dictlearn Documentation

Release 0.0.0

Per Magne

Sep 09, 2018

Contents:

CHAPTER 1

Getting Started

1.1 Overview

dictlearn is a module for signal and image processing. This tool include easy-to-use algorithms for denoising, inpainting, feature enhancement and detection, and image segmentation. Additionally, this tool has some methods designed specifically for medical image processing, among these are vessel segmentation and denoising of large 3D images.

- Multiple algorithms for dictionary learning and sparse coding
- Accelerated with Cython and C
- Built on numpy, scipy and scikit-learn

This module is a part of a masters thesis in applied mathematics, which can be read here.

1.2 Installation

Clone the repository:

```
$ git clone git@github.com:permfl/dictlearn.git
```

1.2.1 Linux/OSX

Install dependencies with:

```
$ pip install -r requirements.txt
```

Make sure *scipy* and *numpy* are linked with *BLAS/lapack*. See the installation guides for numpy and scipy for more details.

Then install the library with:

```
$ python setup.py install
```

1.2.2 Windows

Using Anaconda is strongly recommended. The *PyWavelet* package in *requirement.txt* are not listed in anaconda package repository. Comment out this dependency with #, then install dependencies with *conda install*:

```
$ conda install --file requirements.txt
$ pip install PyWavelets
```

Then install the library with:

```
$ python setup.py install
```

Cython not compiling on Windows

Make sure you have the Microsoft C++ compiler. Download the compiler for python 2.7: https://www.microsoft.com/en-us/download/details.aspx?id=44266

For python 3 you need `Build Tools for Visual Studio 2017`: https://www.visualstudio.com/downloads/#build-tools-for-visual-studio-2017

See here if downloading the above compiler doesn't fix the problem.

1.3 Next steps

See the examples, or the dictionary learning tutorial

CHAPTER 2

Dictionary learning tutorial

Simply put dictionary learning is the method of learning a matrix, called a dictionary, such that we can write a signal as a linear combination of as few columns from the matrix as possible.

When using dictionary learning for images we take advantage of the property that natural images can be represented in a sparse way. This means that if we have a set of basic image features any image can be written as a linear combination of only a few basic features. The matrix we call a *dictionary* is such a set. Each column of the dictionary is one basic image features. In the litterature these feature vectors are called atoms.

We don't work on full images directly, but small image patches. An image patch is simply a small square from the image, and when working with dictionary learning we normally extract all overlapping image patches. If we want to work on patches of size *(8, 8)* we start at *image[0, 0]* and extraxt the first patch *patches[:, 0] = image[0:8, 0:8].flatten()* the second patch will be *patches[:, 1] = image[0:8, 1:9].flatten()*. The inverse transformation, from image patches to image is done adding each patch to its origination? location in the image and then averaging all values for overlapping pixels.

```
>>> import dictlearn as dl
>>> image = dl.imread('examples/images/house.png')
>>> image_patches = dl.Patches(image, 8)
>>> matrix = image_patches.patches
>>> matrix.shape
(64, 62001)
```

With dictionary learning we want to find a dictionary, $\mathbf{D}$, and a vector with coefficients for the linear combinations, $\mathbf{y}$. The vector of an image patches we'll denote by $\mathbf{x}$. Then our goal is to find $\mathbf{D}$ and $\mathbf{y}$ such that the error $\|\mathbf{x} - \mathbf{D}\mathbf{y}\|_2$ is small. Before we said: .. *any image can be written as a linear combination of only a few basic features*. This means $\mathbf{y}$ has to be sparse such that we only use a few of the available atoms in the dictionary. We could try to minimize the error above and hope we get a sparse $\mathbf{y}$, but the would almost certainly not work. Something that does work is to add regularization. Regularization is a way to control how the terms we're minimizing over behaves. In our case we need a type of regularization that is small when $\mathbf{y}$ is sparse and large otherwise. We'll use ℓ_0-regularization, and the minimization problem becomes

$$\underset{\mathbf{D},\,\mathbf{y}}{arg\,min} \quad \frac{1}{2}\|\mathbf{x} - \mathbf{D}\mathbf{y}\|_2^2 + \lambda\|\mathbf{y}\|_0$$

Almost always we'll have have thousands of training signals that each should be represented with a sparse vector. To make this easier we are going to organize our training data as columns in a big matrix **X** with the shape *(signal_size, n_signals)*. Then we have the dictionary which is denoted by **D**. The shape of the dictionary is *(signal_size, n_atoms)*. Finally we have the sparse representation which is the matrix **A** with shape *(n_atoms, n_signals)*, where each column is the representation for the corresponding signal (column i **X**). Now almost all the important parts are introduced and we can look at the definition of the learning problem.

$$\min_{\mathbf{D}\in\mathcal{C},\,\mathbf{A}} \quad \frac{1}{2}\left\|\mathbf{X}-\mathbf{D}\mathbf{A}\right\|_F^2 + \lambda\Psi(\mathbf{A})$$

What this definition says is that we need to the dictionary and sparse representation that minimizes the two terms. The first term measure the error between the real signals **X** and the reconstructed signals **DA**. This first term is essentially the least squares problem. $\|.\|_F$ is the Frobenius norm

The second term is the interesting part. This is the regularization. With regularization we can alter the way the matrix **A** behaves during the minimization. Since we want a sparse representation **A**, the function Ψ needs to be chosen such that the minimization above "prefers" **A** to be sparse. We need Ψ to be a sparsity inducing function. In this package two different functions are used. The first is the ℓ_0- norm which counts the number of nonzero entries. The other is the ℓ_1-norm, this is the sum of the absolute values. When using ℓ_0-norm we say the problem is ℓ_0- regularized, ℓ_1 - regularized in the other case.

