PyDeep Documentation

Release 1.1.0

Jan Melchior

May 19, 2022

Contents:

PyDeep is a machine learning / deep learning library with focus on unsupervised learning. The library has a modular design, is well documented and purely written in Python/Numpy. This allows you to understand, use, modify, and debug the code easily. Furthermore, its extensive use of unittests assures a high level of reliability and correctness.

CHAPTER 1

Welcome

1.1 Welcome

PyDeep is a machine learning / deep learning library with focus on unsupervised learning. The library has a modular design, is well documented and purely written in Python/Numpy. This allows you to understand, use, modify, and debug the code easily. Furthermore, its extensive use of unittests assures a high level of reliability and correctness.

1.1.1 News

- Auto encoder module added including denoising, sparse, contractive, slowness AE's
- Unittests added, examples
- tutorials added
- Upcoming (short-term): Deep Boltzmann machines will be added
- Upcoming (short-term): Feed Forward neural networks will be added
- Future:
- Future: RBM/DBM in tensorFlow

1.1.2 Features index

- Principal Component Analysis (PCA)
  - Zero Phase Component Analysis (ZCA)
- Independent Component Analysis (ICA)
- Autoencoder
  - Centered denoising autoencoder including various noise functions
  - Centered contractive autoencoder

  - Centered sparse autoencoder
  - Centered slowness autoencoder
  - Several regularization methods like l1,l2 norm, Dropout, gradient clipping, . . .
- Restricted Boltzmann machines
  - centered BinaryBinary RBM (BB-RBM)
  - centered GaussianBinary RBM (GB-RBM) with fixed variance
  - centered GaussianBinaryVariance RBM (GB-RBM) with trainable variance
  - centered BinaryBinaryLabel RBM (BBL-RBM)
  - centered GaussianBinaryLabel RBM (GBL-RBM)
  - centered BinaryRect RBM (BR-RBM)
  - centered RectBinary RBM (RB-RBM)
  - centered RectRect RBM (RR-RBM)
  - centered GaussianRect RBM (GR-RBM)
  - centered GaussianRectVariance RBM (GRV-RBM)
  - Sampling Algorithms for RBMs
    - Gibbs Sampling
    - Persistent Gibbs Sampling
    - Parallel Tempering Sampling
    - Independent Parallel Tempering Sampling
  - Training for RBMs
    - Exact gradient (GD)
    - Contrastive Divergence (CD)
    - Persistent Contrastive Divergence (PCD)
    - Independent Parallel Tempering Sampling
  - Log-likelihodd estimation for RBMs
    - Exact Partition function
    - Annealed Importance Sampling (AIS)
    - reverse Annealed Importance Sampling (AIS)

1.1.3 Scientific use

The library contains code I have written during my PhD research allowing you to reproduce the results described in the following publications.

- Gaussian-binary restricted Boltzmann machines for modeling natural image statistics. Melchior, J., Wang, N., & Wiskott, L.. (2017). PLOS ONE, 12(2), 1–24.
- How to Center Deep Boltzmann Machines. Melchior, J., Fischer, A., & Wiskott, L.. (2016). Journal of Machine Learning Research, 17(99), 1–61.

- Gaussian-binary Restricted Boltzmann Machines on Modeling Natural Image statistics Wang, N., Melchior, J., & Wiskott, L.. (2014). (Vol. 1401.5900). arXiv.org e-Print archive.
- How to Center Binary Restricted Boltzmann Machines (Vol. 1311.1354). Melchior, J., Fischer, A., Wang, N., & Wiskott, L.. (2013). arXiv.org e-Print archive.
- An Analysis of Gaussian-Binary Restricted Boltzmann Machines for Natural Images. Wang, N., Melchior, J., & Wiskott, L.. (2012). In Proc. 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Apr 25–27, Bruges, Belgium (pp. 287–292).
- Learning Natural Image Statistics with Gaussian-Binary Restricted Boltzmann Machines. Melchior, J, 29.05.2012. Master's thesis, Applied Computer Science, Univ. of Bochum, Germany.

If you want to use PyDeep in your publication, you can cite it as follows.

```
@misc{melchior2018pydeep,
      title={PyDeep},
      author={Melchior, Jan},
      year={2018},
      publisher={GitHub},
      howpublished={\url{https://github.com/MelJan/PyDeep.git}},
     }
```

1.1.4 Contact

Jan Melchior

CHAPTER 2

Installation

2.1 Installation

To install PyDeep, first download it from GitHub/MelJan. Then simply change to the PyDeep folder and run the setup script:

```
python setup.py install
```

2.1.1 Dependencies

PyDeep has the following dependencies:

2.1.1.1 Hard dependencies:

- numpy
- scipy

2.1.1.2 Soft dependencies

- matplotlib
- cPickle
- encryptedpickle
- paramiko
- mdp

2.1.2 Optimized backend

It is highly recommended to use an multi-threading optimized linear algebra backend such as

- Automatically Tuned Linear Algebra Software (ATLAS)
- Intel® Math Kernel Library (Intel® MKL)

-> Hint: MKL is inlcuded in Enthought which provides a free academic license.

2.1.3 Unit tests

To test whether PyDeep functions properly you can run unittest:

```
python -m unittest discover testunits
```

In this case you test everything, which can take several minutes up to an hour.

CHAPTER 3

Tutorials

3.1 Tutorials

In this section you will find tutorials for several algorithms like PCA, ICA, RBMs, . . . giving you an idea of how you can use the library.

3.1.1 Principal Component Analysis on a 2D example.

Example for Principal Component Analysis (PCA) on a linear 2D mixture.

3.1.1.1 Theory

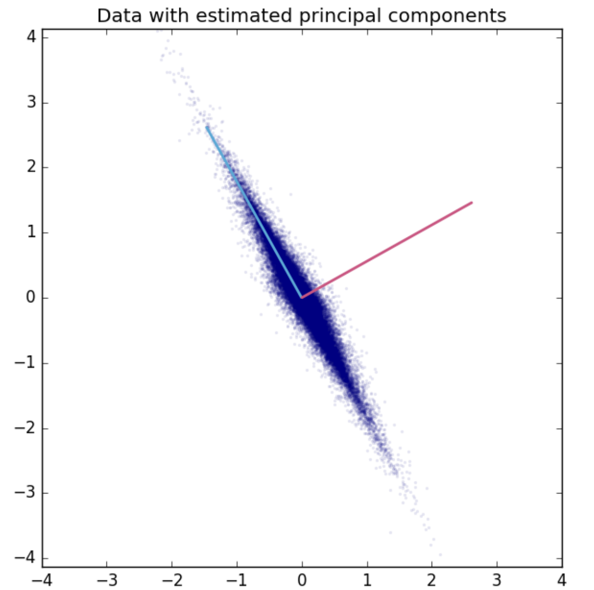
If you are new on PCA, a good theoretical introduction is given by the Course Material in combination with the following video lectures.

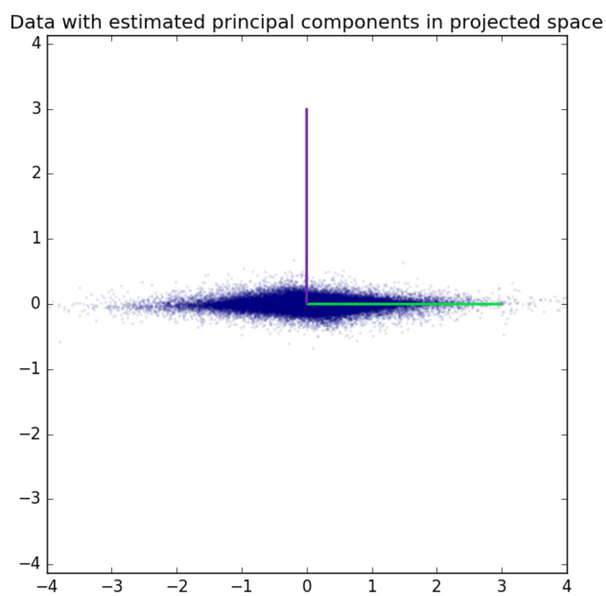
3.1.1.2 Results

The *code* given below produces the following output.

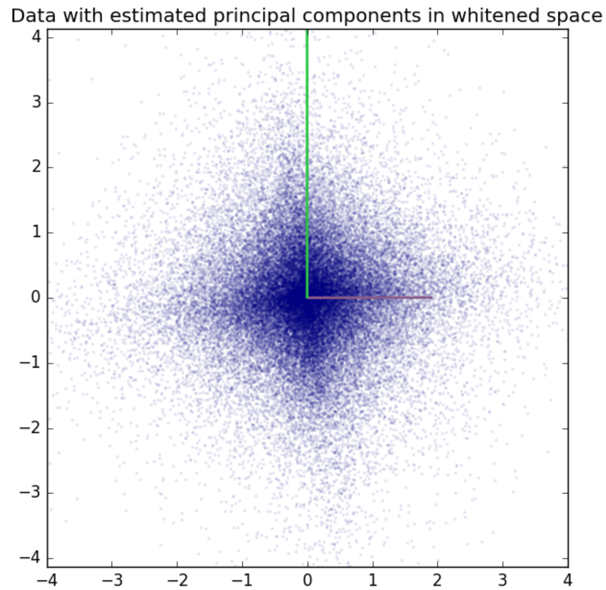
The data is plotted with the extracted principal components.

Data and extracted principal components can also be plotted in the projected space.





The PCA-class can also perform whitening. Data and extracted principal components are plotted in the whitened space.



For a real-world application see the PCA_eigenfaces example.

3.1.1.3 Source code

```
""" Example for the Principal Component Analysis on a 2D example.

    :Version:
        1.1.0

    :Date:
        22.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.
```

(continues on next page)

(continued from previous page)

```
        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""

# Import numpy, numpy extensions, PCA, 2D linear mixture, and visualization module
import numpy as numx
from pydeep.preprocessing import PCA
from pydeep.misc.toyproblems import generate_2d_mixtures
import pydeep.misc.visualization as vis

# Set the random seed
# (optional, if stochastic processes are involved we get the same results)
numx.random.seed(42)

# Create 2D linear mixture, 50000 samples, mean = 0, std = 3
data, _ = generate_2d_mixtures(num_samples=50000,
                               mean=0.0,
                               scale=3.0)

# PCA
pca = PCA(data.shape[1])
pca.train(data)
data_pca = pca.project(data)

# Display results

# For better visualization the principal components are rescaled
scale_factor = 3

# Figure 1 - Data with estimated principal components
vis.figure(0, figsize=[7, 7])
vis.title("Data with estimated principal components")
vis.plot_2d_data(data)
vis.plot_2d_weights(scale_factor*pca.projection_matrix)
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# Figure 2 - Data with estimated principal components in projected space
vis.figure(2, figsize=[7, 7])
vis.title("Data with estimated principal components in projected space")
vis.plot_2d_data(data_pca)
vis.plot_2d_weights(scale_factor*pca.project(pca.projection_matrix.T))
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# PCA with whitening
pca = PCA(data.shape[1], whiten=True)
pca.train(data)
data_pca = pca.project(data)

# Figure 3 - Data with estimated principal components in whitened space
```

(continues on next page)

(continued from previous page)

```
vis.figure(3, figsize=[7, 7])
vis.title("Data with estimated principal components in whitened space")
vis.plot_2d_data(data_pca)
vis.plot_2d_weights(pca.project(pca.projection_matrix.T).T)
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# Show all windows
vis.show()
```

3.1.2 Eigenfaces

Example for Principal Component Analysis (PCA) on face images also known as Eigenfaces

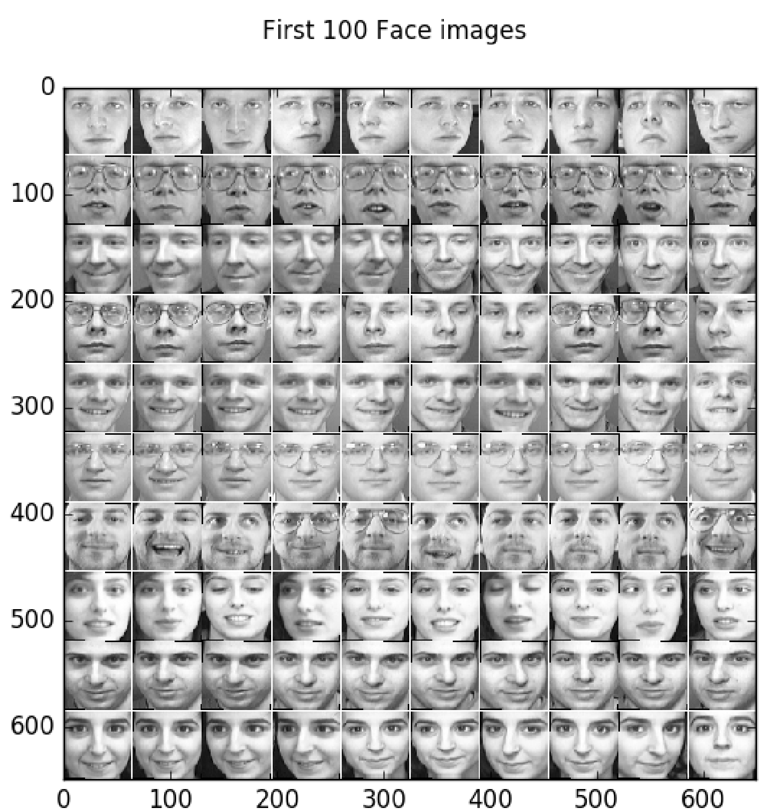
3.1.2.1 Theory

If you are new on PCA, first see PCA_2D_example.

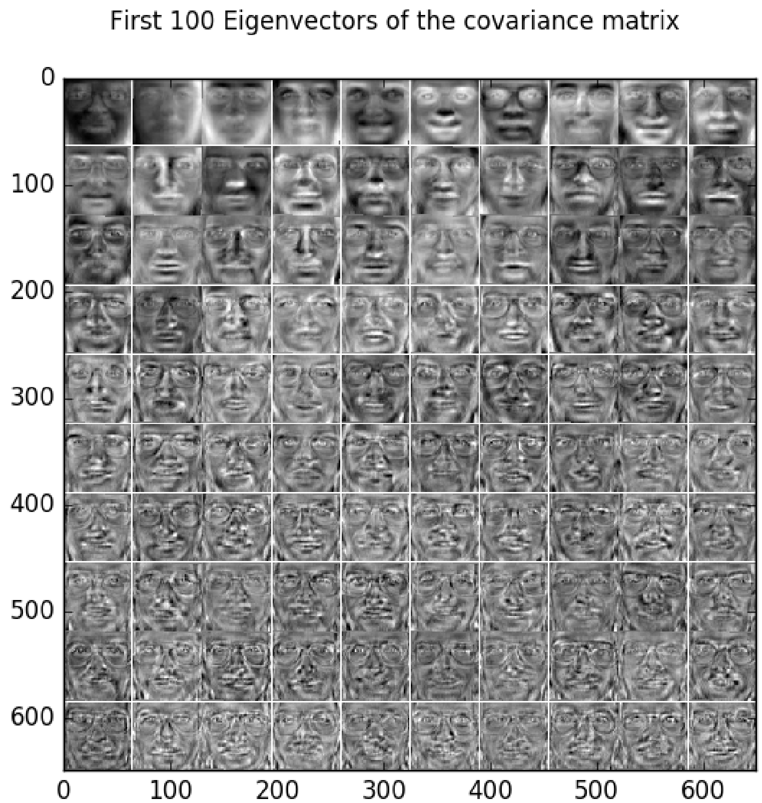
3.1.2.2 Results

The *code* given below produces the following output.

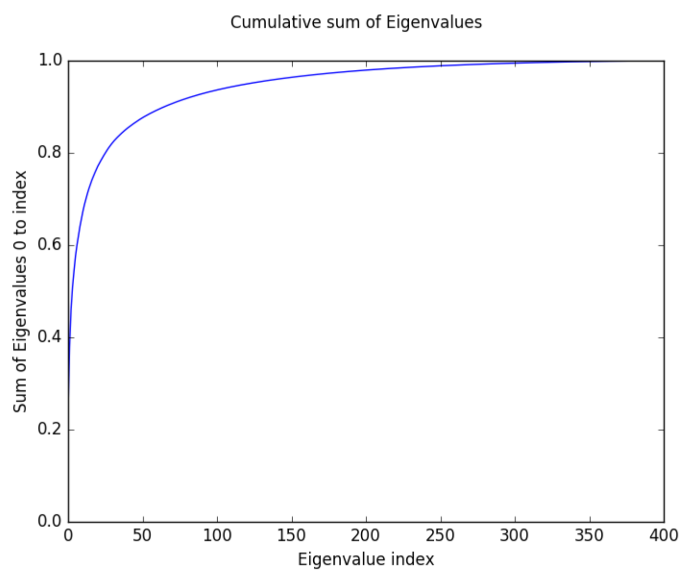
Some examples of the face images of the olivetti face dataset.



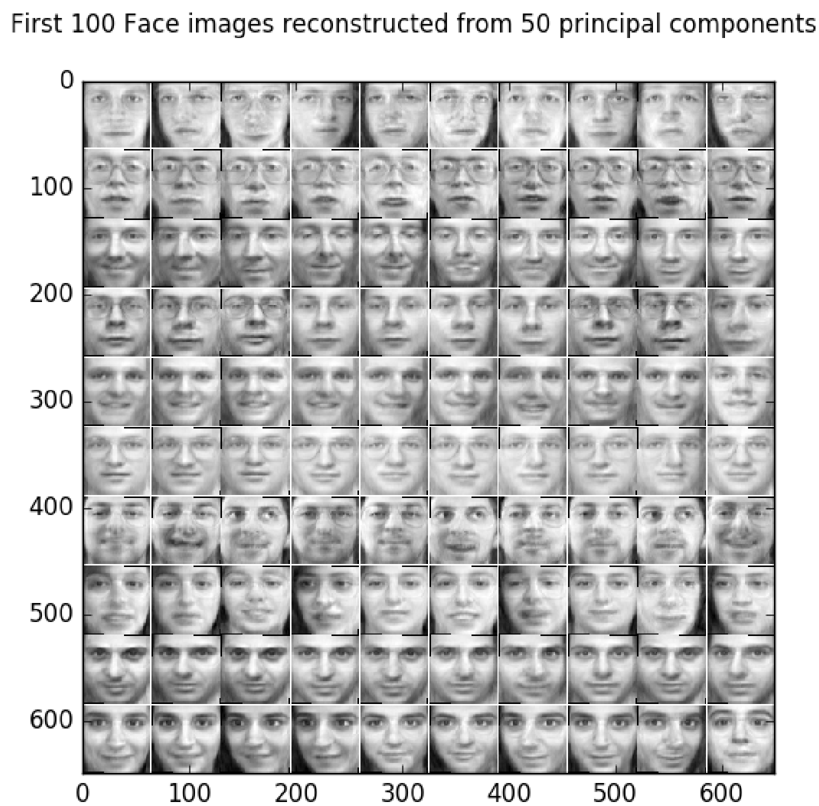
The first 100 principal components extracted from the dataset. The components focus on characteristics like glasses, lighting direction, nose shape, . . .



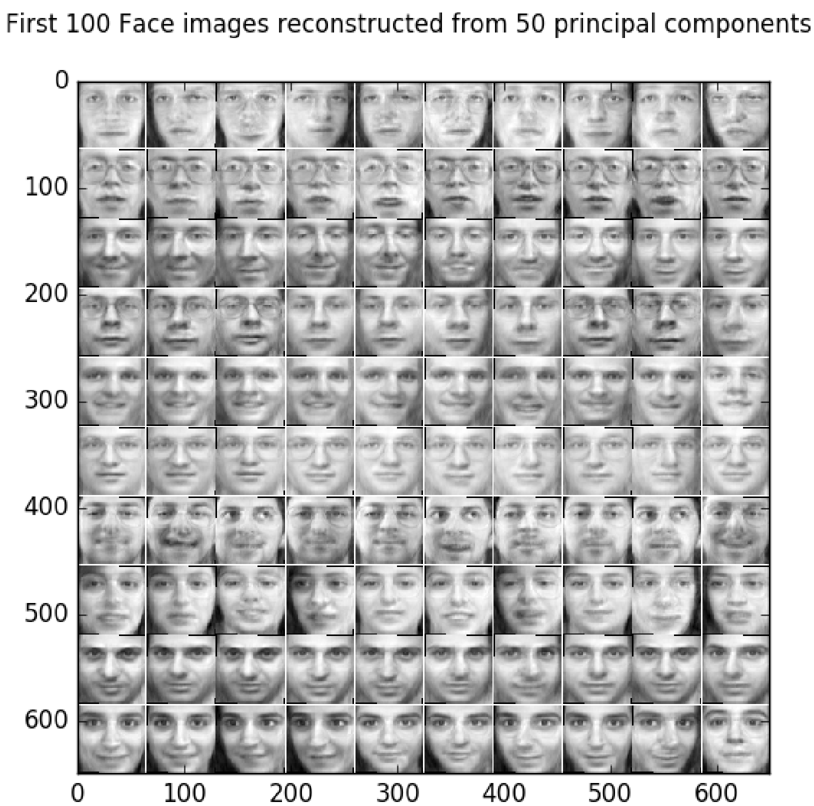
The cumulative sum of the Eigenvalues show how 'compressable' the dataset is.



For example using only the first 50 eigenvectors retains 87,5 % of the variance of data and the reconstructed images look as follows.



For 200 eigenvectors we retain 98,0 % of the variance of the data and the reconstructed images look as follows.



Comparing the results with the original images shows that the data can be compressed to 50 dimensions with an acceptable error.

3.1.2.3 Source code

```
""" Example for Principal component analysis on face images (Eigenfaces).

    :Version:
        1.1.0

    :Date:
        22.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:
```

(continues on next page)

(continued from previous page)

```
        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# Import numpy, PCA, input output module, and visualization module
import numpy as numx
from pydeep.preprocessing import PCA
import pydeep.misc.io as io
import pydeep.misc.visualization as vis

# Set the random seed
# (optional, if stochastic processes are involved we get the same results)
numx.random.seed(42)

# Load data (download is not existing)
data = io.load_olivetti_faces(path='olivettifaces.mat')

# Specify image width and height for displaying
width = height = 64

# PCA
pca = PCA(input_dim=width * height)
pca.train(data=data)

# Show the first 100 eigenvectors of the covariance matrix
eigenvectors = vis.tile_matrix_rows(matrix=pca.projection_matrix,
                                    tile_width=width,
                                    tile_height=height,
                                    num_tiles_x=10,
                                    num_tiles_y=10,
                                    border_size=1,
                                    normalized=True)
vis.imshow_matrix(matrix=eigenvectors,
                  windowtitle='First 100 Eigenvectors of the covariance matrix')

# Show the first 100 images
images = vis.tile_matrix_rows(matrix=data[0:100].T,
                              tile_width=width,
                              tile_height=height,
                              num_tiles_x=10,
                              num_tiles_y=10,
                              border_size=1,
```

(continues on next page)

(continued from previous page)

```
                              normalized=True)
vis.imshow_matrix(matrix=images,
                  windowtitle='First 100 Face images')

# Plot the cumulative sum of teh Eigenvalues.
eigenvalue_sum = numx.cumsum(pca.eigen_values / numx.sum(pca.eigen_values))
vis.imshow_plot(matrix=eigenvalue_sum,
                windowtitle="Cumulative sum of Eigenvalues")
vis.xlabel("Eigenvalue index")
vis.ylabel("Sum of Eigenvalues 0 to index")
vis.ylim(0, 1)
vis.xlim(0, 400)

# Show the first 100 Face images reconstructed from 50 principal components
recon = pca.unproject(pca.project(data[0:100], num_components=50)).T
images = vis.tile_matrix_rows(matrix=recon,
                              tile_width=width,
                              tile_height=height,
                              num_tiles_x=10,
                              num_tiles_y=10,
                              border_size=1,
                              normalized=True)
vis.imshow_matrix(matrix=images,
                  windowtitle='First 100 Face images reconstructed from 50 '
                              'principal components')

# Show the first 100 Face images reconstructed from 120 principal components
recon = pca.unproject(pca.project(data[0:100], num_components=200)).T
images = vis.tile_matrix_rows(matrix=recon,
                              tile_width=width,
                              tile_height=height,
                              num_tiles_x=10,
                              num_tiles_y=10,
                              border_size=1,
                              normalized=True)
vis.imshow_matrix(matrix=images,
                  windowtitle='First 100 Face images reconstructed from 200 '
                              'principal components')

# Show all windows.
vis.show()
```

3.1.3 Independent Component Analysis on a 2D example.

Example for Independent Component Analysis (ICA) used for blind source separation on a linear 2D mixture.

3.1.3.1 Theory

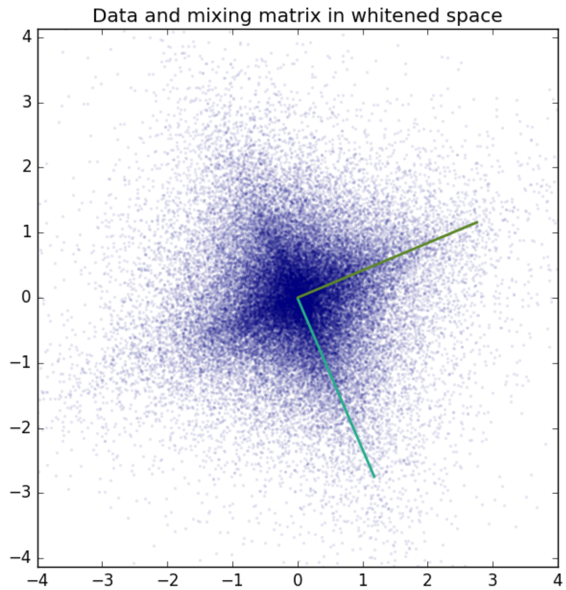
If you are new on ICA and blind source separation, a good theoretical introduction is given by the Course Material in combination with the following video lectures.

and

3.1.3.2 Results

The *code* given below produces the following output.

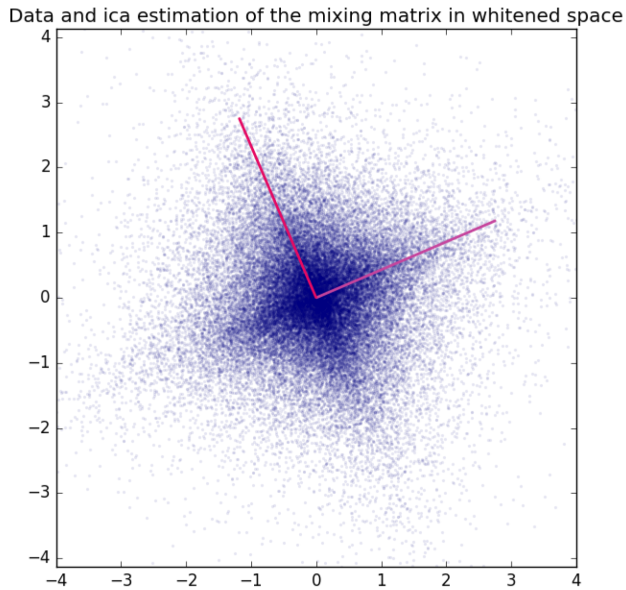
Visualization of the data and true mixing matrix projected to the whitened space.



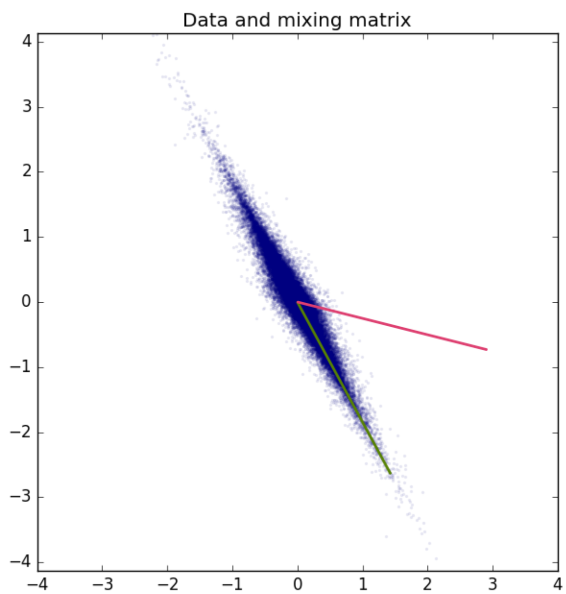
Visualization of the whitened data with the ICA projection matrix, that is the estimation of the whitened mixing matrix. Note that ICA is invariant to sign flips of the sources. The columns of the estimated mixing matrix are most likely a permutation of the columns of the original mixing matrix and can also be a 180 degrees rotated version (original vector multiplied by -1). The Amari distance is invariant to permutations and flips of the matrix columns and can thus be used to compare to mixing matrices.

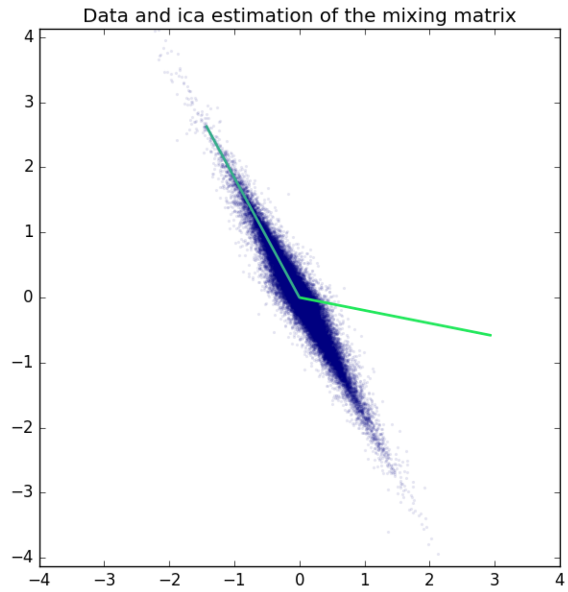
Amari distanca between true mixing matrix and estimated mixing matrix: .. code-block:: Python

0.00989836830489



We can also project the ICA projection matrix back to the original space and compare the results in the original space.





The log-likelihood on all data is:

```
log-likelihood on all data: -2.73863050034
```

For a real-world application see the ICA_natural_images example.

3.1.3.3 Source code

```
""" Example for the Independent Component Analysis on a 2D example.

    :Version:
        1.1.0

    :Date:
        22.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.
```

(continues on next page)

(continued from previous page)

```
        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# Import numpy, numpy extensions, ZCA, ICA, 2D linear mixture, and visualization
import numpy as numx
import pydeep.base.numpyextension as numxext
from pydeep.preprocessing import ZCA, ICA
from pydeep.misc.toyproblems import generate_2d_mixtures
import pydeep.misc.visualization as vis

# Set the random seed
# (optional, if stochastic processes are involved we get the same results)
numx.random.seed(42)

# Create 2D linear mixture, 50000 samples, mean = 0, std = 3
data, mixing_matrix = generate_2d_mixtures(num_samples=50000,
                                           mean=0.0,
                                           scale=3.0)

# Zero Phase Component Analysis (ZCA) - Whitening in original space
zca = ZCA(data.shape[1])
zca.train(data)
whitened_data = zca.project(data)

# Independent Component Analysis (ICA)
ica = ICA(whitened_data.shape[1])

ica.train(whitened_data, iterations=100, status=False)
data_ica = ica.project(whitened_data)

# print the ll on the data
print("Log-likelihood on all data: "+str(numx.mean(
    ica.log_likelihood(data=whitened_data))))

print("Amari distanca between true mixing matrix and estimated mixing matrix: "+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T), ica.projection_matrix.
↪T)))

# For better visualization the principal components are rescaled
scale_factor = 3

# Display results: the matrices are normalized such that the
# column norm equals the scale factor

# Figure 1 - Data and mixing matrix
```

(continues on next page)

(continued from previous page)

```
vis.figure(0, figsize=[7, 7])
vis.title("Data and mixing matrix")
vis.plot_2d_data(data)
vis.plot_2d_weights(numxext.resize_norms(mixing_matrix,
                                         norm=scale_factor,
                                         axis=0))
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# Figure 2 - Data and mixing matrix in whitened space
vis.figure(1, figsize=[7, 7])
vis.title("Data and mixing matrix in whitened space")
vis.plot_2d_data(whitened_data)
vis.plot_2d_weights(numxext.resize_norms(zca.project(mixing_matrix.T).T,
                                         norm=scale_factor,
                                         axis=0))
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# Figure 3 - Data and ica estimation of the mixing matrix in whitened space
vis.figure(2, figsize=[7, 7])
vis.title("Data and ICA estimation of the mixing matrix in whitened space")
vis.plot_2d_data(whitened_data)
vis.plot_2d_weights(numxext.resize_norms(ica.projection_matrix,
                                         norm=scale_factor,
                                         axis=0))
vis.axis('equal')
vis.axis([-4, 4, -4, 4])

# Figure 3 - Data and ica estimation of the mixing matrix
vis.figure(3, figsize=[7, 7])
vis.title("Data and ICA estimation of the mixing matrix")
vis.plot_2d_data(data)
vis.plot_2d_weights(
    numxext.resize_norms(zca.unproject(ica.projection_matrix.T).T,
                         norm=scale_factor,
                         axis=0))
vis.axis('equal')
vis.axis([-4, 4, -4, 4])



# Show all windows
vis.show()
```

3.1.4 Independent Component Analysis on a natural image patches

Example for Independent Component Analysis (ICA) on natural image patches. The independent components (columns of the ICA projection matrix) of natural image patches are edge detector filters.

3.1.4.1 Theory

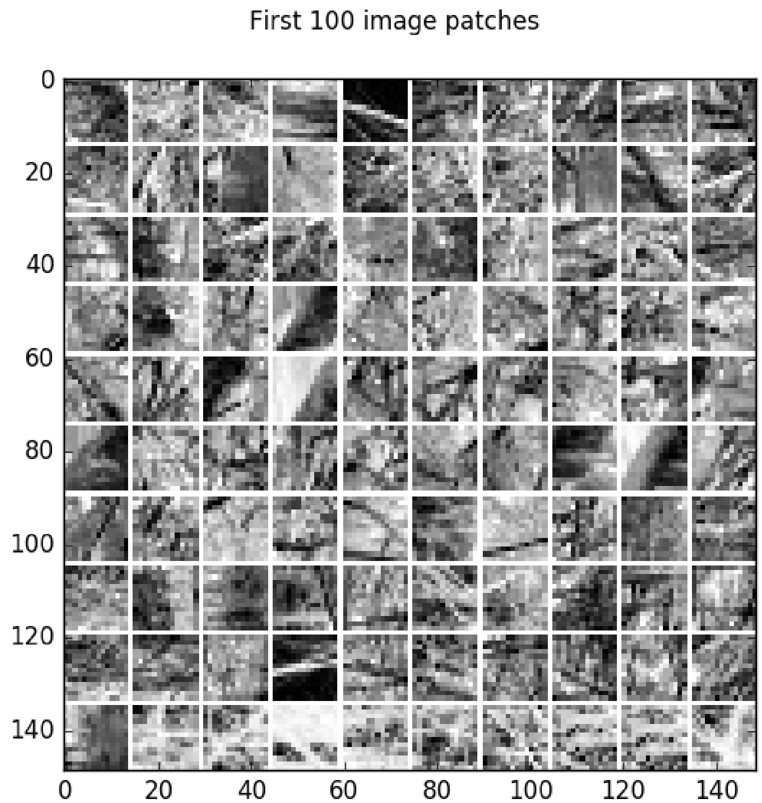
If you are new on ICA and blind source separation, first see ICA_2D_example.

