples to move the window to generate each case. The default value of one maximizes the number of cases generated. If the value is two, the window will be shifted by two sample points for each placement, thus cutting the number of cases generated roughly in half, et cetera.

Nature of target - raw data - The predicted target is just the next value of the time series. This is appropriate when the values of the series itself are to be predicted, and the standard deviation of the series is at least approximately constant. The predictors are based on the raw input series.

Nature of target - log of raw data - The predicted target is the log of the next value of the series. This is appropriate when the values of the series itself are to be predicted, but the data is multiplicative (the average standard deviation during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The predictors are based on the log of the raw input series.

Nature of target - Difference - Each predictive target is computed by taking the difference between the next value of the time series just past the window, and the last value in the window. This is appropriate when the changes in the series are to be predicted, and the variation of these differences is at least roughly constant. The predictors are based on the raw input series.

Nature of target - Difference of log - Each predictive target is computed by taking the difference between the log of the next value of the time series just past the window, and the log of the last value in the window. This is appropriate when the changes in the series are to be predicted, but the values are multiplicative (the average standard deviation of the series during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The classic use of this variable is for equity prices: what we really want to predict is the change in price from today to tomorrow. Moreover, high-priced equities tend to have larger variation than low-priced equities. The predictors are based on the log of the raw input series.

Trim tails - Many series have heavy tails; their values are occasionally far from their central tendency. Outliers, whether in predictors or targets, will usually cause severe problems in the training of models when the training algorithm attempts to accommodate the outliers to the detriment of the masses of central values. Setting this to a value greater than zero will cause the specified percent of the largest and the same percent of smallest values *of the target only* to be removed from the database. The input series itself is not affected, nor are the predictors; trimming applies only to the target.

Skip header record - If this box is checked, the first record in the series file being read is skipped.

Nature of model - Predict - The model type is set to *predictive* (although this can be overridden by the user in the *Supervised training params* menu). The target variable is set to the predictive target. The two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per sign - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per median - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the predictive target relative to its median, with *Lead_Pos* being the class if the predictive target exceeeds its median, and *Lead_Neg* being the class if the predictive target is less than or equal to its median. If no trained model exists at the time the series is read, this dataset will be used for training and hence the median of the predictive target will be computed. If a trained model exists at the time the series is read, this new dataset will (presumably) be used for testing, and the median already computed for the presumed training data will be used for defining class membership.

Center - If this box is checked, the data will be centered to have zero mean. This is almost always the best choice. If this is not checked and the data in a window happens to be significantly offset from zero, application of the mandatory Welch data window will introduce spurious low-frequency components. This is discussed in more detail on Page 54 and the following pages.

Target multiplier - The target variable is multiplied by this quantity.

As noted, the model type (predictive or classifier) is set according to the user-specified model type, although it can be reset using the *Supervised training params* menu. Similarly, the target variable is set to the predictive value or the two class variables according to the user-specified model type. These, too, can be changed with the *Database inputs and targets* menu. However, it is strongly recommended that the user not tamper with either of these settings. There is rarely any good reason for doing so, and other related default behaviors of the program may be impacted in ways that produce confusing results.

Note that if the model is a classifier and the predictors have almost no predictive power, the trained model will have a strong bias toward classifying cases into whichever class is more prevalent in the training set.

Also note that if the ***Nature of model - Classify per median*** option is chosen, the median of the generated target variable will be computed for defining the class membership of each case if and only if no trained model currently exists. If a trained model already exists, the median will not be recomputed, as the assumption is that the new dataset will serve as an independent test set. In particular, bear in mind that even if a new model is trained will newly read data, the median will *not* be recomputed, because at the time the series was read the program would have been unable to read your mind and know in advance that you will be using the new data to train a new model!

The generated predictors will have names of the form *Real_k* and *Imag_k*, where *k* is the index of the Fourier coefficient and *k* ranges from 1 through *n*/2. (If *n* is odd, *n*/2 means half of *n*, with the fraction discarded. So, for example, 15/2=7 in this context.) If the data is not centered, one more predictor called *Offset* will be generated. This is Re(0), the post-windowing data mean. Note that Im(0) is always zero and so no predictor of this name will be generated. Also note that if *n* is even, Im(*n*/2) is always zero, so no imaginary predictor will be generated for this quantity.

The model inputs are preset so that they occur in real/imaginary pairs. Thus, *Offset* will not be a preset indicator. Also, the Nyquist pair will be preset only if the window is an odd length. Naturally, the user can manually change these presets if desired.

Read a Series (Morlet)

This is a very advanced option for reading a time series and automatically generating predictor and target variables. The philosophy is described on Page 67. The user specifies a window size, and this window is marched across the time extent of the series. With each placement, the real and imaginary Morlet wavelet coefficients are computed for the data that is in the window. These define a set of predictor variables.

Exactly three target variables are generated. The first (used for predictive models) is based on the observation (the input series) one sample past the end of the window. It can be the actual value, or the log of the value, or the change from the last observation in the window to the next observation after, or the difference of the logs of these quantities. The other two targets are binary class variables which reflect whether the first target is relatively large or small. In other words, these targets are exactly the same as the targets in the *Simple Series* described in an earlier section.

The following parameters are specified by the user:

Column - The column of this variable in the database file.

Window - The size of the window placed on the data series. Twice this many predictor variables are generated (real and imaginary for each position in the window).

Shift - The number of samples to move the window to generate each case. The default value of one maximizes the number of cases generated. If the value is two, the window will be shifted by two sample points for each placement, thus cutting the number of cases roughly in half, et cetera.

Period - This is the period over which the wave phenomenon repeats. It must be at least two.

Width - This is the time-domain width of the filter, the number of points on *each side* of the center point. In some contexts, this quantity is called the *half-width*. The total number of points examined to compute a single Morlet wavelet transform value is 2 * *width* + 1. Larger values create a more frequency-selective filter. For the user's protection, DEEP imposes the (not theoretically required) limit that the *width* must be at least the *period*. Setting it to twice the period is a good starting point for experimentation.

Lag - This is the number of samples prior to the current sample at which the filter is centered. This ideally equals the width and it must not exceed the width. If you are willing to live dangerously, it can be as small as half of the width. Smaller values, even down to zero, are legal but strongly discouraged due to the huge distortion to the filter's frequency response.

Nature of target - raw data - The predicted target is just the next value of the time series. This is appropriate when the values of the series itself are to be predicted, and the standard deviation of the series is at least approximately constant. The predictors are based on the raw input series.

Nature of target - log of raw data - The predicted target is the log of the next value of the series. This is appropriate when the values of the series itself are to be predicted, but the data is multiplicative (the average standard deviation during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The predictors are based on the log of the raw input series.

Nature of target - Difference - Each predictive target is computed by taking the difference between the next value of the time series just past the window, and the last value in the window. This is appropriate when the changes in the series are to be predicted, and the variation of these differences is at least roughly constant. The predictors are based on the raw input series.

Nature of target - Difference of log - Each predictive target is computed by taking the difference between the log of the next value of the time series just past the window, and the log of the last value in the window. This is appropriate when the changes in the series are to be predicted, but the values are multiplicative (the average standard deviation of the series during a period in time is proportional to the average value of the series in that period of time). In this situation, taking logs stabilizes the variation across time. The classic use of this variable is for equity prices: what we really want to predict is the change in price from today to tomorrow. Moreover, high-priced equities tend to have larger variation than low-priced equities. The predictors are based on the log of the raw input series.

Trim tails - Many series have heavy tails; their values are occasionally far from their central tendency. Outliers, whether in predictors or targets, will usually cause severe problems in the training of models when the training algorithm attempts to accommodate the outliers to the detriment of the masses of central values. Setting this to a value greater than zero will cause the specified percent of the largest and the same percent of smallest values *of the target only* to be removed from the database. The input series itself is not affected, nor are the predictors; trimming applies only to the target.

Skip header record - If this box is checked, the first record in the series file being read is skipped.

Nature of model - Predict - The model type is set to *predictive* (although this can be overridden by the user in the *Supervised training params* menu). The target variable is set to the predictive target. The two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per sign - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the sign of the predictive target, with *Lead_Pos* being the class if the predictive target is positive, and *Lead_Neg* being the class if the predictive target is zero or negative.

Nature of model - Classify per median - The model type is set to *classifier* (although this can be overridden by the user in the *Supervised training params* menu). The target variables are set to *Lead_Pos* and *Lead_Neg*. These two class variables are defined by the predictive target relative to its median, with *Lead_Pos* being the class if the predictive target exceeeds its median, and *Lead_Neg* being the class if the predictive target is less than or equal to its median. If no trained model exists at the time the series is read, this dataset will be used for training and hence the median of the predictive target will be computed. If a trained model exists at the time the series is read, this new dataset will (presumably) be used for testing, and the median already computed for the presumed training data will be used for defining class membership.

Target multiplier - The target variable is multiplied by this quantity

As noted, the model type (predictive or classifier) is set according to the user-specified model type, although it can be reset using the *Supervised training params* menu. Similarly, the target variable is set to the predictive value or the two class variables according to the user-specified model type. These, too, can be changed with the *Database inputs and targets* menu. However, it is strongly recommended that the user not tamper with either of these settings. There is rarely any good reason for doing so, and other related default behaviors of the program may be impacted in ways that produce confusing results.

Note that if the model is a classifier and the predictors have almost no predictive power, the trained model will have a strong bias toward classifying cases into whichever class is more prevalent in the training set.

Also note that if the ***Nature of model - Classify per median*** option is chosen, the median of the generated target variable will be computed for defining the class membership of each case if and only if no trained model currently exists. If a trained model already exists, the median will not be recomputed, as the assumption is that the new dataset will serve as an independent test set. In particular, bear in mind that even if a new model is trained will newly read data, the median will *not* be recomputed, because at the time the series was read the program would have been unable to read your mind and know in advance that you will be using the new data to train a new model!

The generated predictors will have names of the form *Real_k* and *Imag_k*, where *k* is the lag within the window and *k* ranges from 0 (the current point) through one less than the window length.

Read XY Points

This is a very advanced option for reading sets of XY points in a plane and automatically generating predictor and target variables. The philosophy is described on Page 83. Each case must contain the same number of points. Each point is represented by a pair of numbers, the X coordinate followed by the Y coordinate. Fourier coefficients with any of several optional normalizations are computed for each case. These define a set of predictor variables. The last item on each line of the file is an integer specifying the class of the case. The class number must range from zero through one less than the number of classes.

One 'target' variable is defined for each class. For each case, the target variable for that case's class is set to 1.0 and that for all other classes is set to 0.0.

If the user request using the raw data, the predictors will have names of the form *X_k* and *Y_k* for *k* from 0 through one less than the number of points. Fourier predictors will have names of the form *Real_k* and *Imag_k*, where *k* is the index of the Fourier coefficient and *k* ranges from 0 through one less than the number of points.

The following parameters are specified by the user:

Points - Each case (line in the file) contains this many points. Each point is a pair of numbers, with the X coordinate coming first, followed by the Y coordinate.

Classes - This is the number of classes. The last number in each line of the file is the class number of the case, ranging from 0 through *Classes*–1. Each line of the file will thus contain 2 * *Points* + 1 numbers.

Type of Data - Raw XY - The (X, Y) points as read from the file are used as default predictors.

Type of Data - Raw Fourier - All Fourier coefficients of the XY points are used as default predictors.

Type of data - Fourier, location normalized - All Fourier coefficients of the XY points except Re[0] and Im[0] are used as default predictors.

Type of data - Fourier, scale/start normalized - All Fourier coefficients of the XY points except Re[0], Im[0], and the real and imaginary parts of the dominant fundamental are used as default predictors. The coefficients that are used are normalized to be location invariant, scale invariant, and starting-point invariant.

Rotation direction - Force CW - The user guarantees that the direction of perimeter tracing defined by the supplied points is clockwise. This option is taken into account only for scale/start normalization. If the user specifies this incorrectly, serious computational errors will result.

Rotation direction - Force CCW - The user guarantees that the direction of perimeter tracing defined by the supplied points is counter-clockwise. This option is taken into account only for scale/start normalization. If the user specifies this incorrectly, serious computational errors will result.

Rotation direction - Auto rotation - The program automatically computes the rotation direction for each case. However, in most applications it is important that this be consistent, the same for every case. If different cases have different rotation directions, chances are your data is invalid, most likely because the perimeter crosses itself. This option is taken into account only for scale/start normalization.

Max frequency - This is the maximum frequency that goes into the default predictor list. All frequencies are put into the database, so the user can manually change this later if desired.

Read MNIST Image

A standard MNIST image file is read. It is assumed that there will be ten labels. The number of rows and columns is read from the file and not assumed by DEEP, although the common file is 28 rows and columns. In DEEP 1.0 the product of the number of rows and columns must not exceed 4096-10=4086. There is no hard-coded limit on the number of images; it is limited only by available memory.