2.1 Training

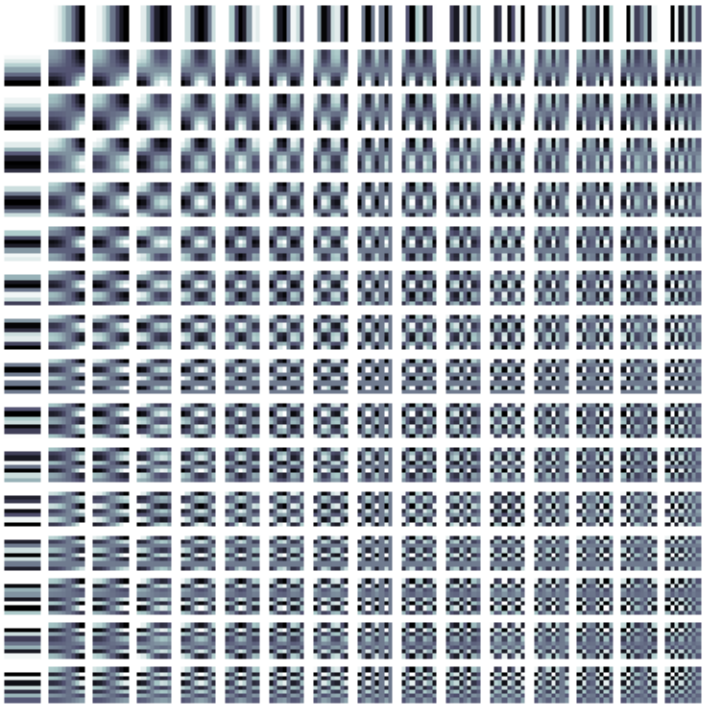
The functions found in *Optimize* are used to train the dictionary. All of these algorithms work in two stages: (1) sparse coding with the current dictionary then (2) using the sparse codes update the dictionary such that the new dictionary will better approximate the image.

To start training we first need an initial guess for the dictionary. This can be pretty much anything as long as the columns are normalized, but some choices give faster convergence than others. One good dictionary is the one created using the *Discrete Cosine Transform* basis functions.

```
import dictlearn as dl
dictionary = dl.dct_dict(256, 8)  # dl.dct_dict(n_atoms, patch_size)
```

The DCT dictionary looks like

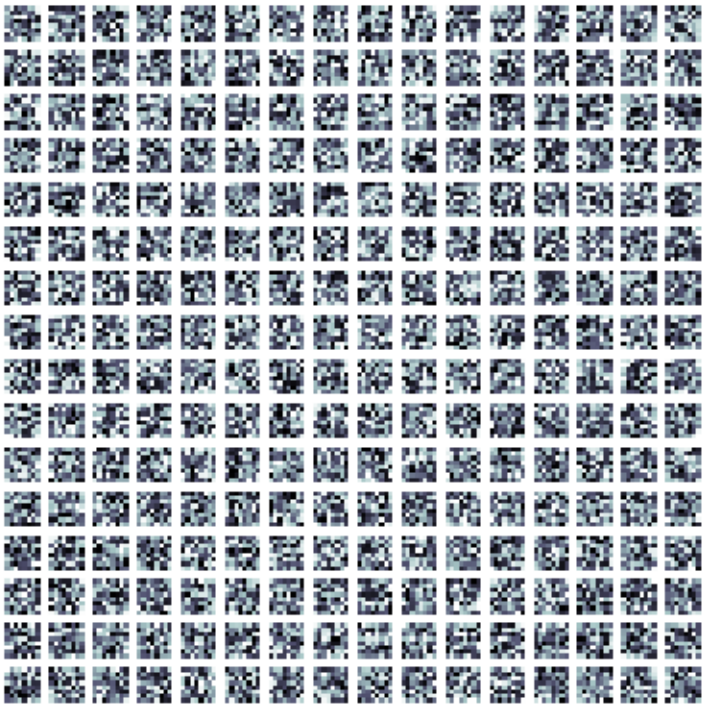
```
dl.visualize_dictionary(dictionary, 16, 16)
```



Each box is one atom. The atoms are reshaped into the shape of the image patches for better visualization.

A random dictionary can also be used. `dl.random_dictionary(rows, columns)` samples from a uniform distribution. This looks like

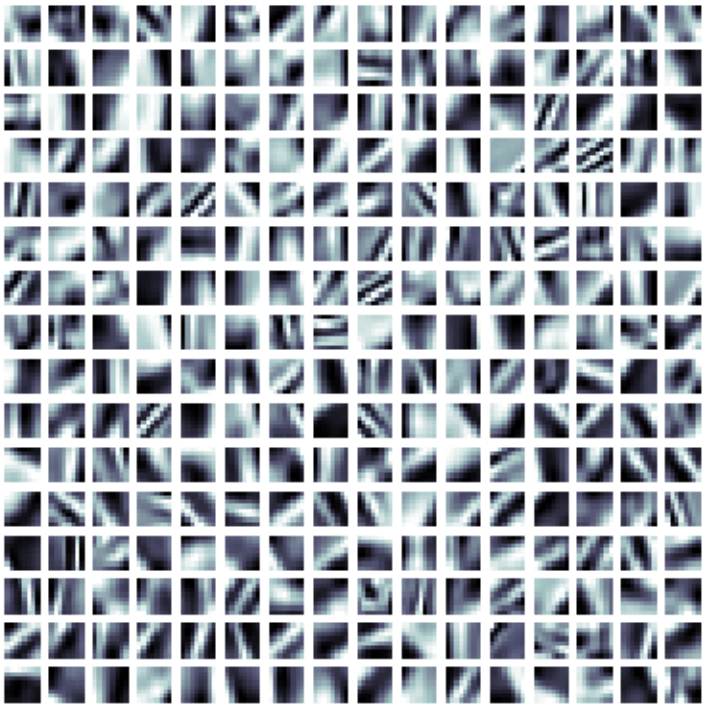
```
import dictlearn as dl
dictionary = dl.random_dictionary(64, 256)
dl.visualize_dictionary(dictionary, 16, 16)
```



The DCT dictionary will give very good results with no training. But training the dictionary will always give better results. In *High-Level Algorithms* a set of high level interfaces are given to train a dictionary, and other methods (denoise, inpaint, . . .). Low level functions are found in *Optimize*. The easiest way to use an image to train a dictionary is to use the class `ImageTrainer`.

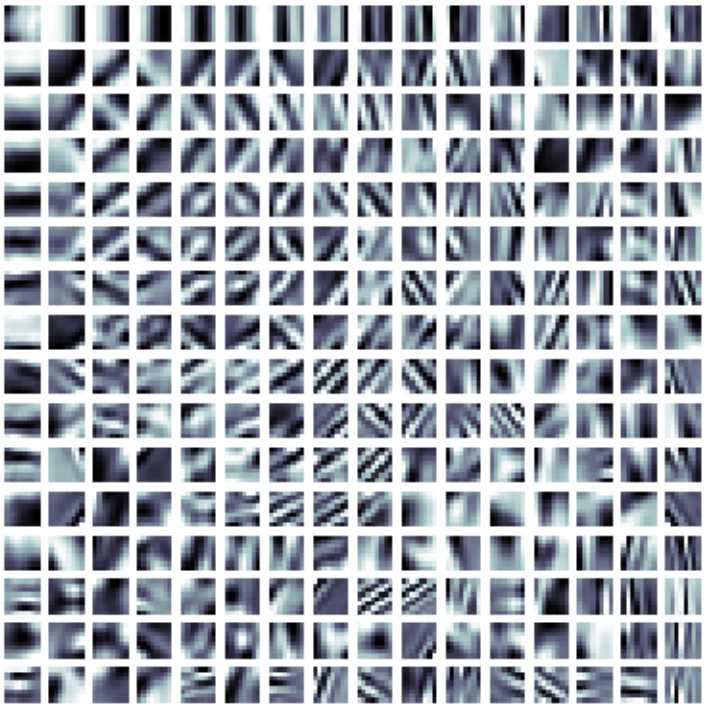
```
trainer = dl.ImageTrainer('examples/images/lena512.png', patch_size=8)
trainer.dictionary = dictionary
trainer.train(iters=100, n_nonzero=8, n_threads=4, verbose=True)

dl.visualize_dictionary(trainer.dictionary, 16, 16)
```