For a comparison of ICA and GRBMs on natural image patches see Gaussian-binary restricted Boltzmann machines for modeling natural image statistics. Melchior et. al. PLOS ONE 2017.

3.1.4.2 Results

The *code* given below produces the following output.

Visualization of 100 examples of the gray scale natural image dataset.



The corresponding whitened image patches.

The learned filters/independent components learned from the whitened natural image patches.

The log-likelihood on all data is:

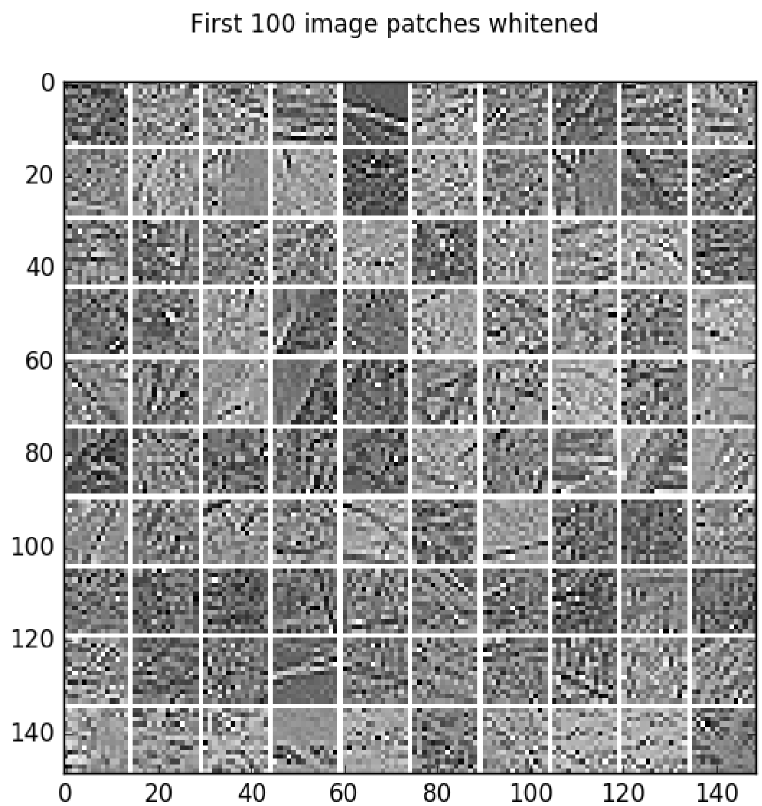
```
log-likelihood on all data: -260.064878919
```

To analyze the optimal response of the learn filters we can fit a Gabor-wavelet parametrized in angle and frequency, and plot the optimal grating, here for 20 filters

as well as the corresponding tuning curves, which show the responds/activities as a function frequency in pixels/cycle (left) and angle in rad (right).

Furthermore, we can plot the histogram of all filters over the frequencies in pixels/cycle (left) and angles in rad (right).

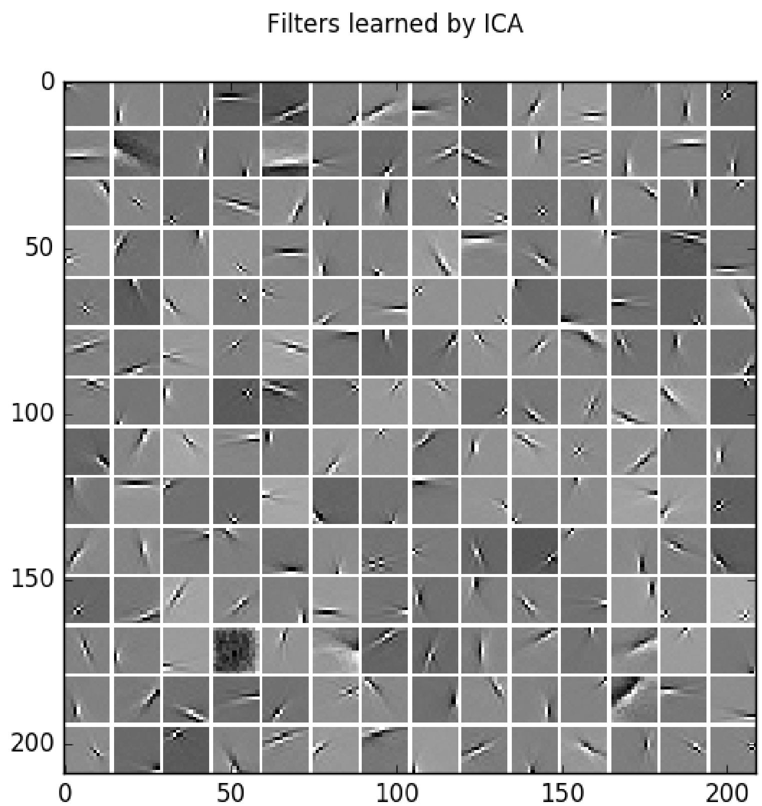
See also GRBM_natural_images. and AE_natural_images.



3.1.4.3 Source code

```
""" Example for the Independent Component Analysis (ICA) on natural image patches.

    :Version:
        1.1.0

    :Date:
        22.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.
```

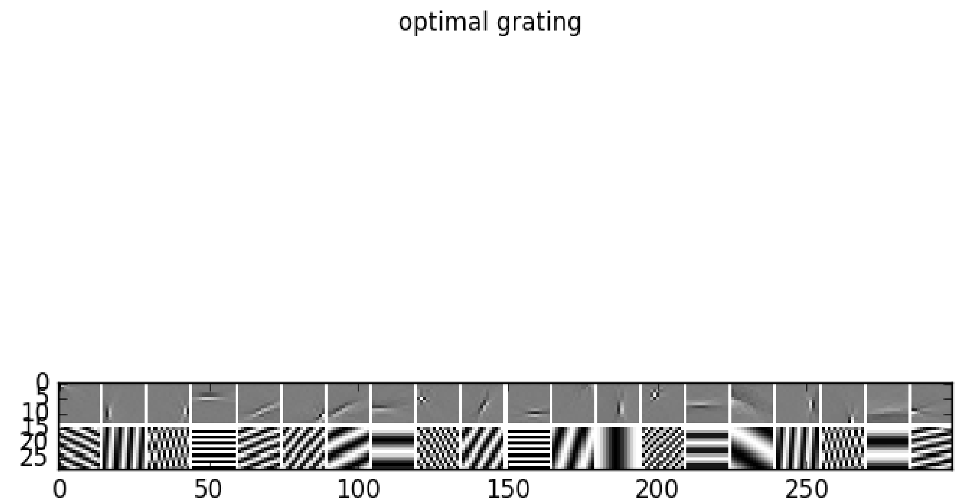
(continues on next page)

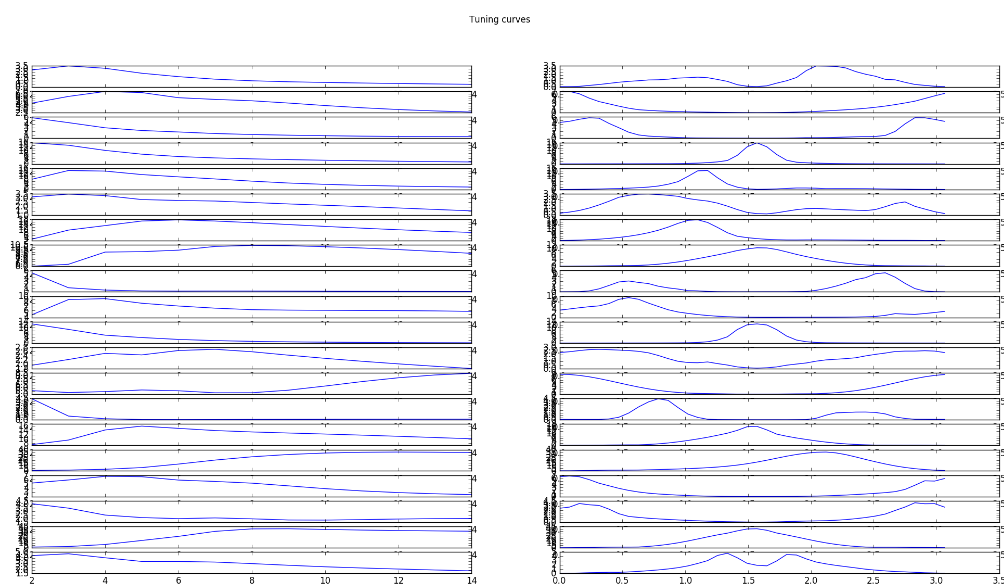


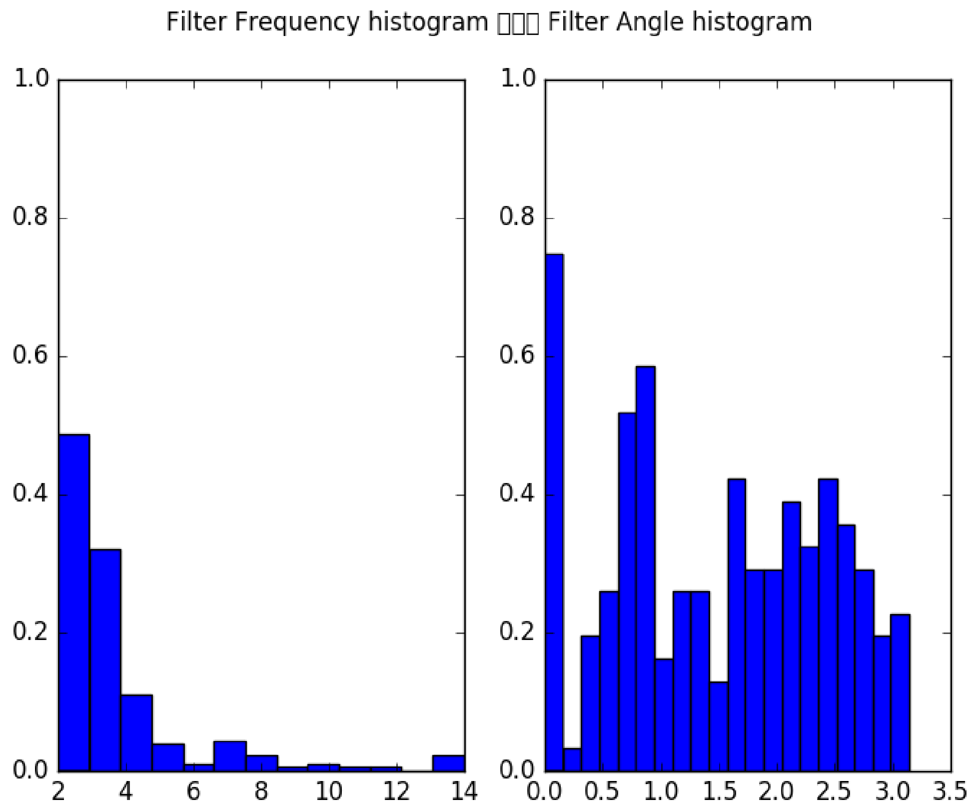
(continued from previous page)

```
        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# Import ZCA, ICA, numpy, input output functions, and visualization functions
import numpy as numx
from pydeep.preprocessing import ICA, ZCA
import pydeep.misc.io as io
import pydeep.misc.visualization as vis

# Set the random seed
# (optional, if stochastic processes are involved we always get the same results)
numx.random.seed(42)

# Load data (download is not existing)
data = io.load_natural_image_patches('NaturalImage.mat')

# Specify image width and height for displaying
```

(continues on next page)


optimal grating
0
5
10
15
20
25
0
50
100
150
200
250


Tuning curves
2
4
6
8
10
12
14
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Filter Frequency histogram □□□ Filter Angle histogram
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
12
14
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

(continued from previous page)

```
width = height = 14

# Use ZCA to whiten the data and train it
# (you could also use PCA whitened=True + unproject for visualization)
zca = ZCA(input_dim=width * height)
zca.train(data=data)

# ZCA projects the whitened data back to the original space, thus does not
# perform a dimensionality reduction but a whitening in the original space
whitened_data = zca.project(data)

# Create a ZCA node and train it (you could also use PCA whitened=True)
ica = ICA(input_dim=width * height)
ica.train(data=whitened_data,
          iterations=100,
          convergence=1.0,
          status=True)

# Show whitened images
images = vis.tile_matrix_rows(matrix=data[0:100].T,
                              tile_width=width,
                              tile_height=height,
                              num_tiles_x=10,
                              num_tiles_y=10,
                              border_size=1,
                              normalized=True)
vis.imshow_matrix(matrix=images,
                  windowtitle='First 100 image patches')

# Show some whitened images
images = vis.tile_matrix_rows(matrix=whitened_data[0:100].T,
                              tile_width=width,
                              tile_height=height,
                              num_tiles_x=10,
                              num_tiles_y=10,
                              border_size=1,
                              normalized=True)
vis.imshow_matrix(matrix=images,
                  windowtitle='First 100 image patches whitened')

# Show the ICA filters/bases
ica_filters = vis.tile_matrix_rows(matrix=ica.projection_matrix,
                                   tile_width=width,
                                   tile_height=height,
                                   num_tiles_x=width,
                                   num_tiles_y=height,
                                   border_size=1,
                                   normalized=True)
vis.imshow_matrix(matrix=ica_filters,
                  windowtitle='Filters learned by ICA')

# Get the optimal gabor wavelet frequency and angle for the filters
opt_frq, opt_ang = vis.filter_frequency_and_angle(ica.projection_matrix,
                                                  num_of_angles=40)

# Show some tuning curves
num_filters = 20
```

(continues on next page)

(continued from previous page)

```
vis.imshow_filter_tuning_curve(ica.projection_matrix[:,0:num_filters],
                               num_of_ang=40)

# Show some optima grating
vis.imshow_filter_optimal_gratings(ica.projection_matrix[:,0:num_filters],
                                   opt_frq[0:num_filters],
                                   opt_ang[0:num_filters])

# Show histograms of frequencies and angles.
vis.imshow_filter_frequency_angle_histogram(opt_frq=opt_frq,
                                            opt_ang=opt_ang,
                                            max_wavelength=14)

print("log-likelihood on all data: "+str(numx.mean(
    ica.log_likelihood(data=whitened_data))))

# Show all windows.
vis.show()
```

3.1.5 Feed Forward Neural Network on MNIST

Example for training a Feed Forward Neural Network on the MNIST handwritten digit dataset.

3.1.5.1 Results

The *code* given below produces the following output that is quite similar to the results produced by an RBM.

```
1    0.1     0.0337166666667         0.0396
2    0.1     0.023                   0.0285
3    0.1     0.0198666666667         0.0276
4    0.1     0.0154                  0.0264
5    0.1     0.01385                 0.0239
6    0.1     0.01255                 0.0219
7    0.1     0.012                   0.0229
8    0.1     0.00926666666667        0.0207
9    0.1     0.0117                  0.0237
10   0.1     0.00881666666667        0.0214
11   0.1     0.007                   0.0191
12   0.1     0.00778333333333        0.0199
13   0.1     0.0067                  0.0183
14   0.1     0.00666666666667        0.0194
15   0.1     0.00665                 0.0197
16   0.1     0.00583333333333        0.0197
17   0.1     0.00563333333333        0.0193
18   0.1     0.005                   0.0181
19   0.1     0.00471666666667        0.0186
20   0.1     0.00431666666667        0.0191
```

Showing the Epoch / Learning Rate / Training Error / Test Error

See also RBM_MNIST_big.

3.1.5.2 Source code

```
''' Toy example using FNN on MNIST.

    :Version:
        3.0

    :Date
        25.05.2019

    :Author:
        Jan Melchior

    :Contact:
        pydeep@gmail.com

    :License:

        Copyright (C) 2019  Jan Melchior

        This program is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

'''

import numpy as numx

import pydeep.fnn.model as MODEL
import pydeep.fnn.layer as LAYER
import pydeep.fnn.trainer as TRAINER
import pydeep.base.activationfunction as ACT
import pydeep.base.costfunction as COST
import pydeep.base.corruptor as CORR
import pydeep.misc.io as IO
import pydeep.base.numpyextension as npExt


# Set random seed (optional)
numx.random.seed(42)


# Load data and whiten it
```

(continues on next page)

(continued from previous page)

```
train_data,train_label,valid_data, valid_label,test_data, test_label = IO.load_mnist(
↪"mnist.pkl.gz",False)
train_data = numx.vstack((train_data,valid_data))
train_label = numx.hstack((train_label,valid_label)).T
train_label = npExt.get_binary_label(train_label)
test_label = npExt.get_binary_label(test_label)

# Create model
l1 = LAYER.FullConnLayer(input_dim = train_data.shape[1],
                         output_dim = 1000,
                         activation_function=ACT.ExponentialLinear(),
                         initial_weights='AUTO',
                         initial_bias=0.0,
                         initial_offset=numx.mean(train_data,axis = 0).reshape(1,
↪train_data.shape[1]),
                         connections=None,
                         dtype=numx.float64)
l2 = LAYER.FullConnLayer(input_dim = 1000,
                         output_dim = train_label.shape[1],
                         activation_function=ACT.SoftMax(),
                         initial_weights='AUTO',
                         initial_bias=0.0,
                         initial_offset=0.0,
                         connections=None,
                         dtype=numx.float64)
model = MODEL.Model([l1,l2])

# Choose an Optimizer
trainer = TRAINER.ADAGDTrainer(model)
#trainer = TRAINER.GDTrainer(model)

# Train model
max_epochs =20
batch_size = 20
eps = 0.1
print 'Training'
for epoch in range(1, max_epochs + 1):
    train_data, train_label = npExt.shuffle_dataset(train_data, train_label)
    for b in range(0, train_data.shape[0], batch_size):
        trainer.train(data=train_data[b:b + batch_size, :],
                      labels=[None,train_label[b:b + batch_size, :]],
                      costs = [None,COST.CrossEntropyError()],
                      reg_costs = [0.0,1.0],
                      #momentum=[0.0]*model.num_layers,
                      epsilon = [eps]*model.num_layers,
                      update_offsets = [0.0]*model.num_layers,
                      corruptor = [CORR.Dropout(0.2),CORR.Dropout(0.5),None],
                      reg_L1Norm = [0.0]*model.num_layers,
                      reg_L2Norm = [0.0]*model.num_layers,
                      reg_sparseness  = [0.0]*model.num_layers,
                      desired_sparseness = [0.0]*model.num_layers,
                      costs_sparseness = [None]*model.num_layers,
                      restrict_gradient = [0.0]*model.num_layers,
                      restriction_norm = 'Mat')
    print epoch,'\t',eps,'\t',
    print numx.mean(npExt.compare_index_of_max(model.forward_propagate(train_data),
↪train_label)),'\t',
```

(continues on next page)

(continued from previous page)

```
    print numx.mean(npExt.compare_index_of_max(model.forward_propagate(test_data),
↪test_label))
```

3.1.6 Small binary RBM on MNIST

Example for training a centered and normal Binary Restricted Boltzmann machine on the MNIST handwritten digit dataset and its flipped version (1-MNIST). The model is small enough to calculate the exact log-Likelihood. For comparision annealed importance sampling and reverse annealed importance sampling are used for estimating the partition function.

It allows to reproduce the results from the publication How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016.

3.1.6.1 Theory

For an analysis of the advantage of centering in RBMs see How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016.

If you are new on RBMs, you can have a look into my master's theses

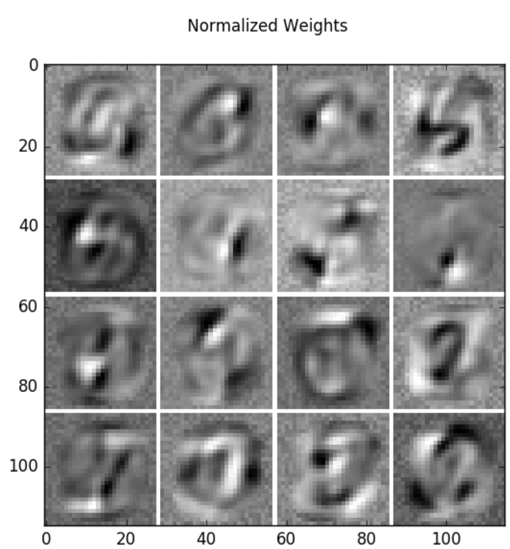
A good theoretical introduction is also given by the Course Material in combination with the following video lectures.

and

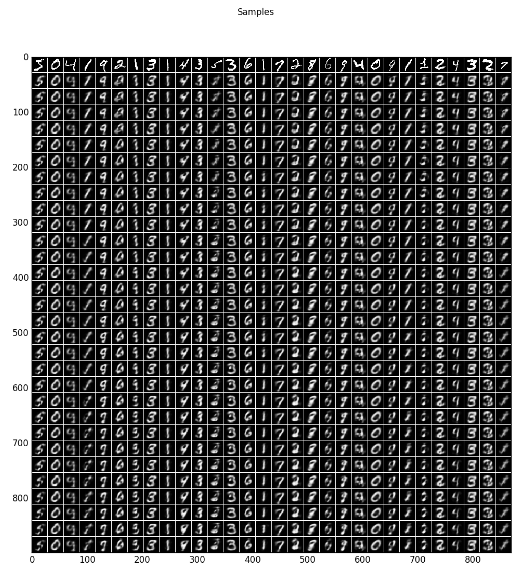
3.1.6.2 Results

The *code* given below produces the following output.

Learned filters of a centered binary RBM on the MNIST dataset. The filters have been normalized such that the structure is more prominent.



Sampling results for some examples. The first row shows the training data and the following rows are the results after one Gibbs-sampling step starting from the previous row.



The Log-Likelihood is calculated using the exact Partition function, an annealed importance sampling estimation (optimistic) and reverse annealed importance sampling estimation (pessimistic).

```
True Partition:         310.18444704  (LL train: -143.149739926, LL test: -142.
↪56382054)
AIS Partition:          309.693954732 (LL train: -142.659247618, LL test: -142.
↪073328232)
reverse AIS Partition:  316.30736142  (LL train: -149.272654305, LL test: -148.
↪686734919)
```

The code can also be executed without centering by setting

```
update_offsets = 0.0
```

Resulting in the following weights and sampling steps.

The Log-Likelihood for this model is worse (6.5 nats lower).

```
True Partition:         190.951945786 (LL train: -149.605105935, LL test: -149.
↪053303204)
AIS Partition:          191.095934868 (LL train: -149.749095017, LL test: -149.
↪197292286)
reverse AIS Partition:  191.192036843 (LL train: -149.845196992, LL test: -149.
↪293394261)
```

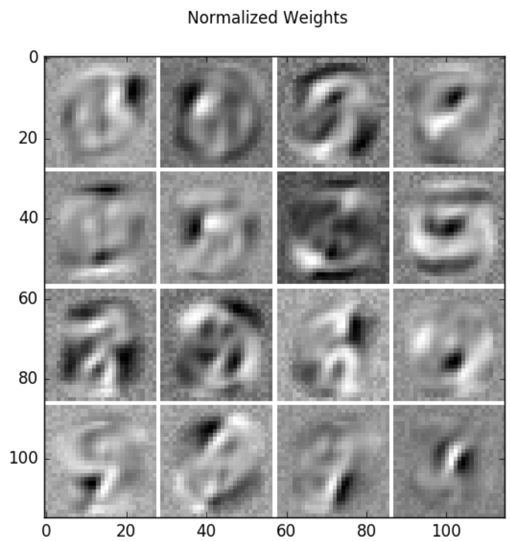
Further, the models can be trained on the flipped version of MNIST (1-MNIST).

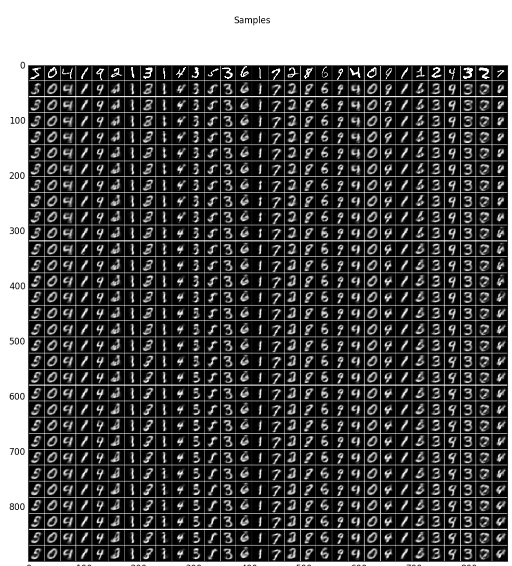
```
flipped = True
```

While the centered model has a similar performance on the flipped version,

```
True Partition:         310.245654321 (LL train: -142.812529437, LL test: -142.
↪08692014)
AIS Partition:          311.177617039 (LL train: -143.744492155, LL test: -143.
↪018882858)
```

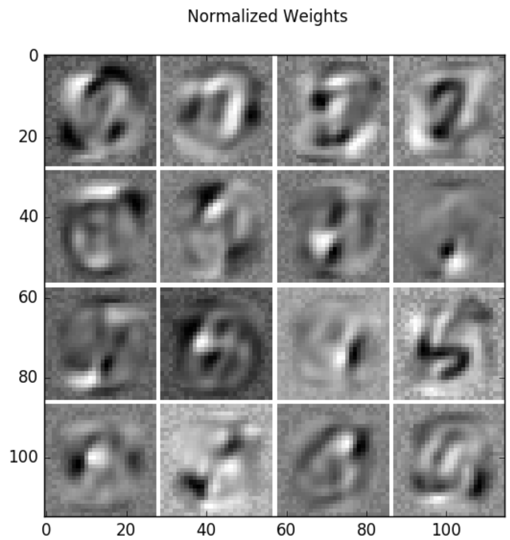
(continues on next page)

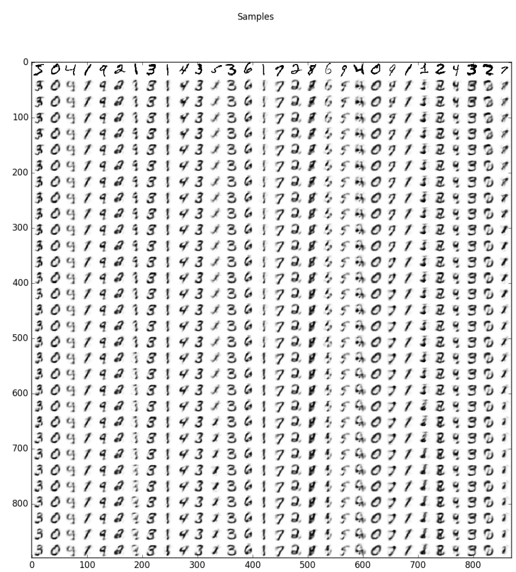

Normalized Weights
0
20
40
60
80
100
0
20
40
60
80
100


Samples
0
100
200
300
400
500
600
700
800
0
100
200
300
400
500
600
700
800

(continued from previous page)

```
reverse AIS Partition:  309.188366165 (LL train: -141.755241282, LL test: -141.
↪029631984)
```

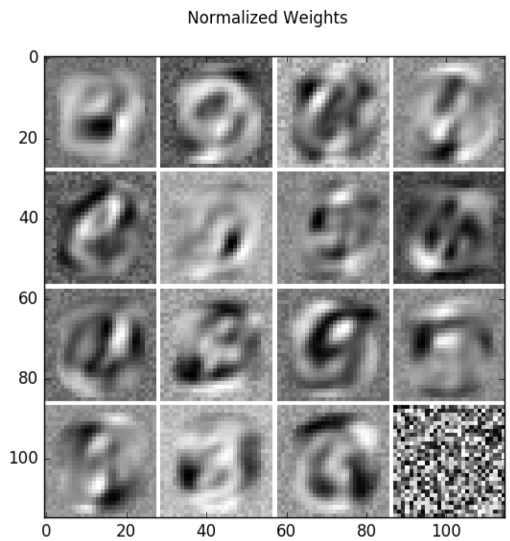


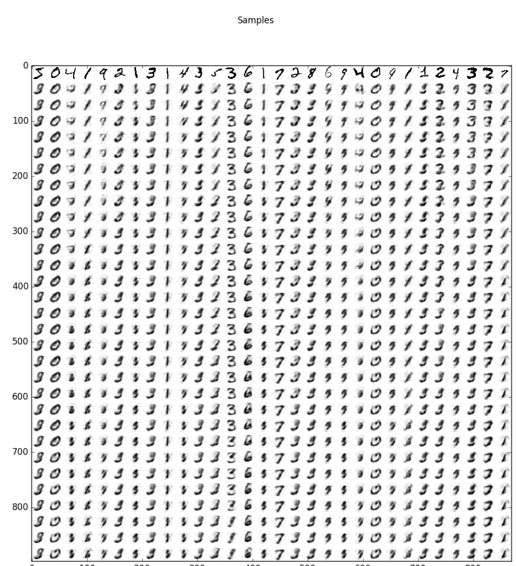


The normal RBM has not.

```
True Partition:         3495.27200694 (LL train: -183.259299994, LL test: -183.
↪359988079)
AIS Partition:          3495.25941111 (LL train: -183.246704163, LL test: -183.
↪347392249)
reverse AIS Partition:  3495.20117625 (LL train: -183.188469308, LL test: -183.
↪289157393)
```

For a large number of hidden units see RBM_MNIST_big.


Normalized Weights
0
20
40
60
80
100
0
20
40
60
80
100


Samples
0
100
200
300
400
500
600
700
800
0
100
200
300
400
500
600
700
800

3.1.6.3 Source code

```
""" Example using a small BB-RBMs on the MNIST handwritten digit database.

    :Version:
        1.1.0

    :Date:
        20.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# model, trainer, and estimator
import pydeep.rbm.model as model
import pydeep.rbm.trainer as trainer
import pydeep.rbm.estimator as estimator

# Import numpy, input output functions, visualization, and measurement
import numpy as numx
import pydeep.misc.io as io
import pydeep.misc.visualization as vis
import pydeep.misc.measuring as mea

# Choose normal/centered RBM and normal/flipped MNIST

# normal/centered RBM --> 0.0/0.01
update_offsets = 0.01
```

(continues on next page)

(continued from previous page)

```
# Flipped/Normal MNIST --> True/False
flipped = False

# Set random seed (optional)
numx.random.seed(42)

# Input and hidden dimensionality
v1 = v2 = 28
h1 = h2 = 4

# Load data (download is not existing)
train_data, _, valid_data, _, test_data, _ = io.load_mnist("mnist.pkl.gz", True)
train_data = numx.vstack((train_data, valid_data))

# Flip the dataset if chosen
if flipped:
    train_data = 1 - train_data
    test_data = 1 - test_data
    print("Flipped MNIST")
else:
    print("Normal MNIST")

# Training parameters
batch_size = 100
epochs = 50

# Create centered or normal model
if update_offsets <= 0.0:
    rbm = model.BinaryBinaryRBM(number_visibles=v1 * v2,
                                number_hiddens=h1 * h2,
                                data=train_data,
                                initial_visible_offsets=0.0,
                                initial_hidden_offsets=0.0)
    print("Normal RBM")
else:
    rbm = model.BinaryBinaryRBM(number_visibles=v1 * v2,
                                number_hiddens=h1 * h2,
                                data=train_data,
                                initial_visible_offsets='AUTO',
                                initial_hidden_offsets='AUTO')
    print("Centered RBM")

# Create trainer
trainer_pcd = trainer.PCD(rbm, num_chains=batch_size)

# Measuring time
measurer = mea.Stopwatch()

# Train model
print('Training')
print('Epoch\tRecon. Error\tLog likelihood train\tLog likelihood test\tExpected End-
↪Time')
for epoch in range(epochs):

    # Loop over all batches
    for b in range(0, train_data.shape[0], batch_size):
```

(continues on next page)

(continued from previous page)

```
        batch = train_data[b:b + batch_size, :]
        trainer_pcd.train(data=batch,
                          epsilon=0.01,
                          update_visible_offsets=update_offsets,
                          update_hidden_offsets=update_offsets)

    # Calculate Log-Likelihood, reconstruction error and expected end time every 5th␣
↪epoch
    if (epoch==0 or (epoch+1) % 5 == 0):
        logZ = estimator.partition_function_factorize_h(rbm)
        ll_train = numx.mean(estimator.log_likelihood_v(rbm, logZ, train_data))
        ll_test = numx.mean(estimator.log_likelihood_v(rbm, logZ, test_data))
        re = numx.mean(estimator.reconstruction_error(rbm, train_data))
        print('{}\t\t{:.4f}\t\t\t{:.4f}\t\t\t\t{:.4f}\t\t\t{}'.format(
        epoch+1, re, ll_train, ll_test, measurer.get_expected_end_time(epoch+1,␣
↪epochs)))
    else:
        print(epoch+1)

measurer.end()

# Print end/training time
print("End-time: \t{}".format(measurer.get_end_time()))
print("Training time:\t{}".format(measurer.get_interval()))

# Calculate true partition function
logZ = estimator.partition_function_factorize_h(rbm, batchsize_exponent=h1,␣
↪status=False)
print("True Partition: {} (LL train: {}, LL test: {})".format(logZ,
    numx.mean(estimator.log_likelihood_v(rbm, logZ, train_data)),
    numx.mean(estimator.log_likelihood_v(rbm, logZ, test_data))))

# Approximate partition function by AIS (tends to overestimate)
logZ_approx_AIS = estimator.annealed_importance_sampling(rbm)[0]
print("AIS Partition: {} (LL train: {}, LL test: {})".format(logZ_approx_AIS,
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_AIS, train_data)),
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_AIS, test_data))))

# Approximate partition function by reverse AIS (tends to underestimate)
logZ_approx_rAIS = estimator.reverse_annealed_importance_sampling(rbm)[0]
print("reverse AIS Partition: {} (LL train: {}, LL test: {})".format(
    logZ_approx_rAIS,
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_rAIS, train_data)),
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_rAIS, test_data))))

# Reorder RBM features by average activity decreasingly
reordered_rbm = vis.reorder_filter_by_hidden_activation(rbm, train_data)

# Display RBM parameters
vis.imshow_standard_rbm_parameters(reordered_rbm, v1, v2, h1, h2)

# Sample some steps and show results
samples = vis.generate_samples(rbm, train_data[0:30], 30, 1, v1, v2, False, None)
vis.imshow_matrix(samples, 'Samples')

# Display results
vis.show()
```

3.1.7 Big binary RBM on MNIST

Example for training a centered and normal binary restricted Boltzmann machine on the MNIST handwritten digit dataset. The model has 500 hidden units, is trained for 200 epochs (That takes a while, reduce it if you like), and the log-likelihood is evaluated using annealed importance sampling.