Models in DEEP 2.0 can be either classifiers, in which case the output layer is SoftMax, or predictive, in which case the output layer is linear with no range limiting, and it makes numeric predictions. When MNIST data is read, the classifier form of model is used by default. For all other data, the default is numeric prediction. In both cases, the user can override the default and force the model to be a classifier or predictive.

The MNIST image file must be read before a label file can be read.

Read MNIST Image (Fourier)

A standard MNIST image file is read and its two-dimensional Fourier transform computed to generate predictor variables. It is assumed that there will be ten labels. The number of rows and columns is read from the file and not assumed by DEEP, although the common file is 28 rows and columns. In DEEP 2.0 the product of the number of rows and columns must not exceed 4096-10=4086. Also in DEEP 2.0 the number of rows and columns must be even. There is no hard-coded limit on the number of images; it is limited only by available memory.

Models in DEEP 2.0 can be either classifiers, in which case the output layer is SoftMax, or predictive, in which case the output layer is linear with no range limiting, and it makes numeric predictions. When MNIST data is read, the classifier form of model is used by default. For all other data, the default is numeric prediction. In both cases, the user can override the default and force the model to be a classifier or predictive.

The MNIST image file must be read before a label file can be read.

The names of the transform variables begin with either the letter *R* for a real part or *I* for an imaginary part, followed by the horizontal frequency (never negative) and finally followed by the vertical frequency (also never negative). For example, the variable R_3_7 is the real part of the coefficient for a horizontal frequency of 3 and a vertical frequency of 7. Note the following properties, which are discussed in more detail on Page 100:

- The horizontal frequency ranges from zero through the number of columns minus one.

- The vertical frequency ranges from zero through half of the number of rows (the Nyquist frequency). This is in deference to the symmetry depicted in Figure 4.1 on Page 105.

- At a vertical frequency of zero, as well as at the vertical Nyquist frequency, the coefficients at a horizontal frequency of zero and at the horizontal Nyquist frequency are strictly real. Therefore, no actual imaginary parts will be produced as variables for these four coefficients. However, so that 'complex' pairs are produced to facilitate complex-domain processing, for these four variables DEEP will generate names starting with *Z* and whose values duplicate the real parts of these four numbers.

- At a vertical frequency of zero, as well as at the vertical Nyquist frequency, the coefficients in the horizontal direction are symmetric (complex conjugates) around the horizontal Nyquist frequency. Therefore, for these two rows, only coefficients through half of the number of columns are generated.

Read MNIST Labels

A standard MNIST label file is read. It is assumed that there are ten possible labels. The label file cannot be read until the image file has been read.

Write Activation File

This option writes a text file containing the activation of a single neuron for all cases, one line per case. The user specifies whether the neuron to be written is in the unsupervised or supervised section, which layer within that section it is in (with 1 being the first layer), and the neuron number within that layer (also with 1 being the first neuron).

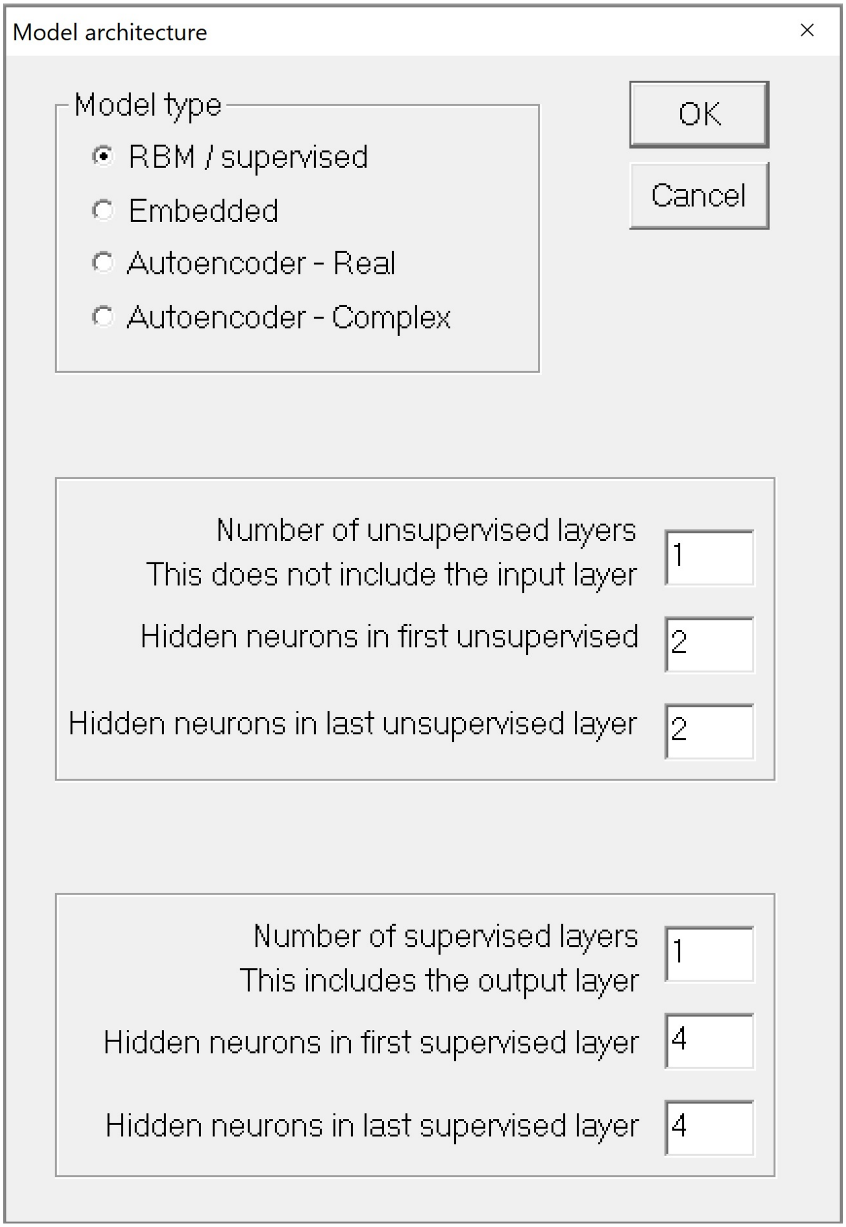
An activation file is mainly for diagnostic use, although some users may find it convenient to pass an activation file to other programs.

Clear All Data

Sometimes the user will want to test a trained model on data that the model has not yet seen, often called a test set or out-of-sample (OOS) data. This can be done by reading the training data, training the model, clicking *Clear all data*, reading the test set, and clicking *Test*.

When a trained model exists and data is cleared, subsequently read data must have the same variables in the same order as the data that was used to train the model.

Model Architecture



Model type

● RBM / supervised

○ Embedded

○ Autoencoder - Real

○ Autoencoder - Complex

Number of unsupervised layers
This does not include the input layer: 1

Hidden neurons in first unsupervised: 2

Hidden neurons in last unsupervised layer: 2

Number of supervised layers
This includes the output layer: 1

Hidden neurons in first supervised layer: 4

Hidden neurons in last supervised layer: 4

Several model architectures are available in DEEP 2.0:

- An *RBM / Supervised* model consists of zero or more unsupervised layers created by greedy RBM training, followed by one or more supervised layers trained by using the outputs of the final unsupervised layer (or the raw data if there are no unsupervised layers) as inputs and targets as outputs.

- An *Embedded* model consists of a stack of one or more layers (not counting the *input* or *visible* layer). These are greedily trained, as usual for RBMs. The layer just prior to the final (top) layer has class identifier neurons appended, as discussed in Chapter 2.

- An *Autoencoder - Real* model consists of zero or more unsupervised layers created by greedy autoencoder training, followed by one or more supervised layers trained by using the outputs of the final unsupervised layer (or the raw data if there are no unsupervised layers) as inputs and targets as outputs. The entire model operates in the real domain. This family is discussed in depth starting on Page 120.

- An *Autoencoder - Complex* model is identical to the above except that the entire model operates in the complex domain. For the final layer outputs, the imaginary part is ignored; only the real part is used for prediction and classification.

The user defines the architecture by specifying the following quantities:

Number of unsupervised layers - For *Unsupervised / Supervised* and *Autoencoder* architectures, this may be zero to create a model that is entirely supervised. For *Embedded* architecture this must be at least one.

Hidden neurons in first unsupervised layer - This refers to the bottom layer, the one closest to the input data.

Hidden neurons in last unsupervised layer - This refers to the topmost unsupervised layer, the one that feeds the supervised section. If there is only one unsupervised layer, this must equal *Hidden neurons in first unsupervised layer*. If there are multiple layers, interior sizes are linearly interpolated.

Number of supervised layers - This is valid only for *Unsupervised / Supervised* and *Autoencoding* architectures. It must be at least one (the output layer), which is the usual case when there are one or more unsupervised layers. But it is legal for an unsupervised section to feed a 'traditional' supervised model, one having one or more hidden layers prior to the output layer. It is also possible to use DEEP 2.0 for strictly supervised models.

Hidden neurons in first supervised layer - This is valid only for *Unsupervised / Supervised* and *Autoencoding* architectures. It is relevant only if *Number of supervised layers* is greater than one, in which case it is the number of hidden neurons in the first layer encountered by the unsupervised layer outputs or the raw data if there are no unsupervised layers.

Hidden neurons in last supervised layer - This is valid only for *Unsupervised / Supervised* and *Autoencoding* architectures. It refers to the last hidden layer before the output layer. If *Number of supervised layers* is two (one hidden, plus output), this must equal *Hidden neurons in first supervised layer*. If there are multiple hidden layers (*Number of supervised layers* exceeds two), interior sizes are linearly interpolated.

Database Inputs and Targets

This option is used to specify the variable(s) that will be used as inputs to the model (the *predictors*) and the variable(s) that will be predicted (the *targets*). One or more of each can be selected using standard Windows methods: dragging across a range, holding down *Shift* while clicking the first and last in a range, or holding down *Control* to select individual variables.

If *Read a database* was used to read the training data, then the user must specify the input(s) and target(s). But if the data is anything else then the inputs and targets are automatically preset. Nonetheless, the user is free to use this menu option to change the preset selection.

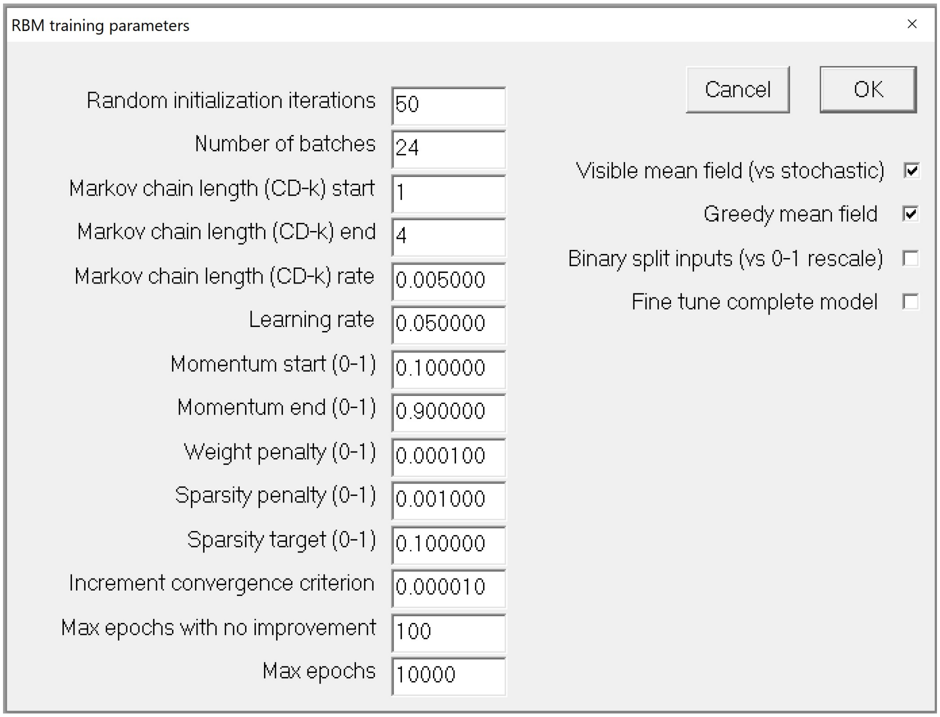
All MNIST input variables will follow the naming convention of *P_row_column* to identify the location of each pixel in the input grid, with the naming origin (first row/column) being zero. Thus, the upper-left pixel will be *P_0_0*.

The MNIST target variables will be named *Label_digit* to identify the digit with which each class is associated. Thus, the targets will be named *Label_0* through *Label_9*.

For MNIST data, the model will be a classifier with SoftMax outputs by default. For training data read any other ways, the model will by default be predictive, attempting to predict numeric values for each target. But a supervised training option (described later) allows the user to force the model to be a classifier or predictor. For a forced classifier, the user must specify at least two targets using the *Data inputs and target* menu option, and for each case, the target having maximum value will be assumed to identify the class of the case.