After training the random dictionary above for 100 iterations it now looks like



The low level methods in *Optimize* can also be used. Training the DCT dictionary for 50 iterations will look like:

```
import dictlearn as dl

image = dl.imread('examples/images/lena512.png').astype(float)
image_patches = dl.Patches(image, 8) # Creates 8x8 image patches
dictionary = dl.dct_dict(256, 8)  # Initial dictionary
dictionary = dl.ksvd(image_patches, dictionary, 50, n_nonzero=8,
                     n_threads=4, verbose=True)

dl.visualize_dictionary(dictionary, 16, 16)
```



2.1.1 Denoise

Here we'll show how to denoise a grayscale (colors in future release) image using patch-based denoising. First, import the library

```
import dictlearn as dl
```

Then you can open an image and create an instance of the denoising class:

```
denoiser = dl.Denoise('path/to/noisy_image.png', patch_size=8, method='online')
```

The denoise class handles both training the dictionary and denoising the image (source: dictlearn/algorithms.py:Denoise). The **first** parameter is the image data, this can be a path, image as a numpy array or an instance of *preprocess.Patches*. The **second** is the size of one dimension in the image patches. As the patches are squares the total patch size in this case is 64, 8*8. The patch size will impact the blurriness of the denoised image, bigger patches gives more blur. More on this later. The **third** argument is the training algorithm. The choices are *{'online', 'batch', 'ksvd'}*. *Online* training uses [ODL](#), while *batch* and *ksvd* both uses [K-SVD](#). The difference between the last two is that *batch* uses an additional step with Orthogonal-MP after training to denoise the image, while *ksvd* uses the sparse coefficients from training to denoise the image.

To train the dictionary, call the *train* method

```
denoiser.train()
```

Calling *train* without arguments like this will give good results for most cases. If you're using *method='ksvd'*, train require the argument *sigma* - the noise variance. You can pass the following keyword arguments to fine tune the training

- **iters**: Number of training iterations. 15 for *batch* or *ksvd* and 5000 for *online* are good starting points
- **n_atoms**: Number of atoms (columns) in the dictionary. Optimally you want your sparse code to use all the atoms. A good default value is 2*total_patch_size. The complexity of the training algorithms with respect to the number of atoms is linear.
- **n_nonzero**: Number of coefficients to use for sparse coding. Has to be atleast one, and cannot be larger than the number of atoms. **Check complexity OMP-Batch**. If you're using too many coefficients you might end up capturing the noise in the trained dictionary.
- **fill in the rest**

Finally, denoise the image:

```
denoised_image = denoiser.denoise(sigma=20, n_threads=4)
```

Image still has a lot of noise, what to do? Try the following, the points has decreasing impact on the denoised image 1. Higher sigma 2. Bigger patches 3. Fewer n_nonzero coeffs 4. Fewer atoms 5. More iterations

Image is very blurry, what to do? Follow the points above, but do the opposite. Ie. *point one: Higher sigma* - you do **lower** sigma.

Full example

```
import dictlearn as dl
import matplotlib.pyplot as plt
from scipy import misc

# Set default pyplot colormat
plt.rcParams['image.cmap'] = 'bone'

clean = misc.imread('images/lena512.png').astype(float)
noisy = misc.imread('images/lena_noisy512.png').astype(float)

denoiser = dl.Denoise(noisy, patch_size=10, method='batch')
denoiser.train(iters=40, n_nonzero=1, n_atoms=256, n_threads=4)
denoised = denoiser.denoise(sigma=33, n_threads=4)

plt.subplot(131)
plt.imshow(clean)
plt.axis('off')
plt.title('Clean')

plt.subplot(132)
plt.imshow(noisy)
plt.axis('off')
plt.title('Noisy, psnr = {}'.format(dl.utils.psnr(clean, noisy, 255)))

plt.subplot(133)
plt.imshow(denoised)
plt.axis('off')
plt.title('Denoised, psnr = {}'.format(dl.utils.psnr(clean, denosied, 255)))
plt.show()
```

2.1.2 Inpaint

Mask

2.1.3 Upscaling

Train two dictionaries

2.1.4 Compression

Distribute compressed sparse codes along with the dictionary

CHAPTER 3

High-Level Algorithms

High level interfaces to the different algorithms and methods. Using algorithms from *optimize* we get denoising, inpainting etc

3.1 Training

class dictlearn.algorithms.**Trainer**(*signals*, *method='online'*, *regularization='l0'*)

Parameters

- **signals** – Training data, shape (n_features, n_samples)
- **method** – Training algorithm, 'online' or 'batch'
- **regularization** – 'l0' or 'l1', 'l0' is faster, but 'l1' can sometimes be more accurate

train(*dictionary=None*, *n_atoms=None*, *iters=None*, *n_nonzero=10*, *tolerance=0*, *n_threads=1*, *verbose=False*, ***kwargs*)

Train a dictionary on training signals using 'Online Dictionary Learning (ODL)' or 'K-SVD'.

Both methods update the dictionary once very iteration. ODL will find the sparse coding on one signal and then update the dictionary using a variant of block coordinate-descent with momentum. K-SVD will sparse code all signals before doing the dictionary update, thus every iteration of K-SVD is a lot slower. Both produces similar results given the same running time

Parameters

- **dictionary** – Optional. Dictionary, ndarray with shape (signal_size, n_atoms)
- **n_atoms** – Number of dictionary atoms, default 2*signal_size
- **iters** – Training iterations, default 10 if 'batch', 1000 for 'online'
- **n_nonzero** – Max number of nonzero coefficients in sparse codes. Default 10
- **tolerance** – Sparse coding tolerance. Adds coefficients to the sparse approximation until the tolerance is achieved, or all coefficients are active

- **n_threads** – Number of threads to use. Default 1
- **verbose** – Print progress if True. Default False

class dictlearn.algorithms.**ImageTrainer**(*image*, *patch_size=8*, *method='online'*, *regularization='l0'*)

Parameters

- **image** – Train dictionary on this image (data). Can be a path to image, numpy array, dl.Patches or dl.Patches3D. If path or numpy array then dl.Patches are created. If the image is to large for keeping all image patches in memory pass dl.Patches3D instance
- **patch_size** – Size of image patches
- **method** – Method for training, 'online', or 'batch'
- **regularization** – Regularization to use, 'l0' or 'l1'

train(*dictionary=None*, *n_atoms=None*, *iters=None*, *n_nonzero=10*, *tolerance=0*, *n_threads=1*, *verbose=False*, ***kwargs*)

Train a dictionary on training signals using 'Online Dictionary Learning (ODL)' or 'K-SVD'.