It allows to reproduce the results from the publication How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016.. Running the code as it is for example reproduces a single trial of the plot in Figure 9. (PCD-1) for dd^b_s.

3.1.7.1 Theory

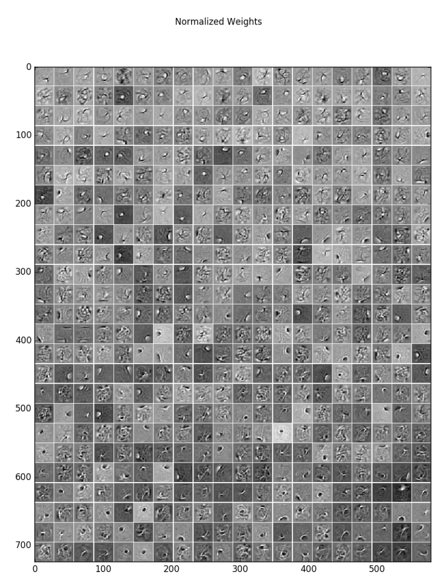
If you are new on RBMs, first see RBM_MNIST_small.

For an analysis of the advantage of centering in RBMs see How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016.

3.1.7.2 Results

The *code* given below produces the following output.

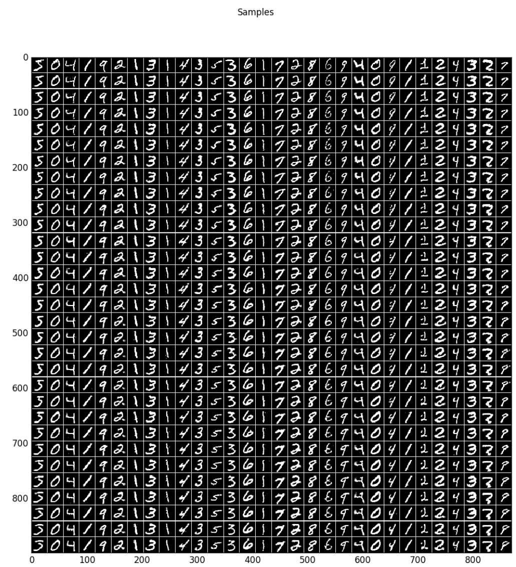
Learned filters of a centered binary RBM with 500 hidden units on the MNIST dataset. The filters have been normalized such that the structure is more prominent.



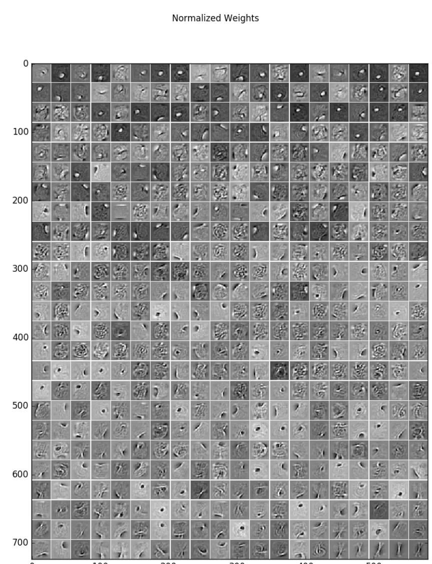
Sampling results for some examples. The first row shows some training data and the following rows are the results after one Gibbs-sampling step starting from the previous row.

The log-Likelihood is estimated using annealed importance sampling (optimistic) and reverse annealed importance sampling (pessimistic).

```
Training time:          1:18:12.536887
AIS Partition:          968.971299741 (LL train: -82.5839850187, LL test: -84.
↪8560508601)
reverse AIS Partition: 980.722421486 (LL train: -94.3351067638, LL test: -96.
↪6071726052)
```



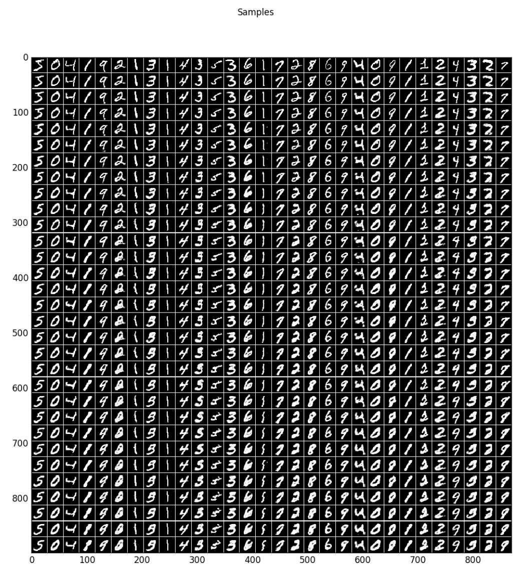
Now we have a look at the filters learned for a normal binary RBM with 500 hidden units on the MNIST dataset. The filters have also been normalized such that the structure is more prominent.



Sampling results for some examples. The first row shows the training data and the following rows are the results after one Gibbs-sampling step starting from the previous row.

```
Training time:          1:16:37.808645
AIS Partition:          959.098055647 (LL train: -128.009777345, LL test: -130.
↪808849443)
reverse AIS Partition: 958.714291654 (LL train: -127.626013352, LL test: -130.
↪42508545)
```

The structure of the filters and the samples are quite similar. But the samples for the centered RBM look a bit sharper



and the log-likelihood is significantly higher. Note that you can reach better values with normal RBMs but this highly depends on the training setup, whereas centering is rather robust to that.

For real valued input see also GRBM_natural_images.

3.1.7.3 Source code

```
""" Example using a big BB-RBMs on the MNIST handwritten digit database.

    :Version:
        1.1.0

    :Date:
        24.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
```

(continues on next page)

(continued from previous page)

```
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

import numpy as numx
import pydeep.rbm.model as model
import pydeep.rbm.trainer as trainer
import pydeep.rbm.estimator as estimator

import pydeep.misc.io as io
import pydeep.misc.visualization as vis
import pydeep.misc.measuring as mea

# normal/centered RBM --> 0.0/0.01
update_offsets = 0.0

# Set random seed (optional)
numx.random.seed(42)

# Input and hidden dimensionality
v1 = v2 = 28
h1 = 25
h2 = 20

# Load data (download is not existing)
train_data, _, valid_data, _, test_data, _ = io.load_mnist("mnist.pkl.gz", True)
train_data = numx.vstack((train_data, valid_data))

# Training paramters
batch_size = 100
epochs = 200

# Create centered or normal model
if update_offsets <= 0.0:
    rbm = model.BinaryBinaryRBM(number_visibles=v1 * v2,
                                number_hiddens=h1 * h2,
                                data=None,
                                initial_weights=0.01,
                                initial_visible_bias=0.0,
                                initial_hidden_bias=0.0,
                                initial_visible_offsets=0.0,
                                initial_hidden_offsets=0.0)
else:
    rbm = model.BinaryBinaryRBM(number_visibles=v1 * v2,
                                number_hiddens=h1 * h2,
                                data=train_data,
                                initial_weights=0.01,
                                initial_visible_bias='AUTO',
```

(continues on next page)

(continued from previous page)

```
                                    initial_hidden_bias='AUTO',
                                    initial_visible_offsets='AUTO',
                                    initial_hidden_offsets='AUTO')

trainer_pcd = trainer.PCD(rbm, num_chains=batch_size)

# Measuring time
measurer = mea.Stopwatch()

# Train model
print('Training')
print('Epoch\t\tRecon. Error\tLog likelihood \tExpected End-Time')
for epoch in range(1, epochs + 1):

    # Loop over all batches
    for b in range(0, train_data.shape[0], batch_size):
        batch = train_data[b:b + batch_size, :]
        trainer_pcd.train(data=batch,
                          epsilon=0.01,
                          update_visible_offsets=update_offsets,
                          update_hidden_offsets=update_offsets)

    # Calculate reconstruction error and expected end time every 10th epoch
    if epoch % 10 == 0:
        RE = numx.mean(estimator.reconstruction_error(rbm, train_data))
        print('{}\t\t{:.4f}\t\t\t{}'.format(
            epoch, RE, measurer.get_expected_end_time(epoch, epochs)))
    else:
        print(epoch)

# Stop time measurement
measurer.end()

# Print end time
print("End-time: \t{}".format(measurer.get_end_time()))
print("Training time:\t{}".format(measurer.get_interval()))

# Approximate partition function by AIS (tends to overestimate)
logZ_approx_AIS = estimator.annealed_importance_sampling(rbm)[0]
print("AIS Partition: {} (LL train: {}, LL test: {})".format(logZ_approx_AIS,
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_AIS, train_data)),
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_AIS, test_data))))

# Approximate partition function by reverse AIS (tends to underestimate)
logZ_approx_rAIS = estimator.reverse_annealed_importance_sampling(rbm)[0]
print("reverse AIS Partition: {} (LL train: {}, LL test: {})".format(
    logZ_approx_rAIS,
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_rAIS, train_data)),
    numx.mean(estimator.log_likelihood_v(rbm, logZ_approx_rAIS, test_data))))

# Reorder RBM features by average activity decreasingly
reordered_rbm = vis.reorder_filter_by_hidden_activation(rbm, train_data)

# Display RBM parameters
vis.imshow_standard_rbm_parameters(reordered_rbm, v1, v2, h1, h2)

# Sample some steps and show results
```

(continues on next page)

(continued from previous page)

```
samples = vis.generate_samples(rbm, train_data[0:30], 30, 1, v1, v2, False, None)
vis.imshow_matrix(samples, 'Samples')

# Display results
vis.show()
```

3.1.8 Deep Boltzmann machines on MNIST

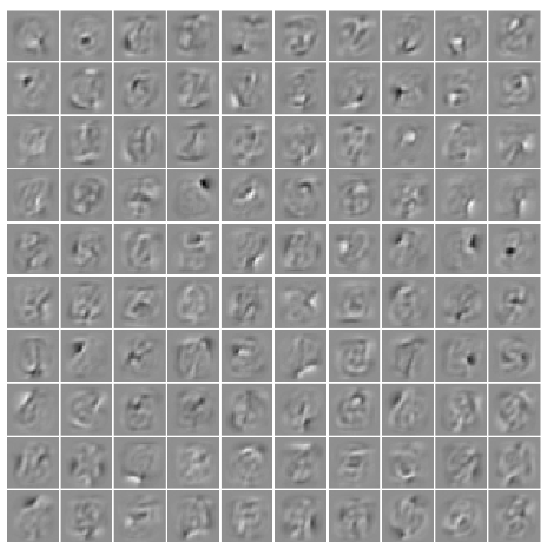
Example for training a centered Deep Boltzmann machine on the MNIST handwritten digit dataset.

It allows to reproduce the results from the publication How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016..

3.1.8.1 Results

The *code* given below produces the following output that is quite similar to the results produced by an RBM.

The learned filters of the first layer

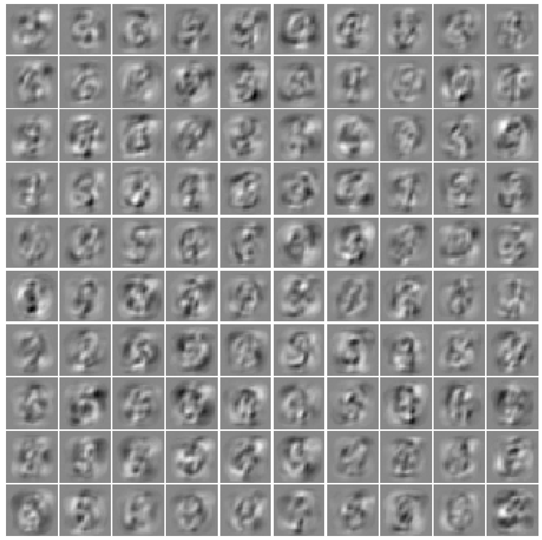


The learned filters of the second layer, linearly back projected

Some generated samples

See also RBM_MNIST_big.

3.1.8.2 Source code



```python
import pydeep.misc.visualization as VIS
import pydeep.misc.io as IO
import pydeep.base.numpyextension as numxExt
from pydeep.dbm.unit_layer import *
from pydeep.dbm.weight_layer import *
from pydeep.dbm.model import *

# Set the same seed value for all algorithms
numx.random.seed(42)

# Load Data
train_data = IO.load_mnist("mnist.pkl.gz", True)[0]

# Set dimensions Layer 1-3
v11 = v12 = 28
v21 = v22 = 10
v31 = v32 = 10
N = v11 * v12
M = v21 * v22
O = v31 * v32

# Create weight layers, which connect the unit layers
wl1 = Weight_layer(input_dim=N,
                   output_dim=M,
                   initial_weights=0.01,
                   dtype=numx.float64)
wl2 = Weight_layer(input_dim=M,
                   output_dim=O,
                   initial_weights=0.01,
                   dtype=numx.float64)

# Create three unit layers
l1 = Binary_layer(None,
                  wl1,
                  data=train_data,
                  initial_bias='AUTO',
                  initial_offsets='AUTO',
                  dtype=numx.float64)

l2 = Binary_layer(wl1,
                  wl2,
                  data=None,
                  initial_bias='AUTO',
                  initial_offsets='AUTO',
                  dtype=numx.float64)

l3 = Binary_layer(wl2,
                  None,
                  data=None,
                  initial_bias='AUTO',
                  initial_offsets='AUTO',
                  dtype=numx.float64)

# Initialize parameters
max_epochs = 10
batch_size = 20
```

(continues on next page)

(continued from previous page)

```
# Sampling Setps positive and negative phase
k_d = 3
k_m = 1

# Set individual learning rates
lr_W1 = 0.01
lr_W2 = 0.01
lr_b1 = 0.01
lr_b2 = 0.01
lr_b3 = 0.01
lr_o1 = 0.01
lr_o2 = 0.01
lr_o3 = 0.01

# Initialize negative Markov chain
x_m = numx.zeros((batch_size, v11 * v12)) + l1.offset
y_m = numx.zeros((batch_size, v21 * v22)) + l2.offset
z_m = numx.zeros((batch_size, v31 * v32)) + l3.offset
chain_m = [x_m, y_m, z_m]

# Reparameterize RBM such that the inital setting is the same for centereing and␣
↪centered training
l1.bias += numx.dot(0.0 - l2.offset, wl1.weights.T)
l2.bias += numx.dot(0.0 - l1.offset, wl1.weights) + numx.dot(0.0 - l3.offset, wl2.
↪weights.T)
l3.bias += numx.dot(0.0 - l2.offset, wl2.weights)

# Finally create model
model = DBM_model([l1, l2, l3])

# Loop over data and batches to traing th emodel
for epoch in range(0, max_epochs):
    rec_sum = 0
    for b in range(0, train_data.shape[0], batch_size):
        # Positive Phase

        # Initialize Markov chains with data or offsets
        x_d = train_data[b:b + batch_size, :]
        y_d = numx.zeros((batch_size, M)) + l2.offset
        z_d = numx.zeros((batch_size, O)) + l3.offset
        chain_d = [x_d, y_d, z_d]

        # Sample for k_d steps mean field estimation inplace, but clamp the data units
        model.meanfield(chain_d, k_d, [True, False, False], True)
        # or sample instead
        #model.sample(chain_d, k_d, [True, False, False], True)

        # Negative Phase

        # PCD, sample k_m steps without clamping
        model.sample(chain_m, k_m, [False, False, False], True)

        # Update the model using the sampled states and learning rates
        model.update(chain_d, chain_m, lr_W1, lr_b1, lr_o1)

    # Print Norms of the Parameters
    print(numx.mean(numxExt.get_norms(wl1.weights)), '\t', numx.mean(numxExt.get_
↪norms(wl2.weights)), '\t')
```

(continues on next page)

(continued from previous page)

```
    print(numx.mean(numxExt.get_norms(l1.bias)), '\t', numx.mean(numxExt.get_norms(l2.
↪bias)), '\t')
    print(numx.mean(numxExt.get_norms(l3.bias)), '\t', numx.mean(l1.offset), '\t',␣
↪numx.mean(l2.offset), '\t', numx.mean(l3.offset))

# Show weights
VIS.imshow_matrix(VIS.tile_matrix_rows(wl1.weights, v11, v12, v21, v22, border_size=1,
↪ normalized=False), 'Weights 1')
VIS.imshow_matrix(
    VIS.tile_matrix_rows(numx.dot(wl1.weights, wl2.weights), v11, v12, v31, v32,␣
↪border_size=1, normalized=False),
    'Weights 2')

# # Samplesome steps
chain_m = [numx.float64(numx.random.rand(10 * batch_size, v11 * v12) < 0.5),
           numx.float64(numx.random.rand(10 * batch_size, v21 * v22) < 0.5),
           numx.float64(numx.random.rand(10 * batch_size, v31 * v32) < 0.5)]
model.sample(chain_m, 100, [False, False, False], True)
# GEt probabilities
samples = l1.activation(None, chain_m[1])[0]
VIS.imshow_matrix(VIS.tile_matrix_columns(samples, v11, v12, 10, batch_size, 1,␣
↪False), 'Samples')

VIS.show()
```

3.1.9 Gaussian-binary restricted Boltzmann machine on a 2D linear mixture.

Example for Gaussian-binary restricted Boltzmann machine used for blind source separation on a linear 2D mixture.

3.1.9.1 Theory

The results are part of the publication Gaussian-binary restricted Boltzmann machines for modeling natural image statistics. Melchior, J., Wang, N., & Wiskott, L.. (2017). PLOS ONE, 12(2), 1–24. .

If you are new on GRBMs, you can have a look into my master's theses

See also ICA_2D_example

3.1.9.2 Results

The *code* given below produces the following output.

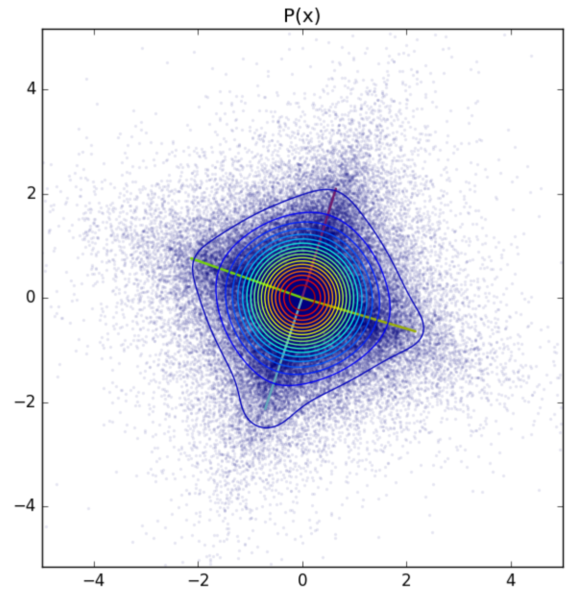
Visualization of the weight vectors learned by the GRBM with 4 hidden units together with the contour plot of the learned probability density function (PDF).

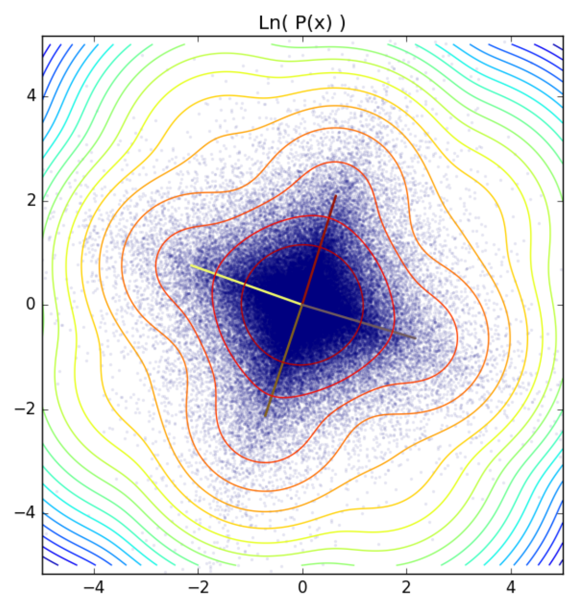
For a better visualization also the log-PDF.

The parameters values and the component scaling values P(h_i) are as follows:

```
Weigths:
[[-2.13559806 -0.71220501  0.64841691  2.17880554]
 [ 0.75840129 -2.13979672  2.09910978 -0.64721076]]
Visible bias:
[[ 0.  0.]]
```

(continues on next page)


P(x)
4
2
0
−2
−4
−4
−2
0
2
4


Ln(P(x))
4
2
0
−2
−4
−4
−2
0
2
4

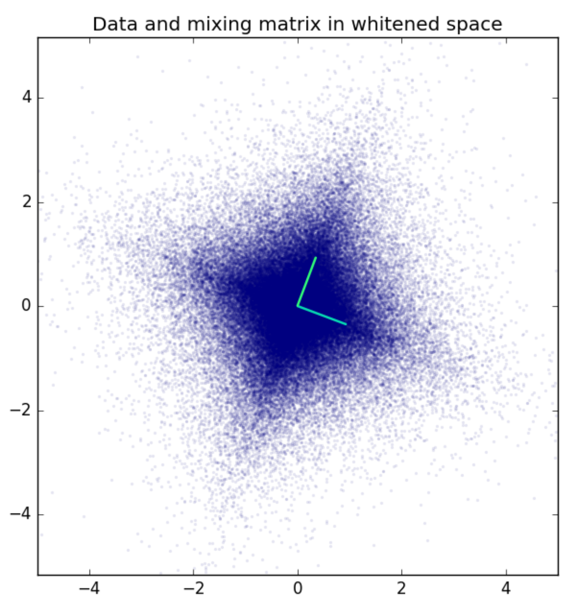
(continued from previous page)

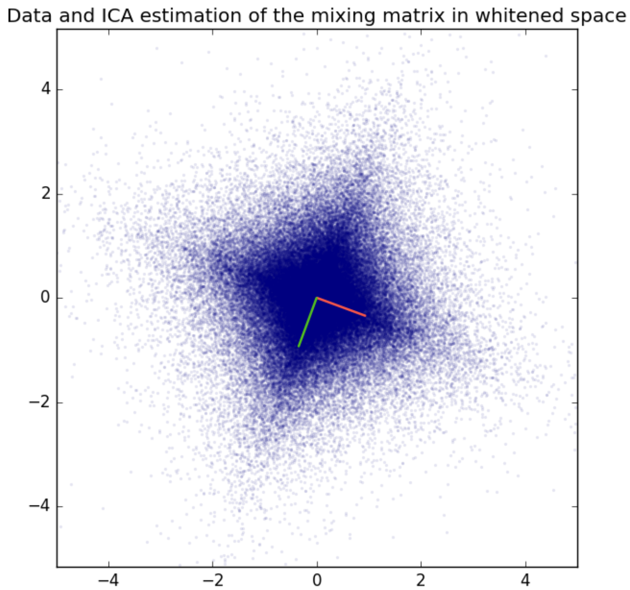
```
Hidden bias:
[[-7.87792514 -7.60603139 -7.73935758 -7.722771  ]]
Sigmas:
[[ 0.74241256  0.73101419]]

Scaling factors:
P(h_0) [[ 0.83734074]]
P(h 1 ) [[ 0.03404849]]
P(h 2 ) [[ 0.04786942]]
P(h 3 ) [[ 0.0329518]]
P(h 4 ) [[ 0.04068302]]
```

The exact log-likelihood, annealed importance sampling estimation, and reverse annealed importance sampling estimation for training and test data are:

```
True log partition:   1.40422867085  ( LL_train:  -2.74117592643 , LL_test:  -2.
↪73620936613  )
AIS  log partition:   1.40390312781  ( LL_train:  -2.74085038339 , LL_test:  -2.
↪73588382309  )
rAIS  log partition:  1.40644042744  ( LL_train:  -2.74338768302 , LL_test:  -2.
↪73842112273  )
```

For comparison here is the original mixing matrix an the corresponding ICA estimation.





The exact log-likelihood for ICA is almost the same as that for the GRBM with 4 hidden units.

```
ICA log-likelihood on train data: -2.74149951412
ICA log-likelihood on test data:  -2.73579105422
```

We can also calculate the Amari distance between true mixing , the ICA estimation, and the GRBM estimation. Since the GRBM has learned 4 weight vectors we calculate teh Amari distance between the true mixing matrix and all sets of 2 weight-vectors of the GRBM.

```
Amari distance between true mixing matrix and ICA estimation:             0.
↪00621143307663
Amari distance between true mixing matrix and GRBM weight vector 1 and 2: 0.
↪0292827450487
Amari distance between true mixing matrix and GRBM weight vector 1 and 3: 0.
↪0397992351592
Amari distance between true mixing matrix and GRBM weight vector 1 and 4: 0.
↪336416964036
Amari distance between true mixing matrix and GRBM weight vector 2 and 3: 0.
↪435997388341
Amari distance between true mixing matrix and GRBM weight vector 2 and 4: 0.
↪0557649366433
Amari distance between true mixing matrix and GRBM weight vector 3 and 4: 0.
↪0666442992135
```

Weight-vectors 1 and 4 as well as 2 and 3 are almost 180 degrees rotated version of each other, which can also be seen from the weight matrix values given above and thus the Amari distance to the mixing matrix is high.

For a real-world application see the GRBM_natural_images example.

3.1.9.3 Source code

```
""" Toy example using GB-RBMs on a blind source seperation toy problem.
```

(continues on next page)

(continued from previous page)

```
    :Version:
        1.1.0

    :Date:
        28.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# Import numpy, numpy extensions
import numpy as numx
import pydeep.base.numpyextension as numxext

# Import models, trainers and estimators
import pydeep.rbm.model as model
import pydeep.rbm.trainer as trainer
import pydeep.rbm.estimator as estimator

# Import linear mixture, preprocessing, and visualization
from pydeep.misc.toyproblems import generate_2d_mixtures
import pydeep.preprocessing as pre
import pydeep.misc.visualization as vis

numx.random.seed(42)

# Create a 2D mxiture
data, mixing_matrix = generate_2d_mixtures(100000, 1, 1.0)
```

(continues on next page)

(continued from previous page)

```
# Whiten data
zca = pre.ZCA(data.shape[1])
zca.train(data)
whitened_data = zca.project(data)

# split training test data
train_data = whitened_data[0:numx.int32(whitened_data.shape[0] / 2.0), :]
test_data = whitened_data[numx.int32(whitened_data.shape[0] / 2.0
                                     ):whitened_data.shape[0], :]

# Input output dims
h1 = 2
h2 = 2
v1 = whitened_data.shape[1]
v2 = 1

# Create model
rbm = model.GaussianBinaryVarianceRBM(number_visibles=v1 * v2,
                                      number_hiddens=h1 * h2,
                                      data=train_data,
                                      initial_weights='AUTO',
                                      initial_visible_bias=0,
                                      initial_hidden_bias=0,
                                      initial_sigma=1.0,
                                      initial_visible_offsets=0.0,
                                      initial_hidden_offsets=0.0,
                                      dtype=numx.float64)

# Set the hidden bias such that the scaling factor is 0.1
rbm.bh = -(numxext.get_norms(rbm.w + rbm.bv.T, axis=0) - numxext.get_norms(
    rbm.bv, axis=None)) / 2.0 + numx.log(0.1)
rbm.bh = rbm.bh.reshape(1, h1 * h2)

# Create trainer
trainer_cd = trainer.CD(rbm)

# Hyperparameters
batch_size = 1000
max_epochs = 50
k = 1
epsilon = [1,0,1,0.1]

# Train model
print 'Training'
print 'Epoch\tRE train\tRE test \tLL train\tLL test '
for epoch in range(1, max_epochs + 1):

    # Shuffle data points
    train_data = numx.random.permutation(train_data)

    # loop over batches
    for b in range(0, train_data.shape[0] / batch_size):
        trainer_cd.train(data=train_data[b:(b + batch_size), :],
                         num_epochs=1,
                         epsilon=epsilon,
                         k=k,
```

(continues on next page)

(continued from previous page)

```
                         momentum=0.0,
                         reg_l1norm=0.0,
                         reg_l2norm=0.0,
                         reg_sparseness=0.0,
                         desired_sparseness=0.0,
                         update_visible_offsets=0.0,
                         update_hidden_offsets=0.0,
                         restrict_gradient=False,
                         restriction_norm='Cols',
                         use_hidden_states=False,
                         use_centered_gradient=False)

    # Calculate Log likelihood and reconstruction error
    RE_train = numx.mean(estimator.reconstruction_error(rbm, train_data))
    RE_test = numx.mean(estimator.reconstruction_error(rbm, test_data))
    logZ = estimator.partition_function_factorize_h(rbm, batchsize_exponent=h1)
    LL_train = numx.mean(estimator.log_likelihood_v(rbm, logZ, train_data))
    LL_test = numx.mean(estimator.log_likelihood_v(rbm, logZ, test_data))
    print '%5d \t%0.5f \t%0.5f \t%0.5f \t%0.5f' % (epoch,
                                                    RE_train,
                                                    RE_test,
                                                    LL_train,
                                                    LL_test)

# Calculate partition function and its AIS approximation
logZ = estimator.partition_function_factorize_h(rbm, batchsize_exponent=h1)
logZ_AIS = estimator.annealed_importance_sampling(rbm,
                                                  num_chains=100,
                                                  k=1,
                                                  betas=1000,
                                                  status=False)[0]
logZ_rAIS = estimator.reverse_annealed_importance_sampling(rbm,
                                                  num_chains=100,
                                                  k=1,
                                                  betas=1000,
                                                  status=False)[0]

# Calculate and print LL
print("")
print("\nTrue log partition: ", logZ, " ( LL_train: ", numx.mean(
    estimator.log_likelihood_v(
        rbm, logZ, train_data)), ",", "LL_test: ", numx.mean(
    estimator.log_likelihood_v(rbm, logZ, test_data)), " )")
print("\nAIS  log partition: ", logZ_AIS, " ( LL_train: ", numx.mean(
    estimator.log_likelihood_v(
        rbm, logZ_AIS, train_data)), ",", "LL_test: ", numx.mean(
    estimator.log_likelihood_v(rbm, logZ_AIS, test_data)), " )")
print("\nrAIS  log partition: ", logZ_rAIS, " ( LL_train: ", numx.mean(
    estimator.log_likelihood_v(
        rbm, logZ_rAIS, train_data)), ",", "LL_test: ", numx.mean(
    estimator.log_likelihood_v(rbm, logZ_rAIS, test_data)), " )")
print("")
# Print parameter
print '\nWeigths:\n', rbm.w
print 'Visible bias:\n', rbm.bv
print 'Hidden bias:\n', rbm.bh
print 'Sigmas:\n', rbm.sigma
```

(continues on next page)

(continued from previous page)

```
print

# Calculate P(h) wich are the scaling factors of the Gaussian components
h_i = numx.zeros((1, h1 * h2))
print("Scaling factors:")
print 'P(h_0)', numx.exp(rbm.log_probability_h(logZ, h_i))
for i in range(h1 * h2):
    h_i = numx.zeros((1, h1 * h2))
    h_i[0, i] = 1
    print 'P(h', (i + 1), ')', numx.exp(rbm.log_probability_h(logZ, h_i))
print


# Independent Component Analysis (ICA)
ica = pre.ICA(train_data.shape[1])
ica.train(train_data, iterations=100,status=False)
data_ica = ica.project(train_data)

# Print ICA log-likelihood
print("ICA log-likelihood on train data: " + str(numx.mean(
    ica.log_likelihood(data=train_data))))
print("ICA log-likelihood on test data: " + str(numx.mean(
    ica.log_likelihood(data=test_data))))
print("")

# Print Amari distances
print("Amari distanca between true mixing matrix and ICA estimation: "+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T), ica.projection_matrix.
↪T)))

print("Amari distanca between true mixing matrix and GRBM weight vector 1 and 2:
↪"+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[0:1],rbm.w.T[1:2])))))

print("Amari distanca between true mixing matrix and GRBM weight vector 1 and 3:
↪"+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[0:1],rbm.w.T[2:3])))))

print("Amari distanca between true mixing matrix and GRBM weight vector 1 and 4:
↪"+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[0:1],rbm.w.T[3:4])))))

print("Amari distanca between true mixing matrix and GRBM weight vector 2 and 3:
↪"+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[1:2],rbm.w.T[2:3])))))

print("Amari distanca between true mixing matrix and GRBM weight vector 2 and 4:
↪"+str(
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[1:2],rbm.w.T[3:4])))))

print("Amari distanca between true mixing matrix and GRBM weight vector 3 and 4:
↪"+str(
```

(continues on next page)

(continued from previous page)

```
    vis.calculate_amari_distance(zca.project(mixing_matrix.T),
                                 numx.vstack((rbm.w.T[2:3],rbm.w.T[3:4])))))

# Display results
# create a new figure of size 5x5
vis.figure(0, figsize=[7, 7])
vis.title("P(x)")
# plot the data
vis.plot_2d_data(whitened_data)
# plot weights
vis.plot_2d_weights(rbm.w, rbm.bv)
# pass our P(x) as function to plotting function
vis.plot_2d_contour(lambda v: numx.exp(rbm.log_probability_v(logZ, v)))
# No inconsistent scaling
vis.axis('equal')
# Set size of the plot
vis.axis([-5, 5, -5, 5])

# Do the sam efor the LOG-Plot
# create a new figure of size 5x5
vis.figure(1, figsize=[7, 7])
vis.title("Ln( P(x) )")
# plot the data
vis.plot_2d_data(whitened_data)
# plot weights
vis.plot_2d_weights(rbm.w, rbm.bv)
# pass our P(x) as function to plotting function
vis.plot_2d_contour(lambda v: rbm.log_probability_v(logZ, v))
# No inconsistent scaling
vis.axis('equal')
# Set size of the plot
vis.axis([-5, 5, -5, 5])

# Figure 2 - Data and mixing matrix in whitened space
vis.figure(3, figsize=[7, 7])
vis.title("Data and mixing matrix in whitened space")
vis.plot_2d_data(whitened_data)
vis.plot_2d_weights(numxext.resize_norms(zca.project(mixing_matrix.T).T,
                                         norm=1,
                                         axis=0))
vis.axis('equal')
vis.axis([-5, 5, -5, 5])

# Figure 3 - Data and ica estimation of the mixing matrix in whitened space
vis.figure(4, figsize=[7, 7])
vis.title("Data and ICA estimation of the mixing matrix in whitened space")
vis.plot_2d_data(whitened_data)
vis.plot_2d_weights(numxext.resize_norms(ica.projection_matrix,
                                         norm=1,
                                         axis=0))
vis.axis('equal')
vis.axis([-5, 5, -5, 5])

vis.show()
```

3.1.10 Gaussian-binary restricted Boltzmann machine on natural image patches

Example for a Gaussian-binary restricted Boltzmann machine (GRBM) on a natural image patches. The learned filters are similar to those of ICA, see also ICA_natural_images.