All input selections which compute Fourier transform variables or effective real/imaginary pairs (such as series trend path which includes velocity) will preselect inputs in such a way that complex-number pairing is obtained. This facilitates immediate use of complex-domain models. Naturally, the user can employ this option to change these presets if desired.

RBM Training Params



Random initialization iterations 50
Number of batches 24
Markov chain length (CD-k) start 1
Markov chain length (CD-k) end 4
Markov chain length (CD-k) rate 0.005000
Learning rate 0.050000
Momentum start (0-1) 0.100000
Momentum end (0-1) 0.900000
Weight penalty (0-1) 0.000100
Sparsity penalty (0-1) 0.001000
Sparsity target (0-1) 0.100000
Increment convergence criterion 0.000010
Max epochs with no improvement 100
Max epochs 10000

Visible mean field (vs stochastic) ☑
Greedy mean field ☑
Binary split inputs (vs 0-1 rescale) □
Fine tune complete model □

This menu option sets the parameters that are relevant to RBM training. All parameters are preset to defaults that should be reasonable for many or most applications. The following parameters may be set:

Random initialization iterations - The number of trial weight sets that are tested to find a good starting point for stochastic gradient descent training. It is definitely worthwhile doing at least a few dozen trials so that subsequent training begins with reconstruction error that is not outrageous. More than several hundred trials is probably overkill.

Number of batches - The training set is divided into this many batches (though the exact number may be adjusted by the program when necessary) for stochastic gradient descent. Concepts vital to this choice are discussed in Volume I. Here are the basic principles:

- Recall from the cited discussion that the tradeoff between time-per-batch and batches-for-convergence is unbalanced in the direction of favoring many small batches. But...

- Although Windows threads have fairly small overhead, the overhead of launching a CUDA kernel can be considerable. Thus, one should be inclined to use fewer batches if using CUDA processing.

- The automatic learning rate and momentum adjustment algorithms described in Volume I perform best with relatively large batches. This inspires us to use few batches.

- *This is the most important issue in practice*. Most Windows installations impose an upper limit of two seconds for a CUDA kernel, after which it is given the boot. Kernel time is almost linearly related to batch size, so if your screen blacks out and recovers with a message that the driver was reset, increase the number of batches. CUDA.LOG lists kernel times and hence can be used to see how close to criticality you are.

Markov chain length (CD-k) start - When stochastic gradient descent begins, this is the number of iterations taken by executing the Markov chain in the contrastive divergence algorithm. The gradient estimate's accuracy is improved by taking more iterations, with the result that convergence requires fewer epochs. But these samples are very expensive to obtain. Early in training we do not need accurate gradient estimates; a rough approximation is sufficient. This parameter should almost always be left at its default value of one.

Markov chain length (CD-k) end - The number of iterations taken as learning progresses. As convergence nears, it is worthwhile expending computation time to obtain more accurate gradient estimates. The default value of four is good in nearly all applications. In case the user wants to obtain true maximum likelihood parameter estimates (usually pointless in practice), this parameter can be set to a very large value.

Markov chain length (CD-k) rate - The rate at which the chain length increases from the starting value to the end value. Standard exponential smoothing is employed, with the ending chain length being the 'new value' of the series.

Learning rate - The initial learning rate. This should be small, probably smaller than the value the user is accustomed to for other programs. This is because the automatic adjustment algorithm described in Volume I will rapidly move it to an optimal value.

Momentum start - The initial momentum. This should be small, probably smaller than the value the user is accustomed to for other programs. As with the learning rate, the automatic adjustment algorithm described in Volume I will rapidly move it to an optimal value.

Momentum end - As training progresses, the momentum will progress toward this value unless the adjustment algorithm swats it down due to instability in the gradient descent algorithm. Values greater than the default are dangerous, and even the default is pretty high.

Weight penalty - The degree to which large weights are penalized. This must be small in order to allow weights to approach their optimal values. But it should not be zero. If no weight penalty is applied, in unusual but annoying pathological situations one or more weights can blow up to enormous values.

Sparsity penalty - The degree to which hidden neuron activation rates are encouraged to approach the *sparsity target* specified as the next parameter. This is not a critical parameter, and it can safely be set to zero if desired. However, in most cases it is good to gently nudge weights toward values that result in smallish hidden neuron activation rates, such as 0.1 or so. Among other things, this makes the weights more interpretable, as one can then study which patterns are associated with activation of certain hidden neurons. If all hidden neurons are activated about half the time, such interpretation is more difficult than if activation is more rare.

Sparsity target - The value toward which hidden neuron activation rates are nudged by the *sparsity penalty*. This is typically around 0.1 or so. This parameter is ignored if the *sparsity penalty* is zero.

Increment convergence criterion - This is the secondary convergence criterion, as described in Volume I. If the ratio of the magnitude of the largest weight adjustment in an epoch to the magnitude of the largest weight drops below this threshold, convergence is decreed to be complete. This should be very small to avoid early exits from the training algorithm.

Max epochs with no improvement - This is the primary convergence criterion. The ratio of the magnitude of the largest weight adjustment in an epoch to the magnitude of the largest weight is a good (though not perfect) measure of how close we are to a local minimum of the negative log likelihood criterion being minimized. If the specified number of epochs pass without this ratio beating its minimum so far, convergence is said to have been achieved.

Max epochs - This is a backstop, insurance against endless iteration. It should never be used as an actual convergence criterion, as it is a brute force rule, with no intelligence about actual convergence. Make it large, and trust that except in very rare pathological situations, one of the main convergence criteria will handle the situation well.

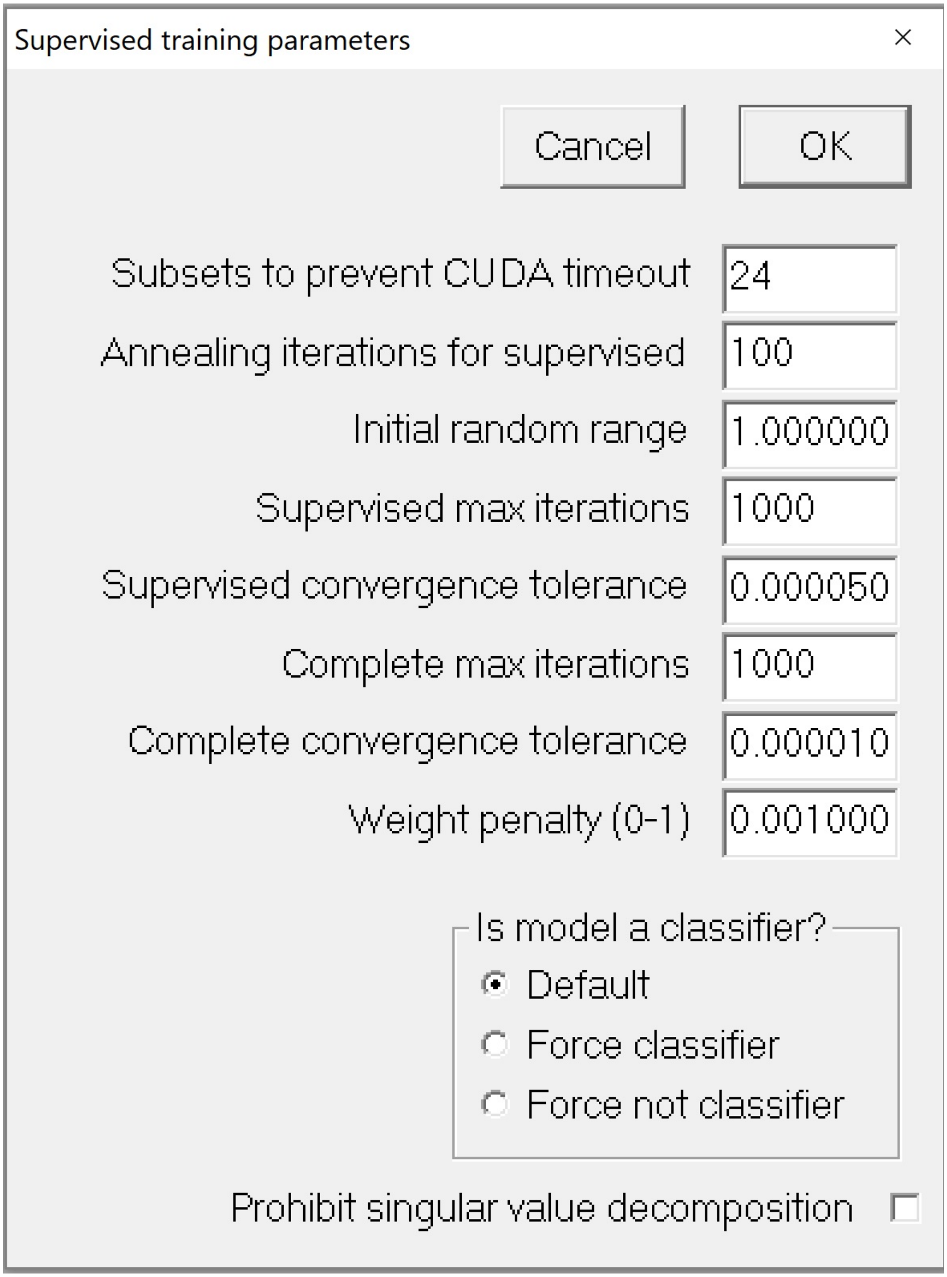
Visible mean field (vs stochastic) - If this box is checked, the reconstruction of the visible layer will use the mean field approximation. If not checked, the reconstruction will sample. It is likely that using the mean field approximation is best, although this is not universally agreed upon. In practice, the difference seems slight.

Greedy mean field - If this box is checked, propagation of input data through early layers for greedy training strictly uses mean field approximations. If not checked, sampling is done for the inputs to the layer being trained (except the first layer, which is never sampled). This topic is discussed in detail in Volume I.

Binary splits - If this box is checked, the raw input data will be quantized to strictly binary data by setting variables above their mean to one and those equal or below the mean to zero. If not checked, the raw input data will be linearly scaled to a range of 0-1.

Fine tune complete model - If this box is checked, after the entire deep belief net is constructed (all RBMs greedily trained, then all subsequent layers trained with supervision), supervised training will be used to tweak the entire model, including the RBM layers. This will always improve in-sample performance and often improve out-of-sample performance. But display of reconstruction samples becomes pointless garbage.

Supervised Training Params



Subsets to prevent CUDA timeout: 24

Annealing iterations for supervised: 100

Initial random range: 1.000000

Supervised max iterations: 1000

Supervised convergence tolerance: 0.000050

Complete max iterations: 1000

Complete convergence tolerance: 0.000010

Weight penalty (0-1): 0.001000

Is model a classifier?

● Default

○ Force classifier

○ Force not classifier

Prohibit singular value decomposition □

This menu option sets the parameters that are relevant to supervised training of the layer(s) following the RBM layer(s), as well as the optional fine tuning of the complete deep belief net. All parameters are preset to defaults that should be reasonable for many or most applications. The following parameters may be set:

Subsets to prevent CUDA timeout - This has no effect whatsoever on the model produced. It affects only the degree to which computations are split up; the results of the computations remain the same. *This is different from batches in RBM training*. RBM batch division does impact the model and the nature of convergence because the weights are updated for each batch. In supervised training, all batches are pooled, with one weight update per epoch (pass through the entire training set). To reduce kernel-launch overhead, the number of subsets should be set as low as possible. But keep an eye on the CUDA time summary in CUDA.LOG and be prepared to use more subsets if any time-per-kernel approaches the two-second Windows limit.

Annealing iterations for supervised - The number of simulated annealing passes used to find a good weight set from which to begin training. This topic is discussed in detail in Volume I. This is usually a fairly cheap operation with good returns for the first few hundred passes. More than a few thousand iterations is probably overkill due to rapidly diminishing returns.

Initial random range - The average range of weight perturbation for simulated annealing. The program will periodically raise and lower the user-specified figure to make this parameter less critical. For this reason, the progress plot of error will have clearly visible periodic variation. This is normal operation. The exact algorithms that govern simulated annealing perturbation are shown in Volume I.

Supervised max iterations - After RBM training is complete, the supervised layer(s) following the RBM layer(s) are trained. This parameter limits the number of epochs in order to prevent wildly excessive run times. It should be set to a very large value and used as insurance only, not as the usual convergence determiner.

Supervised convergence tolerance - This is the primary method for determining convergence of training the supervised layer(s). Training is stopped when the relative change in the error from one epoch to the next falls below this level. Because the supervised training algorithm used in DEEP is deterministic, this can safely be set to a very small value, although doing so is usually without merit because most improvement happens early in training.

Complete max iterations - This is identical to *Supervised max iterations* except that it applies to the optional fine tuning of the complete (unsupervised RBMs plus subsequent supervised layers) deep belief net.