Both methods update the dictionary once very iteration. ODL will find the sparse coding on one signal and then update the dictionary using a variant of block coordinate-descent with momentum. K-SVD will sparse code all signals before doing the dictionary update, thus every iteration of K-SVD is a lot slower. Both produces similar results given the same running time

Parameters

- **dictionary** – Optional. Dictionary, ndarray with shape (signal_size, n_atoms)
- **n_atoms** – Number of dictionary atoms, default 2*signal_size
- **iters** – Training iterations, default 10 if 'batch', 1000 for 'online'
- **n_nonzero** – Max number of nonzero coefficients in sparse codes. Default 10
- **tolerance** – Sparse coding tolerance. Adds coefficients to the sparse approximation until the tolerance is achieved, or all coefficients are active
- **n_threads** – Number of threads to use. Default 1
- **verbose** – Print progress if True. Default False

3.2 Denoise

class dictlearn.algorithms.**Denoise**(*image*, *patch_size=8*, *method='online'*, *regularization='l0'*)

Image Denoising with Dictionary Learning

Train a dictionary on the noisy image, then denoise using sparse coding. If method = 'ksvd' a dictionary is learned using K-SVD and the image is denoised using the sparse coefficients from the last training iteration. If method = 'batch' or 'online' an additional sparse coding step is used to compute the sparse codes for denoising.

Both adaptive and *static* denoising is supported.

Example adaptive denoise:

```
>>> denoiser = Denoise('noisy/image.png', 12)
>>> denoiser.train(iters=5000, n_nonzero=2, n_atoms=256)
>>> cleaned = denoiser.denoise(sigma=30)
```

Example pre-trained (static) dictionary:

```
>>> import numpy as np
>>> dictionary = np.load('dictionary.npy')
>>> denoiser = Denoise('noisy/image.png', 12)
>>> denoiser.dictionary = dictionary
>>> denoiser.train()  # Optional training
>>> cleaned = denoiser.denoise(sigma=30)
```

The size of the image patches and sigma in denoise() will have a large effect on the denoised image. If either are too large the image will look blurry, if too low the difference between the original noisy image and the denoised image will be small. A patch size in [8, 12] is usually a good choice for most images. If the image has very small details then a smaller patch size might be needed. If the structures in the image are large and smooth, larger patches can produce better results.

The value of sigma is highly dependent on the image and its scale. If the image is in [0, 1] then 0 <= sigma <= 1. An image in [0, 255] gives sigma in [0, 255].

Parameters

- **image** – Noisy image. Can be a path to image, numpy array, dl.Patches or dl.Patches3D. If path or numpy array then dl.Patches are created. If the image is to large for keeping all image patches in memory pass dl.Patches3D instance
- **patch_size** – Size of image patches
- **method** – Method for training, 'online', 'batch' or 'ksvd'
- **regularization** – Regularization to use, 'l0' or 'l1'

denoise (*sigma=20*, *n_threads=1*, *noise_gain=1.15*)

Denoise the image

Sigma is the parameter that has the largest effect on the final result. For the best results sigma should be close to the variance of the noise. If the difference between the original and the denoised image is small sigma is probably too low. If the denoised image is very blurry then sigma is too large.

Blurry image? Reduce sigma

Noisy image? Increase sigma

Parameters

- **sigma** – Noise variance
- **n_threads** – Number of threads. Default 1
- **noise_gain** – Average number of nonzero coefficients in sparse approximation. Default 1.15, which has been shown to give good results.

Returns Reconstructed and denoised image

train (*dictionary=None*, *n_atoms=None*, *iters=None*, *n_nonzero=10*, *tolerance=0*, *n_threads=1*, *verbose=False*, ***kwargs*)

Train a dictionary on training signals using 'Online Dictionary Learning (ODL)' or 'K-SVD'.

Both methods update the dictionary once very iteration. ODL will find the sparse coding on one signal and then update the dictionary using a variant of block coordinate-descent with momentum. K-SVD will sparse code all signals before doing the dictionary update, thus every iteration of K-SVD is a lot slower. Both produces similar results given the same running time

Parameters

- **dictionary** – Optional. Dictionary, ndarray with shape (signal_size, n_atoms)

- **n_atoms** – Number of dictionary atoms, default 2*signal_size
- **iters** – Training iterations, default 10 if 'batch', 1000 for 'online'
- **n_nonzero** – Max number of nonzero coefficients in sparse codes. Default 10
- **tolerance** – Sparse coding tolerance. Adds coefficients to the sparse approximation until the tolerance is achieved, or all coefficients are active
- **n_threads** – Number of threads to use. Default 1
- **verbose** – Print progress if True. Default False

3.3 Inpaint

class dictlearn.algorithms.**Inpaint**(*image*, *mask*, *patch_size=8*, *method='online'*)

Image inpainting

Fill in missing areas of an image, or remove unwanted objects. Works very well if the missing areas are fairly small, and smaller than the image patches. If the missing areas are large use TextureSynthesis

```
>>> import dictlearn as dl
>>> import numpy as np
>>> image imread('some/img.png')
>>> # Mask with 60% of pixels marked missing
>>> mask = np.random.rand(*image.shape) < 0.6
>>> corrupted = image * mask
>>> # Plot corrupted
>>> inp = Inpaint(corrupted, mask)
>>> reconstructed = inp.train().inpaint()
```

Parameters

- **image** – Corrupted image
- **mask** – Binary mask for image. Pixels in mask marked 0 is will be inpainted. locations marked 1 is kept as is.
- **patch_size** – Size of image patches. Default 8
- **method** – Inpainting method. 'online' or 'itkrmm'.

inpaint(*n_nonzero=20*, *group_size=60*, *search_space=20*, *stride=4*, *callback=None*, *tol=0*, *verbose=False*)

Parameters

- **n_nonzero** – Number of nonzero coefficients for reconstruction
- **group_size** – Size of group for 'online' inpaint. Finds the 'group_size' most similar image patches and trains a dictionary on this group
- **search_space** – How far from the current pixel (i, j) to search for similar patches. Will search all pixels (i - s, j - s) to (i + s, j + s), s = search_space.
- **stride** – Distance between image patches
- **callback** – Callback function for online inpaint. Called with two arguments

  1. the current reconstruction and
  2. current iteration

- **tol** – For method = 'itkrmm', tolerance for sparse coding reconstruction. Set this the same way as sigma in denoise, to also denoise the image if needed. If the image is noise free, use n_nonzero
- **verbose** – Print progress

Returns Inpainted image

train(***kwargs*)

Train a dictionary on training signals using 'Online Dictionary Learning (ODL)' or 'K-SVD'.