3.1.10.1 Theory

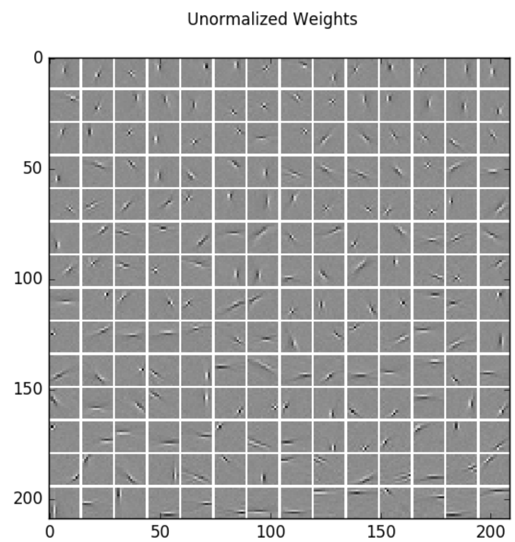
If you are new on GRBMs, first see GRBM_2D_example.

For a theoretical and empirical analysis of on GRBMs on natural image patches see Gaussian-binary restricted Boltzmann machines for modeling natural image statistics. Melchior et. al. PLOS ONE 2017

3.1.10.2 Results

The *code* given below produces the following output.

Visualization of the learned filters, which are very similar to those of ICA.



For a better visualization of the structure, here are the same filters normalized independently.

Sampling results for some examples. The first row shows some training data and the following rows are the results after one step of Gibbs-sampling starting from the previous row.

The log-likelihood and reconstruction error for training and test data

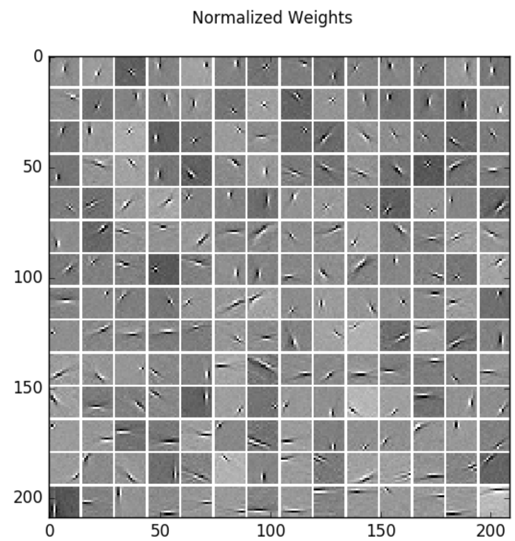
```
              Epoch   RE train        RE test         LL train        LL test
AIS:          200     0.73291         0.75427         -268.34107      -270.82759
reverse AIS:          0.73291         0.75427         -268.34078      -270.82731
```

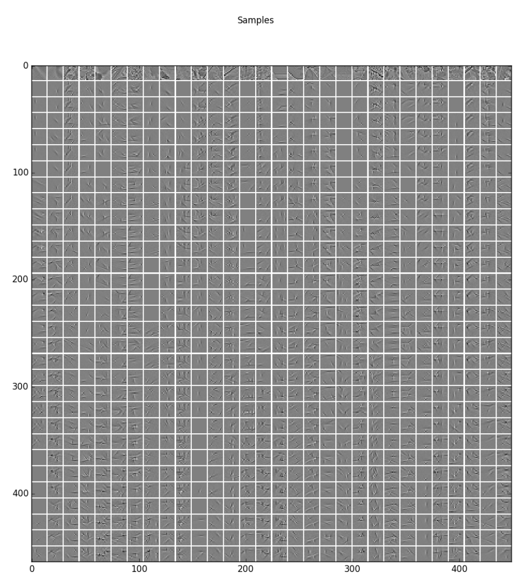
To analyze the optimal response of the learn filters we can fit a Gabor-wavelet parametrized in angle and frequency, and plot the optimal grating, here for 20 filters

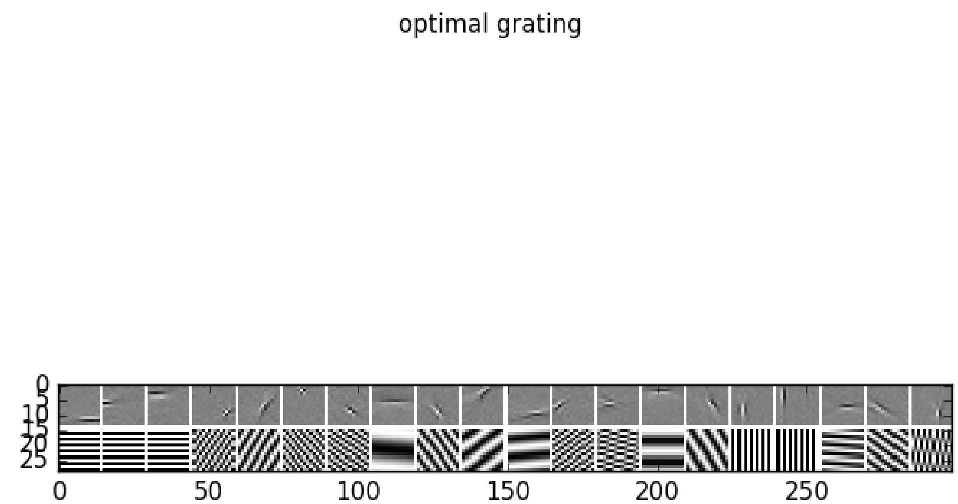
as well as the corresponding tuning curves, which show the responds/activities as a function frequency in pixels/cycle (left) and angle in rad (right).

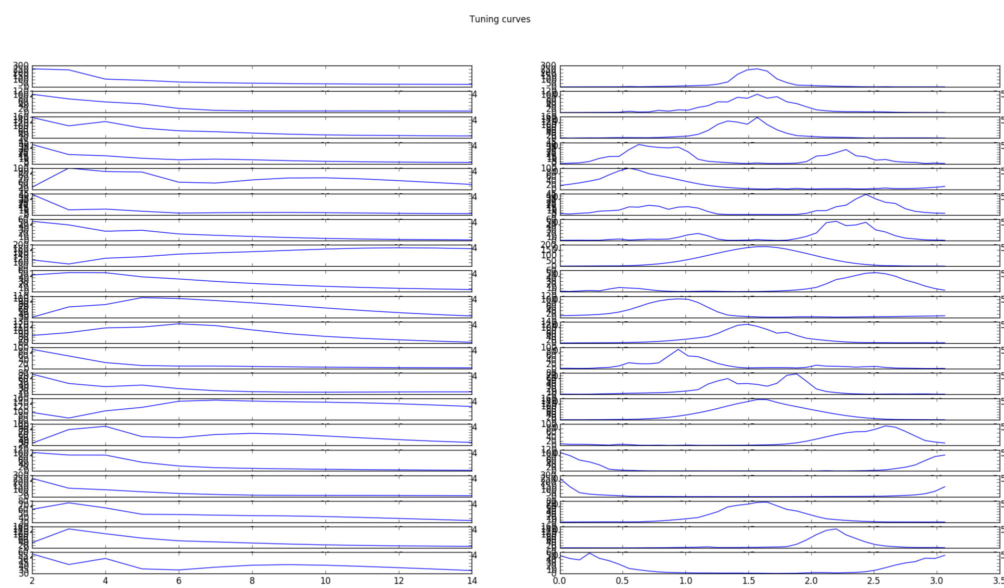
Furthermore, we can plot the histogram of all filters over the frequencies in pixels/cycle (left) and angles in rad (right).

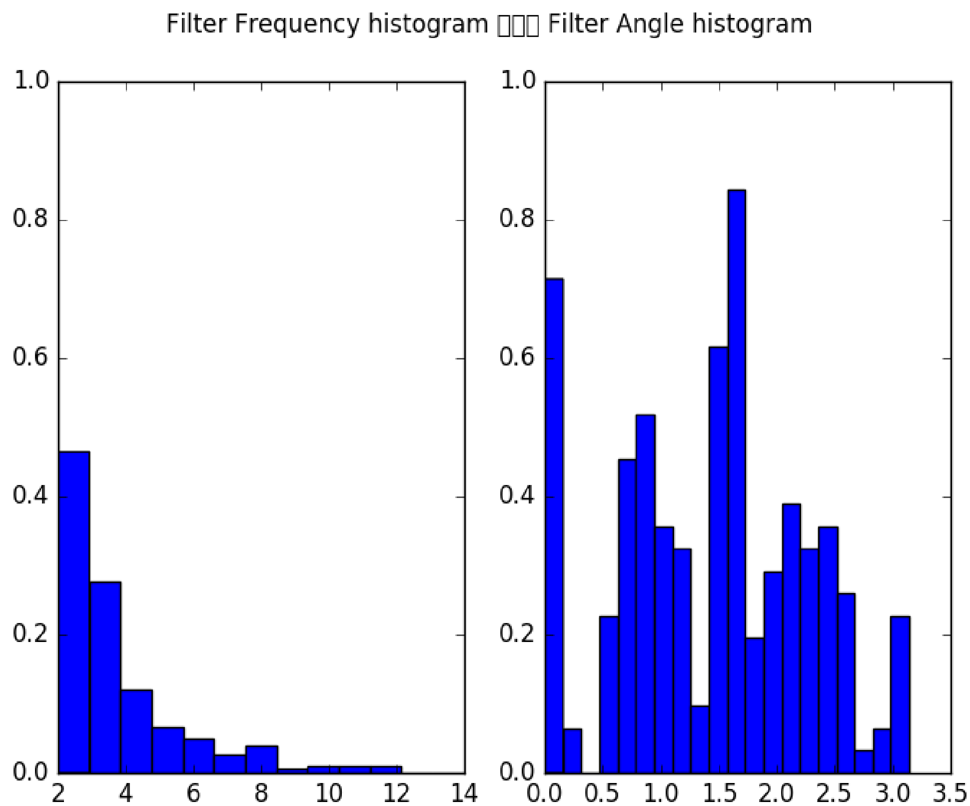
Compare the results with thos of ICA_natural_images, and AE_natural_images..


Normalized Weights
0
50
100
150
200
0
50
100
150
200


Samples
0
100
200
300
400
0
100
200
300
400


optimal grating
0
5
10
15
20
25
0
50
100
150
200
250


Tuning curves



3.1.10.3 Source code

```
""" Example for Gaussian-binary restricted Boltzmann machines (GRBM) on 2D data.

    :Version:
        1.1.0

    :Date:
        25.04.2017

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2017 Jan Melchior

        This file is part of the Python library PyDeep.
```

(continues on next page)

(continued from previous page)

```
        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""

# Import numpy+extensions, i/o functions, preprocessing, and visualization.
import numpy as numx
import pydeep.base.numpyextension as numxext
import pydeep.misc.io as io
import pydeep.preprocessing as pre
import pydeep.misc.visualization as vis

# Model imports: RBM estimator, model and trainer module
import pydeep.rbm.estimator as estimator
import pydeep.rbm.model as model
import pydeep.rbm.trainer as trainer

# Set random seed (optional)
# (optional, if stochastic processes are involved we get the same results)
numx.random.seed(42)

# Load data (download is not existing)
data = io.load_natural_image_patches('NaturalImage.mat')

# Remove the mean of ech image patch separately (also works without)
data = pre.remove_rows_means(data)

# Set input/output dimensions
v1 = 14
v2 = 14
h1 = 14
h2 = 14

# Whiten data using ZCA
zca = pre.ZCA(v1 * v2)
zca.train(data)
data = zca.project(data)

# Split into training/test data
train_data = data[0:40000]
test_data = data[40000:70000]

# Set restriction factor, learning rate, batch size and maximal number of epochs
restrict = 0.01 * numx.max(numxext.get_norms(train_data, axis=1))
eps = 0.1
batch_size = 100
```

(continues on next page)

(continued from previous page)

```
max_epochs = 200

# Create model, initial weights=Glorot init., initial sigma=1.0, initial bias=0,
# no centering (Usually pass the data=training_data for a automatic init. that is
# set the bias and sigma to the data mean and data std. respectively, for
# whitened data centering is not an advantage)
rbm = model.GaussianBinaryVarianceRBM(number_visibles=v1 * v2,
                                      number_hiddens=h1 * h2,
                                      initial_weights='AUTO',
                                      initial_visible_bias=0,
                                      initial_hidden_bias=0,
                                      initial_sigma=1.0,
                                      initial_visible_offsets=0.0,
                                      initial_hidden_offsets=0.0,
                                      dtype=numx.float64)

# Set the hidden bias such that the scaling factor is 0.01
rbm.bh = -(numxext.get_norms(rbm.w + rbm.bv.T, axis=0) - numxext.get_norms(
    rbm.bv, axis=None)) / 2.0 + numx.log(0.01)
rbm.bh = rbm.bh.reshape(1, h1 * h2)

# Training with CD-1
k = 1
trainer_cd = trainer.CD(rbm)

# Train model, status every 10th epoch
step = 10
print('Training')
print('Epoch\tRE train\tRE test \tLL train\tLL test ')
for epoch in range(0, max_epochs + 1, 1):

    # Shuffle training samples (optional)
    train_data = numx.random.permutation(train_data)

    # Print epoch and reconstruction errors every 'step' epochs.
    if epoch % step == 0:
        RE_train = numx.mean(estimator.reconstruction_error(rbm, train_data))
        RE_test = numx.mean(estimator.reconstruction_error(rbm, test_data))
        print('%5d \t%0.5f \t%0.5f' % (epoch, RE_train, RE_test))

    # Train one epoch with gradient restriction/clamping
    # No weight decay, momentum or sparseness is used
    for b in range(0, train_data.shape[0], batch_size):
        trainer_cd.train(data=train_data[b:(b + batch_size), :],
                         num_epochs=1,
                         epsilon=[eps, 0.0, eps, eps * 0.1],
                         k=k,
                         momentum=0.0,
                         reg_l1norm=0.0,
                         reg_l2norm=0.0,
                         reg_sparseness=0,
                         desired_sparseness=None,
                         update_visible_offsets=0.0,
                         update_hidden_offsets=0.0,
                         offset_typ='00',
                         restrict_gradient=restrict,
                         restriction_norm='Cols',
```

(continues on next page)

(continued from previous page)

```
                         use_hidden_states=False,
                         use_centered_gradient=False)

# Calculate reconstruction error
RE_train = numx.mean(estimator.reconstruction_error(rbm, train_data))
RE_test = numx.mean(estimator.reconstruction_error(rbm, test_data))
print '%5d \t%0.5f \t%0.5f' % (max_epochs, RE_train, RE_test)

# Approximate partition function by AIS (tends to overestimate)
logZ = estimator.annealed_importance_sampling(rbm)[0]
LL_train = numx.mean(estimator.log_likelihood_v(rbm, logZ, train_data))
LL_test = numx.mean(estimator.log_likelihood_v(rbm, logZ, test_data))
print 'AIS: \t%0.5f \t%0.5f' % (LL_train, LL_test)

# Approximate partition function by reverse AIS (tends to underestimate)
logZ = estimator.reverse_annealed_importance_sampling(rbm)[0]
LL_train = numx.mean(estimator.log_likelihood_v(rbm, logZ, train_data))
LL_test = numx.mean(estimator.log_likelihood_v(rbm, logZ, test_data))
print 'reverse AIS \t%0.5f \t%0.5f' % (LL_train, LL_test)

# Reorder RBM features by average activity decreasingly
rbmReordered = vis.reorder_filter_by_hidden_activation(rbm, train_data)

# Display RBM parameters
vis.imshow_standard_rbm_parameters(rbmReordered, v1, v2, h1, h2)

# Sample some steps and show results
samples = vis.generate_samples(rbm, train_data[0:30], 30, 1, v1, v2, False, None)
vis.imshow_matrix(samples, 'Samples')

# Get the optimal gabor wavelet frequency and angle for the filters
opt_frq, opt_ang = vis.filter_frequency_and_angle(rbm.w, num_of_angles=40)

# Show some tuning curves
num_filters =20
vis.imshow_filter_tuning_curve(rbm.w[:,0:num_filters], num_of_ang=40)

# Show some optima grating
vis.imshow_filter_optimal_gratings(rbm.w[:,0:num_filters],
                                   opt_frq[0:num_filters],
                                   opt_ang[0:num_filters])

# Show histograms of frequencies and angles.
vis.imshow_filter_frequency_angle_histogram(opt_frq=opt_frq,
                                            opt_ang=opt_ang,
                                            max_wavelength=14)

# Show all windows.
vis.show()
```

3.1.11 Autoencoder on a natural image patches

Example for Autoencoders (Autoencoder) on natural image patches.

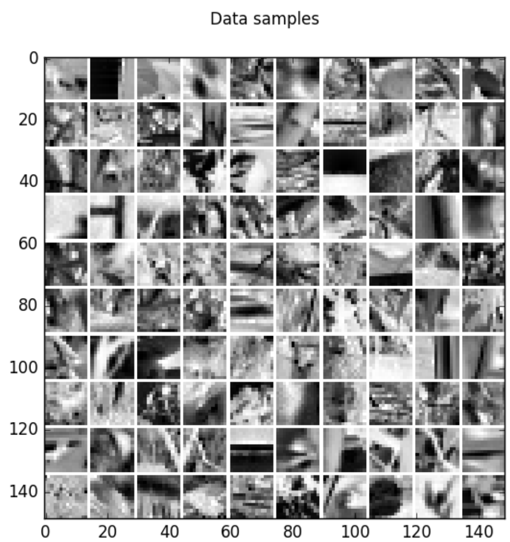
3.1.11.1 Theory

If you are new on Autoencoders visit Autoencoder tutorial or watch the video course by Andrew Ng.

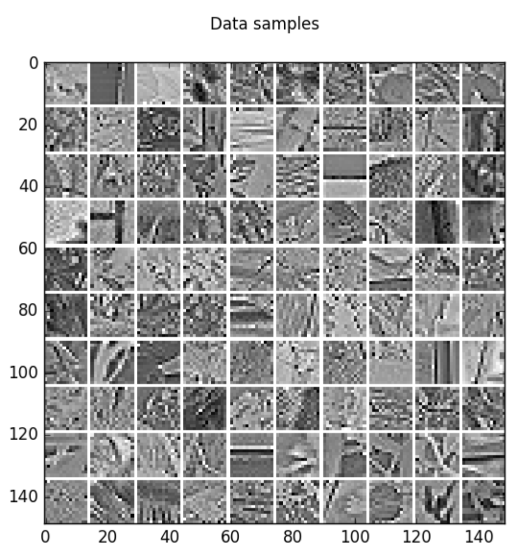
3.1.11.2 Results

The *code* given below produces the following output that is impressively similar to the results produced by ICA or GRBMs.

Visualization of 100 examples of the gray scale natural image dataset.

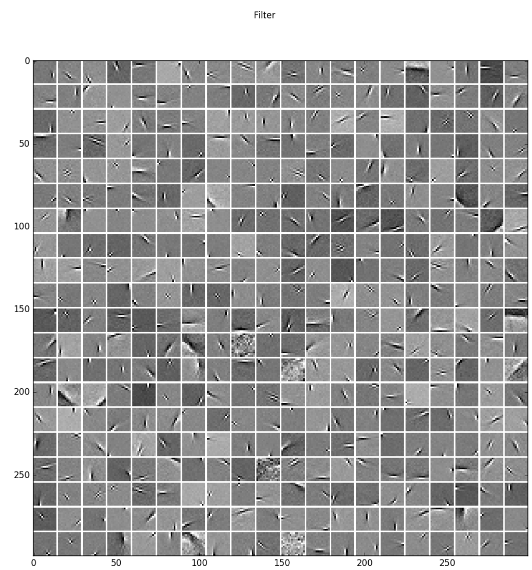


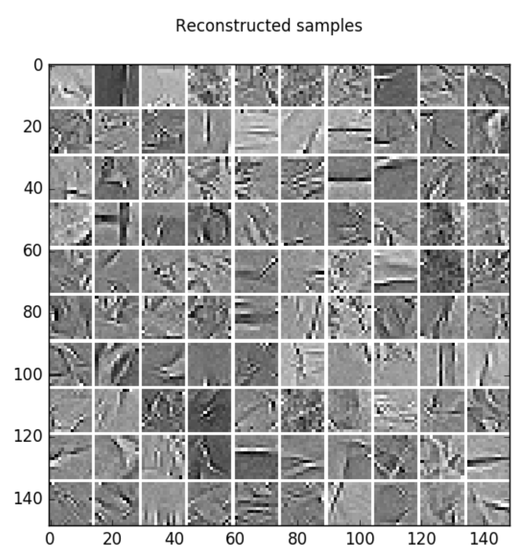
The corresponding whitened image patches.



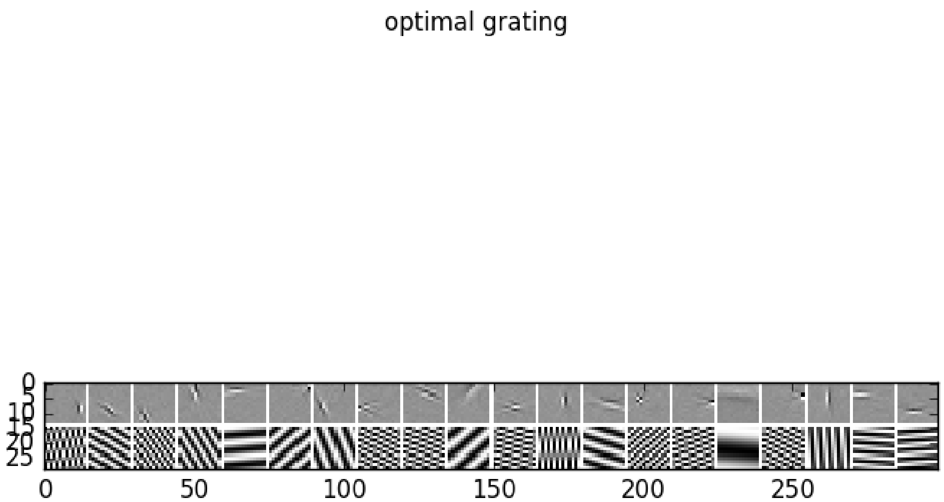
The learned filters from the whitened natural image patches.

The corresponding reconstruction of the model, that is the encoding followed by the decoding.


Filter
0
50
100
150
200
250
0
50
100
150
200
250


Reconstructed samples
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140

To analyze the optimal response of the learn filters we can fit a Gabor-wavelet parametrized in angle and frequency, and plot the optimal grating, here for 20 filters



as well as the corresponding tuning curves, which show the responds/activities as a function frequency in pixels/cycle (left) and angle in rad (right).

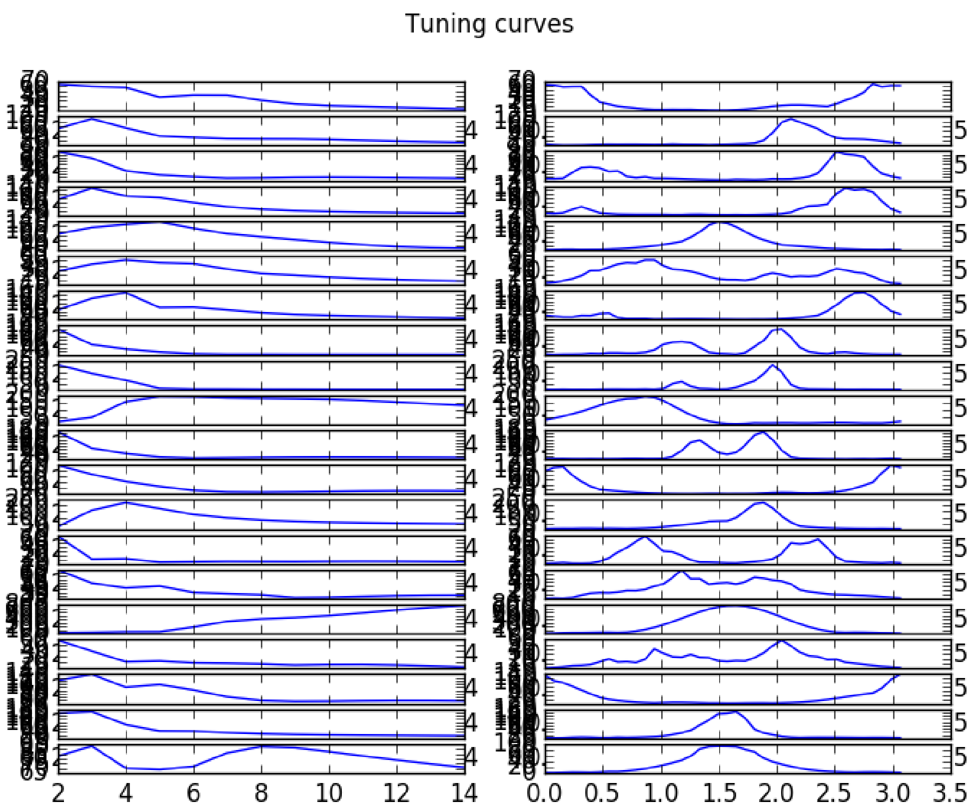
Furthermore, we can plot the histogram of all filters over the frequencies in pixels/cycle (left) and angles in rad (right).

We can also train the model on the unwhitened data leading to the following filters that cover also lower frequencies.

See also GRBM_natural_images, and ICA_natural_images.

3.1.11.3 Source code

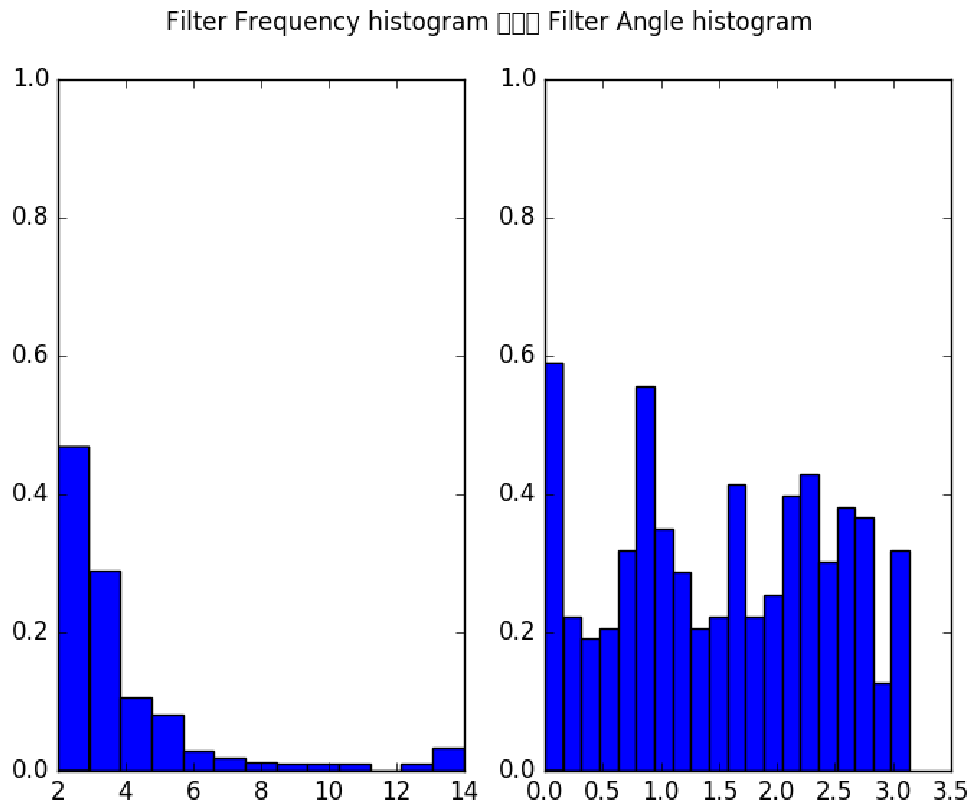
```
""" Example for sparse Autoencoder (SAE) on natural image patches.

    :Version:
        1.0.0

    :Date:
        25.01.2018

    :Author:
        Jan Melchior

    :Contact:
        JanMelchior@gmx.de
```
(continues on next page)

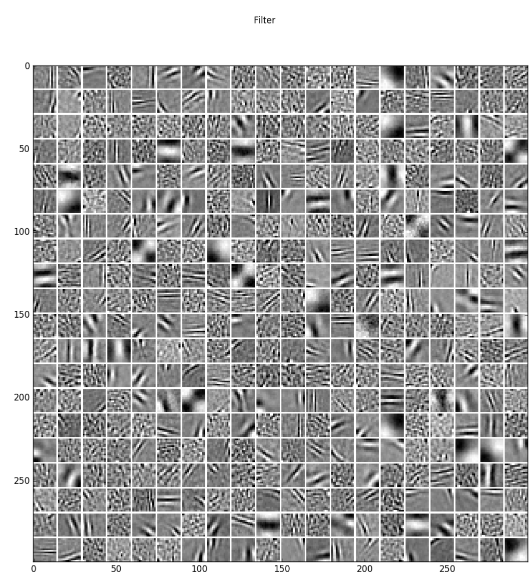


(continued from previous page)

```
    :License:

        Copyright (C) 2018 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""
# Import numpy, i/o functions, preprocessing, and visualization.
import numpy as numx
import pydeep.misc.io as io
import pydeep.misc.visualization as vis
```

(continues on next page)


Filter Frequency histogram
Filter Angle histogram
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
12
14
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Filter
0
50
100
150
200
250

(continued from previous page)

```
import pydeep.preprocessing as pre

# Import cost functions, activation function, Autencoder and trainer module
import pydeep.base.activationfunction as act
import pydeep.base.costfunction as cost
import pydeep.ae.model as aeModel
import pydeep.ae.trainer as aeTrainer

# Set random seed
numx.random.seed(42)

# Load data (download is not existing)
data = io.load_natural_image_patches('NaturalImage.mat')

# Remove mean individually
data = pre.remove_rows_means(data)

# Shuffle data
data = numx.random.permutation(data)

# Specify input and hidden dimensions
h1 = 20
h2 = 20
v1 = 14
v2 = 14

# Whiten data using ZCA or change it to STANDARIZER for unwhitened results
zca = pre.ZCA(v1 * v2)
zca.train(data)
data = zca.project(data)

# Split in tarining and test data
train_data = data[0:50000]
test_data = data[50000:70000]

# Set hyperparameters batchsize and number of epochs
batch_size = 10
max_epochs = 20

# Create model with sigmoid hidden units, linear output units, and squared error.
ae = aeModel.AutoEncoder(v1*v2,
                         h1*h2,
                         data = train_data,
                         visible_activation_function = act.Identity(),
                         hidden_activation_function = act.Sigmoid(),
                         cost_function = cost.SquaredError(),
                         initial_weights = 0.01,
                         initial_visible_bias = 0.0,
                         initial_hidden_bias = -2.0,
# Set initially the units to be inactive, speeds up learning a little bit
                         initial_visible_offsets = 0.0,
                         initial_hidden_offsets = 0.02,
                         dtype = numx.float64)

# Initialized gradient descent trainer
trainer = aeTrainer.GDTrainer(ae)
```

(continues on next page)

(continued from previous page)

```
# Train model
print 'Training'
print 'Epoch\tRE train\t\tRE test\t\t\tSparsness train\t\tSparsness test '
for epoch in range(0,max_epochs+1,1) :

    # Shuffle data
    train_data = numx.random.permutation(train_data)

    # Print reconstruction errors and sparseness for Training and test data
    print epoch, ' \t\t', numx.mean(ae.reconstruction_error(train_data)), \
        ' \t', numx.mean(ae.reconstruction_error(test_data)),\
        ' \t', numx.mean(ae.encode(train_data)), \
        ' \t', numx.mean(ae.encode(test_data))
    for b in range(0,train_data.shape[0],batch_size):

        trainer.train(data = train_data[b:(b+batch_size),:],
                      num_epochs=1,
                      epsilon=0.1,
                      momentum=0.0,
                      update_visible_offsets=0.0,
                      update_hidden_offsets=0.01,
                      reg_L1Norm=0.0,
                      reg_L2Norm=0.0,
                      corruptor=None,
                      # Rather strong sparsity regularization
                      reg_sparseness = 2.0,
                      desired_sparseness=0.001,
                      reg_contractive=0.0,
                      reg_slowness=0.0,
                      data_next=None,
# The gradient restriction is important for fast learning, see also GRBMs
                      restrict_gradient=0.1,
                      restriction_norm='Cols')

# Show filters/features
filters = vis.tile_matrix_rows(ae.w, v1,v2,h1,h2, border_size = 1,
                               normalized = True)
vis.imshow_matrix(filters, 'Filter')

# Show samples
samples = vis.tile_matrix_rows(train_data[0:100].T, v1,v2,10,10,
                               border_size = 1,normalized = True)
vis.imshow_matrix(samples, 'Data samples')

# Show reconstruction
samples = vis.tile_matrix_rows(ae.decode(ae.encode(train_data[0:100])).T,
                               v1,v2,10,10, border_size = 1,
                               normalized = True)
vis.imshow_matrix(samples, 'Reconstructed samples')

# Get the optimal gabor wavelet frequency and angle for the filters
opt_frq, opt_ang = vis.filter_frequency_and_angle(ae.w, num_of_angles=40)

# Show some tuning curves
num_filters =20
vis.imshow_filter_tuning_curve(ae.w[:,0:num_filters], num_of_ang=40)
```

(continues on next page)

(continued from previous page)

```
# Show some optima grating
vis.imshow_filter_optimal_gratings(ae.w[:,0:num_filters],
                                   opt_frq[0:num_filters],
                                   opt_ang[0:num_filters])

# Show histograms of frequencies and angles.
vis.imshow_filter_frequency_angle_histogram(opt_frq=opt_frq,
                                            opt_ang=opt_ang,
                                            max_wavelength=14)

# Show all windows.
vis.show()
```

3.1.12 Autoencoder on MNIST

Example for training a centered Autoencoder on the MNIST handwritten digit dataset with and without contractive penalty, dropout, . . .

It allows to reproduce the results from the publication How to Center Deep Boltzmann Machines. Melchior et al. JMLR 2016..

3.1.12.1 Theory

If you are new on Autoencoders visit Autoencoder tutorial or watch the video course by Andrew Ng.

3.1.12.2 Results

The *code* given below produces the following output that is quite similar to the results produced by an RBM.

Visualization of 100 test samples.

The learned filters without regularization.

The corresponding reconstruction of the model, that is the encoding followed by the decoding.

The learned filters when a contractive penalty is used, leading to much more localized and less noisy filters.