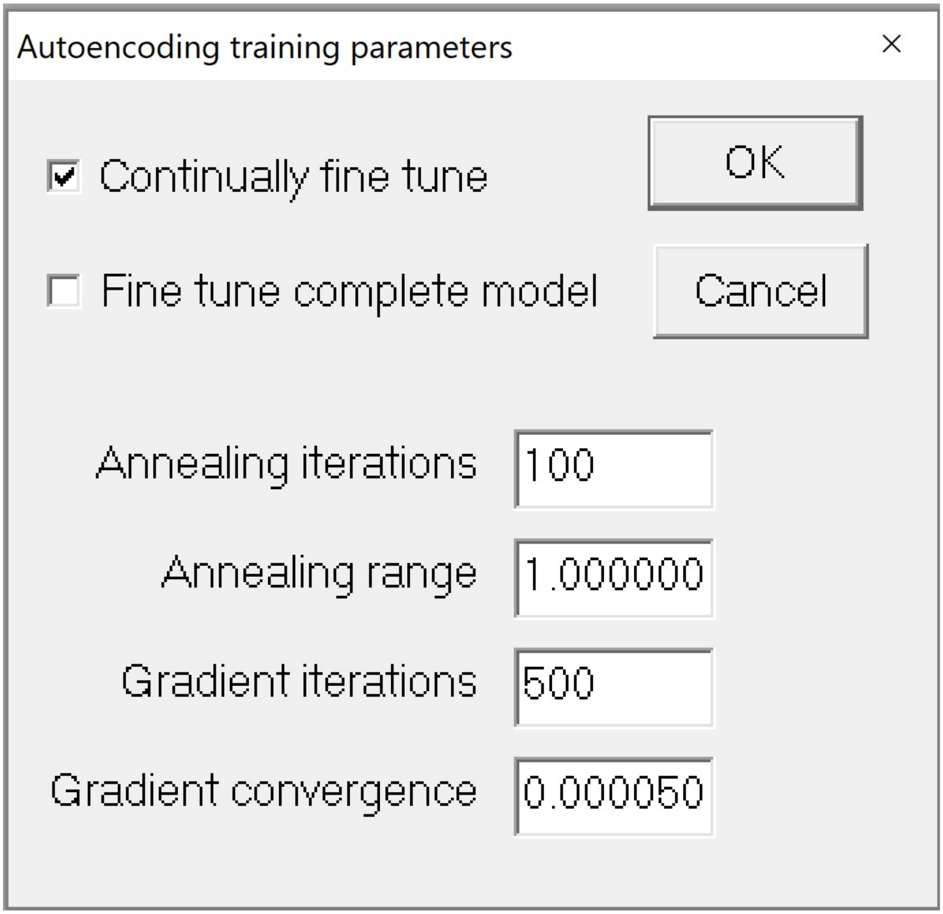
Complete convergence tolerance - This is identical to *Supervised convergence tolerance* except that it applies to the optional fine tuning of the complete (unsupervised RBMs plus subsequent supervised layers) deep belief net.

Weight penalty - This penalty discourages large weights during supervised training. It should nearly always be set to a very small value, small enough that it does not have an overly strong impact on learning 'best' weights, yet large enough that it prevents the large weights that can happen in some unusual pathological situations which are especially likely when the inputs to the supervised section are strongly correlated. This topic is discussed in detail in Volume I.

Is model a classifier - By default, MNIST data produces a classifier model, and all other data produces a predictive model. This option allows the user to override the default. If database data is read and the user forces the model to be a classifier, at least two targets must be selected, and for each case the target having greatest value is assumed to be the correct case.

Prohibit singular value decomposition - The section beginning on Page 133 discusses how the extremely efficient singular value decomposition (SVD) algorithm can be used to explicitly compute optimal output weights for a predictive model and discover excellent starting weights for iterative training of classifiers. But for gigantic problems, and in some very rare pathological situations, SVD can fail, even (very rarely) producing not-a-number results. For this reason, SVD is disabled if there are more than 400 inputs to the output layer. Moreover, the user may choose to disable SVD. Because SVD is such an enormous help in achieving rapid and high-quality convergence, it should always be allowed if at all possible.

Autoencoding Training Params



☑ Continually fine tune

□ Fine tune complete model

Annealing iterations 100

Annealing range 1.000000

Gradient iterations 500

Gradient convergence 0.000050

Continually fine tune - If this box is not checked, each unsupervised autoencoding layer will be trained individually and independently. If this box is checked, after two layers are trained (each separately), these two layers will be pooled and the two-layer network trained to autoencode the inputs. If a third layer is then added (by independent training), it will then be added to the pool and this three-layer network trained as an input autoencoder. Et cetera. This algorithm is described in detail on Page 125.

Fine tune complete model - If this box is checked, after the entire deep belief net is constructed (all autoencoding layers greedily trained, then all subsequent layers trained with supervision), supervised training will be used to tweak the entire model, including the unsupervised layers. This will always improve in-sample performance and often improve out-of-sample performance.

Annealing iterations - The number of simulated annealing passes used to find a good weight set from which to begin greedy training of an autoencoding layer. This is usually a fairly cheap operation with good returns for the first few hundred passes. More than a few thousand iterations is probably overkill due to rapidly diminishing returns.

Annealing range - The average range of weight perturbation for the simulated annealing described in the prior option. The program will periodically raise and lower the user-specified figure to make this parameter less critical. For this reason, the progress plot of error will have clearly visible periodic variation. This is normal operation. The exact algorithms that govern simulated annealing perturbation are shown in Volume I.

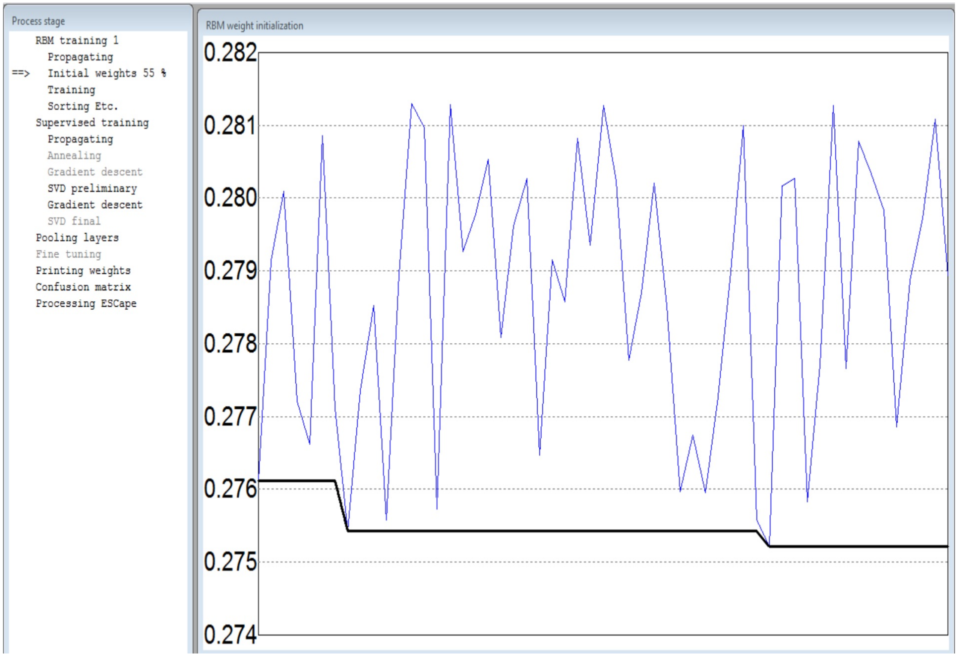
Gradient iterations - This parameter applies to gradient-descent optimization of individual autoencoding layers as well as continuous fine tuning of autoencoding layers. It limits the number of epochs in order to prevent wildly excessive run times. It should be set to a very large value and used as insurance only, not as the usual convergence determiner.

Gradient convergence - This parameter applies to gradient-descent optimization of individual autoencoding layers as well as continuous fine tuning of autoencoding layers. This is the primary method for determining convergence of training the autoencoding layer(s). Training is stopped when the relative change in the error from one epoch to the next falls below this level. Because the training algorithm used in DEEP is deterministic, this can safely be set to a very small value, although doing so is usually without merit because most improvement happens early in training.

Train

The *Train* selection trains the entire deep belief net. If this is an RBM / Supervised model or an autoencoding model, then first, all RBM or autoencoding layers are trained with unsupervised greedy training. Then, all subsequent layers (typically just one, the output) are trained using supervision. Finally and optionally, the entire deep belief net is fine tuned with supervision. The steps for complete training are shown at the left side of the screen. Those that will not be used in the current configuration are grayed out. A marker arrow identifies the step currently executing, and particularly slow operations indicate the percent completion.

The first step in RBM training is finding initial weights by randomly generating weight sets and finding the one with minimum reconstruction error. In Figure 6.5 below we see this operation in progress. The top line on the left side says that we are training RBM layer 1. The initial weight operation is 55 percent complete. The graph is the RMS reconstruction error, with the light blue line showing the individual tries, and the heavy black line showing the best so far.



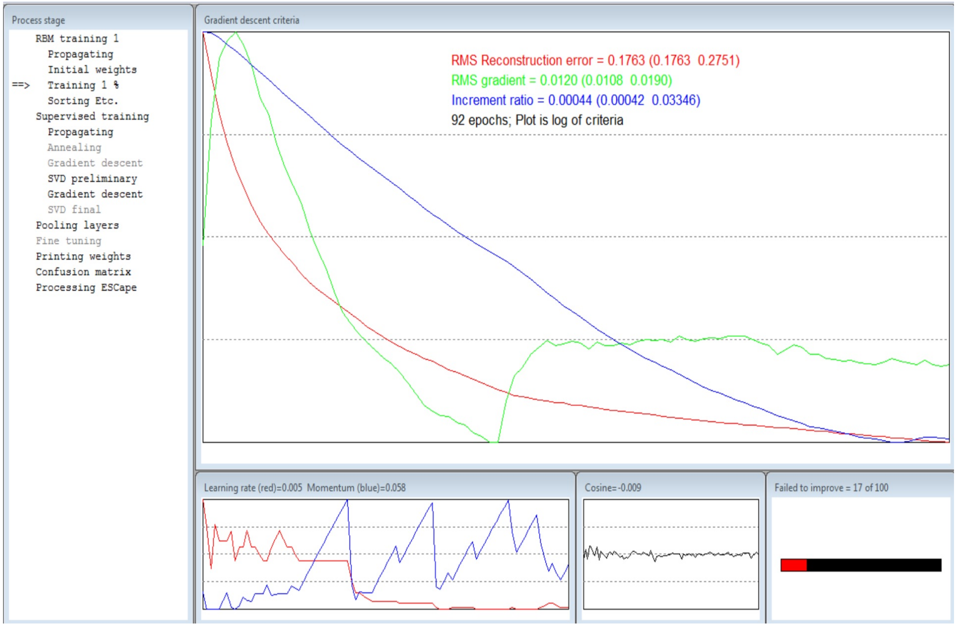
```
Process stage
    RBM training 1
      Propagating
==>   Initial weights 55 %
      Training
      Sorting Etc.
    Supervised training
      Propagating
      Annealing
      Gradient descent
      SVD preliminary
      Gradient descent
      SVD final
    Pooling layers
    Fine tuning
    Printing weights
    Confusion matrix
    Processing ESCape
```

Figure 6.5: Finding initial weights for RBM training

After initial weight selection is complete, the program trains the RBM using stochastic gradient descent. The screen will resemble Figure 6.6 below.

At the left side we see that we are in the *Training* operation and we are 1 percent done. This percentage is relative to the *Max epochs* parameter which, as stated earlier, should always be set overly large and used only as a backstop. Hence, this percentage will nearly always be very pessimistic relative to actual training progress.

The largest window plots three values, whose current, minimum, and maximum values are written in the top-center of the plot. The *reconstruction error* is in red and it typically drops off fast and then levels out. The *increment ratio* (maximum increment divided by maximum weight) typically decreases fairly linearly before hitting a sharp knee and flattening, with a few subsequent small bounces. The RMS gradient often displays peculiar behavior, with very gradual decrease punctuated by sharp jumps up as blocks of weights suddenly go from near zero to larger, more useful values.



```
    RBM training 1
      Propagating
      Initial weights
==>   Training 1 %
      Sorting Etc.
    Supervised training
      Propagating
      Annealing
      Gradient descent
      SVD preliminary
      Gradient descent
      SVD final
    Pooling layers
    Fine tuning
    Printing weights
    Confusion matrix
    Processing ESCape
```

RMS Reconstruction error = 0.1763 (0.1763 0.2751)
RMS gradient = 0.0120 (0.0108 0.0190)
Increment ratio = 0.00044 (0.00042 0.03346)
92 epochs; Plot is log of criteria

Learning rate (red)=0.005 Momentum (blue)=0.058

Cosine=-0.009

Failed to improve = 17 of 100

Figure 6.6: RBM training

Be aware that these plots are the logs of the values, not the actual values. Also, each plot is scaled so that the entire historical range of the parameter exactly covers the vertical extent of the plot. The net effect is that as training progresses and values become small, tiny changes in the actual values are magnified to large changes in the plot. This magnification is useful in that it shows in great detail exactly what is happening. Unfortunately, it can be deceptive, making the user think that violent gyrations are occurring when, in fact, the changes in the actual values are miniscule.