Both methods update the dictionary once very iteration. ODL will find the sparse coding on one signal and then update the dictionary using a variant of block coordinate-descent with momentum. K-SVD will sparse code all signals before doing the dictionary update, thus every iteration of K-SVD is a lot slower. Both produces similar results given the same running time

Parameters

- **dictionary** – Optional. Dictionary, ndarray with shape (signal_size, n_atoms)
- **n_atoms** – Number of dictionary atoms, default 2*signal_size
- **iters** – Training iterations, default 10 if 'batch', 1000 for 'online'
- **n_nonzero** – Max number of nonzero coefficients in sparse codes. Default 10
- **tolerance** – Sparse coding tolerance. Adds coefficients to the sparse approximation until the tolerance is achieved, or all coefficients are active
- **n_threads** – Number of threads to use. Default 1
- **verbose** – Print progress if True. Default False

class dictlearn.algorithms.**TextureSynthesis**(*image*, *mask*, *patch_size=8*, *method='online'*)

Inpaint by texture synthesis

For each missing pixel we create an image patch centered at this pixel, and search the image for the most similar patch. Then the missing pixel is replaced with the center pixel in the most similar patch. Repeat until all pixels are filled.

denoise(*sigma=20*, *n_threads=1*, *noise_gain=1.15*)

Denoise the image

Sigma is the parameter that has the largest effect on the final result. For the best results sigma should be close to the variance of the noise. If the difference between the original and the denoised image is small sigma is probably too low. If the denoised image is very blurry then sigma is too large.

Blurry image? Reduce sigma

Noisy image? Increase sigma

Parameters

- **sigma** – Noise variance
- **n_threads** – Number of threads. Default 1
- **noise_gain** – Average number of nonzero coefficients in sparse approximation. Default 1.15, which has been shown to give good results.

Returns Reconstructed and denoised image

inpaint(*max_iters=None*, *verbose=False*)

Parameters

- **max_iters** – If None, run until the entire image is filled. Otherwise stop after 'max_iters' iterations
- **verbose** – Print progress

train (***kwargs*)

Train a dictionary on training signals using 'Online Dictionary Learning (ODL)' or 'K-SVD'.

Both methods update the dictionary once very iteration. ODL will find the sparse coding on one signal and then update the dictionary using a variant of block coordinate-descent with momentum. K-SVD will sparse code all signals before doing the dictionary update, thus every iteration of K-SVD is a lot slower. Both produces similar results given the same running time

Parameters

- **dictionary** – Optional. Dictionary, ndarray with shape (signal_size, n_atoms)
- **n_atoms** – Number of dictionary atoms, default 2*signal_size
- **iters** – Training iterations, default 10 if 'batch', 1000 for 'online'
- **n_nonzero** – Max number of nonzero coefficients in sparse codes. Default 10
- **tolerance** – Sparse coding tolerance. Adds coefficients to the sparse approximation until the tolerance is achieved, or all coefficients are active
- **n_threads** – Number of threads to use. Default 1
- **verbose** – Print progress if True. Default False

3.4 Structure Detection

dictlearn.detection.**smallest_cluster** (*features*, *n_clusters*, *verbose=False*)

Extract the smallest cluster of samples for the data 'features'.

See examples/vessel_enhancement.py

Parameters

- **features** – array of features, shape (n_samples, n_features)
- **n_clusters** – Number of features
- **verbose** –

Returns Prediction vector of shape (n_samples,) where prediction[i] == True if features i belongs to the smallest cluster and prediction[i] == False if belongs to any other cluster

CHAPTER 4

Sparse Coding

The goal of these algorithms is to find a sparse coefficient matrix (or vector) for some signals given a dictionary of signal features.

The two norms are defined as:

$\|\mathbf{x}\|_0 = \#\{\mathbf{x}_i \neq 0 : \forall \mathbf{x}_i \in \mathbf{x}\}$

$\|\mathbf{x}\|_1 = \sum_i |x_i|$

Thus the two problem formulations are:

$$\hat{a} = \underset{a}{argmin} \quad \frac{1}{2} \|x - Da\|_F^2 + \lambda \|a\|_0$$

$$\hat{a} = \underset{a}{argmin} \quad \frac{1}{2} \|x - Da\|_F^2 + \lambda \|a\|_1$$

With signal x, dictionary D, and $\hat{a}$ the sparse coefficients

4.1 ℓ_0-regularization

`dictlearn.sparse.`**`omp_batch`**(*signals*, *dictionary*, *n_nonzero=10*, *tol=0*, *n_threads=1*)

Batch Orthogonal Matching Pursuit. A more effective version than omp_cholesky if the number of signals is high. Saves time and calculations by doing some pre-computations. See [2] for details

```
>>> import dictlearn as dl
>>> image = dl.imread('some/image.png')
>>> dictionary = dl.load_dictionary('some-dictionary')
>>> image_patches = dl.Patches(image, 8)
>>> sparse_codes = dl.omp_batch(image_patches, dictionary, n_nonzero=4)
>>> sparse_approx = dictionary.dot(sparse_codes)
```

Parameters

- **signals** – Signals to encode. numpy.ndarray shape (signal_size, n_signals) or dictlearn.preprocess.Patches
- **dictionary** – ndarray, shape (signal_size, n_atoms)
- **n_nonzero** – Default 10. Max number of nonzero coeffs for sparse codes
- **tol** – Default 0. Add nonzero coeffs until `norm(signal - dict*sparse_code) < tol`
- **n_threads** – Default 1. Number of threads to use.