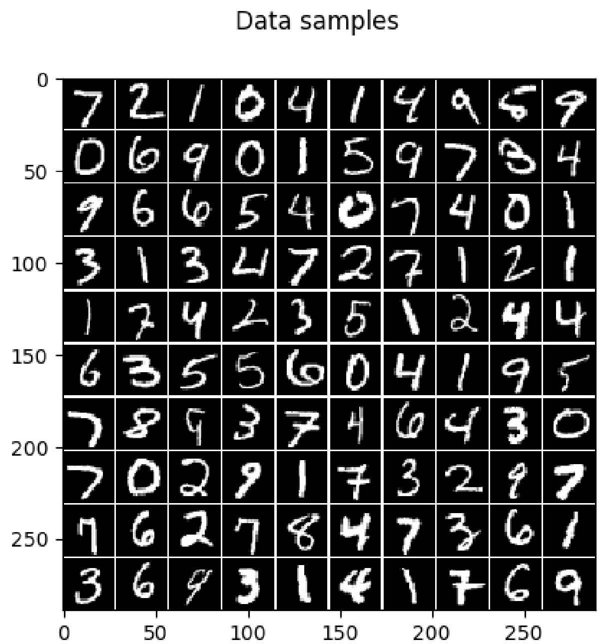
And the corresponding reconstruction of the model.

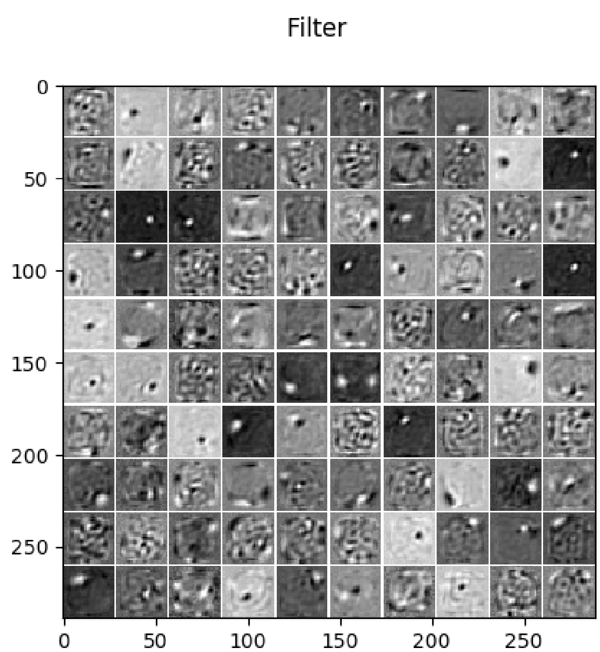
See also RBM_MNIST_big.

3.1.12.3 Source code

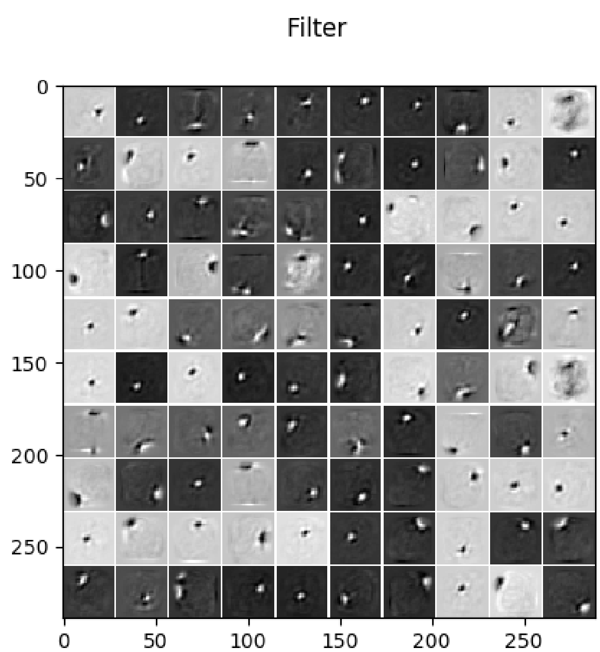
```
""" Example for contractive Autoencoder (SAE) on MNIST.

    :Version:
        1.0.0

    :Date:
        28.01.2018

    :Author:
        Jan Melchior
```

(continues on next page)


Data samples
0
50
100
150
200
250
0
50
100
150
200
250


Filter
0
50
100
150
200
250
0
50
100
150
200
250


Reconstructed samples
0
50
100
150
200
250
0 50 100 150 200 250


Filter
0
50
100
150
200
250
0 50 100 150 200 250

(continued from previous page)

```
    :Contact:
        JanMelchior@gmx.de

    :License:

        Copyright (C) 2018 Jan Melchior

        This file is part of the Python library PyDeep.

        PyDeep is free software: you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/licenses/>.

"""
# Import numpy, i/o functions, preprocessing, and visualization.
import numpy as numx
```

(continues on next page)

(continued from previous page)

```
import pydeep.misc.io as io
import pydeep.misc.visualization as vis
import pydeep.preprocessing as pre

# Import cost functions, activation function, Autencoder and trainer module
import pydeep.base.activationfunction as act
import pydeep.base.costfunction as cost
import pydeep.ae.model as aeModel
import pydeep.ae.trainer as aeTrainer

# Set random seed (optional)
numx.random.seed(42)

# Input and hidden dimensionality
v1 = v2 = 28
h1 = 10
h2 = 10

# Load data , get it from 'deeplearning.net/data/mnist/mnist.pkl.gz'
train_data, _, _, _, test_data, _ = io.load_mnist("mnist.pkl.gz", False)

# Set hyperparameters batchsize and number of epochs
batch_size = 10
max_epochs = 10

# Create model with sigmoid hidden units, linear output units, and squared error loss.
ae = aeModel.AutoEncoder(v1*v2,
                         h1*h2,
                         data = train_data,
                         visible_activation_function = act.Sigmoid(),
                         hidden_activation_function = act.Sigmoid(),
                         cost_function = cost.CrossEntropyError(),
                         initial_weights = 'AUTO',
                         initial_visible_bias = 'AUTO',
                         initial_hidden_bias = 'AUTO',
                         initial_visible_offsets = 'AUTO',
                         initial_hidden_offsets = 'AUTO',
                         dtype = numx.float64)

# Initialized gradient descent trainer
trainer = aeTrainer.GDTrainer(ae)

# Train model
print 'Training'
print 'Epoch\tRE train\t\tRE test\t\t\tSparsness train\t\tSparsness test '
for epoch in range(0,max_epochs+1,1) :

    # Shuffle data
    train_data = numx.random.permutation(train_data)

    # Print reconstruction errors and sparseness for Training and test data
    print epoch, ' \t\t', numx.mean(ae.reconstruction_error(train_data)), ' \t',\
        numx.mean(ae.reconstruction_error(test_data)), ' \t', \
        numx.mean(ae.encode(train_data)), ' \t',\
        numx.mean(ae.encode(test_data))
    for b in range(0,train_data.shape[0],batch_size):
```

(continues on next page)

(continued from previous page)

```
        trainer.train(data = train_data[b:(b+batch_size),:],
                      num_epochs=1,
                      epsilon=0.1,
                      momentum=0.0,
                      update_visible_offsets=0.0,
                      update_hidden_offsets=0.01,
                      reg_L1Norm=0.0,
                      reg_L2Norm=0.0,
                      corruptor=None,
                      reg_sparseness = 0.0,
                      desired_sparseness=0.0,
                      # Set to 0.0 to disable contractive penalty
                      reg_contractive=0.3,
                      reg_slowness=0.0,
                      data_next=None,
                      restrict_gradient=0.0,
                      restriction_norm='Cols')

# Show filters/features
filters = vis.tile_matrix_rows(ae.w, v1,v2,h1,h2, border_size = 1,
                               normalized = True)
vis.imshow_matrix(filters, 'Filter')

# Show samples
samples = vis.tile_matrix_rows(test_data[0:100].T, v1,v2,10,10,
                               border_size = 1,
                               normalized = True)
vis.imshow_matrix(samples, 'Data samples')

# Show reconstruction
samples = vis.tile_matrix_rows(ae.decode(ae.encode(test_data[0:100])).T,
                               v1,v2,10,10,
                               border_size = 1,
                               normalized = True)
vis.imshow_matrix(samples, 'Reconstructed samples')

# Show all windows.
vis.show()
```

The tutorials show how to reproduce results described in the following publications

- Gaussian-binary restricted Boltzmann machines for modeling natural image statistics. Melchior, J., Wang, N., & Wiskott, L.. (2017). PLOS ONE, 12(2), 1–24.
- How to Center Deep Boltzmann Machines. Melchior, J., Fischer, A., & Wiskott, L.. (2016). Journal of Machine Learning Research, 17(99), 1–61.
- Gaussian-binary Restricted Boltzmann Machines on Modeling Natural Image statistics Wang, N., Melchior, J., & Wiskott, L.. (2014). (Vol. 1401.5900). arXiv.org e-Print archive.
- How to Center Binary Restricted Boltzmann Machines (Vol. 1311.1354). Melchior, J., Fischer, A., Wang, N., & Wiskott, L.. (2013). arXiv.org e-Print archive.
- An Analysis of Gaussian-Binary Restricted Boltzmann Machines for Natural Images. Wang, N., Melchior, J., & Wiskott, L.. (2012). In Proc. 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Apr 25–27, Bruges, Belgium (pp. 287–292).
- Learning Natural Image Statistics with Gaussian-Binary Restricted Boltzmann Machines. Melchior, J, 29.05.2012. Master's thesis, Applied Computer Science, Univ. of Bochum, Germany.

For an introduction to Restricted Boltzmann machines especially for Gaussian input variables you can have a look into my master's theses

A good introduction into several machine learning topics with exercises and video lectures can be found in the here course Material .

CHAPTER 4

Documentation

4.1 Documentation

API documentation for PyDeep.

4.1.1 pydeep

Root package directory containing all subpackages og the library.

Version 1.1.0

Date 19.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

This file is part of the Python library PyDeep.

PyDeep is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

4.1.1.1 ae

Module initializer includes all sub-modules for the autoencoder module.

Version 1.0

Date 21.01.2018

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2018 Jan Melchior

4.1.1.1.1 model

This module provides a general implementation of a 3 layer tied weights Auto-encoder (x-h-y). The code is focused on readability and clearness, while keeping the efficiency and flexibility high. Several activation functions are available for visible and hidden units which can be mixed arbitrarily. The code can easily be adapted to AEs without tied weights. For deep AEs the FFN code can be adapted.

Implemented

- AE - Auto-encoder (centered)
- DAE - Denoising Auto-encoder (centered)
- SAE - Sparse Auto-encoder (centered)
- CAE - Contractive Auto-encoder (centered)
- SLAE - Slow Auto-encoder (centered)

Info http://ufldl.stanford.edu/wiki/index.php/Sparse_Coding:_Autoencoder_Interpretation

Version 1.0

Date 08.02.2016

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2016 Jan Melchior

4.1.1.1.1.1 AutoEncoder

class pydeep.ae.model.**AutoEncoder**(*number_visibles*, *number_hiddens*, *data=None*, *visible_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *hidden_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *cost_function=<class 'pydeep.base.costfunction.CrossEntropyError'>*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Class for a 3 Layer Auto-encoder (x-h-y) with tied weights.

_AutoEncoder__get_sparse_penalty_gradient_part(*h*, *desired_sparseness*)

This function computes the desired part of the gradient for the sparse penalty term. Only used for efficiency.

Parameters

h: hidden activations -type: numpy array [num samples, input dim]

desired_sparseness: Desired average hidden activation. -type: float

Returs

The computed gradient part is returned

-type: numpy array [1, hidden dim]

__init__(*number_visibles*, *number_hiddens*, *data=None*, *visible_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *hidden_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *cost_function=<class 'pydeep.base.costfunction.CrossEntropyError'>*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

number_visibles: Number of the visible variables. -type: int

number_hiddens Number of hidden variables. -type: int

data: The training data for parameter

initialization if 'AUTO' is chosen.

-type: None or numpy array [num samples, input dim] or List of numpy arrays [num samples, input dim]

visible_activation_function: A non linear transformation function

for the visible units (default: Sigmoid)

-type: Subclass of ActivationFunction()

hidden_activation_function: A non linear transformation function

for the hidden units (default: Sigmoid)

-type: Subclass of ActivationFunction

cost_function A cost function (default: CrossEntropyError()) -type: subclass of FNNCostFunction()

initial_weights: Initial weights.'AUTO' is random

-type: 'AUTO', scalar or numpy array [input dim, output_dim]

initial_visible_bias: Initial visible bias.

'AUTO' is random 'INVERSE_SIGMOID' is the inverse Sigmoid of

the visilbe mean

-type: 'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]

initial_hidden_bias: Initial hidden bias.

'AUTO' is random 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean

-type: 'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]

initial_visible_offsets: Initial visible mean values.

AUTO=data mean or 0.5 if not data is given.

-type: 'AUTO', scalar or numpy array [1, input dim]

initial_hidden_offsets: Initial hidden mean values.

AUTO = 0.5

-type: 'AUTO', scalar or numpy array [1, output_dim]

dtype: Used data type i.e. numpy.float64

-type: numpy.float32 or numpy.float64 or numpy.longdouble

`_decode` (*h*)

The function propagates the activation of the hidden layer reverse through the network to the input layer.

Parameters

h: Output of the network -type: numpy array [num samples, hidden dim]

Returns

Input of the network.

-type: array [num samples, input dim]

`_encode` (*x*)

The function propagates the activation of the input layer through the network to the hidden/output layer.

Parameters

x: Input of the network. -type: numpy array [num samples, input dim]

Returns

Pre and Post synaptic output.

-type: List of arrays [num samples, hidden dim]

`_get_contractive_penalty`(*a_h*, *factor*)
Calculates contractive penalty cost for a data point x.

Parameters

a_h: Pre-synaptic activation of h: a_h = (Wx+c). -type: numpy array [num samples, hidden dim]

factor: Influence factor (lambda) for the penalty. -type: float

Returns

Contractive penalty costs for x.

-type: numpy array [num samples]

`_get_contractive_penalty_gradient`(*x*, *a_h*, *df_a_h*)
This function computes the gradient for the contractive penalty term.

Parameters

x: Training data. -type: numpy array [num samples, input dim]

a_h: Untransformed hidden activations -type: numpy array [num samples, input dim]

df_a_h: Derivative of untransformed hidden activations -type: numpy array [num samples, input dim]

Returs

The computed gradient is returned

-type: numpy array [input dim, hidden dim]

`_get_gradients`(*x*, *a_h*, *h*, *a_y*, *y*, *reg_contractive*, *reg_sparseness*, *desired_sparseness*, *reg_slowness*, *x_next*, *a_h_next*, *h_next*)
Computes the gradients of weights, visible and the hidden bias. Depending on whether contractive penalty and or sparse penalty is used the gradient changes.

Parameters

x: Training data. -type: numpy array [num samples, input dim]

a_h: Pre-synaptic activation of h: a_h = (Wx+c). -type: numpy array [num samples, output dim]

h Post-synaptic activation of h: h = f(a_h). -type: numpy array [num samples, output dim]

a_y: Pre-synaptic activation of y: a_y = (Wh+b). -type: numpy array [num samples, input dim]

y Post-synaptic activation of y: y = f(a_y). -type: numpy array [num samples, input dim]

reg_contractive: Contractive influence factor (lambda). -type: float

reg_sparseness: Sparseness influence factor (lambda). -type: float

desired_sparseness: Desired average hidden activation. -type: float

reg_slowness: Slowness influence factor. -type: float

x_next: Next Training data in Sequence. -type: numpy array [num samples, input dim]

a_h_next: Next pre-synaptic activation of h: a_h = (Wx+c). -type: numpy array [num samples, output dim]

h_next Next post-synaptic activation of h: h = f(a_h). -type: numpy array [num samples, input dim]

`_get_slowness_penalty` (*h*, *h_next*, *factor*)

Calculates slowness penalty cost for a data point x.

> **Warning:** Different penalties are used depending on the hidden activation function.

Parameters

h: hidden activation. -type: numpy array [num samples, hidden dim]

h_next: hidden activation of the next data point in a sequence. -type: numpy array [num samples, hidden dim]

factor: Influence factor (beta) for the penalty. -type: float

Returns

Sparseness penalty costs for x.

-type: numpy array [num samples]

`_get_slowness_penalty_gradient` (*x*, *x_next*, *h*, *h_next*, *df_a_h*, *df_a_h_next*)

This function computes the gradient for the slowness penalty term.

Parameters

x: Training data. -type: numpy array [num samples, input dim]

x_next: Next training data points in Sequence. -type: numpy array [num samples, input dim]

h: Corresponding hidden activations. -type: numpy array [num samples, output dim]

h_next: Corresponding next hidden activations. -type: numpy array [num samples, output dim]

df_a_h: Derivative of untransformed hidden activations. -type: numpy array [num samples, input dim]

df_a_h_next: Derivative of untransformed next hidden activations. -type: numpy array [num samples, input dim]

Returs

The computed gradient is returned

-type: numpy array [input dim, hidden dim]

`_get_sparse_penalty` (*h*, *factor*, *desired_sparseness*)

Calculates sparseness penalty cost for a data point x.

> **Warning:** Different penalties are used depending on the hidden activation function.

Parameters

h: hidden activation. -type: numpy array [num samples, hidden dim]

factor: Influence factor (beta) for the penalty. -type: float

desired_sparseness: Desired average hidden activation. -type: float

Returns

Sparseness penalty costs for x.

-type: numpy array [num samples]

`_get_sparse_penalty_gradient` (*h*, *df_a_h*, *desired_sparseness*)
This function computes the gradient for the sparse penalty term.

Parameters

h: hidden activations -type: numpy array [num samples, input dim]

df_a_h: Derivative of untransformed hidden activations -type: numpy array [num samples, input dim]

desired_sparseness: Desired average hidden activation. -type: float

Returs

The computed gradient part is returned

-type: numpy array [1, hidden dim]

`decode` (*h*)

The function propagates the activation of the hidden layer reverse through the network to the input layer.

Parameters

h: Output of the network -type: numpy array [num samples, hidden dim]

Returns

Pre and Post synaptic input.

-type: List of arrays [num samples, input dim]

`encode` (*x*)

The function propagates the activation of the input layer through the network to the hidden/output layer.

Parameters

x: Input of the network. -type: numpy array [num samples, input dim]

Returns

Output of the network.

-type: array [num samples, hidden dim]

`energy` (*x*, *contractive_penalty=0.0*, *sparse_penalty=0.0*, *desired_sparseness=0.01*, *x_next=None*, *slowness_penalty=0.0*)
Calculates the energy/cost for a data point x.

Parameters

x: Data points. -type: numpy array [num samples, input dim]

contractive_penalty: If a value > 0.0 is given the cost is also
calculated on the contractive penalty.

-type: float

sparse_penalty: If a value > 0.0 is given the cost is also
calculated on the sparseness penalty.

-type: float

desired_sparseness: Desired average hidden activation. -type: float

x_next: Next data points. -type: None or numpy array [num samples, input dim]

slowness_penalty: If a value > 0.0 is given the cost is also
calculated on the slowness penalty.

-type: float

Returns

Costs for x.

-type: numpy array [num samples]

finit_differences(*data*, *delta*, *reg_sparseness*, *desired_sparseness*, *reg_contractive*, *reg_slowness*, *data_next*)
Finite differences test for AEs. The finite differences test involves all functions of the model except init and reconstruction_error

data: The training data -type: numpy array [num samples, input dim]

delta: The learning rate. -type: numpy array[num parameters]

reg_sparseness: The parameter (epsilon) for the sparseness regularization. -type: float

desired_sparseness: Desired average hidden activation. -type: float

reg_contractive: The parameter (epsilon) for the contractive regularization. -type: float

reg_slowness: The parameter (epsilon) for the slowness regularization. -type: float

data_next: The next training data in the sequence. -type: numpy array [num samples, input dim]

reconstruction_error(*x*, *absolut=False*)
Calculates the reconstruction error for given training data.

Parameters

x: Datapoints -type: numpy array [num samples, input dim]

absolut: If true the absolute error is caluclated. -type: bool

Returns

Reconstruction error.

-type: List of arrays [num samples, 1]

4.1.1.1.2 sae

Helper class for stacked auto encoder networks.

Version 1.1.0

Date 21.01.2018

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.1.2.1 SAE

class pydeep.ae.sae.**SAE**(*list_of_autoencoders*)

Stack of auto encoders.

__init__(*list_of_autoencoders*)

Initializes the network with auto encoders.

Parameters **list_of_autoencoders** (*list*) – List of auto-encoders

backward_propagate(*output_data*)

Propagates the output back through the input.

Parameters **output_data** (*numpy array [batchsize x output dim]*) – Output data.

Returns Input of the network.

Return type numpy array [batchsize x input dim]

forward_propagate(*input_data*)

Propagates the data through the network.

Parameters **input_data** (*numpy array [batchsize x input dim]*) – Input data.

Returns Output of the network.

Return type numpy array [batchsize x output dim]

4.1.1.1.3 trainer

This module provides implementations for training different variants of Auto-encoders, modifications on standard gradient decent are provided (centering, denoising, dropout, sparseness, contractiveness, slowness L1-decay, L2-decay, momentum, gradient restriction)

Implemented

- GDTrainer

Info http://ufldl.stanford.edu/wiki/index.php/Sparse_Coding:_Autoencoder_Interpretation

Version 1.0

Date 21.01.2018

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.1.3.1 GDTrainer

class `pydeep.ae.trainer.`**`GDTrainer`**(*model*)

Auto encoder trainer using gradient descent.

`__init__`(*model*)

The constructor takes the model as input

Parameters

model: An auto-encoder object which should be trained. -type: AutoEncoder

`_train`(*data*, *epsilon*, *momentum*, *update_visible_offsets*, *update_hidden_offsets*, *corruptor*, *reg_L1Norm*, *reg_L2Norm*, *reg_sparseness*, *desired_sparseness*, *reg_contractive*, *reg_slowness*, *data_next*, *restrict_gradient*, *restriction_norm*)

The training for one batch is performed using gradient descent.

Parameters

data: The training data -type: numpy array [num samples, input dim]

epsilon: The learning rate. -type: numpy array[num parameters]

momentum: The momentum term. -type: numpy array[num parameters]

update_visible_offsets: The update step size for the models

visible offsets. Good value if functionality is used: 0.001

-type: float

update_hidden_offsets: The update step size for the models hidden

offsets. Good value if functionality is used: 0.001

-type: float

corruptor: Defines if and how the data gets corrupted.

(e.g. Gauss noise, dropout, Max out)

-type: corruptor

reg_L1Norm: The parameter for the L1 regularization -type: float

reg_L2Norm: The parameter for the L2 regularization,

also know as weight decay.

-type: float

reg_sparseness: The parameter (epsilon) for the sparseness regularization. -type: float

desired_sparseness: Desired average hidden activation. -type: float

reg_contractive: The parameter (epsilon) for the contractive regularization. -type: float

reg_slowness: The parameter (epsilon) for the slowness regularization. -type: float

data_next: The next training data in the sequence. -type: numpy array [num samples, input dim]

restrict_gradient: If a scalar is given the norm of the

weight gradient is restricted to stay below this value.

-type: None, float

restriction_norm: restricts the column norm, row norm or

Matrix norm.

-type: string: 'Cols','Rows', 'Mat'

train(*data*, *num_epochs=1*, *epsilon=0.1*, *momentum=0.0*, *update_visible_offsets=0.0*, *update_hidden_offsets=0.0*, *corruptor=None*, *reg_L1Norm=0.0*, *reg_L2Norm=0.0*, *reg_sparseness=0.0*, *desired_sparseness=0.01*, *reg_contractive=0.0*, *reg_slowness=0.0*, *data_next=None*, *restrict_gradient=False*, *restriction_norm='Mat'*)

The training for one batch is performed using gradient descent.

Parameters

data: The data used for training.

-type: list of numpy arrays [num samples input dimension]

num_epochs: Number of epochs to train. -type: int

epsilon: The learning rate. -type: numpy array[num parameters]

momentum: The momentum term. -type: numpy array[num parameters]

update_visible_offsets: The update step size for the models

visible offsets. Good value if functionality is used: 0.001

-type: float

update_hidden_offsets: The update step size for the models hidden

offsets. Good value if functionality is used: 0.001

-type: float

corruptor: Defines if and how the data gets corrupted. -type: corruptor

reg_L1Norm: The parameter for the L1 regularization -type: float

reg_L2Norm: The parameter for the L2 regularization, also know as weight decay. -type: float

reg_sparseness: The parameter (epsilon) for the sparseness regularization. -type: float

desired_sparseness: Desired average hidden activation. -type: float

reg_contractive: The parameter (epsilon) for the contractive regularization. -type: float

reg_slowness: The parameter (epsilon) for the slowness regularization. -type: float

data_next: The next training data in the sequence. -type: numpy array [num samples, input dim]

restrict_gradient: If a scalar is given the norm of the

weight gradient is restricted to stay below this value.

-type: None, float

restriction_norm: restricts the column norm, row norm or

Matrix norm.

-type: string: 'Cols','Rows', 'Mat'

4.1.1.2 base

Package providing basic/fundamental functions/structures such as cost-functions, activation-functions, preprocessing …

Version 1.1.0

Date 13.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

4.1.1.2.1 activationfunction

Different kind of non linear activation functions and their derivatives.

Implemented

 # Unbounded

 # Linear

- Identity

 # Piecewise-linear

- Rectifier
- RestrictedRectifier (hard bounded)
- LeakyRectifier

 # Soft-linear

- ExponentialLinear
- SigmoidWeightedLinear
- SoftPlus

 # Bounded

 # Step

- Step

 # Soft-Step

- Sigmoid
- SoftSign
- HyperbolicTangent
- SoftMax
- K-Winner takes all

 # Symmetric, periodic

- Radial Basis function
- Sinus

Info http://en.wikipedia.org/wiki/Activation_function

Version 1.1.1

Date 16.01.2018

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.2.1.1 Identity

class pydeep.base.activationfunction.**Identity**

Identity function.

Info http://www.wolframalpha.com/input/?i=line

classmethod ddf(*x*)

Calculates the second derivative of the identity function value for a given input x.

Parameters x (*scalar or numpy array.*) – Inout data.

Returns Value of the second derivative of the identity function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df(*x*)

Calculates the derivative of the identity function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the identity function for x.

Return type scalar or numpy array with the same shape as x.

classmethod dg(*y*)

Calculates the derivative of the inverse identity function value for a given input y.

Parameters y (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the inverse identity function for y.

Return type scalar or numpy array with the same shape as y.

classmethod f(*x*)

Calculates the identity function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the identity function for x.

Return type scalar or numpy array with the same shape as x.

classmethod g(*y*)

Calculates the inverse identity function value for a given input y.

Parameters y (*scalar or numpy array.*) – Input data.

Returns Value of the inverse identity function for y.

Return type scalar or numpy array with the same shape as y.

4.1.1.2.1.2 Rectifier

class pydeep.base.activationfunction.**Rectifier**

Rectifier activation function function.

Info http://www.wolframalpha.com/input/?i=max%280%2Cx%29&dataset=&asynchronous=false&equal=Submit

classmethod ddf(*x*)

Calculates the second derivative of the Rectifier function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the 2nd derivative of the Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df (*x*)
Calculates the derivative of the Rectifier function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

classmethod f (*x*)
Calculates the Rectifier function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.3 RestrictedRectifier

class pydeep.base.activationfunction.**RestrictedRectifier** (*restriction=1.0*)
Restricted Rectifier activation function function.

Info http://www.wolframalpha.com/input/?i=max%280%2Cx%29&dataset=&asynchronous=false&equal=Submit

__init__ (*restriction=1.0*)
Constructor.

Parameters restriction (*float.*) – Restriction value / upper limit value.

df (*x*)
Calculates the derivative of the Restricted Rectifier function value for a given input x.

Parameters x (*scalar or numpy array*) – Input data.

Returns Value of the derivative of the Restricted Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

f (*x*)
Calculates the Restricted Rectifier function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the Restricted Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.4 LeakyRectifier

class pydeep.base.activationfunction.**LeakyRectifier** (*negativeSlope=0.01, positiveSlope=1.0*)
Leaky Rectifier activation function function.

Info https://en.wikipedia.org/wiki/Activation_function

__init__ (*negativeSlope=0.01, positiveSlope=1.0*)
Constructor.

Parameters

- **negativeSlope** (*scalar*) – Slope when x < 0
- **positiveSlope** (*scalar*) – Slope when x >= 0

df (*x*)
Calculates the derivative of the Leaky Rectifier function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the derivative of the Leaky Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

f (*x*)
Calculates the Leaky Rectifier function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the Leaky Rectifier function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.5 ExponentialLinear

class pydeep.base.activationfunction.**ExponentialLinear** (*alpha=1.0*)
Exponential Linear activation function function.

Info https://en.wikipedia.org/wiki/Activation_function

__init__ (*alpha=1.0*)
Constructor.

Parameters **alpha** (*scalar*) – scaling factor

df (*x*)
Calculates the derivative of the Exponential Linear function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Exponential Linear function for x.

Return type scalar or numpy array with the same shape as x.

f (*x*)
Calculates the Exponential Linear function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the Exponential Linear function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.6 SigmoidWeightedLinear

class pydeep.base.activationfunction.**SigmoidWeightedLinear** (*beta=1.0*)
Sigmoid weighted linear units (also named Swish)

Info https://arxiv.org/pdf/1702.03118v1.pdf and for Swish: https://arxiv.org/pdf/1710.05941.pdf

__init__ (*beta=1.0*)
Constructor.

Parameters **beta** (*scalar*) – scaling factor

df (*x*)
Calculates the derivative of the Sigmoid weighted linear function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Sigmoid weighted linear function for x.

Return type scalar or numpy array with the same shape as x.

f (*x*)
Calculates the Sigmoid weighted linear function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the Sigmoid weighted linear function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.7 SoftPlus

class pydeep.base.activationfunction.**SoftPlus**
Soft Plus function.

Info http://www.wolframalpha.com/input/?i=log%28exp%28x%29%2B1%29

classmethod ddf (*x*)
Calculates the second derivative of the SoftPlus function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the 2nd derivative of the SoftPlus function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df (*x*)
Calculates the derivative of the SoftPlus function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the SoftPlus function for x.

Return type scalar or numpy array with the same shape as x.

classmethod dg (*y*)
Calculates the derivative of the inverse SoftPlus function value for a given input y.

Parameters **y** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the inverse SoftPlus function for x.

Return type scalar or numpy array with the same shape as y.

classmethod f (*x*)
Calculates the SoftPlus function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the SoftPlus function for x.

Return type scalar or numpy array with the same shape as x.

classmethod g (*y*)
Calculates the inverse SoftPlus function value for a given input y.

Parameters **y** (*scalar or numpy array.*) – Input data.

Returns Value of the inverse SoftPlus function for y.

Return type scalar or numpy array with the same shape as y.

4.1.1.2.1.8 Step

class pydeep.base.activationfunction.**Step**
Step activation function function.

classmethod ddf (*x*)
Calculates the second derivative of the step function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Step function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df (*x*)
Calculates the derivative of the step function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the step function for x.

Return type scalar or numpy array with the same shape as x.

classmethod f (*x*)
Calculates the step function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the step function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.9 Sigmoid

class pydeep.base.activationfunction.**Sigmoid**
Sigmoid function.

Info http://www.wolframalpha.com/input/?i=sigmoid

classmethod ddf (*x*)
Calculates the second derivative of the Sigmoid function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the second derivative of the Sigmoid function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df (*x*)
Calculates the derivative of the Sigmoid function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Sigmoid function for x.

Return type scalar or numpy array with the same shape as x.

classmethod dg(*y*)
Calculates the derivative of the inverse Sigmoid function value for a given input y.

Parameters **y** (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the inverse Sigmoid function for y.

Return type scalar or numpy array with the same shape as y.

classmethod f(*x*)
Calculates the Sigmoid function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the Sigmoid function for x.

Return type scalar or numpy array with the same shape as x.

classmethod g(*y*)
Calculates the inverse Sigmoid function value for a given input y.

Parameters **y** (*scalar or numpy array.*) – Input data.

Returns Value of the inverse Sigmoid function for y.

Return type scalar or numpy array with the same shape as y.

4.1.1.2.1.10 SoftSign

class pydeep.base.activationfunction.**SoftSign**
SoftSign function.

Info http://www.wolframalpha.com/input/?i=x%2F%281%2Babs%28x%29%29

classmethod ddf(*x*)
Calculates the second derivative of the SoftSign function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the 2nd derivative of the SoftSign function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df(*x*)
Calculates the derivative of the SoftSign function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the SoftSign function for x.

Return type scalar or numpy array with the same shape as x.

classmethod f(*x*)
Calculates the SoftSign function value for a given input x.

Parameters **x** (*scalar or numpy array.*) – Input data.

Returns Value of the SoftSign function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.11 HyperbolicTangent

class pydeep.base.activationfunction.**HyperbolicTangent**
HyperbolicTangent function.

Info http://www.wolframalpha.com/input/?i=tanh

classmethod ddf(*x*)
Calculates the second derivative of the Hyperbolic Tangent function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the second derivative of the Hyperbolic Tangent function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df(*x*)
Calculates the derivative of the Hyperbolic Tangent function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the derivative of the Hyperbolic Tangent function for x.

Return type scalar or numpy array with the same shape as x.

classmethod dg(*y*)
Calculates the derivative of the inverse Hyperbolic Tangent function value for a given input y.

Parameters y (*scalar or numpy array.*) – Input data.

Returns Value the derivative of the inverse Hyperbolic Tangent function for x.

Return type scalar or numpy array with the same shape as y.

classmethod f(*x*)
Calculates the Hyperbolic Tangent function value for a given input x.

Parameters x (*scalar or numpy array.*) – Input data.

Returns Value of the Hyperbolic Tangent function for x.

Return type scalar or numpy array with the same shape as x.

classmethod g(*y*)
Calculates the inverse Hyperbolic Tangent function value for a given input y.

Parameters y (*scalar or numpy array.*) – Input data.

Returns alue of the inverse Hyperbolic Tangent function for y.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.12 SoftMax

class pydeep.base.activationfunction.**SoftMax**
Soft Max function.

Info https://en.wikipedia.org/wiki/Activation_function

classmethod df(*x*)
Calculates the derivative of the SoftMax function value for a given input x.

Parameters x (*scalar or numpy array*) – Input data.

Returns Value of the derivative of the SoftMax function for x.

Return type scalar or numpy array with the same shape as x.

classmethod f(*x*)
Calculates the function value of the SoftMax function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the SoftMax function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.13 RadialBasis

class pydeep.base.activationfunction.**RadialBasis**(*mean=0.0*, *variance=1.0*)
Radial Basis function.