The lower-left graph shows the dynamically adjusted learning rate and momentum, also scaled so that the historical values exactly fill the vertical extent of the plot. Typically, the learning rate will show a net decrease, dropping to a very small value after several dozen iterations during which it bounces. The momentum only rarely stabilizes, climbing steadily until it becomes excessive and causes an overshoot which results in backtracking, at which point the adjustment algorithm slaps it back down for a while.

The bottom-center graph shows the cosine of the angle between successive gradients, scaled to a fixed range of minus one to one. It should always be near the center, indicating that the weight increments are neither undershooting nor overshooting.

The lower-right bar graph shows the number of contiguous failures of the increment ratio to decrease, relative to the user-specified limit. When the red interior reaches the right side of the bar's outline, training will terminate. This is the primary convergence criterion.

Supervised training of the post-RBM layers, as well as the optional fine tuning, also cause graphs of the error to be displayed as training progresses. There is nothing fancy or confusing about them, so we'll dispense with a detailed discussion.

If this is an autoencoding model, the display is a lot simpler. Only single-layer greedy training and the optional greedy fine tuning are displayed, and in a manner which should be self explanatory.

Test

The *Test* selection tests the trained model on the current dataset. There is little point in training and then immediately testing a model, as the test would just reproduce the same results given when training is complete. However, this selection facilitates testing the model on new data.

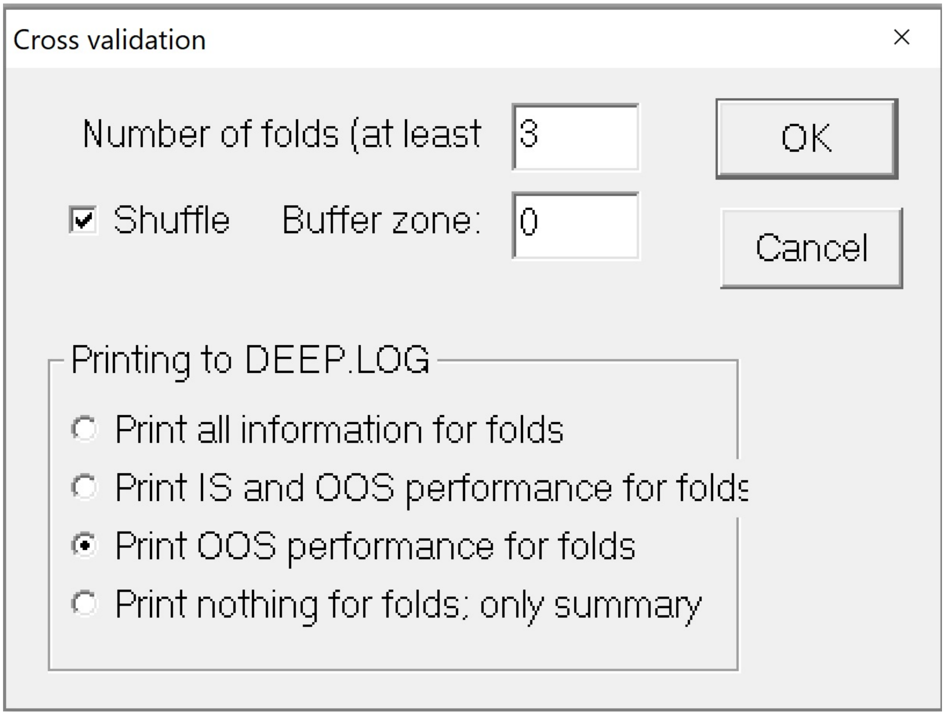
The usual procedure for training and testing a model is as follows:

- Read the training data
- Define the architecture
- Select the predictor and target variables if they are not preset. (Only 'Read a Database' does not preset them.)
- Set training parameters if something other than the default is desired
- Train
- Clear all data
- Read the test data
- Test

The test dataset must contain the same variables in the same order as the training dataset. The user must not change the architecture or the predictor/target variables.

NOTE... The *Test* option does not use CUDA processing. If the model was trained with CUDA enhancement, it is possible that the slightly different floating point computations with and without CUDA may result in slightly different test results. Any differences should be small.

Cross Validate



Cross validation

Number of folds (at least 3

☑ Shuffle Buffer zone: 0

Printing to DEEP.LOG

○ Print all information for folds
○ Print IS and OOS performance for folds
● Print OOS performance for folds
○ Print nothing for folds; only summary

The model's capability is evaluated by cross validation. In this process, the dataset is divided into a user-specified number of subsets, as equal in size as possible. One at a time, each of these subsets is used as a test set after the model is trained by combining the remaining subsets into a single training set. After each subset has been processed this way, which is called a *fold*, the test set results for all folds are pooled into a single grand performance measure. The beauty of this technique is that we do not have to sacrifice data by designating part of the dataset as a one-time-only training set and the other part as a one-time-only test set. Each case in the dataset serves as a test case exactly once, which is an extremely efficient use of the data! Moreover, each time we train the model we are using the large majority of the training cases, which encourages stability. The fact that the pooled test results are nearly unbiased is almost miraculous. Of course, the price paid is that we must retrain the model for each fold, which can be very expensive in some applications. There's no free lunch.

The user must specify several options. These are:

Number of folds - The dataset will be divided into this many subsets, as equal in size as possible. Each subset will be used as the independent test set exactly once, with the other subsets used as the training set.

Printing to DEEP.LOG - This choice controls how much information is printed to the DEEP.LOG log file. The choices are:

> ***Print all information*** - All training and test information, including trained weights, is printed for each fold. This might be voluminous.
>
> ***Print IS and OOS performance*** - In-sample (training set) and out-of-sample (test set) performance figures are printed for each fold. Weights are not printed.
>
> ***Print OOS performance*** - For each fold, only out-of-sample (test) performance is printed.
>
> ***Print nothing for folds*** - No information is printed for individual folds. Only the pooled summary performance report appears.

Shuffle - If this box is checked, the dataset is shuffled before folds are assigned. This is mandatory if the data is 'grouped' in any way, perhaps by inherent serial correlation or perhaps by experimental design. Suppose, for example, the design incorporates three experimental conditions, each grouped together and we employ three folds of cross validation. If the data is not shuffled, then for each fold we will be testing data from a condition that did not appear in the training set for that fold!

Buffer zone - This is required if one or more predictors have serial correlation *and* one or more targets have serial correlation. In this situation, the specified number of cases are temporarily removed from the training set on the upper and lower fold boundaries. The test set is left unaltered. This is illustrated in Figure 6.8. In this figure, there are five folds, delineated with tick marks, and the fold depicted happens to be the one with the test set in the center. Note how part of the training set on each side of the boundary is removed. To determine the correct buffer zone size, find the maximum distance of serial correlation among the predictors and the maximum among the targets. Take the minimum of these two numbers to get the optimal buffer zone.

For example, let the predictors have a maximum correlation distance of 5 cases, and the target 3. The optimal buffer would be Min(5,3)=3. Consider the left boundary. Define time 0 as the first case in the test set. Then consider the following discussion:

Suppose we did not use a guard buffer zone. The last training case at the left boundary would be at time -1. Its targets could be correlated up to the case at time 2. The first test case could have predictors correlated back to time -5, meaning that its predictors could be similar to several cases in the training set. This alone is of no consequence. But recall that the last training case's targets could be similar to those up though the case at time 2. So this first test case can *also* have targets similar to those in the training set. Having an 'independent' test case substantially artificially represented in the training set introduces bias.

It's important to understand the problem here. There is nothing wrong with having serial correlation in *either* the predictors alone (a very common situation) *or* the targets alone (not so common). The problem happens when *both* have serial correlation. In this situation, the training cases near the boundary can be substantially similar to the test cases near the same boundary. The test set, which is supposed to be independent of the training set, is actually not totally independent, because serial correlation has destroyed independence near the boundaries.

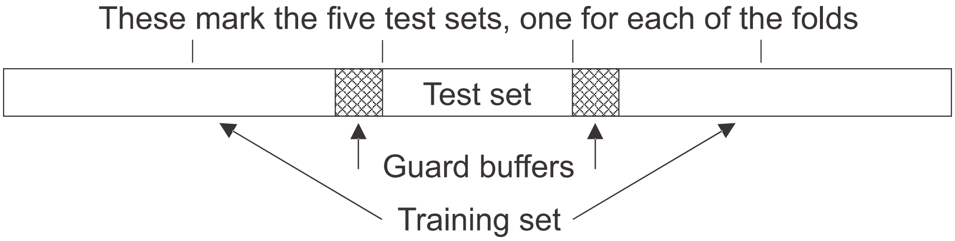


Figure 6.8: Cross validation with guard buffers

Now suppose we remove the three guard zone training cases. The last training set case at this boundary is at time -4. Its targets could be correlated with those of cases through time -1. But the target correlation has vanished by the time we get to the first test case. We're safe.

A similar effect happens at the right boundary. Let the first training case in the upper section be at time 0. Then the last test case below it is at time -1, and its targets could be correlated with the targets through the case at time 2. But after removal of the guard zone, the first used training case will be at time 3.

We can reverse the correlation pattern, with the correlation distance for predictors 3 cases and that for targets 5. The guard zone is still 3 cases. At the left boundary, let the first test case be at time 0. Its predictors can be correlated with those of the cases back to time -3. The last 'original' training case is at time -1, but after removing the guard zone the last actually used training case is at time -4. I leave it as an exercise for the reader to check the right boundary.

Analyze

This selection computes and prints to the DEEP.LOG file two tables of information for RBM models (only!). The first is a comparison, for each input variable, of the probability of its being activated in the training set versus the probability of its being activated in the reconstructed input layer. Here is a short segment illustrating this table:

```
Variable   Visible  Reconstructed

 P_8_10     0.616      0.617
 P_8_11     0.551      0.547
 P_8_12     0.522      0.519
 P_8_13     0.516      0.513
 P_8_14     0.517      0.511
 P_8_15     0.520      0.514
 P_8_16     0.517      0.513
 P_8_17     0.514      0.510
 P_8_18     0.539      0.536
 P_8_19     0.606      0.603
 P_8_20     0.706      0.706
 P_8_21     0.806      0.810
 P_8_22     0.887      0.891
 P_8_23     0.942      0.943
```

The other analysis output is the probability (across the training set) of each final (topmost) layer hidden neuron being activated. Here is an example of this table:

```
Hidden   Activation

   1       0.837
   2       0.449
   3       0.723
   4       0.596
   5       0.578
   6       0.501
   7       0.501
   8       0.418
```

Receptive Field

The *receptive field* of a hidden neuron in an RBM is (loosely) defined as the pattern of weights connecting the input layer to the hidden neuron. If the input happens to be an image, such as is the case with MNIST data, then it is possible to display these weights in the same dimensions as the input image. Figure 6.9 below shows the receptive fields of a dozen neurons trained with MNIST data. Large positive weights are white, large negative weights are black, and intermediate values are shades of gray. A color display is also an option, with positive weights colored cyan and negative weights colored red, and brightness corresponding to magnitude. The gray areas around the perimeter are pixels that are constant for all cases and hence omitted from the model.

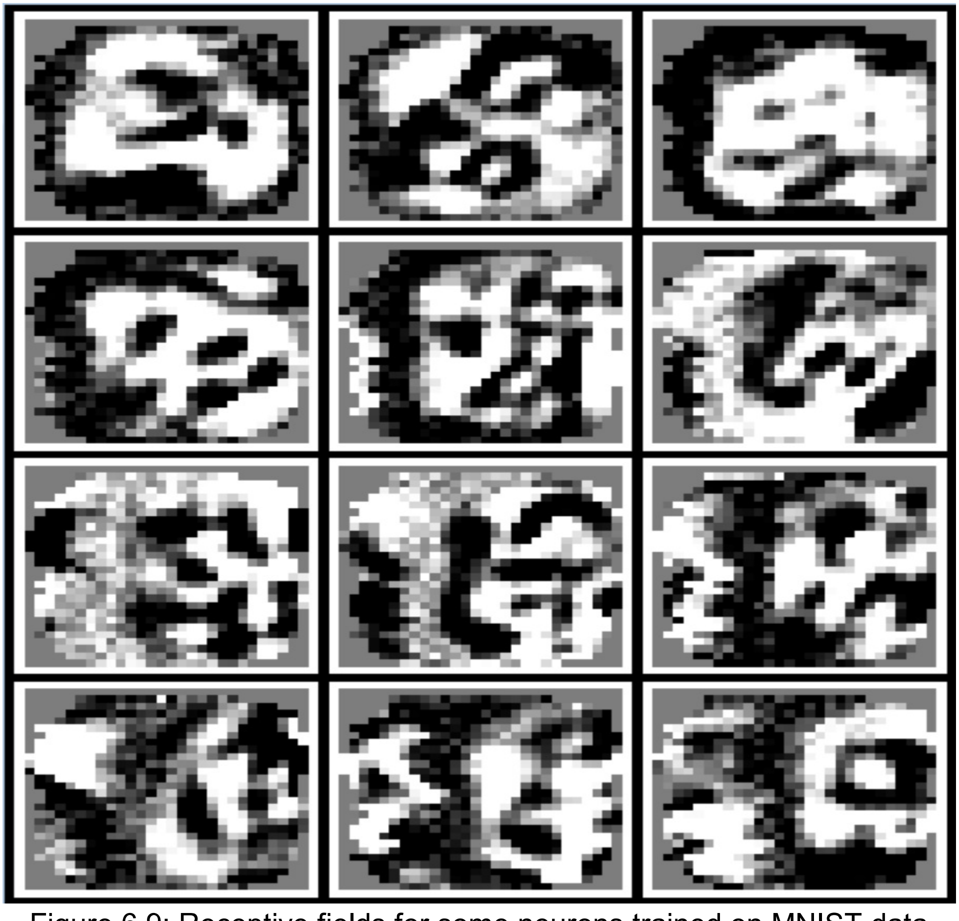


Figure 6.9: Receptive fields for some neurons trained on MNIST data

Generative Sample

We saw in Volume I and on Page 114 of this book that a trained RBM or set of greedily trained layers can be made to spit out random samples from the distribution that it has learned. Examination of such random samples can be interesting because they show examples of the primitive patterns that the model has learned.