Returns Sparse codes, shape (n_atoms, n_signals)

`dictlearn.sparse.`**omp_cholesky**(*signals*, *dictionary*, *n_nonzero=10*, *tol=0*, *n_threads=1*)

Cholesky Orthogonal Matching pursuits. Use omp_batch if many signals need to be sparse coded. See [2] for details

```
>>> import dictlearn as dl
>>> image = dl.imread('some/image.png')
>>> dictionary = dl.load_dictionary('some-dictionary')
>>> image_patches = dl.Patches(image, 8)
>>> sparse_codes = dl.omp_cholesky(image_patches, dictionary, n_nonzero=4)
>>> sparse_approx = dictionary.dot(sparse_codes)
```

Parameters

- **signals** – Signals to sparse code, shape (signal_size,) or (signal_size, n_signals)
- **dictionary** – ndarray, shape (signal_size, n_atoms)
- **n_nonzero** – Default 10. Max number of nonzero coeffs for sparse codes
- **tol** – Default 0. Add nonzero coeffs until norm(signal - dict*sparse_code) < tol
- **n_threads** – Default 1. Number of threads to use.

Returns Sparse codes, shape (n_atoms, n_signals)

`dictlearn.sparse.`**omp_mask**(*signals*, *masks*, *dictionary*, *n_nonzero=None*, *tol=1e-06*, *n_threads=1*, *verbose=False*)

Orthogonal Matching Pursuit for masked data. Tries to reconstruct the full set of signals, by ignoring data points signals[i, j] where mask[i, j] == 0.

```
>>> import dictlearn as dl
>>> broken_image = dl.imread('some-broken-image')
>>> mask = dl.imread('mask-for-some-image')
>>> # Create patches from broken_image and mask + get dictionary
>>> sparse_codes = dl.omp_mask(broken_image, mask, dictionary)
>>> reconstructed_image_patches = dictionary.dot(sparse_codes)
```

Parameters

- **signals** – Corrupted signals, shape (size, n_signals)
- **masks** – Masks, shape (size, n_signals). masks[:, i] is the mask for signals[:, i]
- **dictionary** – Trained dictionary, shape (size, n_atoms)
- **n_nonzero** – Default None. Max number of nonzero coeffs to use

- **tol** – Default 1e-6. Stop if signal approximation is within the accuracy. Overwrites n_nonzero
- **n_threads** – Not used
- **verbose** – Default False. Print progress

Returns Sparse approximation to signals, shape (n_atoms, n_signals)

dictlearn.sparse.**iterative_hard_thresholding**(*signal*, *dictionary*, *iters*, *step_size*, *initial*, *n_nonzero=None*, *penalty=None*)

If penalty make sure sqrt(2*penalty) is not bigger than all elements in initial_a, if that's the case the solution is just the zero vector

Requires tuning of the hyper parameters step_size, initial_a and penalty. optimize.omp_cholesky may be a better choice.

If reg_param is supplied this solves:

min 0.5*||signal - D*alpha||_2^2 + reg_param||alpha||_0

If n_nonzero is supplied then the following is solved for alpha

min || signal - D*alpha||_2^2 such that ||alpha||_0 <= n_nonzero

Parameters

- **signal** – Signal in R^m
- **dictionary** – Dictionary (D) in R^(m,p)
- **iters** – Number of iterations
- **step_size** – Step size for gradient descent step
- **initial** – Initial sparse coefficients. Need ||initial_a||_0 <= nonzero if n_nonzero not None
- **n_nonzero** – Sparsity target
- **penalty** – Penalty parameter

Returns Sparse decomposition of signal

4.2 ℓ_1-regularization

dictlearn.sparse.**lars**(*signals*, *dictionary*, *n_nonzero=0*, *alpha=0*, *lars_params=None*, ***kwargs*)

"Homotopy" algorithm for solving the Lasso

argmin 0.5*||X - DA||_2^2 + r*||A||_1

for all r.

This algorithm is supposedly the most accurate for l1 regularization.

This is terribly slow, and not very accurate. ~20x slower than OMP. Find this strange as OMP solves a NP-Hard problem and this a convex

Parameters

- **signals** – Signals to encode. Shape (signal_size, n_signals) or (signal_size,)
- **dictionary** – Dictionary, shape (signal_size, n_atoms)

- **n_nonzero** – Number of nonzero coefficients to use
- **alpha** – Regularization parameter. Overwrites n_nonzero
- **lars_params** – See sklearn.linear_models.LassoLars docs
- **kwargs** – Not used. Just to make calling API for all regularization algorithms the same

Returns Sparse codes, shape (n_atoms, n_signals) or (n_atoms,)

dictlearn.sparse.**lasso**(*signals*, *dictionary*, *alpha*, *lasso_params=None*, ***kwargs*)

Parameters

- **signals** – Signals to encode, shape (signal_size,) or (signal_size, n_signals)
- **dictionary** – Dictionary, shape (signal_size, n_atoms)
- **alpha** – Regularization parameter. <~0.9 yields more accurate results than OMP, but slower
- **lasso_params** – Other parameters. See sklearn.linear_model.Lasso
- **kwargs** – Not used, just for making calling compatible with other sparse coding methods

Returns Sparse codes, shape (n_atoms,) or (n_atoms, n_signals)

dictlearn.sparse.**iterative_soft_thresholding**(*signal*, *dictionary*, *initial*, *reg_param=None*, *n_nonzero=None*, *step_size=0.1*, *iters=10*)

l1 reg using iterative soft thresholding

if regularization parameter is given solve:

min_a 1/2 * || x - D*alpha||_2^2 + reg_param*||alpha||_1

if number of nonzero is given solve:

min_a ||signal - D*alpha||_2^2 such that ||alpha||_1 <= n_nonzero

This method requires tuning of the initial value for alpha, step_size and iters/res_param.

Parameters

- **signal** – Signal, shape (signal_size,)
- **dictionary** – Dictionary, shape (signal_size, n_atoms)
- **initial** – Initial sparse codes, shape (n_atoms,)
- **reg_param** – Regularization parameter
- **n_nonzero** – Max number of nonzero coeffs
- **step_size** – Gradient descent step size
- **iters** – Number of iterations

Returns Sparse codes, shape (n_atoms,)

dictlearn.sparse.**l1_ball**(*vector*, *target*)

Create sparse approximation of 'vector' by projecting onto to l1-ball keeping at most 'target' coefficients active

Parameters

- **vector** – Vector to project to sparse
- **target** – Max number of nonzero coefficients

Returns Sparse vector, same shape as 'vector'

4.3 References

[1] Rubinstein, Ron, Michael Zibulevsky, and Michael Elad. "Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit." Cs Technion 40.8 (2008): 1-15.

CHAPTER 5

Optimize

Optimization methods for learning dictionaries.

5.1 Standard algorithms

Algorithms for training on complete data (ie. when you don't need to mask your data). These are the algorithms needed for most use cases.