Info http://www.wolframalpha.com/input/?i=Gaussian

__init__(*mean=0.0*, *variance=1.0*)
Constructor.

Parameters

- **mean** (*scalar or numpy array*) – Mean of the function.
- **variance** (*scalar or numpy array*) – Variance of the function.

ddf(*x*)
Calculates the second derivative of the Radial Basis function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the second derivative of the Radial Basis function for x.

Return type scalar or numpy array with the same shape as x.

df(*x*)
Calculates the derivative of the Radial Basis function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the derivative of the Radial Basis function for x.

Return type scalar or numpy array with the same shape as x.

f(*x*)
Calculates the Radial Basis function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the Radial Basis function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.14 Sinus

class pydeep.base.activationfunction.**Sinus**
Sinus function.

Info http://www.wolframalpha.com/input/?i=sin(x)

classmethod ddf(*x*)
Calculates the second derivative of the Sinus function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the second derivative of the Sinus function for x.

Return type scalar or numpy array with the same shape as x.

classmethod df (*x*)
Calculates the derivative of the Sinus function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the derivative of the Sinus function for x.

Return type scalar or numpy array with the same shape as x.

classmethod f (*x*)
Calculates the function value of the Sinus function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the Sinus function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.1.15 KWinnerTakeAll

class pydeep.base.activationfunction.**KWinnerTakeAll** (*k*, *axis=1*, *activation_function=<pydeep.base.activationfunction.Identity object>*)

K Winner take all activation function.

WARNING The derivative gets already calcluated in the forward pass. Thus, for the same datapoint the order should always be forward_pass, backward_pass!

__init__ (*k*, *axis=1*, *activation_function=<pydeep.base.activationfunction.Identity object>*)
Constructor.

Parameters

- **k** (*Instance of an activation function*) – Number of active units.
- **axis** (*int*) – Axis to compute the maximum.
- **k** – activation_function

df (*x*)
Calculates the derivative of the KWTA function.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Derivative of the KWTA function

Return type scalar or numpy array with the same shape as x.

f (*x*)
Calculates the K-max function value for a given input x.

Parameters **x** (*scalar or numpy array*) – Input data.

Returns Value of the Kmax function for x.

Return type scalar or numpy array with the same shape as x.

4.1.1.2.2 basicstructure

This module provides basic structural elements, which different models have in common.

Implemented

- BipartiteGraph
- StackOfBipartiteGraphs

Version 1.1.0

Date 06.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

This file is part of the Python library PyDeep.

PyDeep is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

4.1.1.2.2.1 BipartiteGraph

class pydeep.base.basicstructure.**BipartiteGraph**(*number_visibles*, *number_hiddens*, *data=None*, *visible_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *hidden_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a bipartite graph structure.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *visible_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *hidden_activation_function=<class 'pydeep.base.activationfunction.Sigmoid'>*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.

- **number_hiddens** (*int*) – Number of the hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **visible_activation_function** (*pydeep.base.activationFunction*) – Activation function for the visible units.
- **hidden_activation_function** (*pydeep.base.activationFunction*) – Activation function for the hidden units.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visible mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64.

_add_hidden_units(*num_new_hiddens, position=0, initial_weights='AUTO', initial_bias='AUTO', initial_offsets='AUTO'*)

This function adds new hidden units at the given position to the model. .. Warning:: If the parameters are changed. the trainer needs to be reinitialized.

Parameters

- **num_new_hiddens** (*int*) – The number of new hidden units to add.
- **position** (*int*) – Position where the units should be added.
- **initial_weights** (*'AUTO' or scalar or numpy array [input_dim, num_new_hiddens]*) – The initial weight values for the hidden units.
- **initial_bias** (*'AUTO' or scalar or numpy array [1, num_new_hiddens]*) – The initial hidden bias values.
- **initial_offsets** (*'AUTO' or scalar or numpy array [1, num_new_hiddens]*) – The initial hidden mean values.

_add_visible_units(*num_new_visibles, position=0, initial_weights='AUTO', initial_bias='AUTO', initial_offsets='AUTO', data=None*)

This function adds new visible units at the given position to the model.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters

- **num_new_visibles** (*int*) – The number of new hidden units to add
- **position** (*int*) – Position where the units should be added.
- **initial_weights** (*'AUTO' or scalar or numpy array [num_new_visibles, output_dim]*) – The initial weight values for the hidden units.
- **initial_bias** (*numpy array [1, num_new_visibles]*) – The initial hidden bias values.
- **initial_offsets** (*numpy array [1, num_new_visibles]*) – The initial visible offset values.
- **data** (*numpy array [num datapoints, num_new_visibles]*) – Data for AUTO initialization.

_hidden_post_activation(*pre_act_h*)

Computes the Hidden (post) activations from hidden pre-activations.

Parameters **pre_act_h** (*numpy array [num data points, output_dim]*) – Hidden pre-activations.

Returns Hidden activations.

Return type numpy array [num data points, output_dim]

_hidden_pre_activation(*v*)

Computes the Hidden pre-activations from visible activations.

Parameters **v** (*numpy array [num data points, input_dim]*) – Visible activations.

Returns Hidden pre-synaptic activations.

Return type numpy array [num data points, output_dim]

_remove_hidden_units(*indices*)

This function removes the hidden units whose indices are given. .. Warning:: If the parameters are changed. the trainer needs to be reinitialized.

Parameters **indices** (*int or list of int or numpy array of int*) – Indices to remove.

_remove_visible_units(*indices*)

This function removes the visible units whose indices are given.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters **indices** (*int or list of int or numpy array of int*) – Indices of units to be remove.

_visible_post_activation(*pre_act_v*)

Computes the visible (post) activations from visible pre-activations.

Parameters **pre_act_v** (*numpy array [num data points, input_dim]*) – Visible pre-activations.

Returns Visible activations.

Return type numpy array [num data points, input_dim]

_visible_pre_activation(*h*)
Computes the visible pre-activations from hidden activations.

Parameters **h** (*numpy array [num data points, output_dim]*) – Hidden activations.

Returns Visible pre-synaptic activations.

Return type numpy array [num data points, input_dim]

get_parameters()
This function returns all model parameters in a list.

Returns The parameter references in a list.

Return type list

hidden_activation(*v*)
Computes the Hidden (post) activations from visible activations.

Parameters **v** (*numpy array [num data points, input_dim]*) – Visible activations.

Returns Hidden activations.

Return type numpy array [num data points, output_dim]

update_offsets(*new_visible_offsets=0.0*, *new_hidden_offsets=0.0*, *update_visible_offsets=1.0*, *update_hidden_offsets=1.0*)

This function updates the visible and hidden offsets. | –> update_offsets(0,0,1,1) reparameterizes to the normal binary RBM.

Parameters

- **new_visible_offsets** (*numpy arrays [1, input dim]*) – New visible means.
- **new_hidden_offsets** (*numpy arrays [1, output dim]*) – New hidden means.
- **update_visible_offsets** (*float*) – Update/Shifting factor for the visible means.
- **update_hidden_offsets** (*float*) – Update/Shifting factor for the hidden means.

update_parameters(*updates*)
This function updates all parameters given the updates derived by the training methods.

Parameters **updates** (*list of numpy arrays (num para. x [para. shape])*) – Parameter gradients.

visible_activation(*h*)
Computes the visible (post) activations from hidden activations.

Parameters **h** (*numpy array [num data points, output_dim]*) – Hidden activations.

Returns Visible activations.

Return type numpy array [num data points, input_dim]

4.1.1.2.2.2 StackOfBipartiteGraphs

class pydeep.base.basicstructure.**StackOfBipartiteGraphs**(*list_of_layers*)
Stacked network layers

__init__(*list_of_layers*)
Initializes the network with auto encoders.

Parameters **list_of_layers** (*list*) – List of Layers i.e. BipartiteGraph.

_check_network()
Check whether the network is consistent and raise an exception if it is not the case.

append_layer(*layer*)
Appends the model to the network.

Parameters **layer** (*Layer object i.e. BipartiteGraph.*) – Layer object.

backward_propagate(*output_data*)
Propagates the output back through the input.

Parameters **output_data** (*numpy array [batchsize x output dim]*) – Output data.

Returns Input of the network.

Return type numpy array [batchsize x input dim]

depth
Networks depth/ number of layers.

forward_propagate(*input_data*)
Propagates the data through the network.

Parameters **input_data** (*numpy array [batchsize x input dim]*) – Input data.

Returns Output of the network.

Return type numpy array [batchsize x output dim]

num_layers
Networks depth/ number of layers.

pop_last_layer()
Removes/pops the last layer in the network.

reconstruct(*input_data*)
Reconstructs the data by propagating the data to the output and back to the input.

Parameters **input_data** (*numpy array [batchsize x input dim]*) – Input data.

Returns Output of the network.

Return type numpy array [batchsize x output dim]

save(*path*, *save_states=False*)
Saves the network.

Parameters

- **path** (*string.*) – Filename+path.
- **save_states** (*bool*) – If true the current states are saved.

4.1.1.2.3 corruptor

This module provides implementations for corrupting the training data.

Implemented

- Identity
- Sampling Binary
- BinaryNoise
- Additive Gauss Noise
- Multiplicative Gauss Noise
- Dropout
- Random Permutation
- KeepKWinner
- KWinnerTakesAll

Info http://ufldl.stanford.edu/wiki/index.php/Sparse_Coding:_Autoencoder_Interpretation

Version 1.1.0

Date 13.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.2.3.1 Identity

class pydeep.base.corruptor.**Identity**

Dummy corruptor object.

classmethod corrupt(*data*)

The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.2 AdditiveGaussNoise

class pydeep.base.corruptor.**AdditiveGaussNoise**(*mean*, *std*)
An object that corrupts data by adding Gauss noise.

__init__(*mean*, *std*)
The function corrupts the data.

Parameters

- **mean** (*float*) – Constant the data is shifted
- **std** (*float*) – Standard deviation Added to the data.

corrupt(*data*)
The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.3 MultiGaussNoise

class pydeep.base.corruptor.**MultiGaussNoise**(*mean*, *std*)
An object that corrupts data by multiplying Gauss noise.

__init__(*mean*, *std*)
Corruptor contructor.

Parameters

- **mean** (*float*) – Constant the data is shifted
- **std** (*float*) – Standard deviation Added to the data.

corrupt(*data*)
The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.4 SamplingBinary

class pydeep.base.corruptor.**SamplingBinary**
Sample binary states (zero out) corruption.

classmethod corrupt (*data*)
The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.5 Dropout

class pydeep.base.corruptor.**Dropout** (*dropout_percentage=0.2*)
Dropout (zero out) corruption.

__init__ (*dropout_percentage=0.2*)
Corruptor contructor.

Parameters dropout_percentage (*float*) – Dropout percentage

corrupt (*data*)
The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.6 RandomPermutation

class pydeep.base.corruptor.**RandomPermutation** (*permutation_percentage=0.2*)
RandomPermutation corruption, a fix number of units change their activation values.

__init__ (*permutation_percentage=0.2*)
Corruptor contructor.

Parameters permutation_percentage (*float*) – permutation_percentage: Percentage of states to permute

corrupt (*data*)
The function corrupts the data.

Parameters data (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.7 KeepKWinner

class pydeep.base.corruptor.**KeepKWinner** (*k=10, axis=0*)
Implements K Winner stay. Keep the k max values and set the rest to 0.

__init__ (*k=10, axis=0*)
Corruptor contructor.

Parameters

- **k** (*int*) – Keep the k max values and set the rest to 0.
- **axis** (*int*) – Axis =0 across min batch, axis = 1 across hidden units

corrupt (*data*)

The function corrupts the data.

Parameters **data** (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.3.8 KWinnerTakesAll

class pydeep.base.corruptor.**KWinnerTakesAll**(*k=10*, *axis=0*)

Implements K Winner takes all. Keep the k max values and set the rest to 0.

__init__(*k=10*, *axis=0*)

Corruptor constructor.

Parameters

- **k** (*int*) – Keep the k max values and set the rest to 0.
- **axis** (*int*) – Axis =0 across min batch, axis = 1 across hidden units

corrupt (*data*)

The function corrupts the data.

Parameters **data** (*numpy array [num samples, layer dim]*) – Input of the layer.

Returns Corrupted data.

Return type numpy array [num samples, layer dim]

4.1.1.2.4 costfunction

Different kind of cost functions and their derivatives.

Implemented

- Squared error
- Absolute error
- Cross entropy
- Negative Log-likelihood

Version 1.1.0

Date 13.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.2.4.1 SquaredError

class `pydeep.base.costfunction.`**`SquaredError`**

Mean Squared error.

classmethod df (*x*, *t*)

Calculates the derivative of the Squared Error value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales.

Returns Value of the derivative of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

classmethod f (*x*, *t*)

Calculates the Squared Error value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales

Returns Value of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

4.1.1.2.4.2 AbsoluteError

class `pydeep.base.costfunction.`**`AbsoluteError`**

Absolute error.

classmethod df (*x*, *t*)

Calculates the derivative of the absolute error value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales.

Returns Value of the derivative of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

classmethod f (*x*, *t*)
Calculates the absolute error value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales

Returns Value of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

4.1.1.2.4.3 CrossEntropyError

class pydeep.base.costfunction.**CrossEntropyError**
Cross entropy functions.

classmethod df (*x*, *t*)
Calculates the derivative of the cross entropy value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales.

Returns Value of the derivative of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

classmethod f (*x*, *t*)
Calculates the cross entropy value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales

Returns Value of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

4.1.1.2.4.4 NegLogLikelihood

class pydeep.base.costfunction.**NegLogLikelihood**
Negative log likelihood function.

classmethod df (*x*, *t*)
Calculates the derivative of the negative log-likelihood value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales.

Returns Value of the derivative of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

classmethod f (*x*, *t*)
Calculates the negative log-likelihood value for a given input x and target t.

Parameters

- **x** (*scalar or numpy array*) – Input data.
- **t** (*scalar or numpy array*) – Target vales

Returns Value of the cost function for x and t.

Return type scalar or numpy array with the same shape as x and t.

4.1.1.2.5 numpyextension

This module provides different math functions that extend the numpy library.

Implemented

- log_sum_exp
- log_diff_exp
- get_norms
- multinominal_batch_sampling
- restrict_norms
- resize_norms
- angle_between_vectors
- get_2D_gauss_kernel
- generate_binary_code
- get_binary_label
- compare_index_of_max
- shuffle_dataset
- rotationSequence
- generate_2D_connection_matrix

Version 1.1.0

Date 13.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.2.5.1 log_sum_exp

`numpyextension.log_sum_exp(axis=0)`

Calculates the logarithm of the sum of e to the power of input 'x'. The method tries to avoid overflows by using the relationship: log(sum(exp(x))) = alpha + log(sum(exp(x-alpha))).

Parameters

- **x** (*float or numpy array*) – data.
- **axis** (*int*) – Sums along the given axis.

Returns Logarithm of the sum of exp of x.

Return type float or numpy array.

4.1.1.2.5.2 log_diff_exp

`numpyextension.log_diff_exp(axis=0)`

Calculates the logarithm of the diffs of e to the power of input 'x'. The method tries to avoid overflows by using the relationship: log(diff(exp(x))) = alpha + log(diff(exp(x-alpha))).

Parameters

- **x** (*float or numpy array*) – data.
- **axis** (*int*) – Diffs along the given axis.

Returns Logarithm of the diff of exp of x.

Return type float or numpy array.

4.1.1.2.5.3 multinominal_batch_sampling

`numpyextension.multinominal_batch_sampling(isnormalized=True)`

Sample states where only one entry is one and the rest is zero according to the given probablities.

Parameters

- **probabilties** (*numpy array [batchsize, number of states]*) – Matrix containing probabilities the rows have to sum to one, otherwise chosen normalized=False.
- **isnormalized** (*bool*) – If True the probabilities are assumed to be normalized. If False the probabilities are normalized.

Returns Sampled multinominal states.

Return type numpy array [batchsize, number of states]

4.1.1.2.5.4 get_norms

`numpyextension.get_norms(axis=0)`

Computes the norms of the matrix along a given axis.

Parameters

- **matrix** (*numpy array [num rows, num columns]*) – Matrix to get the norm of.

- **axis** (*int, None*) – Axis along the norm should be calculated. 0 = rows, 1 = cols, None = Matrix norm

Returns Norms along the given axis.

Return type numpy array or float

4.1.1.2.5.5 restrict_norms

numpyextension.**restrict_norms**(*max_norm*, *axis=0*)

This function restricts a matrix, its columns or rows to a given norm.

Parameters

- **matrix** (*numpy array [num rows, num columns]*) – Matrix that should be restricted.
- **max_norm** (*float*) – The maximal data norm.
- **axis** (*int, None*) – Restriction of the matrix along the given axis or the full matrix.

Returns Restricted matrix

Return type numpy array [num rows, num columns]

4.1.1.2.5.6 resize_norms

numpyextension.**resize_norms**(*norm*, *axis=0*)

This function resizes a matrix, its columns or rows to a given norm.

Parameters

- **matrix** (*numpy array [num rows, num columns]*) – Matrix that should be resized.
- **norm** (*float*) – The norm to restrict the matrix to.
- **axis** (*int, None*) – Resize of the matrix along the given axis.

Returns Resized matrix, however it is inplace

Return type numpy array [num rows, num columns]

4.1.1.2.5.7 angle_between_vectors

numpyextension.**angle_between_vectors**(*v2*, *degree=True*)

Computes the angle between two vectors.

Parameters

- **v1** (*numpy array*) – Vector 1.
- **v2** (*numpy array*) – Vector 2.
- **degree** (*bool*) – If true degrees is return, rad otherwise.

Returns Angle

Return type float

4.1.1.2.5.8 get_2d_gauss_kernel

numpyextension.**get_2d_gauss_kernel**(*height, shift=0, var=[1.0, 1.0]*)

Creates a 2D Gauss kernel of size NxM with variance 1.

Parameters

- **width** (*int*) – Number of pixels first dimension.
- **height** (*int*) – Number of pixels second dimension.
- **shift** (*int, 1D numpy array*) –

  The Gaussian is shifted by this amount from the center of the image.
  Passing a scalar -> x,y shifted by the same value
  Passing a vector -> x,y shifted accordingly
- **var** (*int, 1D numpy array or 2D numpy array*) –

  Variances or Covariance matrix.
  Passing a scalar -> Isotropic Gaussian
  Passing a vector -> Spherical covariance with vector values on the diagonals.
  Passing a matrix -> Full Gaussian

Returns Bit array containing the states.

Return type numpy array [num samples, bit_length]

4.1.1.2.5.9 generate_binary_code

numpyextension.**generate_binary_code**(*batch_size_exp=None, batch_number=0*)

This function can be used to generate all possible binary vectors of length 'bit_length'. It is possible to generate only a particular batch of the data, where 'batch_size_exp' controls the size of the batch (batch_size = 2**batch_size_exp) and 'batch_number' is the index of the batch that should be generated.

Example

bit_length = 2, batchSize = 2
-> All combination = 2^bit_length = 2^2 = 4
-> All_combinations / batchSize = 4 / 2 = 2 batches
-> _generate_bit_array(2, 2, 0) = [0,0],[0,1]
-> _generate_bit_array(2, 2, 1) = [1,0],[1,1]

Parameters

- **bit_length** (*int*) – Length of the bit vectors.
- **batch_size_exp** (*int*) – Size of the batch of data. Here: batch_size = 2**batch_size_exp
- **batch_number** (*int*) – Index of the batch.

Returns Bit array containing the states .

Return type numpy array [num samples, bit_length]

4.1.1.2.5.10 get_binary_label

`numpyextension.get_binary_label()`

This function converts a 1D-array with integers labels into a 2D-array containing binary labels.

Example

-> [3,1,0]|

-> [[1,0,0,0],[0,0,1,0],[0,0,0,1]]

Parameters **int_array** (*int*) – 1D array containing integers

Returns 2D array with binary labels.

Return type numpy array [num samples, num labels]

4.1.1.2.5.11 compare_index_of_max

`numpyextension.compare_index_of_max(target)`

Compares data rows by comparing the index of the maximal value e.g. Classifier output and true labels.

Example

[0.3,0.5,0.2],[0.2,0.6,0.2] -> 0

[0.3,0.5,0.2],[0.6,0.2,0.2] -> 1

Parameters

- **output** (*numpy array [batchsize, output_dim]*) – vectors usually containing label probabilties.
- **target** (*numpy array [batchsize, output_dim]*) – vectors usually containing true labels.

Returns Int array containging 0 is the two rows hat the maximum at the same index, 1 otherwise.

Return type numpy array [num samples, num labels]

4.1.1.2.5.12 shuffle_dataset

`numpyextension.shuffle_dataset(label)`

Shuffles the data points and the labels correspondingly.

Parameters

- **data** (*numpy array [num_datapoints, dim_datapoints]*) – Datapoints.
- **label** (*numpy array [num_datapoints]*) – Labels.

Returns Shuffled datapoints and labels.

Return type List of numpy arrays

4.1.1.2.5.13 rotation_sequence

`numpyextension.rotation_sequence(width, height, steps)`

Rotates a 2D image given as a 1D vector with shape[width*height] in 'steps' number of steps.

Parameters

- **image** (*int*) – Image as 1D vector.
- **width** (*int*) – Width of the image such that image.shape[0] = width*height.
- **height** (*int*) – Height of the image such that image.shape[0] = width*height.
- **steps** (*int*) – Number of rotation steps e.g. 360 each steps is 1 degree.

Returns Bool array containging True is the two rows hat the maximum at the same index, False otherwise.

Return type numpy array [num samples, num labels]

4.1.1.2.5.14 generate_2d_connection_matrix

`numpyextension.`**`generate_2d_connection_matrix`**(*input_y_dim*, *field_x_dim*, *field_y_dim*, *overlap_x_dim*, *overlap_y_dim*, *wrap_around=True*)

This function constructs a connection matrix, which can be used to force the weights to have local receptive fields.

Example

input_x_dim = 3,
input_y_dim = 3,
field_x_dim = 2,
field_y_dim = 2,
overlap_x_dim = 1,
overlap_y_dim = 1,
wrap_around=False)
leads to numx.array([[1,1,0,1,1,0,0,0,0],
[0,1,1,0,1,1,0,0,0],
[0,0,0,1,1,0,1,1,0],
[0,0,0,0,1,1,0,1,1]]).T

Parameters

- **input_x_dim** (*int*) – Input dimension.
- **input_y_dim** (*int*) – Output dimension.
- **field_x_dim** (*int*) – Size of the receptive field in dimension x.
- **field_y_dim** (*int*) – Size of the receptive field in dimension y.
- **overlap_x_dim** (*int*) – Overlap of the receptive fields in dimension x.
- **overlap_y_dim** (*int*) – Overlap of the receptive fields in dimension y.
- **wrap_around** (*bool*) – If true teh overlap has warp around in both dimensions.

Returns Connection matrix.

Return type numpy arrays [input dim, output dim]

4.1.1.3 misc

Package providing miscellaneous functionalities such as datsets, input-output, visualization, profiling methods ...

Version 1.1.0

Date 19.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

4.1.1.3.1 io

This class contains methods to read and write data.

Implemented

- Save/Load arbitrary objects.
- Save/Load images.
- Load MNIST.
- Load CIFAR.
- Load Caltech.
- Load olivietti face dataset
- Load nactural image patches
- Load UCI binary dataset
- Adult dataset
- Connect4 dataset
- Nips dataset
- Web dataset
- RCV1 dataset
- Mushrooms dataset
- DNA dataset
- OCR_letters dataset

Version 1.1.0

Date 29.03.2018

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2018 Jan Melchior

4.1.1.3.1.1 save_object

io.**save_object**(*path*, *info=True*, *compressed=True*)

Saves an object to file.

Parameters

- **obj** (*object*) – object to be saved.
- **path** (*string*) – Path and name of the file
- **info** (*bool*) – Prints statements if True
- **compressed** (*bool*) – Object will be compressed before storage.

Returns

Return type

4.1.1.3.1.2 save_image

io.**save_image**(*path*, *ext='bmp'*)

Saves a numpy array to an image file.

Parameters

- **array** (*numpy array [width, height]*) – Data to save
- **path** (*string*) – Path and name of the directory to save the image at.
- **ext** (*string*) – Extension for the image.

4.1.1.3.1.3 load_object

io.**load_object**(*info=True*, *compressed=True*)

Loads an object from file.

Parameters

- **path** (*string*) – Path and name of the file

- **info** (*bool*) – If True, prints status information.
- **compressed** (*bool*) –

Returns Loaded object

Return type object

4.1.1.3.1.4 load_image

io.**load_image**(*grayscale=False*)
Loads an image to numpy array.

Parameters

- **path** (*string*) – Path and name of the directory to save the image at.
- **grayscale** (*bool*) – If true image is converted to gray scale.

Returns Loaded image.

Return type numpy array [width, height]

4.1.1.3.1.5 download_file

io.**download_file**(*path*, *buffer_size=1048576*)
Downloads an saves a dataset from a given url.

Parameters

- **url** (*string*) – URL including filename (e.g. www.testpage.com/file1.zip)
- **path** (*string, None*) – Path the dataset should be stored including filename (e.g. /home/file1.zip).
- **buffer_size** (*int*) – Size of the streaming buffer in bytes.

4.1.1.3.1.6 load_mnist

io.**load_mnist**(*binary=False*)
Loads the MNIST digit data in binary [0,1] or real values [0,1].

Parameters

- **path** (*string*) – Path and name of the file to load.
- **binary** (*bool*) – If True returns binary images, real valued between [0,1] if False.

Returns MNIST dataset [train_set, train_lab, valid_set, valid_lab, test_set, test_lab]

Return type list of numpy arrays

4.1.1.3.1.7 load_caltech

io.**load_caltech**()
Loads the Caltech dataset.

Parameters **path** (*string*) – Path and name of the file to load.

Returns CAltech dataset [train_set, train_lab, valid_set, valid_lab, test_set, test_lab]

Return type list of numpy arrays

4.1.1.3.1.8 load_cifar

io.**load_cifar**(*grayscale=True*)

Loads the CIFAR dataset in real values [0,1]

Parameters

- **path** (*string*) – Path and name of the file to load.
- **grayscale** (*bool*) – If true converts the data to grayscale.

Returns CIFAR data and labels.

Return type list of numpy arrays ([# samples, 1024],[# samples])

4.1.1.3.1.9 load_natural_image_patches

io.**load_natural_image_patches**()

Loads the natural image patches used in the publication 'Gaussian-binary restricted Boltzmann machines for modeling na

See also:

http://journals.plos.org/plosone/article/authors?id=10.1371/journal.pone.0171015

Parameters **path** (*string*) – Path and name of the file to load.

Returns Natural image dataset

Return type numpy array

4.1.1.3.1.10 load_olivetti_faces

io.**load_olivetti_faces**(*correct_orientation=True*)

Loads the Olivetti face dataset 400 images, size 64x64

Parameters

- **path** (*string*) – Path and name of the file to load.
- **correct_orientation** (*bool*) – Corrects the orientation of the images.

Returns Olivetti face dataset

Return type numpy array

4.1.1.3.2 measuring

This module provides functions for measuring like time measuring for executed code.

Version 1.1.0

Date 19.03.2017

4.1.1.3.2.1 print_progress

measuring.**print_progress**(*num_steps*, *gauge=False*, *length=50*, *decimal_place=1*)

Prints the progress of a system at state 'step'.

Parameters

- **step** (*int*) – Current step between 0 and num_steps-1.
- **num_steps** (*int*) – Total number of steps.
- **gauge** (*bool*) – If true prints a gauge
- **length** (*int*) – Length of the gauge (in number of chars)
- **decimal_place** (*int*) – Number of decimal places to display.

4.1.1.3.2.2 Stopwatch

class pydeep.misc.measuring.**Stopwatch**

This class provides a stop watch for measuring the execution time of code.

__init__()

Constructor sets the starting time to the current time.

Info Will be overwritten by calling start()!

end()

Stops/ends the time measuring.

get_end_time()

Returns the end time.

Returns End time:

Return type datetime

get_expected_end_time(*iteration*, *num_iterations*)

Returns the expected end time.

Parameters

- **iteration** (*int*) – Current iteration

- **num_iterations** (*int*) – Total number of iterations.

Returns Expected end time.

Return type datetime

get_expected_interval(*iteration*, *num_iterations*)
Returns the expected interval/Time needed till ending.

Parameters

- **iteration** (*int*) – Current iteration
- **num_iterations** (*int*) – Total number of iterations.

Returns Expected interval.

Return type timedelta

get_interval()
Returns the current interval.

Returns Current interval:

Return type timedelta

get_start_time()
Returns the starting time.

Returns Starting time:

Return type datetime

pause()
Pauses the time measuring.

resume()
Resumes the time measuring.

start()
Sets the starting time to the current time.

update(*factor=1.0*)

Updates the internal variables. | Factor can be used to sum up not regular events in a loop: | Lets assume you have a loop over 100 sets and only every 10th | step you execute a function, then use update(factor=0.1) to | measure it.

Parameters **factor** (*float*) – Sums up factor*current interval

4.1.1.3.3 sshthreadpool

Provides a thread/script pooling mechanism based on ssh + screen.

Version 1.1.0

Date 19.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.3.3.1 SSHConnection

class pydeep.misc.sshthreadpool.**SSHConnection**(*hostname*, *username*, *password*, *max_cpus_usage=2*)

Handles a SSH connection.

__init__(*hostname*, *username*, *password*, *max_cpus_usage=2*)

Constructor takes hostname, username, password.

Parameters

- **hostname** (*string*) – Hostname or address of host.
- **username** (*string*) – SSH username.
- **password** (*string*) – SSH password.
- **max_cpus_usage** (*int*) – Maximal number of cores to be used

connect()

Connects to the server.

Returns turns True is the connection was sucessful

Return type bool

classmethod decrypt(*connection*, *password*)

Decrypts a connection object and returns it

Parameters

- **connection** (*string*) – SSHConnection to be decrypted
- **password** (*string*) – Encryption password

Returns Decrypted object

Return type *SSHConnection*

disconnect()

Disconnects from the server.

encrypt(*password*)

Encrypts the connection object.

Parameters **password** (*string*) – Encryption password

Returns Encrypted object

Return type object

execute_command(*command*)
Executes a command on the server and returns stdin, stdout, and stderr

Parameters **command** (*string*) – Command to be executed.

Returns stdin, stdout, and stderr

Return type list

execute_command_in_screen(*command*)

Executes a command in a screen on the server which is automatically detached and returns stdin, stdout, and stderr done.

Parameters **command** (*string*) – Command to be executed.

Returns stdin, stdout, and stderr

Return type list

get_number_users_processes()
Gets number of processes of the user on the server.

Returns number of processes

Return type int or None

get_number_users_screens()
Gets number of users screens on the server.

Returns number of users screens on the server.

Return type int or None

get_server_info()
Get the server info like number of cpus, meomory size and stores it in the corresponding variables.

Returns online or offline FLAG

Return type string

get_server_load()
Get the current cpu and memory of the server.

Returns

Average CPU(s) usage last 1 min,
Average CPU(s) usage last 5 min,
Average CPU(s) usage last 15 min,
Average memory usage,

Return type list

kill_all_processes()
Kills all processes.

Returns stdin, stdout, and stderr

Return type list

kill_all_screen_processes()
Kills all acreen processes.

Returns stdin, stdout, and stderr

Return type list

renice_processes (*value*)
Renices all processes.

Parameters **value** (*int or string*) – The New nice value -40 … 20

Returns stdin, stdout, and stderr

Return type list

4.1.1.3.3.2 SSHJob

class pydeep.misc.sshthreadpool.**SSHJob** (*command*, *num_threads=1*, *nice=19*)
Handles a SSH JOB.

__init__ (*command*, *num_threads=1*, *nice=19*)
Saves the encrypted serverlist to path.

Parameters

- **command** (*string*) – Command to be executed.
- **num_threads** (*int*) – Number of threads the job needs.
- **nice** (*int*) – Nice value for this job.

4.1.1.3.3.3 SSHPool

class pydeep.misc.sshthreadpool.**SSHPool** (*servers*)
Handles a pool of servers and allows to distribute jobs over the pool.

__init__ (*servers*)
Constructor takes a list of SSHConnections.

Parameters **servers** (*list*) – List of SSHConnections.

broadcast_command (*command*)
Executes a command an all servers.

Parameters **command** (*string*) – Command to be executed

Returns list of all stdin, stdout, and stderr

Return type list

broadcast_kill_all ()
Kills all processes on the server of the corresponding user.

Returns list of all stdin, stdout, and stderr

Return type list

broadcast_kill_all_screens ()
Kills all screens on the server of the corresponding user.

Returns list of all stdin, stdout, and stderr

Return type list

distribute_jobs (*jobs*, *status=False*, *ignore_load=False*, *sort_server=True*)
Distributes the jobs over the servers.

Parameters

- **jobs** (*string or SSHConnection*) – List of SSHJobs to be executeed on the servers.
- **status** (*bool*) – If true prints info about which job was started on which server.
- **ignore_load** (*bool*) – If true starts the job without caring about the current load.
- **sort_server** (*bool*) – If True Servers will be sorted by load.

Returns List of all started jobs and list of all remaining jobs

Return type list, list

execute_command (*host*, *command*)
Executes a command on a given server servers.

Parameters

- **host** (*string or SSHConnection*) – Hostname or connection object
- **command** (*string*) – Command to be executed

Returns

Return type

execute_command_in_screen (*host*, *command*)
Executes a command in a screen on a given server servers.

Parameters

- **host** (*string or SSHConnection*) – Hostname or connection object
- **command** (*string*) – Command to be executed

Returns list of all stdin, stdout, and stderr

Return type list

get_servers_info (*status=True*)

Reads the status of all servers, the information is stored in the SSHConnection objects. Additionally print to the console if status == True.

Parameters **status** (*bool*) – If true prints info.

get_servers_status ()
Reads the status of all servers and returns it a list. Additionally print to the console if status == True.

Returns list of header and list corresponding status information

Return type list, list

load_server (*path*, *password*, *append=True*)

Parameters

- **path** (*string*) – Path and filename.
- **password** (*string*) – Encrption password.
- **append** (*bool*) – If true, servers get append to list, if false server list gets replaced.

save_server (*path*, *password*)
Saves the encrypted serverlist to path.

Parameters

- **path** (*string*) – Path and filename
- **password** (*string*) – Encrption password

4.1.1.3.4 toyproblems

This class contains some example toy problems for RBMs.

Implemented

- Bars and Stripes dataset
- Shifting bars dataset
- 2D mixture of Laplacians

Version 1.1.0

Date 19.03.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

This file is part of the Python library PyDeep.

PyDeep is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

4.1.1.3.4.1 generate_2d_mixtures

toyproblems.**generate_2d_mixtures** (*mean=0.0*, *scale=0.7071067811865476*)
Creates a dataset containing 2D data points from a random mixtures of two independent Laplacian distributions.