This option is valid only for MNIST images and *simple series* data. The user must specify the number of rows and columns of samples to display. Each of the *nrows*ncolumns* images is a separate sample.

As discussed in Volume I, there are two ways to begin the Markov chain whose final value will be the computed sample. One can begin with a member of the training set. To do this, set the *First case* field to a positive number, the sequential number of the training case that will be used for the first sample. Subsequent samples will start from subsequent training cases. The degree to which the final reconstruction resembles the starting pattern is an indication of the quality of training and the degree to which efficient mixing is taking place in the Markov chain.

Figure 6.10 on the next page shows the first 12 cases from the MNIST test set of ten thousand cases. Figure 6.11 shows generative samples obtained from these cases using ten thousand iterations. What makes this interesting is that this was derived from a single RBM layer having just 15 hidden neurons! The degree to which this tiny model has encapsulated training set patterns is astounding.

Alternatively, one can set the topmost hidden neuron layer to random values, thus divorcing the computed samples from training data. This lets us see the actual primitive patterns which the model is recognizing. Figure 6.12 shows 108 random samples obtained from an RBM having 100 hidden neurons, using 50,000 iterations. This is not an *embedded* model, which allow class-conditional sample generation. So rather than often seeing *digits* we are more likely to see the *components* of the digit images which the model has learned.

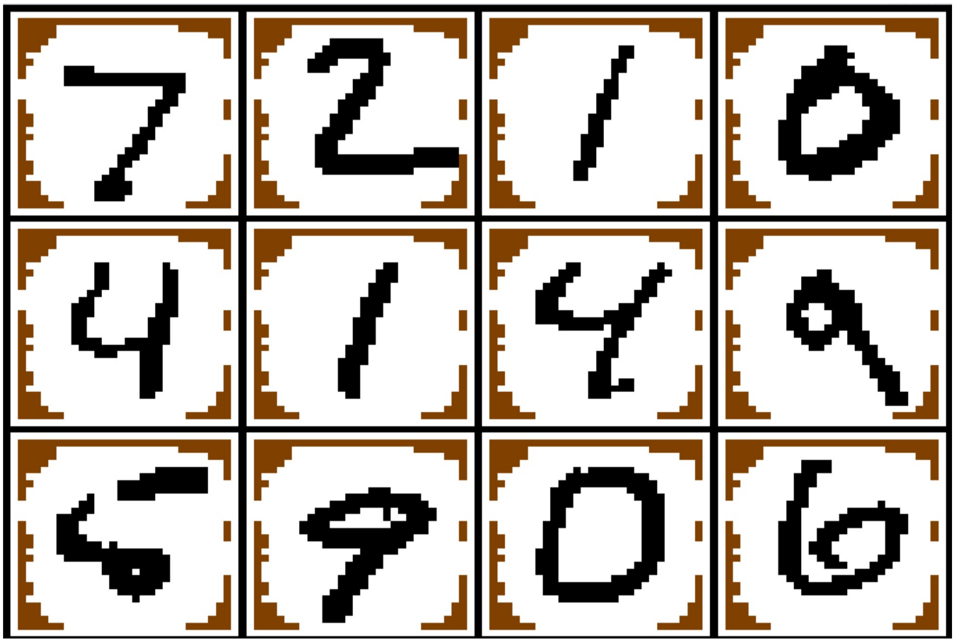


Figure 6.10: First 12 cases of MNIST test set

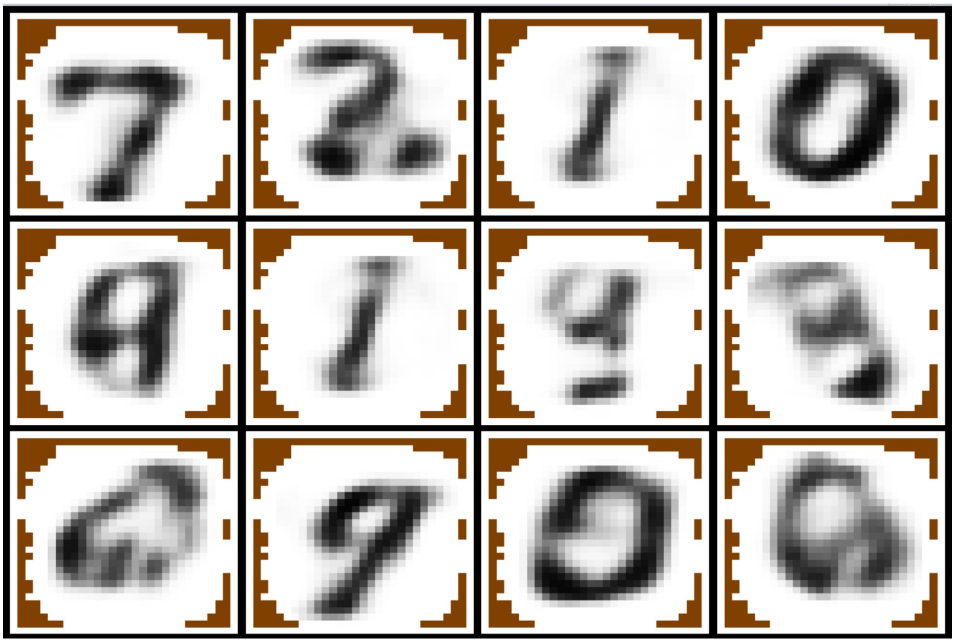


Figure 6.11: Generative samples after 10,000 iterations

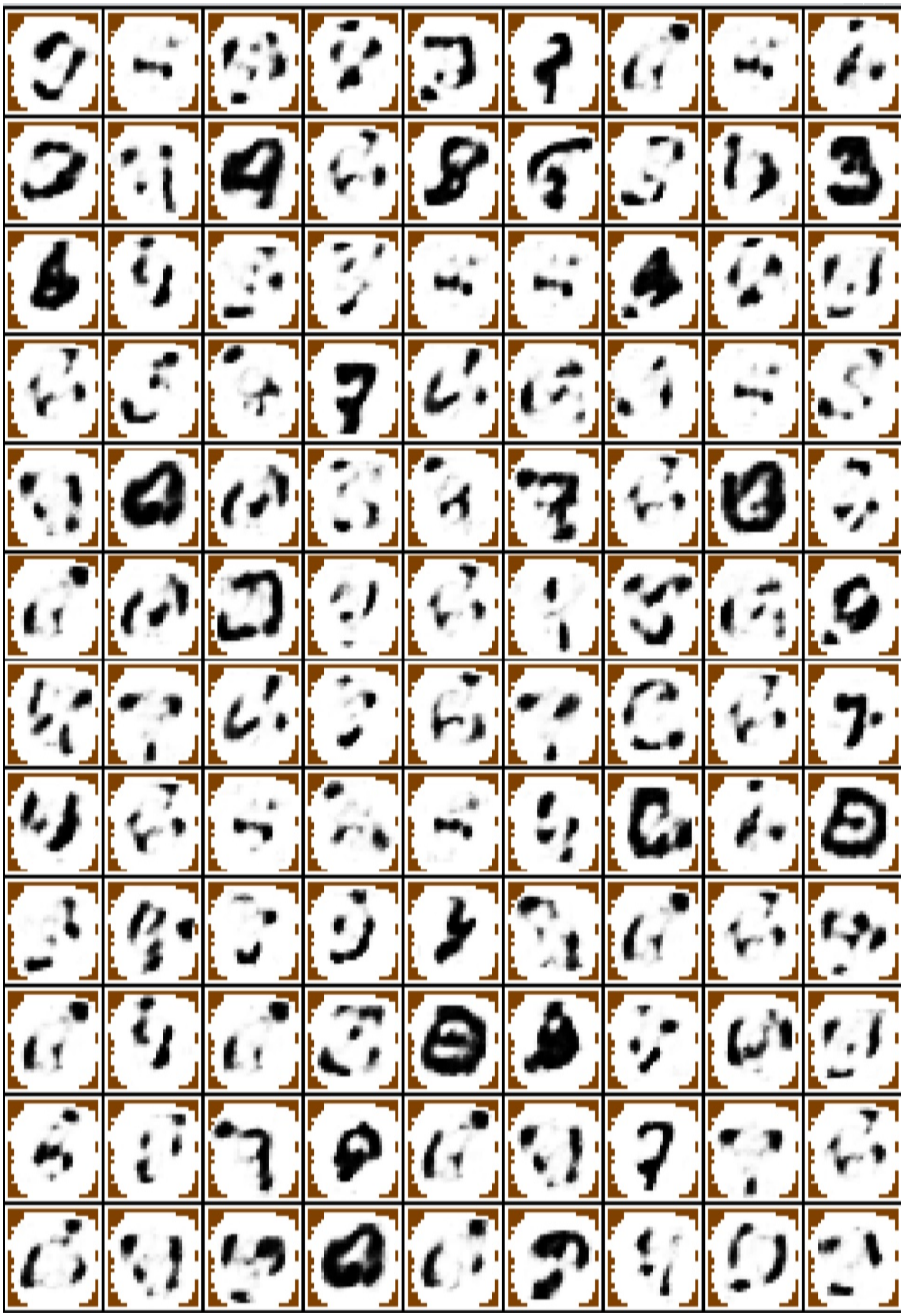


Figure 6.12: Samples using 100 hidden neurons randomly set

Samples from an Embedded Model

We saw on Page 10 that by embedding class labels in the visible layer corresponding to the top-level RBM, we can perform generative sampling from each class separately. When an embedded model has been trained, two additional selections may be made by the user.

The first choice is the check box *Clamp class label*. This would almost always be left at its default of checked, as clamping a class label to force class-conditional sampling is usually the point of using an embedded model (at least for me!). Unchecking this box lets generation run free, without regard to class labels, and thus provides samples from the entire population.

The other choice is *Class for Clamped Random*. This is relevant only if the user specified zero for the *First class label*. With zero, the Markov chain is initialized to random values, so the *Class for Clamped Random* specification is needed to tell the program which class to sample. If the *First class label* is positive then the class of each training case used for initialization will be clamped.

Figure 6.13 on the next page shows some generated samples from the MNIST ‘0' class, and Figure 6.14 shows samples from the ‘1' class.

If the user specified that binary thresholding be performed for RBM training (*Binary splits* on Page 254) then generated samples will also be quantized to binary. Figures 6.15 and 6.16 show binary generated samples from the MNIST ‘0' and ‘1' classes, respectively. Note the significant amount of exact or near duplication, indicating that the RBM has learned some ‘favorite’ patterns and converges strongly to them.

If the data input is a simple series (Page 33) then the user must specify the *Row resolution*. This is the vertical resolution for the display, the number of rows over which the signal is quantified. The default of 20 is reasonable for most situations, although it can be argued that setting the row resolution approximately equal to the window length is also a good choice. An example of generative samples of a simple series can be seen in Figure 3.3 on Page 45.