`dictlearn.optimize.`**`ksvd`**(*signals*, *dictionary*, *iters*, *n_nonzero=0*, *omp_tol=0*, *tol=0*, *verbose=False*, *n_threads=1*, *retcodes=False*)

Iterative batch algorithm [1, 2] for fitting a dictionary D to a set of signals X. Each iteration consists of two stages:

1. Fix **D**. Find sparse codes **A** such that **X** is approx equal **DA**
2. Fix the sparse codes. Find **D_new** such that `norm(X - D_new*A) < norm(X - DA)`

Need one of (or both) `n_nonzero` and `omp_tol` different from zero. If `n_nonzero > 0` and `omp_tol == 0` then KSVD finds an approximate solution to:

$$\min_{\mathbf{D},\mathbf{A}} \quad \|\mathbf{X} - \mathbf{DA}\|_F^2 \text{ such that } \|\mathbf{A}\|_0 \leq n\_nonzero$$

If omp_tol is not None then KSVD finds an approximate solution to:

$$\underset{\mathbf{D},\mathbf{A}}{argmin} \quad \|\mathbf{A}\|_0 \text{ such that } \|\mathbf{X} - \mathbf{DA}\|_F^2 \leq omp\_tol$$

```
>>> import dictlearn as dl
>>> from numpy import linalg as LA
>>> image = dl.Patches(dl.imread('some-image'), 8).patches
>>> dictionary = dl.random_dictionary(8*8, 128)
>>> sparse_1 = dl.omp_batch(image, dictionary, 10)
>>> new_dict, _ = dl.ksvd(image, dictionary, 20, 10)
>>> err_initial = LA.norm(image - dictionary.dot(sparse_1))
```

(continues on next page)

(continued from previous page)

```
>>> sparse_2 = dl.omp_batch(image, new_dict, 10)
>>> err_trained = LA.norm(image - new_dict.dot(sparse_2))
>>> assert err_trained < err_initial
```

Parameters

- **signals** – Training signals. One signal per column numpy.ndarray of shape (signal_size, n_signals)
- **dictionary** – Initial dictionary, shape (signal_size, n_atoms)
- **iters** – Max number of iterations
- **n_nonzero** – Default 0. Max nonzero coefficients in sparse decomposition
- **omp_tol** – Default 0. Tolerance of sparse approximation. Overrides n_nonzero
- **tol** – Default 0. Stop learning if `norm(signals - dictionary.dot(sparse_codes) < tol`
- **verbose** – Print progress
- **n_threads** – Default 1. Number of threads to use for sparse coding.
- **retcodes** – Return sparse codes from last iteration

Returns dictionary[, sparse decomposition if retcodes = True]

`dictlearn.optimize.odl`(*signals*, *dictionary*, *iters=1000*, *n_nonzero=10*, *tol=0*, *verbose=False*, *batch_size=1*, *n_threads=1*, *seed=None*)

Online dictionary learning algorithm

This algorithm sparsely encode one training signal at the time and updates the dictionary given this signal. The number if iterations also determines how many of the training signals are used. If the number of iterations is less than the number of signals, then `iters` signals is drawn at random. If iters is equal to the number of signals all signals are used in a random order.

The dictionary atoms are updated using block-coordinate descent. See [4] for details

Parameters

- **signals** – Training signals. One signal per column numpy.ndarray of shape (signal_size, n_signals)
- **dictionary** – Initial dictionary, shape (signal_size, n_atoms)
- **iters** – Default 1000. Number of training iterations to use. This is also equal to the number of signals used in training.
- **n_nonzero** – Default 10. Max nonzero coefficients in sparse decomposition
- **tol** – Default 0. Tolerance of sparse approximation. Overrides n_nonzero
- **verbose** – Print progress
- **batch_size** – The number of signals to use for each dictionary update
- **seed** – Seed the drawing of random signals

Returns Trained and improved dictionary

dictlearn.optimize.**mod**(*signals*, *dictionary*, *n_nonzero*, *iters*, *n_threads=1*)
Method of optimal directions

The first alternate minimization algorithm [3] for dictionary learning.

1. Find sparse codes A given signals X and dictionary D
2. Update D given the new A by approximately solving for D in X = DA. That is D = X*pinv(A), with pinv(A) = A.T*(A*A.T)^-1

Parameters

- **signals** – Training signals
- **dictionary** – Initial dictionary
- **n_nonzero** – Sparsity target for signal approximation
- **iters** – Number of dictionary update iterations
- **n_threads** – Default 1. Number of threads to use for sparse coding step

Returns New dictionary

5.2 Masked data

Use these algorithms when you need to explicitly mark which data points to use and which to discard/ignore. All masks should have the same shape as the training data, with values [0, 1]. A data point is ignored if 0.

dictlearn.optimize.**itkrmm**(*signals*, *masks*, *dictionary*, *n_nonzero*, *iters*, *low_rank=None*, *verbose=False*)
Train a dictionary from corrupted image patches.

Need signals and masks of same shape. Data point `signals[i, j]` is used if the corresponding point in the mask, `masks[i, j] == True`. All points `signals[i, j]` with `masks[i, j] == False` are ignored.

See [5] for details.

Parameters

- **signals** – Corrupted image patches, shape (patch_size, n_patches)
- **masks** – Binary mask for signals, same shape as signal.
- **dictionary** – Initial dictionary (patch_size, n_atoms)
- **n_nonzero** – Number of nonzero coeffs to use for training
- **iters** – Max number of iterations
- **low_rank** – Matrix of low rank components, shape (patch_size, n_low_rank)
- **verbose** – Print progress

Returns Dictionary. Shape (patch_size, n_atoms + n_low_rank)

dictlearn.optimize.**reconstruct_low_rank**(*signals*, *masks*, *n_low_rank*, *initial=None*, *iters=10*)
Reconstruct low rank components from image patches, by ITKrMM.