Info Every sample is a 2-dimensional mixture of two sources. The sources can either be super_gauss or sub_gauss.

If x is one sample generated by mixing s, i.e. x = A*s, then the mixing_matrix is A.

Parameters

- **num_samples** (*int*) – The number of training samples.
- **mean** (*float*) – The mean of the two independent sources.
- **scale** (*float*) – The scale of the two independent sources.

Returns Data and mixing matrix.

Return type list of numpy arrays ([num samples, 2], [2,2])

4.1.1.3.4.2 generate_bars_and_stripes

`toyproblems.generate_bars_and_stripes(num_samples)`

Creates a dataset containing samples showing bars or stripes.

Parameters

- **length** (*int*) – Length of the bars/stripes.
- **num_samples** (*int*) – Number of samples

Returns Samples.

Return type numpy array [num_samples, length*length]

4.1.1.3.4.3 generate_bars_and_stripes_complete

`toyproblems.generate_bars_and_stripes_complete()`

Creates a dataset containing all possible samples showing bars or stripes.

Parameters **length** (*int*) – Length of the bars/stripes.

Returns Samples.

Return type numpy array [num_samples, length*length]

4.1.1.3.4.4 generate_shifting_bars

`toyproblems.generate_shifting_bars(bar_length, num_samples, random=False, flipped=False)`

Creates a dataset containing random positions of a bar of length "bar_length" in a strip of "length" dimensions.

Parameters

- **length** (*int*) – Number of dimensions
- **bar_length** (*int*) – Length of the bar
- **num_samples** (*int*) – Number of samples to generate
- **random** (*bool*) – If true dataset gets shuffled
- **flipped** (*bool*) – If true dataset gets flipped 0–>1 and 1–>0

Returns Samples of the shifting bars dataset.

Return type numpy array [samples, dimensions]

4.1.1.3.4.5 generate_shifting_bars_complete

`toyproblems.generate_shifting_bars_complete(bar_length, random=False, flipped=False)`

Creates a dataset containing all possible positions of a bar of length "bar_length" can take in a strip of "length" dimensions.

Parameters

- **length** (*int*) – Number of dimensions

- **bar_length** (*int*) – Length of the bar
- **random** (*bool*) – If true dataset gets shuffled
- **flipped** (*bool*) – If true dataset gets flipped 0–>1 and 1–>0

Returns Complete shifting bars dataset.

Return type numpy array [samples, dimensions]

4.1.1.3.5 visualization

This module provides functions for displaying and visualize data. It extends the matplotlib.pyplot.

Implemented

- Tile a matrix rows
- Tile a matrix columns
- Show a matrix
- Show plot
- Show a histogram
- Plot data
- Plot 2D weights
- Plot PDF-contours
- Show RBM parameters
- hidden_activation
- reorder_filter_by_hidden_activation
- generate_samples
- filter_frequency_and_angle
- filter_angle_response
- calculate_amari_distance
- Show the tuning curves
- Show the optimal gratings
- Show the frequency angle histogram

Version 1.1.0

Date 19.03.2017

Author Jan Melchior, Nan Wang

Contact JanMelchior@gmx.de

4.1.1.3.5.1 tile_matrix_columns

visualization.**tile_matrix_columns**(*tile_width*, *tile_height*, *num_tiles_x*, *num_tiles_y*, *border_size=1*, *normalized=True*)

Creates a matrix with tiles from columns.

Parameters

- **matrix** (*numpy array 2D*) – Matrix to display.
- **tile_width** (*int*) – Tile width dimension.
- **tile_height** (*int*) – Tile height dimension.
- **num_tiles_x** (*int*) – Number of tiles horizontal.
- **num_tiles_y** (*int*) – Number of tiles vertical.
- **border_size** (*int*) – Size of the border.
- **normalized** (*bool*) – If true each image gets normalized to be between 0..1.

Returns Matrix showing the 2D patches.

Return type 2D numpy array

4.1.1.3.5.2 tile_matrix_rows

visualization.**tile_matrix_rows**(*tile_width*, *tile_height*, *num_tiles_x*, *num_tiles_y*, *border_size=1*, *normalized=True*)

Creates a matrix with tiles from rows.

Parameters

- **matrix** (*numpy array 2D*) – Matrix to display.
- **tile_width** (*int*) – Tile width dimension.
- **tile_height** (*int*) – Tile height dimension.
- **num_tiles_x** (*int*) – Number of tiles horizontal.
- **num_tiles_y** (*int*) – Number of tiles vertical.
- **border_size** (*int*) – Size of the border.
- **normalized** (*bool*) – If true each image gets normalized to be between 0..1.

Returns Matrix showing the 2D patches.

Return type 2D numpy array

4.1.1.3.5.3 imshow_matrix

visualization.**imshow_matrix**(*windowtitle*, *interpolation='nearest'*)

Displays a matrix in gray-scale.

Parameters

- **matrix** (*numpy array*) – Data to display
- **windowtitle** (*string*) – Figure title
- **interpolation** (*string*) – Interpolation style

4.1.1.3.5.4 imshow_plot

visualization.**imshow_plot**(*windowtitle*)

Plots the colums of a matrix.

Parameters

- **matrix** (*numpy array*) – Data to plot
- **windowtitle** (*string*) – Figure title

4.1.1.3.5.5 imshow_histogram

visualization.**imshow_histogram**(*windowtitle*, *num_bins=10*, *normed=False*, *cumulative=False*, *log_scale=False*)

Shows a image of the histogram.

Parameters

- **matrix** (*numpy array 2D*) – Data to display
- **windowtitle** (*string*) – Figure title
- **num_bins** (*int*) – Number of bins
- **normed** (*bool*) – If true histogram is being normed to 0..1
- **cumulative** (*bool*) – Show cumulative histogram
- **log_scale** (*bool*) – Use logarithm Y-scaling

4.1.1.3.5.6 plot_2d_weights

visualization.**plot_2d_weights**(*bias=array([[0., 0.]])*, *scaling_factor=1.0*, *color='random'*, *bias_color='random'*)

Parameters

- **weights** (*numpy array [2,2]*) – Weight matrix (weights per column).
- **bias** (*numpy array [1,2]*) – Bias value.
- **scaling_factor** (*float*) – If not 1.0 the weights will be scaled by this factor.
- **color** (*string*) – Color for the weights.
- **bias_color** (*string*) – Color for the bias.

4.1.1.3.5.7 plot_2d_data

visualization.**plot_2d_data**(*alpha=0.1*, *color='navy'*, *point_size=5*)
Plots the data into the current figure.

Parameters

- **data** (*numpy array*) – Data matrix (Datapoint x dimensions).
- **alpha** (*float*) – ranspary value 0.0 = invisible, 1.0 = solid.
- **color** (*string (color name)*) – Color for the data points.
- **point_size** (*int*) – Size of the data points.

4.1.1.3.5.8 plot_2d_contour

visualization.**plot_2d_contour**(*value_range=[-5.0, 5.0, -5.0, 5.0]*, *step_size=0.01*, *levels=20*, *stylev=None*, *colormap='jet'*)
Plots the data into the current figure.

Parameters

- **probability_function** (*python method*) – Probability function must take 2D array [number of datapoint x 2]
- **value_range** (*list with four float entries*) – Min x, max x , min y, max y.
- **step_size** (*float*) – Step size for evaluating the pdf.
- **levels** (*int*) – Number of contour lines or array of contour height.
- **stylev** (*string or None*) – None as normal contour, 'filled' as filled contour, 'image' as contour image
- **colormap** (*string*) – Selected colormap .. seealso:: http://www.scipy.org/Cookbook/Matplotlib/…/Show_colormaps

4.1.1.3.5.9 imshow_standard_rbm_parameters

visualization.**imshow_standard_rbm_parameters**(*v1*, *v2*, *h1*, *h2*, *whitening=None*, *window_title=''*)
Saves the weights and biases of a given RBM at the given location.

Parameters

- **rbm** (*RBM object*) – RBM which weights and biases should be saved.
- **v1** (*int*) – Visible bias and the single weights will be saved as an image with size
- **v2** (*int*) – Visible bias and the single weights will be saved as an image with size
- **h1** (*int*) – Hidden bias and the image containing all weights will be saved as an image with size h1 x h2.
- **h2** (*int*) – Hidden bias and the image containing all weights will be saved as an image with size h1 x h2.
- **whitening** (*preprocessing object or None*) – If the data is PCA whitened it is useful to dewhiten the filters to wee the structure!

- **window_title** (*string*) – Title for this rbm.

4.1.1.3.5.10 hidden_activation

visualization.**hidden_activation**(*data*, *states=False*)
Calculates the hidden activation.

Parameters

- **rbm** (*RBM model object*) – RBM model object.
- **data** (*numpy array [num samples, dimensions]*) – Data for the activation calculation.
- **states** (*bool*) – If True uses states rather then probabilities by rounding to 0 or 1.

Returns hidden activation and the mean and standard deviation over the data.

Return type numpy array, float, floa

4.1.1.3.5.11 reorder_filter_by_hidden_activation

visualization.**reorder_filter_by_hidden_activation**(*data*)
Reorders the weights by its activation over the data set in decreasing order.

Parameters

- **rbm** (*RBM model object*) – RBM model object.
- **data** (*numpy array [num samples, dimensions]*) – Data for the activation calculation.

Returns RBM with reordered weights.

Return type RBM object.

4.1.1.3.5.12 generate_samples

visualization.**generate_samples**(*data*, *iterations*, *stepsize*, *v1*, *v2*, *sample_states=False*, *whitening=None*)
Generates samples from the given RBM model.

Parameters

- **rbm** (*RBM model object.*) – RBM model.
- **data** (*numpy array [num samples, dimensions]*) – Data to start sampling from.
- **iterations** (*int*) – Number of Gibbs sampling steps.
- **stepsize** (*int*) – After how many steps a sample should be plotted.
- **v1** (*int*) – X-Axis of the reorder image patch.
- **v2** (*int*) – Y-Axis of the reorder image patch.
- **sample_states** (*bool*) – If true returns the sates , probabilities otherwise.
- **whitening** (*preprocessing object or None*) – If the data has been preprocessed it needs to be undone.

Returns Matrix with image patches order along X-Axis and it's evolution in Y-Axis.

Return type numpy array

4.1.1.3.5.13 imshow_filter_tuning_curve

`visualization.`**`imshow_filter_tuning_curve`**(*num_of_ang=40*)

Plot the tuning curves of the filter's changes in frequency and angles.

Parameters

- **`filters`** (*`numpy array`*) – Filters to analyze.
- **`num_of_ang`** (*`int`*) – Number of orientations to check.

4.1.1.3.5.14 imshow_filter_optimal_gratings

`visualization.`**`imshow_filter_optimal_gratings`**(*opt_frq*, *opt_ang*)

Plot the filters and corresponding optimal gating pattern.

Parameters

- **`filters`** (*`numpy array`*) – Filters to analyze.
- **`opt_frq`** (*`int`*) – Optimal frequencies.
- **`opt_ang`** (*`int`*) – Optimal frequencies.

4.1.1.3.5.15 imshow_filter_frequency_angle_histogram

`visualization.`**`imshow_filter_frequency_angle_histogram`**(*opt_ang*, *max_wavelength=14*)

lots the histograms of the optimal frequencies and angles.

Parameters

- **`opt_frq`** (*`int`*) – Optimal frequencies.
- **`opt_ang`** (*`int`*) – Optimal angle.
- **`max_wavelength`** (*`int`*) – Maximal wavelength.

4.1.1.3.5.16 filter_frequency_and_angle

`visualization.`**`filter_frequency_and_angle`**(*num_of_angles=40*)

Analyze the filters by calculating the responses when gratings, i.e. sinusoidal functions, are input to them.

Info Hyv/"arinen, A. et al. (2009) Natural image statistics, Page 144-146

Parameters

- **`filters`** (*`numpy array`*) – Filters to analyze
- **`num_of_angles`** (*`int`*) – Number of angles steps to check

Returns The optimal frequency (pixels/cycle) of the filters, the optimal orientation angle (rad) of the filters

Return type numpy array, numpy array

4.1.1.3.5.17 filter_frequency_response

visualization.**filter_frequency_response**(*num_of_angles=40*)

Compute the response of filters w.r.t. different frequency.

Parameters

- **filters** (*numpy array*) – Filters to analyze
- **num_of_angles** (*int*) – Number of angles steps to check

Returns Frequency response as output_dim x max_wavelength-1 index of the

Return type numpy array, numpy array

4.1.1.3.5.18 filter_angle_response

visualization.**filter_angle_response**(*num_of_angles=40*)

Compute the angle response of the given filter.

Parameters

- **filters** (*numpy array*) – Filters to analyze
- **num_of_angles** (*int*) – Number of angles steps to check

Returns Angle response as output_dim x num_of_ang, index of angles

Return type numpy array, numpy array

4.1.1.3.5.19 calculate_amari_distance

visualization.**calculate_amari_distance**(*matrix_two*, *version=1*)

Calculate the Amari distance between two input matrices.

Parameters

- **matrix_one** (*numpy array*) – the first matrix
- **matrix_two** (*numpy array*) – the second matrix
- **version** (*int*) – Variant to use.

Returns The amari distance between two input matrices.

Return type float

4.1.1.4 preprocessing

This module contains several classes for data preprocessing.

Implemented

- Standarizer
- Principal Component Analysis (PCA)
- Zero Phase Component Analysis (ZCA)
- Independent Component Analysis (ICA)
- Binarize data

- Rescale data
- Remove row means
- Remove column means

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.4.1 binarize_data

`preprocessing.`**`binarize_data`**`()`

Converts data to binary values. For data out of [a,b] a data point p will become zero if p < 0.5*(b-a) one otherwise.

Parameters **`data`** (*`numpy array [num data point, data dimension]`*) – Data to be binarized.

Returns Binarized data.

Return type numpy array [num data point, data dimension]

4.1.1.4.2 rescale_data

`preprocessing.`**`rescale_data`**`(`*new_min=0.0*, *new_max=1.0*`)`

Normalize the values of a matrix. e.g. [min,max] -> [new_min,new_max]

Parameters

- **`data`** (*`numpy array [num data point, data dimension]`*) – Data to be normalized.
- **`new_min`** (*`float`*) – New min value.
- **`new_max`** (*`float`*) – Rescaled data

Returns

Return type numpy array [num data point, data dimension]

4.1.1.4.3 remove_rows_means

preprocessing.**remove_rows_means**(*return_means=False*)

Remove the individual mean of each row.

Parameters

- **data** (*numpy array [num data point, data dimension]*) – Data to be normalized
- **return_means** (*bool*) – If True returns also the means

Returns Data without row means, row means (optional).

Return type numpy array [num data point, data dimension], Means of the data (optional)

4.1.1.4.4 remove_cols_means

preprocessing.**remove_cols_means**(*return_means=False*)

Remove the individual mean of each column.

Parameters

- **data** (*numpy array [num data point, data dimension]*) – Data to be normalized
- **return_means** (*bool*) – If True returns also the means

Returns Data without column means, column means (optional).

Return type numpy array [num data point, data dimension], Means of the data (optional)

4.1.1.4.5 STANDARIZER

class pydeep.preprocessing.**STANDARIZER**(*input_dim*)

Shifts the data having zero mean and scales it having unit variances along the axis.

__init__(*input_dim*)

Constructor.

Parameters **input_dim** (*int*) – Data dimensionality.

project(*data*)

Projects the data to normalized space.

Parameters **data** (*numpy array [num data point, data dimension]*) – Data to project.

Returns Projected data.

Return type numpy array [num data point, data dimension]

train(*data*)

Training the model (full batch).

Parameters **data** (*numpy array [num data point, data dimension]*) – Data for training.

unproject(*data*)

Projects the data back to the input space.

Parameters data (*numpy array [num data point, data dimension]*) – Data to unproject.

Returns Projected data.

Return type numpy array [num data point, data dimension]

4.1.1.4.6 PCA

class pydeep.preprocessing.**PCA**(*input_dim*, *whiten=False*)
Principle component analysis (PCA) using Singular Value Decomposition (SVD)

__init__(*input_dim*, *whiten=False*)
Constructor.

Parameters

- **input_dim** (*int*) – Data dimensionality.
- **whiten** (*bool*) – If true the projected data will be de-correlated in all directions.

project(*data*, *num_components=None*)
Projects the data to Eigenspace.

Info projection_matrix has its projected vectors as its columns. i.e. if we project x by W into y where W is the projection_matrix, then y = W.T * x

Parameters

- **data** (*numpy array [num data point, data dimension]*) – Data to project.
- **num_components** (*int or None*) –

Returns Projected data.

Return type numpy array [num data point, data dimension]

train(*data*)
Training the model (full batch).

Parameters data (*numpy array [num data point, data dimension]*) – data for training.

unproject(*data*, *num_components=None*)
Projects the data from Eigenspace to normal space.

Parameters

- **data** (*numpy array [num data point, data dimension]*) – Data to be unprojected.
- **num_components** (*int*) – Number of components to project.

Returns Unprojected data.

Return type numpy array [num data point, num_components]

4.1.1.4.7 ZCA

class pydeep.preprocessing.**ZCA**(*input_dim*)
Principle component analysis (PCA) using Singular Value Decomposition (SVD).

__init__(*input_dim*)
Constructor.

Parameters **input_dim** (*int*) – Data dimensionality.

train(*data*)
Training the model (full batch).

Parameters **data** (*numpy array [num data point, data dimension]*) – data for training.

4.1.1.4.8 ICA

class pydeep.preprocessing.**ICA**(*input_dim*)
Independent Component Analysis using FastICA.

__init__(*input_dim*)
Constructor.

Parameters **input_dim** (*int*) – Data dimensionality.

log_likelihood(*data*)
Calculates the Log-Likelihood (LL) for the given data.

Parameters **data** (*numpy array [num data point, data dimension]*) – data to calculate the Log-Likelihood for.

Returns log-likelihood.

Return type numpy array [num data point]

train(*data*, *iterations=1000*, *convergence=0.0*, *status=False*)
Training the model (full batch).

Parameters

- **data** (*numpy array [num data point, data dimension]*) – data for training.
- **iterations** (*int*) – Number of iterations
- **convergence** (*double*) – If the angle (in degrees) between filters of two updates is less than the given value, training is terminated.
- **status** (*bool*) – If true the progress is printed to the console.

4.1.1.5 rbm

Package providing rbm models and corresponding sampler, trainer and estimator.

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

This file is part of the Python library PyDeep.

4.1.1.5.1 dbn

Helper class for deep believe networks.

Version 1.1.0

Date 06.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

4.1.1.5.1.1 DBN

class `pydeep.rbm.dbn.`**`DBN`**(*list_of_rbms*)

Deep believe network.

`__init__`(*list_of_rbms*)

Initializes the network with rbms.

Parameters **`list_of_rbms`** (*`list`*) – List of rbms.

`backward_propagate`(*output_data*, *sample=False*)

Propagates the output back through the input.

Parameters

- **`output_data`** (*`numpy array [batchsize x output dim]`*) – Output data.
- **`sample`** (*`bool`*) – If true the states are sampled, otherwise the probabilities are used.

Returns Input of the network.

Return type numpy array [batchsize x input dim]

forward_propagate(*input_data*, *sample=False*)
Propagates the data through the network.

Parameters

- **input_data** (*numpy array [batchsize x input dim]*) – Input data
- **sample** (*bool*) – If true the states are sampled, otherwise the probabilities are used.

Returns Output of the network.

Return type numpy array [batchsize x output dim]

reconstruct(*input_data*, *sample=False*)
Reconstructs the data by propagating the data to the output and back to the input.

Parameters

- **input_data** (*numpy array [batchsize x input dim]*) – Input data.
- **sample** (*bool*) – If true the states are sampled, otherwise the probabilities are used.

Returns Output of the network.

Return type numpy array [batchsize x output dim]

reconstruct_sample_top_layer(*input_data*, *sampling_steps=100*, *sample_forward_backward=False*)
Reconstructs data by propagating the data forward, sampling the top most layer and propagating the result backward.

Parameters

- **input_data** (*numpy array [batchsize x input dim]*) – Input data
- **sampling_steps** (*int*) – Number of Sampling steps.
- **sample_forward_backward** (*bool*) – If true the states for the forward and backward phase are sampled.

Returns reconstruction of the network.

Return type numpy array [batchsize x output dim]

sample_top_layer(*sampling_steps=100*, *initial_state=None*, *sample=True*)
Samples the top most layer, if initial_state is None the current state is used otherwise sampling is started from the given initial state

Parameters

- **sampling_steps** (*int*) – Number of Sampling steps.
- **initial_state** (*numpy array [batchsize x output dim]*) – Output data
- **sample** (*bool*) – If true the states are sampled, otherwise the probabilities are used (Mean field estimate).

Returns Output of the network.

Return type numpy array [batchsize x output dim]

4.1.1.5.2 estimator

This module provides methods for estimating the model performance (running on the CPU). Provided performance measures are for example the reconstruction error (RE) and the log-likelihood (LL). For estimating the LL we need to know the value of the partition function Z. If at least one layer is binary it is possible to calculate the value by factorizing over the binary values. Since it involves calculating all possible binary states, it is only possible for small models i.e. less than 25 (e.g. ~2^25 = 33554432 states). For bigger models we can estimate the partition function using annealed importance sampling (AIS).

Implemented

- kth order reconstruction error
- Log likelihood for visible data.
- Log likelihood for hidden data.
- True partition by factorization over the visible units.
- True partition by factorization over the hidden units.
- Annealed importance sampling to approximated the partition function.
- Reverse annealed importance sampling to approximated the partition function.

Info For the derivations .. seealso:: https://www.ini.rub.de/PEOPLE/wiskott/Reprints/Melchior-2012-MasterThesis-RBMs.pdf

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.5.2.1 reconstruction_error

`estimator.`**`reconstruction_error`**(*data*, *k=1*, *beta=None*, *use_states=False*, *absolut_error=False*)

This function calculates the reconstruction errors for a given model and data.

Parameters

- **model** (*Valid RBM model*) – The model.
- **data** (*numpy array [num samples, num dimensions] or numpy array [num batches, num samples in batch, num dimensions]*) – The data as 2D array or 3D array.

- **k** (*int*) – Number of Gibbs sampling steps.
- **beta** (*None, float or numpy array [batchsize,1]*) – Temperature(s) for the models energy.
- **use_states** (*bool*) – If false (default) the probabilities are used as reconstruction, if true states are sampled.
- **absolut_error** (*boll*) – If false (default) the squared error is used, the absolute error otherwise.

Returns Reconstruction errors of the data.

Return type nump array [num samples]

4.1.1.5.2.2 log_likelihood_v

estimator.**log_likelihood_v**(*logz*, *data*, *beta=None*)

Computes the log-likelihood (LL) for a given model and visible data given its log partition function.

Info logz needs to be the partition function for the same beta (i.e. beta = 1.0)!

Parameters

- **model** (*Valid RBM model.*) – The model.
- **logz** (*float*) – The logarithm of the partition function.
- **data** (*2D array [num samples, num input dim] or 3D type numpy array [num batches, num samples in batch, num input dim]*) – The visible data.
- **beta** (*None, float, numpy array [batchsize,1]*) – Inverse temperature(s) for the models energy.

Returns The log-likelihood for each sample.

Return type numpy array [num samples]

4.1.1.5.2.3 log_likelihood_h

estimator.**log_likelihood_h**(*logz*, *data*, *beta=None*)

Computes the log-likelihood (LL) for a given model and hidden data given its log partition function.

Info logz needs to be the partition function for the same beta (i.e. beta = 1.0)!

Parameters

- **model** (*Valid RBM model.*) – The model.
- **logz** (*float*) – The logarithm of the partition function.
- **data** (*2D array [num samples, num output dim] or 3D type numpy array [num batches, num samples in batch, num output dim]*) – The hidden data.
- **beta** (*None, float, numpy array [batchsize,1]*) – Inverse temperature(s) for the models energy.

Returns The log-likelihood for each sample.

Return type numpy array [num samples]

4.1.1.5.2.4 partition_function_factorize_v

estimator.**partition_function_factorize_v**(*beta=None*, *batchsize_exponent='AUTO'*, *status=False*)

Computes the true partition function for the given model by factoring over the visible units.

Info Exponential increase of computations by the number of visible units. (16 usually ~ 20 seconds)

Parameters

- **model** (*Valid RBM model.*) – The model.
- **beta** (*None, float, numpy array [batchsize,1]*) – Inverse temperature(s) for the models energy.
- **batchsize_exponent** (*int*) – 2^batchsize_exponent will be the batch size.
- **status** (*bool*) – If true prints the progress to the console.

Returns Log Partition function for the model.

Return type float

4.1.1.5.2.5 partition_function_factorize_h

estimator.**partition_function_factorize_h**(*beta=None*, *batchsize_exponent='AUTO'*, *status=False*)

Computes the true partition function for the given model by factoring over the hidden units.

Info Exponential increase of computations by the number of visible units. (16 usually ~ 20 seconds)

Parameters

- **model** (*Valid RBM model.*) – The model.
- **beta** (*None, float, numpy array [batchsize,1]*) – Inverse temperature(s) for the models energy.
- **batchsize_exponent** (*int*) – 2^batchsize_exponent will be the batch size.
- **status** (*bool*) – If true prints the progress to the console.

Returns Log Partition function for the model.

Return type float

4.1.1.5.2.6 annealed_importance_sampling

estimator.**annealed_importance_sampling**(*num_chains=100*, *k=1*, *betas=10000*, *status=False*)

Approximates the partition function for the given model using annealed importance sampling.

See also:

Accurate and Conservative Estimates of MRF Log-likelihood using Reverse Annealing http://arxiv.org/pdf/1412.8566.pdf

Parameters

- **model** (*Valid RBM model.*) – The model.

- **num_chains** (*int*) – Number of AIS runs.
- **k** (*int*) – Number of Gibbs sampling steps.
- **betas** (*int, numpy array [num_betas]*) – Number or a list of inverse temperatures to sample from.
- **status** (*bool*) – If true prints the progress on console.

Returns

Mean estimated log partition function,
Mean +3std estimated log partition function,
Mean -3std estimated log partition function.

Return type float

4.1.1.5.2.7 reverse_annealed_importance_sampling

estimator.**reverse_annealed_importance_sampling**(*num_chains=100*, *k=1*, *betas=10000*, *status=False*, *data=None*)

Approximates the partition function for the given model using reverse annealed importance sampling.

See also:

Accurate and Conservative Estimates of MRF Log-likelihood using Reverse Annealing http://arxiv.org/pdf/1412.8566.pdf

Parameters

- **model** (*Valid RBM model.*) – The model.
- **num_chains** (*int*) – Number of AIS runs.
- **k** (*int*) – Number of Gibbs sampling steps.
- **betas** (*int, numpy array [num_betas]*) – Number or a list of inverse temperatures to sample from.
- **status** (*bool*) – If true prints the progress on console.
- **data** (*numpy array*) – If data is given, initial sampling is started from data samples.

Returns

Mean estimated log partition function,
Mean +3std estimated log partition function,
Mean -3std estimated log partition function.

Return type float

4.1.1.5.3 model

This module provides restricted Boltzmann machines (RBMs) with different types of units. The structure is very close to the mathematical derivations to simplify the understanding. In addition, the modularity helps to create other kind of RBMs without adapting the training algorithms.

Implemented

- centered BinaryBinary RBM (BB-RBM)
- centered GaussianBinary RBM (GB-RBM) with fixed variance
- centered GaussianBinaryVariance RBM (GB-RBM) with trainable variance

Models without implementation of p(v),p(h),p(v,h) -> AIS, PT, true gradient, . . . cannot be used! - centered BinaryBinaryLabel RBM (BBL-RBM) - centered GaussianBinaryLabel RBM (GBL-RBM)

Models with intractable p(v),p(h),p(v,h) -> AIS, PT, true gradient, . . . cannot be used! - centered BinaryRect RBM (BR-RBM) - centered RectBinary RBM (RB-RBM) - centered RectRect RBM (RR-RBM) - centered GaussianRect RBM (GR-RBM) - centered GaussianRectVariance RBM (GRV-RBM)

Info For the derivations .. seealso:: https://www.ini.rub.de/PEOPLE/wiskott/Reprints/Melchior-2012-MasterThesis-RBMs.pdf

A usual way to create a new unit is to inherit from a given RBM class and override the functions that changed, e.g. Gaussian-Binary RBM inherited from the Binary-Binary RBM.

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.5.3.1 BinaryBinaryRBM

class `pydeep.rbm.model.`**BinaryBinaryRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered restricted Boltzmann machine with binary visible and binary hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_visible_units(*num_new_visibles*, *position=0*, *initial_weights='AUTO'*, *initial_bias='AUTO'*, *initial_offsets='AUTO'*, *data=None*)

This function adds new visible units at the given position to the model. .. Warning:: If the parameters are changed.
reinitialized.

Parameters

- **num_new_visibles** (*int*) – The number of new hidden units to add
- **position** (*int*) – Position where the units should be added.
- **initial_weights** (*'AUTO', scalar or numpy array [input num_new_visibles, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_bias** (*'AUTO' or scalar or numpy array [1, num_new_visibles]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_offsets** (*'AUTO' or scalar or numpy array [1, num_new_visibles]*) – The initial visible offset values.

- **data** (*numpy array [num datapoints, num_new_visibles]*) – If data is given and the offset and bias is initzialized accordingly, if 'AUTO' is chosen.

_base_log_partition(*use_base_model=False*)
Returns the base partition function for a given visible bias. .. Note:: that for AIS we need to be able to calculate the partition function of the base distribution exactly. Furthermore it is beneficial if the base distribution is a good approximation of the target distribution. A good choice is therefore the maximum likelihood estimate of the visible bias, given the data.

Parameters **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Partition function for zero parameters.

Return type float

_calculate_hidden_bias_gradient(*h*)
This function calculates the gradient for the hidden biases.

Parameters **h** (*numpy arrays [batch size, output dim]*) – Hidden activations.

Returns Hidden bias gradient.

Return type numpy arrays [1, output dim]

_calculate_visible_bias_gradient(*v*)
This function calculates the gradient for the visible biases.

Parameters **v** (*numpy arrays [batch_size, input dim]*) – Visible activations.

Returns Visible bias gradient.

Return type numpy arrays [1, input dim]

_calculate_weight_gradient(*v, h*)
This function calculates the gradient for the weights from the visible and hidden activations.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – Visible activations.
- **h** (*numpy arrays [batchsize, output dim]*) – Hidden activations.

Returns Weight gradient.

Return type numpy arrays [input dim, output dim]

_getbasebias()
Returns the maximum likelihood estimate of the visible bias, given the data. If no data is given the RBMs bias value is return, but is highly recommended to pass the data.

Returns Base bias.

Return type numpy array [1, input dim]

_remove_visible_units(*indices*)

This function removes the visible units whose indices are given.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters **indices** (*int or list of int or numpy array of int*) – Indices of units to be remove.

calculate_gradients(*v*, *h*)

This function calculates all gradients of this RBM and returns them as a list of arrays. This keeps the flexibility of adding parameters which will be updated by the training algorithms.

Parameters

- **v** (*numpy arrays [batch size, output dim]*) – Visible activations.
- **h** (*numpy arrays [batch size, output dim]*) – Hidden activations.

Returns Gradients for all parameters.

Return type list of numpy arrays (num parameters x [parameter.shape])

energy(*v*, *h*, *beta=None*, *use_base_model=False*)

Compute the energy of the RBM given observed variable states v and hidden variables state h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Energy of v and h.

Return type numpy array [batch size,1]

log_probability_h(*logz*, *h*, *beta=None*, *use_base_model=False*)

Computes the log-probability / LogLikelihood(LL) for the given hidden units for this model. To estimate the LL we need to know the logarithm of the partition function Z. For small models it is possible to calculate Z, however since this involves calculating all possible hidden states, it is intractable for bigger models. As an estimation method annealed importance sampling (AIS) can be used instead.

Parameters

- **logz** (*float*) – The logarithm of the partition function.
- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Log probability for hidden_states.

Return type numpy array [batch size, 1]

log_probability_v(*logz*, *v*, *beta=None*, *use_base_model=False*)

Computes the log-probability / LogLikelihood(LL) for the given visible units for this model. To estimate the LL we need to know the logarithm of the partition function Z. For small models it is possible to calculate Z, however since this involves calculating all possible hidden states, it is intractable for bigger models. As an estimation method annealed importance sampling (AIS) can be used instead.

Parameters

- **logz** (*float*) – The logarithm of the partition function.
- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.None is equivalent to pass the value 1.0.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Log probability for visible_states.

Return type numpy array [batch size, 1]

log_probability_v_h(*logz*, *v*, *h*, *beta=None*, *use_base_model=False*)

Computes the joint log-probability / LogLikelihood(LL) for the given visible and hidden units for this model. To estimate the LL we need to know the logarithm of the partition function Z. For small models it is possible to calculate Z, however since this involves calculating all possible hidden states, it is intractable for bigger models. As an estimation method annealed importance sampling (AIS) can be used instead.

Parameters

- **logz** (*float*) – The logarithm of the partition function.
- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Joint log probability for v and h.

Return type numpy array [batch size, 1]

probability_h_given_v(*v*, *beta=None*, *use_base_model=False*)

Calculates the conditional probabilities of h given v.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – DUMMY variable, since we do not use a base hidden bias.

Returns Conditional probabilities h given v.

Return type numpy array [batch size, output dim]

probability_v_given_h(*h*, *beta=None*, *use_base_model=False*)

Calculates the conditional probabilities of v given h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.

- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Conditional probabilities v given h.

Return type numpy array [batch size, input d

sample_h (*h*, *beta=None*, *use_base_model=False*)
Samples the hidden variables from the conditional probabilities h given v.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Conditional probabilities of h given v.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for h.

Return type numpy array [batch size, output dim]

sample_v (*v*, *beta=None*, *use_base_model=False*)
Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

unnormalized_log_probability_h (*h*, *beta=None*, *use_base_model=False*)
Computes the unnormalized log probabilities of h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.None is equivalent to pass the value 1.0.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Unnormalized log probability of h.

Return type numpy array [batch size, 1]

unnormalized_log_probability_v(*v*, *beta=None*, *use_base_model=False*)
Computes the unnormalized log probabilities of v.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.None is equivalent to pass the value 1.0.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Unnormalized log probability of v.

Return type numpy array [batch size, 1]

4.1.1.5.3.2 GaussianBinaryRBM

class pydeep.rbm.model.**GaussianBinaryRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)
Implementation of a centered Restricted Boltzmann machine with Gaussian visible and binary hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)
This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.