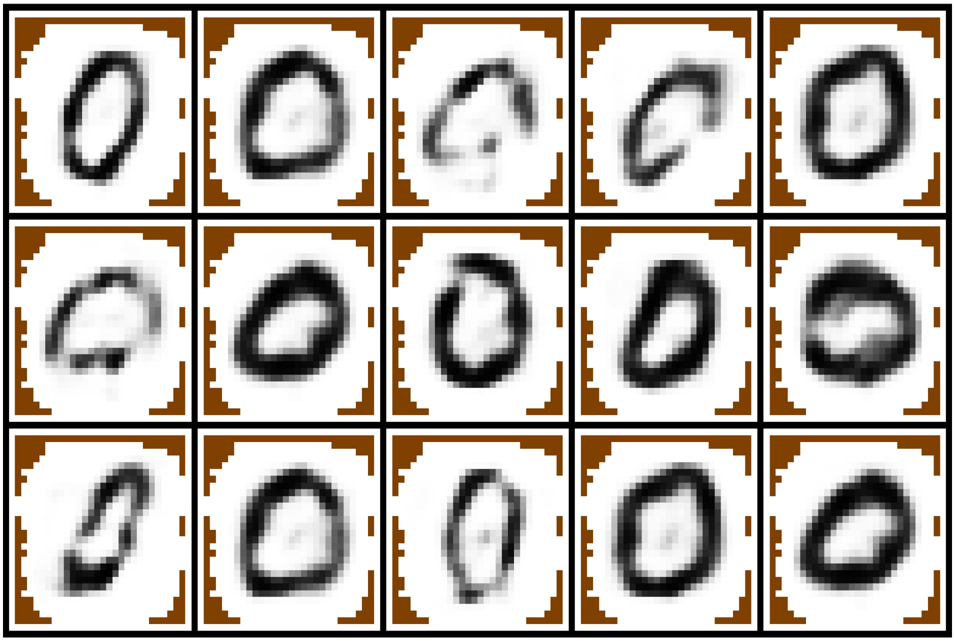


Figure 6.13: Generative samples from MNIST ‘0' class

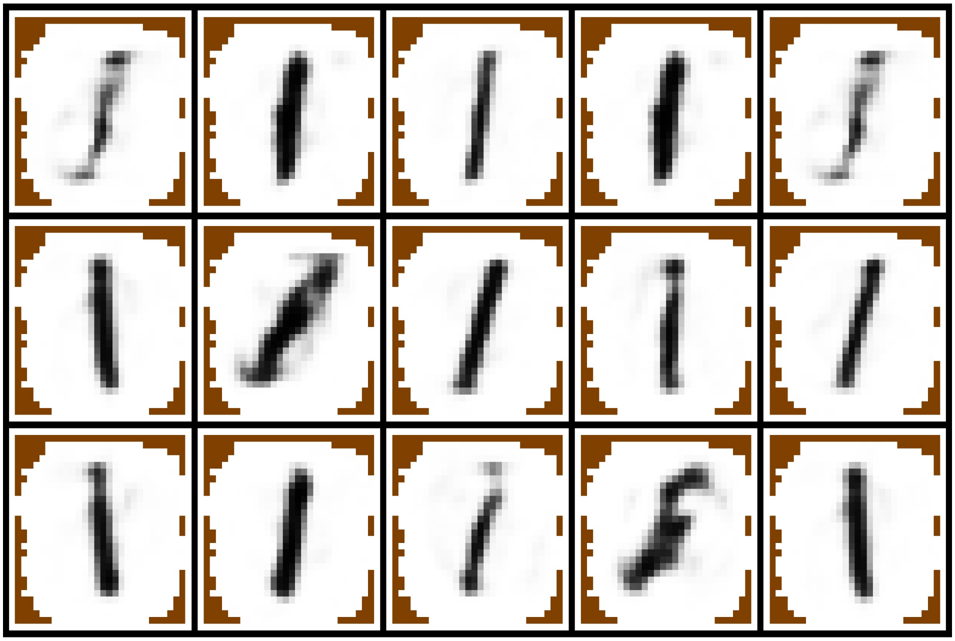


Figure 6.14: Generative samples from MNIST ‘1' class

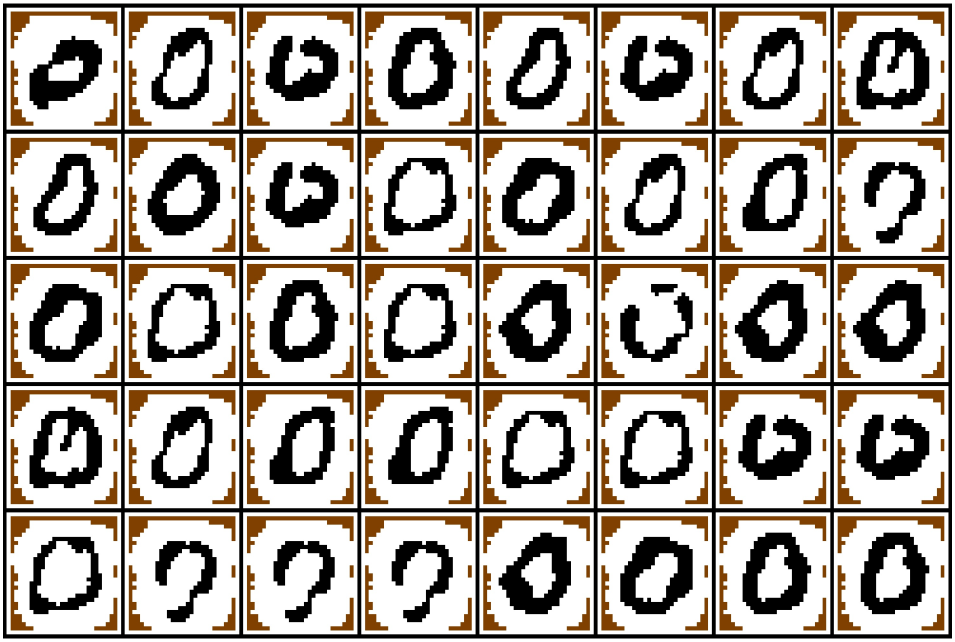


Figure 6.15: Binary generative samples from MNIST ‘0' class

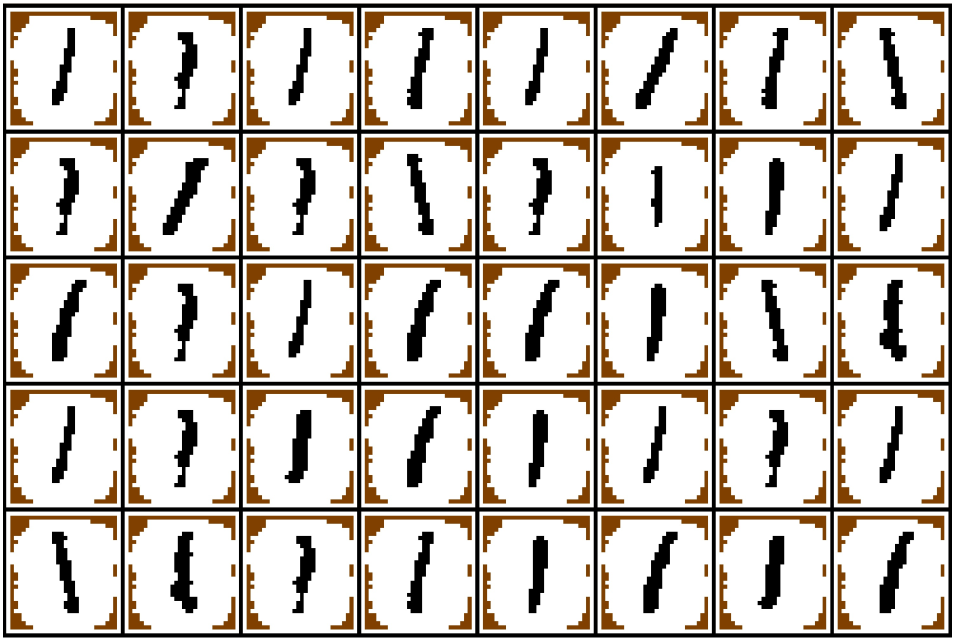


Figure 6.16: Binary generative samples from MNIST ‘1' class

Samples from a Path Series

We saw a discussion of the 'path' series type on Page 46, and specific instructions for generating a path series on Page 228. Figure 6.17 below shows a few generative samples from a path series modeling the OEX market index. The values of the trends are blue, and the velocities are red. Observe that, as expected, when the velocity is high the values are increasing, and when the velocity is low the values are decreasing.

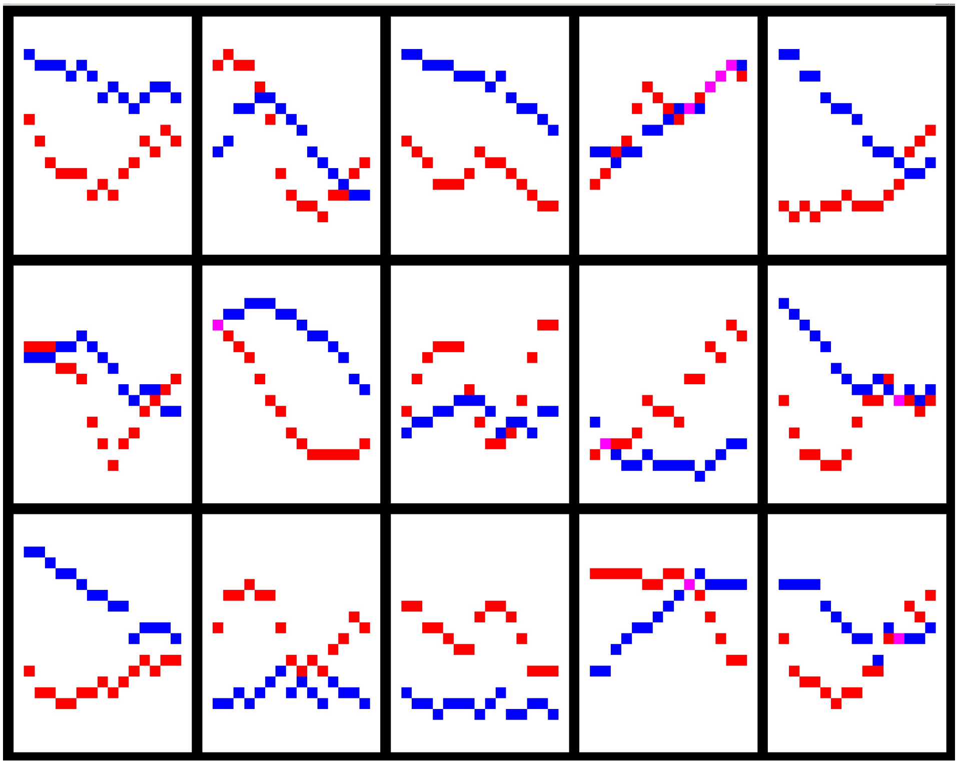


Figure 6.17: Generative samples from a Path series

The DEEP.LOG File

When a database or other file is read, the program creates a new file called DEEP.LOG in the same directory as the data file being read. If a file of that name already exists, it is erased. This log file begins by showing the directory in which it is created, along with the date and time. It then lists the mean and standard deviation of every variable read. Here is a typical example:

```
Deep (D:\DEEP\TEST\DEEP.LOG)  1/26/15  15:42:16

Found 23 variables in input file D:\DEEP\TEST\SYNTH.TXT

6304 cases read

Means and standard deviations...

    Variable      Mean        StdDev

     RAND0      0.00711     0.57541
     RAND1      0.01422     0.58043
     RAND2      0.01027     0.57694
     RAND3     -0.00765     0.58143
     RAND4      0.00713     0.57911
     RAND5     -0.01166     0.57263
     RAND6     -0.00648     0.57742
     RAND7     -0.01424     0.58015
     RAND8      0.00659     0.57533
     RAND9     -0.00366     0.57733
```

It then shows the architecture of the model, including the unsupervised and supervised sections:

```
Beginning training a model with the following architecture:

  There are 1 unsupervised layers, not including input
    Hidden layer has 5 neurons

  There are 1 supervised layers, including output
```

Since there is at least one RBM layer, the training parameters for this layer are listed:

```
Restricted Boltzmann Machine training parameters...
  Initial random iterations for starting weights = 50
  Number of batches = 24
  Markov chain length start = 1
  Markov chain length end = 4
  Markov chain length rate = 0.0050
  Learning rate = 0.05000
  Starting momentum = 0.10000
  Ending momentum = 0.90000
  Weight penalty = 0.00010
  Sparsity penalty = 0.00100
  Sparsity target = 0.10000
  Increment convergence criterion = 0.00001
  Max epochs with no improvement = 500
  Max epochs = 10000
  Visible layer using mean field, not stochastic
  Inputs will be rescaled to cover a range of 0-1
  Unsupervised section weights will be fine tuned by supervised training
```

The training parameters for the supervised section are also listed:

```
Supervised layer(s) training parameters...
  Initial annealing iterations for starting weights = 100
  Initial random range for starting weights = 1.00000
  Supervised optimization max iterations = 1000
  Supervised optimization convergence tolerance = 0.0000500
  Complete model optimization max iterations = 2000
  Complete model optimization convergence tolerance = 0.0000100
  Weight penalty = 0.00100
```

The results of training the unsupervised layer are printed first:

```
Training unsupervised layer 1
  Initial weight search RMS reconstruction error = 0.27098
  Unsupervised training complete; RMS reconstruction error = 0.31654
```

There is one curious issue in that result. The initial weight search gave a reproduction error of 0.27098, but after real training was done we see that the reproduction error has increased to 0.31654. How did this happen?