- **initial_sigma** (*'AUTO', scalar or numpy array [1, input_dim]*) – Initial standard deviation for the model.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_hidden_units(*num_new_hiddens*, *position=0*, *initial_weights='AUTO'*, *initial_bias='AUTO'*, *initial_offsets='AUTO'*)

This function adds new hidden units at the given position to the model.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters

- **num_new_hiddens** (*int*) – The number of new hidden units to add.
- **position** (*int*) – Position where the units should be added.
- **initial_weights** (*'AUTO' or scalar or numpy array [input_dim, num_new_hiddens]*) – The initial weight values for the hidden units.
- **initial_bias** (*'AUTO' or scalar or numpy array [1, num_new_hiddens]*) – The initial hidden bias values.
- **initial_offsets** (*'AUTO' or scalar or numpy array [1, num_new_hidden*) – he initial hidden mean values.

_add_visible_units(*num_new_visibles*, *position=0*, *initial_weights='AUTO'*, *initial_bias='AUTO'*, *initial_sigmas=1.0*, *initial_offsets='AUTO'*, *data=None*)

This function adds new visible units at the given position to the model.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters

- **num_new_visibles** (*int*) – The number of new hidden units to add
- **position** (*int*) – Position where the units should be added.
- **initial_weights** (*'AUTO', scalar or numpy array [input num_new_visibles, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_bias** (*'AUTO' or scalar or numpy array [1, num_new_visibles]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.

- **initial_sigmas** (*'AUTO' or scalar or numpy array [1, num_new_visibles]*) – The initial standard deviation for the model.
- **initial_offsets** (*'AUTO' or scalar or numpy array [1, num_new_visibles]*) – The initial visible offset values.
- **data** (*numpy array [num datapoints, num_new_visibles]*) – If data is given and the offset and bias is initzialized accordingly, if 'AUTO' is chosen.

_base_log_partition (*use_base_model=False*)
Returns the base partition function which needs to be calculateable.

Parameters **use_base_model** (*bool*) – DUMMY sicne the integral does not change if the mean is shifted.

Returns Partition function for zero parameters.

Return type float

_calculate_visible_bias_gradient (*v*)
This function calculates the gradient for the visible biases.

Parameters **v** (*numpy arrays [batch_size, input dim]*) – Visible activations.

Returns Visible bias gradient.

Return type numpy arrays [1, input dim]

_calculate_weight_gradient (*v, h*)
This function calculates the gradient for the weights from the visible and hidden activations.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – Visible activations.
- **h** (*numpy arrays [batchsize, output dim]*) – Hidden activations.

Returns Weight gradient.

Return type numpy arrays [input dim, output dim]

_remove_visible_units (*indices*)

This function removes the visible units whose indices are given.

> **Warning:** If the parameters are changed. the trainer needs to be reinitialized.

Parameters **indices** (*int or list of int or numpy array of int*) – Indices of units to be remove.

energy (*v, h, beta=None, use_base_model=False*)
Compute the energy of the RBM given observed variable states v and hidden variables state h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0

- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Energy of v and h.

Return type numpy array [batch size,1]

probability_h_given_v(*v*, *beta=None*, *use_base_model=False*)
Calculates the conditional probabilities h given v.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states / data.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Conditional probabilities h given v.

Return type numpy array [batch size, output dim]

probability_v_given_h(*h*, *beta=None*, *use_base_model=False*)
Calculates the conditional probabilities of v given h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Conditional probabilities v given h.

Return type numpy array [batch size, input dim]

sample_v(*v*, *beta=None*, *use_base_model=False*)
Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

unnormalized_log_probability_h(*h*, *beta=None*, *use_base_model=False*)
Computes the unnormalized log probabilities of h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.

- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.None is equivalent to pass the value 1.0.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Unnormalized log probability of h.

Return type numpy array [batch size, 1]

unnormalized_log_probability_v(*v*, *beta=None*, *use_base_model=False*)

Computes the unnormalized log probabilities of v. ln(z*p(v)) = ln(p(v))-ln(z)+ln(z) = ln(p(v)).

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.None is equivalent to pass the value 1.0.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Unnormalized log probability of v.

Return type numpy array [batch size, 1]

4.1.1.5.3.3 GaussianBinaryVarianceRBM

class pydeep.rbm.model.**GaussianBinaryVarianceRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets=0.0*, *initial_hidden_offsets=0.0*, *dtype=<type 'numpy.float64'>*)

Implementation of a Restricted Boltzmann machine with Gaussian visible having trainable variances and binary hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets=0.0*, *initial_hidden_offsets=0.0*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.

- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_sigma** (*'AUTO', scalar or numpy array [1, input_dim]*) – Initial standard deviation for the model.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_calculate_sigma_gradient(*v*, *h*)

This function calculates the gradient for the variance of the RBM.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – States of the visible variables.
- **h** (*numpy arrays [batchsize, output dim]*) – Probs/States of the hidden variables.

Returns Sigma gradient.

Return type list of numpy arrays [input dim,1]

calculate_gradients(*v*, *h*)

his function calculates all gradients of this RBM and returns them as an ordered array. This keeps the flexibility of adding parameters which will be updated by the training algorithms.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – States of the visible variables.
- **h** (*numpy arrays [batchsize, output dim]*) – Probabilities of the hidden variables.

Returns Gradients for all parameters.

Return type numpy arrays (num parameters x [parameter.shape])

get_parameters()

This function returns all mordel parameters in a list.

Returns The parameter references in a list.

Return type list

4.1.1.5.3.4 BinaryBinaryLabelRBM

class pydeep.rbm.model.**BinaryBinaryLabelRBM**(*number_visibles*, *number_labels*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered Restricted Boltzmann machine with Binary visible plus Softmax label units and binary hidden units.

__init__(*number_visibles*, *number_labels*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_labels** (*int*) – Number of the label variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_visible_units()

Not available!

_base_log_partition()

Not available!

_remove_visible_units()
Not available!

energy()
Not available!

log_probability_h()
Not available!

log_probability_v()
Not available!

log_probability_v_h()
Not available!

sample_v(*v*, *beta=None*, *use_base_model=False*)
Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (`numpy array [batch size, input dim]`) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

unnormalized_log_probability_h()
Not available!

unnormalized_log_probability_v()
Not available!

4.1.1.5.3.5 SoftMaxSigmoid

4.1.1.5.3.6 GaussianBinaryLabelRBM

class pydeep.rbm.model.**GaussianBinaryLabelRBM**(*number_visibles*, *number_labels*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)
Implementation of a centered Restricted Boltzmann machine with Gaussian visible plus Softmax label units and binary hidden units.

__init__(*number_visibles*, *number_labels*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)
This function initializes all necessary parameters and data structures. It is recommended to pass the

training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_labels** (*int*) – Number of the label variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_sigma** (*'AUTO', scalar or numpy array [1, input_dim]*) – Initial standard deviation for the model.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_visible_units()
Not available!

_base_log_partition()
Not available!

_remove_visible_units()
Not available!

energy()
Not available!

log_probability_h()
Not available!

log_probability_v()
Not available!

log_probability_v_h()
Not available!

sample_v(*v*, *beta=None*, *use_base_model=False*)
Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

unnormalized_log_probability_h()
Not available!

unnormalized_log_probability_v()
Not available!

4.1.1.5.3.7 SoftMaxLinear

4.1.1.5.3.8 BinaryRectRBM

class pydeep.rbm.model.**BinaryRectRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered Restricted Boltzmann machine with Binary visible and Noisy linear rectified hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random,

'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.

- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_visible_units()
Not available!

_base_log_partition()
Not available!

_remove_visible_units()
Not available!

energy()
Not available!

log_probability_h()
Not available!

log_probability_v()
Not available!

log_probability_v_h()
Not available!

probability_h_given_v(*v*, *beta=None*)
Calculates the conditional probabilities h given v.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states / data.
- **beta** (*float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.

Returns Conditional probabilities h given v.

Return type numpy array [batch size, output dim]

sample_h(*h*, *beta=None*, *use_base_model=False*)
Samples the hidden variables from the conditional probabilities h given v.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Conditional probabilities of h given v.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for h.

Return type numpy array [batch size, output dim]

unnormalized_log_probability_h()
Not available!

unnormalized_log_probability_v()
Not available!

4.1.1.5.3.9 RectBinaryRBM

class pydeep.rbm.model.**RectBinaryRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered Restricted Boltzmann machine with Noisy linear rectified visible units and binary hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

_add_visible_units()
: Not available!

_base_log_partition()
: Not available!

_getbasebias()
: Not available!

_remove_visible_units()
: Not available!

energy()
: Not available!

log_probability_h()
: Not available!

log_probability_v()
: Not available!

log_probability_v_h()
: Not available!

probability_v_given_h(*h*, *beta=None*, *use_base_model=False*)
: Calculates the conditional probabilities of v given h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Conditional probabilities v given h.

Return type numpy array [batch size, input d

sample_v(*v*, *beta=None*, *use_base_model=False*)
: Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable. The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

unnormalized_log_probability_h()
: Not available!

unnormalized_log_probability_v()
: Not available!

4.1.1.5.3.10 RectRectRBM

class pydeep.rbm.model.**RectRectRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered Restricted Boltzmann machine with Noisy linear rectified visible and hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. It is recommended to pass the training data to initialize the network automatically.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of hidden variables.
- **data** (*None or numpy array [num samples, input dim]*) – The training data for parameter initialization if 'AUTO' is chosen for the corresponding parameter.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights. 'AUTO' and a scalar are random init.
- **initial_visible_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, input dim]*) – Initial visible bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the visilbe mean. If a scalar is passed all values are initialized with it.
- **initial_hidden_bias** (*'AUTO','INVERSE_SIGMOID', scalar or numpy array [1, output_dim]*) – Initial hidden bias. 'AUTO' is random, 'INVERSE_SIGMOID' is the inverse Sigmoid of the hidden mean. If a scalar is passed all values are initialized with it.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible offset values. AUTO=data mean or 0.5 if no data is given. If a scalar is passed all values are initialized with it.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden offset values. AUTO = 0.5 If a scalar is passed all values are initialized with it.
- **dtype** (*numpy.float32 or numpy.float64 or numpy.longdouble*) – Used data type i.e. numpy.float64

probability_v_given_h(*h*, *beta=None*, *use_base_model=False*)

Calculates the conditional probabilities of v given h.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Hidden states.
- **beta** (*None, float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously. None is equivalent to pass the value 1.0

- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values.

Returns Conditional probabilities v given h.

Return type numpy array [batch size, input d

sample_v(*v*, *beta=None*, *use_base_model=False*)

Samples the visible variables from the conditional probabilities v given h.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Conditional probabilities of v given h.
- **beta** (*None*) – DUMMY Variable The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for v.

Return type numpy array [batch size, input dim]

4.1.1.5.3.11 GaussianRectRBM

class pydeep.rbm.model.**GaussianRectRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

Implementation of a centered Restricted Boltzmann machine with Gaussian visible and Noisy linear rectified hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets='AUTO'*, *initial_hidden_offsets='AUTO'*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. See comments for automatically chosen values.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of the hidden variables.
- **data** (*None or numpy array [num samples, input dim] or List of numpy arrays [num samples, input dim]*) – The training data for initializing the visible bias.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights.
- **initial_visible_bias** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible bias.
- **initial_hidden_bias** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden bias.

- **initial_sigma** (*'AUTO', scalar or numpy array [1, input_dim]*) – Initial standard deviation for the model.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible mean values.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden mean values.
- **dtype** (*numpy.float32, numpy.float64 and, numpy.longdouble*) – Used data type.

_add_visible_units()
Not available!

_base_log_partition()
Not available!

_remove_visible_units()
Not available!

energy()
Not available!

log_probability_h()
Not available!

log_probability_v()
Not available!

log_probability_v_h()
Not available!

probability_h_given_v(*v*, *beta=None*)
Calculates the conditional probabilities h given v.

Parameters

- **v** (*numpy array [batch size, input dim]*) – Visible states / data.
- **beta** (*float or numpy array [batch size, 1]*) – Allows to sample from a given inverse temperature beta, or if a vector is given to sample from different betas simultaneously.

Returns Conditional probabilities h given v.

Return type numpy array [batch size, output dim]

sample_h(*h*, *beta=None*, *use_base_model=False*)
Samples the hidden variables from the conditional probabilities h given v.

Parameters

- **h** (*numpy array [batch size, output dim]*) – Conditional probabilities of h given v.
- **beta** (*None*) – DUMMY Variable The sampling in other types of units like Gaussian-Binary RBMs will be affected by beta.
- **use_base_model** (*bool*) – If true uses the base model, i.e. the MLE of the bias values. (DUMMY in this case)

Returns States for h.

Return type numpy array [batch size, output dim]

unnormalized_log_probability_h()
Not available!

unnormalized_log_probability_v()
Not available!

4.1.1.5.3.12 GaussianRectVarianceRBM

class pydeep.rbm.model.**GaussianRectVarianceRBM**(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets=0.0*, *initial_hidden_offsets=0.0*, *dtype=<type 'numpy.float64'>*)

Implementation of a Restricted Boltzmann machine with Gaussian visible having trainable variances and noisy rectified hidden units.

__init__(*number_visibles*, *number_hiddens*, *data=None*, *initial_weights='AUTO'*, *initial_visible_bias='AUTO'*, *initial_hidden_bias='AUTO'*, *initial_sigma='AUTO'*, *initial_visible_offsets=0.0*, *initial_hidden_offsets=0.0*, *dtype=<type 'numpy.float64'>*)

This function initializes all necessary parameters and data structures. See comments for automatically chosen values.

Parameters

- **number_visibles** (*int*) – Number of the visible variables.
- **number_hiddens** (*int*) – Number of the hidden variables.
- **data** (*None or numpy array [num samples, input dim] or List of numpy arrays [num samples, input dim]*) – The training data for initializing the visible bias.
- **initial_weights** (*'AUTO', scalar or numpy array [input dim, output_dim]*) – Initial weights.
- **initial_visible_bias** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible bias.
- **initial_hidden_bias** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden bias.
- **initial_sigma** (*'AUTO', scalar or numpy array [1, input_dim]*) – Initial standard deviation for the model.
- **initial_visible_offsets** (*'AUTO', scalar or numpy array [1, input dim]*) – Initial visible mean values.
- **initial_hidden_offsets** (*'AUTO', scalar or numpy array [1, output_dim]*) – Initial hidden mean values.
- **dtype** (*numpy.float32, numpy.float64 and, numpy.longdouble*) – Used data type.

_calculate_sigma_gradient(*v*, *h*)

This function calculates the gradient for the variance of the RBM.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – States of the visible variables.
- **h** (*numpy arrays [batchsize, output dim]*) – Probabilities of the hidden variables.

Returns Sigma gradient.

Return type list of numpy arrays [input dim,1]

calculate_gradients(*v*, *h*)

This function calculates all gradients of this RBM and returns them as an ordered array. This keeps the flexibility of adding parameters which will be updated by the training algorithms.

Parameters

- **v** (*numpy arrays [batchsize, input dim]*) – States of the visible variables.
- **h** (*numpy arrays [batchsize, output dim]*) – Probabilities of the hidden variables.

Returns Gradients for all parameters.

Return type numpy arrays (num parameters x [parameter.shape])

get_parameters()

This function returns all model parameters in a list.

Returns The parameter references in a list.

Return type list

4.1.1.5.4 sampler

This module provides different sampling algorithms for RBMs running on CPU. The structure is kept modular to simplify the understanding of the code and the mathematics. In addition the modularity helps to create other kind of sampling algorithms by inheritance.

Implemented

- Gibbs Sampling
- Persistent Gibbs Sampling
- Parallel Tempering Sampling
- Independent Parallel Tempering Sampling

Info For the derivations .. seealso:: https://www.ini.rub.de/PEOPLE/wiskott/Reprints/Melchior-2012-MasterThesis-RBMs.pdf

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

License Copyright (C) 2017 Jan Melchior

This file is part of the Python library PyDeep.

4.1.1.5.4.1 GibbsSampler

class pydeep.rbm.sampler.**GibbsSampler**(*model*)

Implementation of k-step Gibbs-sampling for bipartite graphs.

__init__(*model*)

Initializes the sampler with the model.

Parameters **model** (*Valid model class like BinaryBinary-RBM.*) – The model to sample from.

sample(*vis_states*, *k=1*, *betas=None*, *ret_states=True*)

Performs k steps Gibbs-sampling starting from given visible data.

Parameters

- **vis_states** (*numpy array [num samples, input dimension]*) – The initial visible states to sample from.
- **k** (*int*) – The number of Gibbs sampling steps.
- **betas** (*None, float, numpy array [num_betas,1]*) – Inverse temperature to sample from.(energy based models)
- **ret_states** (*bool*) – If False returns the visible probabilities instead of the states.

Returns The visible samples of the Markov chains.

Return type numpy array [num samples, input dimension]

sample_from_h(*hid_states*, *k=1*, *betas=None*, *ret_states=True*)

Performs k steps Gibbs-sampling starting from given hidden states.

Parameters

- **hid_states** (*numpy array [num samples, output dimension]*) – The initial hidden states to sample from.
- **k** (*int*) – The number of Gibbs sampling steps.
- **betas** (*(energy based models)*) – Inverse temperature to sample from.
- **ret_states** (*bool*) – If False returns the visible probabilities instead of the states.

Returns The visible samples of the Markov chains.

Return type numpy array [num samples, input dimension]

4.1.1.5.4.2 PersistentGibbsSampler

class pydeep.rbm.sampler.**PersistentGibbsSampler**(*model*, *num_chains*)
Implementation of k-step persistent Gibbs sampling.

__init__(*model*, *num_chains*)
Initializes the sampler with the model.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **num_chains** (*int*) – The number of Markov chains. .. Note:: Optimal performance is achieved if the number of samples and the number of chains equal the batch_size.

sample(*num_samples*, *k=1*, *betas=None*, *ret_states=True*)
Performs k steps persistent Gibbs-sampling.

Parameters

- **num_samples** (*int, numpy array*) – The number of samples to generate. .. Note:: Optimal performance is achieved if the number of samples and the number of chains equal the batch_size.
- **k** (*int*) – The number of Gibbs sampling steps.
- **betas** (*None, float, numpy array [num_betas,1]*) – Inverse temperature to sample from.(energy based models)
- **ret_states** (*bool*) – If False returns the visible probabilities instead of the states.

Returns The visible samples of the Markov chains.

Return type numpy array [num samples, input dimension]

4.1.1.5.4.3 ParallelTemperingSampler

class pydeep.rbm.sampler.**ParallelTemperingSampler**(*model*, *num_chains=3*, *betas=None*)
Implementation of k-step parallel tempering sampling.

__init__(*model*, *num_chains=3*, *betas=None*)
Initializes the sampler with the model.

Parameters

- **model** (*Valid model Class.*) – The model to sample from.
- **num_chains** (*int*) – The number of Markov chains.
- **betas** (*int, None*) – Array of inverse temperatures to sample from, its dimensionality needs to equal the number of chains or if None is given the inverse temperatures are initialized linearly from 0.0 to 1.0 in 'num_chains' steps.

classmethod _swap_chains(*chains*, *hid_states*, *model*, *betas*)
Swaps the samples between the Markov chains according to the Metropolis Hastings Ratio.

Parameters

- **chains** (*[num samples, input dimension]*) – Chains with visible data.
- **hid_states** (*[num samples, output dimension]*) – Hidden states.

- **model** (*Valid RBM Class.*) – The model to sample from.
- **betas** (*int, None*) – Array of inverse temperatures to sample from, its dimensionality needs to equal the number of chains or if None is given the inverse temperatures are initialized linearly from 0.0 to 1.0 in 'num_chains' steps.

sample (*num_samples, k=1, ret_states=True*)
Performs k steps parallel tempering sampling.

Parameters

- **num_samples** (*int, numpy array*) – The number of samples to generate. .. Note:: Optimal performance is achieved if the number of samples and the number of chains equal the batch_size.
- **k** (*int*) – The number of Gibbs sampling steps.
- **ret_states** (*bool*) – If False returns the visible probabilities instead of the states.

Returns The visible samples of the Markov chains.

Return type numpy array [num samples, input dimension]

4.1.1.5.4.4 IndependentParallelTemperingSampler

class pydeep.rbm.sampler.**IndependentParallelTemperingSampler** (*model, num_samples, num_chains=3, betas=None*)

Implementation of k-step independent parallel tempering sampling. IPT runs an PT instance for each sample in parallel. This speeds up the sampling but also decreases the mixing rate.

__init__ (*model, num_samples, num_chains=3, betas=None*)
Initializes the sampler with the model.

Parameters

- **model** (*Valid model Class.*) – The model to sample from.
- **num_samples** – The number of samples to generate. .. Note:: Optimal performance (ATLAS,MKL) is achieved if the number of samples equals the batchsize.
- **num_chains** (*int*) – The number of Markov chains.
- **betas** (*int, None*) – Array of inverse temperatures to sample from, its dimensionality needs to equal the number of chains or if None is given the inverse temperatures are initialized linearly from 0.0 to 1.0 in 'num_chains' steps.

classmethod _swap_chains (*chains, num_chains, hid_states, model, betas*)
Swaps the samples between the Markov chains according to the Metropolis Hastings Ratio.

Parameters

- **chains** (*[num samples*num_chains, input dimension]*) – Chains with visible data.
- **hid_states** (*[num samples*num_chains, output dimension]*) – Hidden states.
- **model** (*Valid RBM Class.*) – The model to sample from.

- **betas** (*int, None*) – Array of inverse temperatures to sample from, its dimensionality needs to equal the number of chains or if None is given the inverse temperatures are initialized linearly from 0.0 to 1.0 in 'num_chains' steps.

sample (*num_samples='AUTO', k=1, ret_states=True*)

Performs k steps independent parallel tempering sampling.

Parameters

- **num_samples** (*int or 'AUTO'*) – The number of samples to generate. .. Note:: Optimal performance is achieved if the number of samples and the number of chains equal the batch_size. -> AUTO
- **k** (*int*) – The number of Gibbs sampling steps.
- **ret_states** (*bool*) – If False returns the visible probabilities instead of the states.

Returns The visible samples of the Markov chains.

Return type numpy array [num samples, input dimension]

4.1.1.5.5 trainer

This module provides different types of training algorithms for RBMs running on CPU. The structure is kept modular to simplify the understanding of the code and the mathematics. In addition the modularity helps to create other kind of training algorithms by inheritance.

Implemented

- CD (Contrastive Divergence)
- PCD (Persistent Contrastive Divergence)
- PT (Parallel Tempering)
- IPT (Independent Parallel Tempering)
- GD (Exact Gradient descent (only for small binary models))

Info For the derivations .. seealso:: https://www.ini.rub.de/PEOPLE/wiskott/Reprints/Melchior-2012-MasterThesis-RBMs.pdf

Version 1.1.0

Date 04.04.2017

Author Jan Melchior

Contact JanMelchior@gmx.de

4.1.1.5.5.1 CD

class pydeep.rbm.trainer.**CD**(*model*, *data=None*)

Implementation of the training algorithm Contrastive Divergence (CD).

INFO A fast learning algorithm for deep belief nets, Geoffrey E. Hinton and Simon Osindero Yee-Whye Teh Department of Computer Science University of Toronto Yee-Whye Teh 10 Kings College Road National University of Singapore.

__init__(*model*, *data=None*)

The constructor initializes the CD trainer with a given model and data.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **data** (*numpy array [num. samples x input dim]*) – Data for initialization, only has effect if the centered gradient is used.

_adapt_gradient(*pos_gradients*, *neg_gradients*, *batch_size*, *epsilon*, *momentum*, *reg_l1norm*, *reg_l2norm*, *reg_sparseness*, *desired_sparseness*, *mean_hidden_activity*, *visible_offsets*, *hidden_offsets*, *use_centered_gradient*, *restrict_gradient*, *restriction_norm*)

This function updates the parameter gradients.

Parameters

- **pos_gradients** (*numpy array[parameter index, parameter shape]*) – Positive Gradients.
- **neg_gradients** (*numpy array[parameter index, parameter shape]*) – Negative Gradients.
- **batch_size** (*float*) – The batch_size of the data.
- **epsilon** (*numpy array[num parameters]*) – The learning rate.
- **momentum** (*numpy array[num parameters]*) – The momentum term.
- **reg_l1norm** (*float*) – The parameter for the L1 regularization
- **reg_l2norm** (*float*) – The parameter for the L2 regularization also know as weight decay.
- **reg_sparseness** (*None or float*) – The parameter for the desired_sparseness regularization.
- **desired_sparseness** (*None or float*) – Desired average hidden activation or None for no regularization.
- **mean_hidden_activity** (*numpy array [num samples]*) – Average hidden activation <P(h_i=1|x)>_h_i
- **visible_offsets** (*float*) – If not zero the gradient is centered around this value.
- **hidden_offsets** (*float*) – If not zero the gradient is centered around this value.
- **use_centered_gradient** (*bool*) – Uses the centered gradient instead of centering.
- **restrict_gradient** (*None, float*) – If a scalar is given the norm of the weight gradient (along the input dim) is restricted to stay below this value.

- **restriction_norm** (*string, 'Cols','Rows', 'Mat'*) – Restricts the column norm, row norm or Matrix norm.

classmethod _calculate_centered_gradient (*gradients, visible_offsets, hidden_offsets*)
Calculates the centered gradient from the normal CD gradient for the parameters W, bv, bh and the corresponding offset values.

Parameters

- **gradients** (*List of 2D numpy arrays*) – Original gradients.
- **visible_offsets** (*numpy array[1,input dim]*) – Visible offsets to be used.
- **hidden_offsets** (*numpy array[1,output dim]*) – Hidden offsets to be used.

Returns Enhanced gradients for all parameters.

Return type numpy arrays (num parameters x [parameter.shape])

_train (*data, epsilon, k, momentum, reg_l1norm, reg_l2norm, reg_sparseness, desired_sparseness, update_visible_offsets, update_hidden_offsets, offset_typ, use_centered_gradient, restrict_gradient, restriction_norm, use_hidden_states*)
The training for one batch is performed using Contrastive Divergence (CD) for k sampling steps.

Parameters

- **data** (*numpy array [batch_size, input dimension]*) – The data used for training.
- **epsilon** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The learning rate.
- **k** (*int*) – NUmber of sampling steps.
- **momentum** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The momentum term.
- **reg_l1norm** (*float*) – The parameter for the L1 regularization
- **reg_l2norm** (*float*) – The parameter for the L2 regularization also know as weight decay.
- **reg_sparseness** (*None or float*) – The parameter for the desired_sparseness regularization.
- **desired_sparseness** (*None or float*) – Desired average hidden activation or None for no regularization.
- **update_visible_offsets** (*float*) – The update step size for the models visible offsets.
- **update_hidden_offsets** (*float*) – The update step size for the models hidden offsets.
- **offset_typ** (*string*) –

  Different offsets can be used to center the gradient.

  :Example: 'DM' uses the positive phase visible mean and the negative phase hidden mean. 'A0' uses the average of positive and negative phase mean for visible, zero for the hiddens. Possible values are out of {A,D,M,0}x{A,D,M,0}
- **use_centered_gradient** (*bool*) – Uses the centered gradient instead of centering.

- **restrict_gradient** (*None*, *float*) – If a scalar is given the norm of the weight gradient (along the input dim) is restricted to stay below this value.
- **restriction_norm** (*string, 'Cols','Rows', 'Mat'*) – Restricts the column norm, row norm or Matrix norm.
- **use_hidden_states** (*bool*) – If True, the hidden states are used for the gradient calculations, the hiddens probabilities otherwise.

train(*data*, *num_epochs=1*, *epsilon=0.01*, *k=1*, *momentum=0.0*, *reg_l1norm=0.0*, *reg_l2norm=0.0*, *reg_sparseness=0.0*, *desired_sparseness=None*, *update_visible_offsets=0.01*, *update_hidden_offsets=0.01*, *offset_typ='DD'*, *use_centered_gradient=False*, *restrict_gradient=False*, *restriction_norm='Mat'*, *use_hidden_states=False*)

Train the models with all batches using Contrastive Divergence (CD) for k sampling steps.

Parameters

- **data** (*numpy array [batch_size, input dimension]*) – The data used for training.
- **num_epochs** (*int*) – NUmber of epochs (loop through the data).
- **epsilon** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The learning rate.
- **k** (*int*) – NUmber of sampling steps.
- **momentum** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The momentum term.
- **reg_l1norm** (*float*) – The parameter for the L1 regularization
- **reg_l2norm** (*float*) – The parameter for the L2 regularization also know as weight decay.
- **reg_sparseness** (*None or float*) – The parameter for the desired_sparseness regularization.
- **desired_sparseness** (*None or float*) – Desired average hidden activation or None for no regularization.
- **update_visible_offsets** (*float*) – The update step size for the models visible offsets.
- **update_hidden_offsets** (*float*) – The update step size for the models hidden offsets.
- **offset_typ** (*string*) –

  Different offsets can be used to center the gradient.

  Example:'DM' uses the positive phase visible mean and the negative phase hidden mean. 'A0' uses the average of positive and negative phase mean for visible, zero for the hiddens. Possible values are out of {A,D,M,0}x{A,D,M,0}
- **use_centered_gradient** (*bool*) – Uses the centered gradient instead of centering.
- **restrict_gradient** (*None*, *float*) – If a scalar is given the norm of the weight gradient (along the input dim) is restricted to stay below this value.
- **restriction_norm** (*string, 'Cols','Rows', 'Mat'*) – Restricts the column norm, row norm or Matrix norm.

- **use_hidden_states** (*bool*) – If True, the hidden states are used for the gradient calculations, the hiddens probabilities otherwise.

4.1.1.5.5.2 PCD

class pydeep.rbm.trainer.**PCD**(*model*, *num_chains*, *data=None*)

Implementation of the training algorithm Persistent Contrastive Divergence (PCD).

Reference

Training Restricted Boltzmann Machines using Approximations to the
Likelihood Gradient, Tijmen Tieleman, Department of Computer
Science, University of Toronto, Toronto, Ontario M5S 3G4, Canada

__init__(*model*, *num_chains*, *data=None*)

The constructor initializes the PCD trainer with a given model and data.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **num_chains** (*int*) – The number of chains that should be used. .. Note:: You should use the data's batch size!
- **data** (*numpy array [num. samples x input dim]*) – Data for initialization, only has effect if the centered gradient is used.

4.1.1.5.5.3 PT

class pydeep.rbm.trainer.**PT**(*model*, *betas=3*, *data=None*)

Implementation of the training algorithm Parallel Tempering Contrastive Divergence (PT).

Reference

Parallel Tempering for Training of Restricted Boltzmann Machines,
Guillaume Desjardins, Aaron Courville, Yoshua Bengio, Pascal
Vincent, Olivier Delalleau, Dept. IRO, Universite de Montreal P.O.
Box 6128, Succ. Centre-Ville, Montreal, H3C 3J7, Qc, Canada.

__init__(*model*, *betas=3*, *data=None*)

The constructor initializes the IPT trainer with a given model anddata.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **betas** (*int, numpy array [num betas]*) – List of inverse temperatures to sample from. If a scalar is given, the temperatures will be set linearly from 0.0 to 1.0 in 'betas' steps.
- **data** (*numpy array [num. samples x input dim]*) – Data for initialization, only has effect if the centered gradient is used.

4.1.1.5.5.4 IPT

class pydeep.rbm.trainer.**IPT**(*model*, *num_samples*, *betas=3*, *data=None*)

Implementation of the training algorithm Independent Parallel Tempering Contrastive Divergence (IPT). As normal PT but the chain's switches are done only from one batch to the next instead of from one sample to the next.

Reference

Parallel Tempering for Training of Restricted Boltzmann Machines,
Guillaume Desjardins, Aaron Courville, Yoshua Bengio, Pascal
Vincent, Olivier Delalleau, Dept. IRO, Universite de Montreal P.O.
Box 6128, Succ. Centre-Ville, Montreal, H3C 3J7, Qc, Canada.

`__init__`(*model*, *num_samples*, *betas=3*, *data=None*)

The constructor initializes the IPT trainer with a given model and data.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **num_samples** (*int*) – The number of Samples to produce. .. Note:: you should use the batchsize.
- **betas** (*int, numpy array [num betas]*) – List of inverse temperatures to sample from. If a scalar is given, the temperatures will be set linearly from 0.0 to 1.0 in 'betas' steps.
- **data** (*numpy array [num. samples x input dim]*) – Data for initialization, only has effect if the centered gradient is used.

4.1.1.5.5.5 GD

class pydeep.rbm.trainer.**GD**(*model*, *data=None*)

Implementation of the training algorithm Gradient descent. Since it involves the calculation of the partition function for each update, it is only possible for small BBRBMs.

`__init__`(*model*, *data=None*)

The constructor initializes the Gradient trainer with a given model.

Parameters

- **model** (*Valid model class.*) – The model to sample from.
- **data** (*numpy array [num. samples x input dim]*) – Data for initialization, only has effect if the centered gradient is used.

`_train`(*data*, *epsilon*, *k*, *momentum*, *reg_l1norm*, *reg_l2norm*, *reg_sparseness*, *desired_sparseness*, *update_visible_offsets*, *update_hidden_offsets*, *offset_typ*, *use_centered_gradient*, *restrict_gradient*, *restriction_norm*, *use_hidden_states*)

The training for one batch is performed using True Gradient (GD) for k Gibbs-sampling steps.

Parameters

- **data** (*numpy array [batch_size, input dimension]*) – The data used for training.
- **epsilon** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The learning rate.
- **k** (*int*) – NUmber of sampling steps.
- **momentum** (*scalar or numpy array[num parameters] or numpy array[num parameters, parameter shape]*) – The momentum term.

- **reg_l1norm** (*float*) – The parameter for the L1 regularization
- **reg_l2norm** (*float*) – The parameter for the L2 regularization also know as weight decay.
- **reg_sparseness** (*None or float*) – The parameter for the desired_sparseness regularization.
- **desired_sparseness** (*None or float*) – Desired average hidden activation or None for no regularization.
- **update_visible_offsets** (*float*) – The update step size for the models visible offsets.
- **update_hidden_offsets** (*float*) – The update step size for the models hidden offsets.
- **offset_typ** (*string*) –

  Different offsets can be used to center the gradient.

  Example: 'DM' uses the positive phase visible mean and the negative phase hidden mean.

  'A0' uses the average of positive and negative phase mean for visible, zero for the hiddens. Possible values are out of {A,D,M,0}x{A,D,M,0}
- **use_centered_gradient** (*bool*) – Uses the centered gradient instead of centering.
- **restrict_gradient** (*None, float*) – If a scalar is given the norm of the weight gradient (along the input dim) is restricted to stay below this value.
- **restriction_norm** (*string, 'Cols','Rows', 'Mat'*) – Restricts the column norm, row norm or Matrix norm.
- **use_hidden_states** (*bool*) – If True, the hidden states are used for the gradient calculations, the hiddens probabilities otherwise.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

H

I

K

L

M

N

P

R

S

T

U

V

Z