Actually, this is unusual, happening only when the input variables have little or no patterns that the RBM can learn. In this example, the inputs are all random numbers, so there are obviously no patterns. We must remember that the reconstruction error is measured slightly differently during weight initialization and training. In Volume I we see that the initial search reconstruction error is computed in a deterministic manner using mean field approximation in both directions. But during learning we use random sampling of the hidden neuron activations for the reconstruction error. This tends to increase the error somewhat. If the RBM is able to learn real patterns, the difference due to randomization during reconstruction error computation is swamped out by the model's ability to reconstruct authentic patterns. But if there are no patterns to reconstruct, we just get the effect of randomization.

After greedy training of the unsupervised section is complete, the supervised section which follows the unsupervised section is trained. Fine tuning was selected, so the last step is to tweak the entire model, unsupervised plus supervised sections. Here we see that fine tuning produces a huge improvement in the criterion, which is negative log likelihood in this example because a classification model was forced.

```
Optimization of supervised section is complete with negative log likelihood = 0.12270
Fine tuning of the entire model is complete with negative log likelihood = 0.02327
```

The targets are listed, and it is noted that the inputs are rescaled 0-1, so the weights which will be printed soon refer to these rescaled values.

```
Trained weights for this model, predicting the following target(s)...
  RAND1
  RAND2
  RAND3

Each raw input has been rescaled 0-1 to cover the min/max range.
Thus, all weights refer to the rescaled value, not the raw value.
```

The weights for the single unsupervised layer are now printed. If there were multiple layers, each set of weights would appear. These weights are *after* fine tuning.

```
Weights for unsupervised hidden layer 1

                  1         2         3         4         5

    Q mean    0.4522    0.4796    0.4556    0.4138    0.4717
  skewness    0.1310    0.0700    0.1271    0.2440    0.0618

     RAND1   -7.0347   -4.5028    0.9392   -2.5469    1.4879
     RAND2    4.7047   -1.7225   -0.5104    7.0462    2.0824
     RAND3    2.8726    6.0903    1.6480   -4.7952   -2.6467
     RAND4   -0.0131    0.1551   -1.6304   -0.2535    0.3858
     RAND5   -0.0032   -0.3523   -0.0453    0.1271   -0.8947
     RAND6   -0.0619   -0.1453   -1.8291   -0.1889   -0.2881
      BIAS    0.7231   -0.5790    0.8983   -0.6242    0.4237
```

The model was specified to have five hidden neurons, so we have five columns, one for each. At most ten columns are printed. After each unsupervised layer is trained, the hidden neuron weights are sorted so that the hidden neuron having maximum sum of absolute values becomes the first hidden neuron, and so forth. This way, if we examine the weights to obtain hints about interpretation of features detected, we can focus our efforts on the early columns. However, if fine tuning is done, as is the case in this example, this sorting can be subverted. This is not a practical problem, as fine tuning almost always largely or entirely destroys the interpretability of weight patterns that were discovered by the RBM.

The *Q mean* row is the mean activation of each hidden neuron, and the *skewness* row is the statistical skewness of the activations. In general, a positive skewness means that the neuron is usually off, and vice versa. These two values are computed *before* fine tuning; they refer to the actions of the trained RBM before its weights are adjusted by supervised fine tuning.

We then see the weights which connect the (last and only) unsupervised layer to the (first and only) supervised layer. Also, the final value of the optimization criterion, which we saw earlier, is repeated.

```
Weights for final (output) layer

Target 1 of 3: RAND1

   -9.158017  Unsupervised output 1
   -6.844571  Unsupervised output 2
    0.781757  Unsupervised output 3
   -2.436789  Unsupervised output 4
    2.660202  Unsupervised output 5
    6.449515  CONSTANT

Target 2 of 3: RAND2

    5.160418  Unsupervised output 1
   -1.469535  Unsupervised output 2
   -0.629605  Unsupervised output 3
    9.063721  Unsupervised output 4
    2.708100  Unsupervised output 5
   -8.018184  CONSTANT

Target 3 of 3: RAND3

    3.467198  Unsupervised output 1
    8.798016  Unsupervised output 2
    1.170767  Unsupervised output 3
   -8.699801  Unsupervised output 4
   -3.433103  Unsupervised output 5
   -2.159271  CONSTANT

Negative log likelihood = 0.02327
```

Last of all the confusion matrix is shown. Usually, when one is training a classifier, the target vector for each case has 1.0 in the position corresponding to the correct class, and 0.0 in all other positions. But this is just a common convention and is not required in DEEP. Instead, whichever target has the maximum value is defined to be the correct class. So when a model having continuous targets is forced to be a classifier, as is the situation in this example, results are reasonable. In particular, we would expect good classification in this example, since all three targets are also present as inputs! Indeed, we see this to be the case.

```
Confusion matrix... Row is true class, column is predicted class
  In each set of three rows for a true class, the first row is the count,
  the second row is the percent for that row (true class)
  and the third row is the percent of the entire dataset.

       1      2      3

1   2128      3      8
   99.49   0.14   0.37
   33.76   0.05   0.13

2      9   2088     15
    0.43  98.86   0.71
    0.14  33.12   0.24

3      8      7   2038
    0.39   0.34  99.27
    0.13   0.11  32.33

Total misclassification = 0.7931 percent
```

In DEEP 2.0, additional confusion matrices will be printed, showing the effect of using thresholds to classify only cases for which the model has varying degrees of certainty. A table of the thresholds for keeping assorted fractions of the training set is also printed.

Predictive Performance Measures

The example output just shown is for a classification model. We now discuss the performance measures that may be seen with a predictive model. The first and most basic statistics are the mean squared error, root-mean-squared error, and R-squared. This output will resemble the following:

```
Mean squared error and R-squared of target(s)...
  MSE of Lead_1 = 0.00007   RMS = 0.00860   RSQ = 0.00396
```

If there are multiple targets, these quantities will be printed separately for each, followed by a pooled value (all targets). It's important to note that these values are computed very differently in DEEP 2.x versus DEEP 1.x. In version 1, all targets were scaled according to their standard deviations in the training set so that MSE referred to standardized quantities, and these same scale factors were used for any test set(s). There are many advantages to doing it this way, but it led to massive confusion among users, especially when values of R-squared greater than one appeared! For this reason, version 2 now employs the 'traditional' approach of avoiding any scaling of the targets, and R-squared for a test set is now based on the variance in that test set, irrespective of the variance in the training set. Of course, negative R-squared values are still possible when a model is anti-predictive (it behaves worse than just guessing the mean.) This is fundamental to this test statistic.

The next test statistics are based on a performance figure commonly used in evaluating market trading systems. However, it is broadly applicable (and indeed very useful) in any application in which the goal is not so much to predict an exact value of the target as to predict whether the target will be positive or negative. Moreover, this application must not require that a decision be made; the user has the choice of examining a prediction and then choosing whether to act on this prediction. There is little or no penalty for deliberately ignoring a prediction.

For example, suppose our model predicts the price change of a market in the upcoming day. Based on the prediction we may or may not submit an order to trade. Perhaps the prediction is that the market will rise 1 percent and we take a long position (we buy). Now suppose that we are off by 4 percent in the prediction. Maybe the market actually rises by 5 percent. Or maybe it falls by 3 percent. These two possibilities are equal errors, yet the first error is not a problem at all (!) while the second error is a disaster. So our 'performance penalty' must be similarly asymmetric. Moreover, if the prediction is that the market will rise by 0.0001 percent we would probably pass on a trade, while a predictied rise of 8 percent will definitely get our attention. Our performance criterion must take this into account.

Suppose we define a threshold for predictions. Given this threshold, we define a *win* as the true value of a target being positive when the prediction equals or exceed the threshold, or the true value being negative when the prediction is less than the threshold. Similarly, we experience a *loss* when the true value is negative when the prediction equals or exceeds the threshold, or when the true value is positive while the prediction is less than the threshold.

The condition of the prediction equaling or exceeding the threshold is called *long*, and the condition of the prediction being less than the threshold is called *short*, again in deference to related issues in market trading; an automated trading system would take a long position when the prediction is large, and a short position when the prediction is small (very negative).

A table similar to the one shown on the next page is printed. It is preceded by the *long ratio*, the sum of all positive targets divided by the (absolute) sum of all negative targets. The *net* is the sum of all targets, which of course equals the sum of all positive targets minus the (absolute) sum of all negative targets. The corresponding *short* values are the reciprocal and negative, respectively, of the long values. These serve as baselines for comparison.

```
Total Win vs. Total Loss above and below various fractions
  (For all tested cases, long ratio = 1.1218 (net=3195.592) and short ratio = 0.8914
(net=-3195.592)

Threshold  Frac Gtr/Eq  Ratio     Net      Frac Less   Ratio       Net

  -0.901      0.990    1.1261  3265.0136     0.010    1.2484     69.4217
  -0.481      0.950    1.1376  3387.0308     0.050    1.1331    191.4389
  -0.279      0.900    1.1489  3455.2076     0.100    1.0937    259.6157
  -0.042      0.800    1.1687  3495.6538     0.200    1.0574    300.0619
   0.120      0.700    1.1829  3371.6175     0.300    1.0231    176.0256
   0.253      0.600    1.1971  3160.0880     0.400    0.9965    -35.5039
   0.374      0.500    1.2478  3340.0525     0.500    1.0114    144.4606
   0.500      0.400    1.2727  3008.0249     0.600    0.9878   -187.5670
   0.642      0.300    1.3077  2608.6799     0.700    0.9680   -586.9120
   0.819      0.200    1.3017  1828.5688     0.800    0.9366  -1367.0231
   1.072      0.100    1.3382  1106.7696     0.900    0.9166  -2088.8223
   1.307      0.050    1.4954   831.8562     0.950    0.9122  -2363.7357
   1.789      0.010    1.7472   273.0242     0.990    0.8985  -2922.5677
```

The rows of this table are computed so as to at least approximately cover preset fractiles of the data distribution, although this may be subverted if there are numerous tied predictions. The leftmost column shows the numerical values of the thresholds corresponding to the preset fractiles. The second column shows the fraction of cases whose prediction equals or exceeds the corresponding threshold. The fifth column is the complement, the fraction of cases whose prediction is less than the threshold.

The third and fourth columns are relevant to the situation of the prediction equaling or exceeding the threshold. The *Ratio* is the sum of wins (positive targets) divided by the (absolute) sum of losses (negative targets). The *Net* is the sum of all targets for theses cases whose predictions equal or exceed the threshold.

Columns six and seven are for cases whose predictions are less than the threshold. For this ratio, wins (the numerator) are negative targets and losses are positive targets.

We see in the line above the table that the long ratio (sum of all positive targets divided by absolute sum of all negative targets) is 1.1218. So it's no surprise that the ratio in the top row of the third column, 1.1261, is very close to this value, as this subset consists of 99 percent of all cases. As we drop to lower rows, corresponding to increasing thresholds, we see the win/loss ratio steadily increasing, until when we reach the point of examining only the highest 1 percent of predictions, the ratio peaks at 1.7472. The net decreases because we have fewer and fewer cases going into the sum.

The reverse happens in the ratio column for cases below the threshold. In the bottom row (99 percent of cases) we have a ratio of 0.8985, close to the entire-set ratio of 0.8914. But by the time we get to the top line, with just 1 percent of the cases having a prediction this small, the win/loss ratio is up to 1.2484.

Not shown here is the fact that this model happens to have a *negative* R-squared! This phenomenon is common in ultra-high-noise situations. Predictive power may be totally nonexistent throughout the majority of the data distribution, but be respectable in the extreme tails. For this reason, charts like the one shown here can be invaluable.

The chart just shown is always computed based on the distribution in the dataset being tested. This generally reveals the most information. However, for out-of-sample testing, this method does have the disadvantage that in real-time applications, such as financial market trading, these thresholds cannot be known in advance. The chart produced by pooling all OOS cases relies on thresholds that would not be known until the entire OOS set has been processed, an obvious impossibility in real time. This does *not* necessarily introduce any bias, optimistic or pessimistic. However, it is an annoyance.

In order to handle this complaint, the *Test* and *Cross validate* functions print one small additional set of statistics. The three tail thresholds, 0.01, 0.05, and 0.10, both long and short, are preserved for the training set. These same thresholds are then used to compute the win/loss ratios and net sums in any subsequent test set. A typical set of statistics is shown on the next page. For this example, I deliberately used the training data as a test set so that the reader can compare the results here with those in the prior table. (Please do so.) When different datasets are used, the usual situation, the percent of cases above/below the threshold generally varies from the presets. Here, because the same datasets are used for training and testing, they are virtually identical.

```
Out-of-sample performance at tails, based on training-set thresholds

Target variable Lead_1
0.01    Long n=83 (1.01 Pct) Ratio=1.747  Net=273.024
        Short n=81 (0.99 Pct) Ratio=1.248  Net=69.422

0.05    Long n=411 (5.01 Pct) Ratio=1.495  Net=831.856
        Short n=409 (4.99 Pct) Ratio=1.133  Net=191.439

0.10    Long n=821 (10.01 Pct) Ratio=1.338  Net=1106.770
        Short n=819 (9.99 Pct) Ratio=1.094  Net=259.616
```