Low rank components or atoms capture low rank signal features. In the case where signals are image patches low rank atoms can capture average intensities and low variance features in the image. When these are included

in a dictionary most of the signals will use at least one of the low rank atoms leaving the normal atoms to represent more specific image features

Parameters

- **signals** – Image patches
- **masks** – Masks for image patches
- **n_low_rank** – Number of low rank components to reconstruct
- **initial** – Initial low rank dictionary, shape (signals.shape[0], n_low_rank)
- **iters** – Number of iterations for each component

Returns Low rank dictionary, shape (signals.shape[0], n_low_rank)

5.3 References

[1] *M. Aharon, M. Elad, and A. Bruckstein, "The K-SVD: An algorithm for designing of overcomplete dictionaries for sparse representation," IEEE Trans. on Signal Processing, vol. 54, no. 11, pp. 4311–4322, 2006.*

[2] *Rubinstein, Ron, Michael Zibulevsky, and Michael Elad. "Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit." Cs Technion 40.8 (2008): 1-15.*

[3] *Engan, Kjersti, Sven Ole Aase, and J. Hakon Husoy. "Method of optimal directions for frame design." Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on. Vol. 5. IEEE, 1999.*

[4] *Mairal, Julien, et al. "Online dictionary learning for sparse coding." Proceedings of the 26th annual international conference on machine learning. ACM, 2009.*

[5] *Naumova, Valeriya, and Karin Schnass. "Dictionary learning from incomplete data." arXiv preprint arXiv:1701.03655 (2017).*

CHAPTER 6

Preprocessing

class dictlearn.preprocess.**Patches**(*image*, *size*, *stride=1*, *max_patches=None*, *random=None*, *order='C'*)

REMOVE_MEAN = 'remove_mean'

Generate and reconstruct image patches

Parameters

- **image** – ndarray, 2D or 3D
- **size** – Patch size, since all patches are square (cube) this is just the size of the first dimension. Ie 8 for (8, 8) patches
- **stride** – Stride/distance between patches in image. Can be int or list type. If int then the stride is the same in every dimension. If list then each stride[i] denotes the stride on axis i. Patches cannot be reconstructed if the stride in one dimension is larger the than the patch size in the same dimension. Ie. stride[i] > size[i] for any i
- **max_patches** – Maximum number of patches
- **random** – True for taking patches from random locations in image. Overwritten if max_patches=None
- **order** – C or F for C or FORTRAN order on underlying data

check_batch_size_or_raise(*batch_size*)

Check if there's enough memory to store 'batch_size' patches. Raise MemoryError if not

generator(*batch_size*, *callback=False*)

Create and reconstruct a batch iteratively.

If Patches.patches is too large to keep all in memory use this. Only 'batch_size' patches are generated. This requires approximately 'batch_size' times less memory. If batch_size is 100 and Patches.patches need 100 memories then this need only one memory.

```
>>> import numpy as np
>>> volume = np.load('some_image.npy')
```

(continues on next page)

(continued from previous page)

```
>>> size, stride = [10, 10, 10], [1, 1, 1]
>>> patches = Patches(volume, size, stride)
>>> for batch in patches.generator(100):
>>>     # Handle batch
>>>     assert batch.shape[1] == 100
>>>     assert batch.shape[0] == 1000, 'Can fail at last batch, see stride'
```

One matrix of size (patch_size, batch_size) is created per iteration. This generator return (batch, callback) with batch a numpy array of shape (patch_size, batch_size) and callback(batch) reconstruct the part of the volume which contains the given batch. It is required that the argument to callback has shape identical to the batch returned

This can be used if Patches3D.create() requires too much memory. The amount of memory required by this method is batch_size*size[0]*size[1]*size[2]*volume.dtype.itemsize bytes

```
>>> import numpy as np
>>> volume = np.load('some_image_volume.npy')
>>> size, stride = [10, 10, 10], [1, 1, 1]
>>> patches = Patches(volume, size, stride)
>>> for batch, reconstruct in patches.generator(100, callback=True):
>>>    # Handle batch, here we do nothing
>>>    reconstruct(batch)
>>> assert np.array_equal(volume, patches.reconstructed)
```

Parameters

- **batch_size** – Size of batches. The last batch can be smaller if n_patches % batch_size != 0
- **callback** – If True a callback function 'callback(batch)' is returned such the the image can be partially reconstructed

Returns Generator

n_patches

Returns Number of patches

patches

Returns Image patches, shape (size[0]*size[1]*..., n_patches)

reconstruct(*new_patches*, *save=False*)

Reconstruct the image with new_patches. Overlapping regions are averaged. The reconstructed patches are not saved by default

self.patches are the same object before and after this method is called, as long as save=False

Parameters

- **new_patches** – *ndarray* (patch_size, n_patches). Patches returned from Patches.patches
- **save** – Overwrite current patches with new_patches

Returns Reconstructed image

remove_mean(*add_back=True*)

Remove the mean from every patch, this is automatically added back if the image is reconstructed

Parameters **add_back** – Automatically add back the mean to patches on reconstruction

shape
Shape of patch matrix, (patch_size, n_patches)

size
Size of patches

class dictlearn.preprocess.**Patches3D**(*volume*, *size*, *stride*)
Create and reconstruct image patches from 3D volume.

Parameters

- **volume** – 3D ndarray
- **size** – Size of image patches, (x, y, z)
- **stride** – Stride between each patch, (i, j, k). 'volume' cannot be reconstructed if i > x, j > y or k > z

create_batch_and_reconstruct(*batch_size*)

Create and reconstruct a batch iteratively.

One matrix of 'batch_size' is created per iteration. This generator return (batch, callback) with batch a numpy array of shape (n, batch_size) and callback(batch) reconstruct the part of the volume which contains the given batch.

This can be used if Patches3D.create() requires too much memory. The amount of memory required by this method is batch_size*size[0]*size[1]*size[2]*volume.dtype.itemsize bytes

```
>>> import numpy as np
>>> import dictlearn as dl
>>> dictionary = np.load('some_dictionary.npy')
>>> volume = np.load('some_image_volume.npy')
>>> size, stride = [1, 1, 1], [1, 1, 1]
```

```
>>> patches = Patches3D(volume, size, stride)
>>> for batch, reconstruct in patches.create_batch_and_
↪reconstruct(100):
>>>     new_batch = dl.omp_batch(batch, dictionary)
>>>     reconstruct(new_batch)
```

```
>>> reconstructed_volume = patches.reconstructed
```

Parameters **batch_size** – Number of patches per batch.

Returns Generator, next() returns (batch, reconstruct(new_batch)

next_batch(*batch_size*)

Parameters **batch_size** – Number of image patches per batch

Returns Generator, next() returns a ndarray of shape (n, batch_size)

dictlearn.preprocess.**center**(*data*, *dim=0*, *retmean=False*, *inplace=False*)
Remove the mean at dim from every patch

Parameters

- **data** – ndarray, data to center
- **dim** – Dimension to calculate mean, default 0 (columns)
- **retmean** – Return mean if True

- **inplace** – Change argument data directly if True, returns mean only

Returns Centered patches and mean if retmean is True. Or just mean if inplace is True

dictlearn.preprocess.**normalize**(*patches*, *lim=0.2*)

L2 normalization. If l2 norm of a patch is smaller than lim the the patch is divided element wise by lim

Parameters

- **patches** – ndarray, (size, n_patches)
- **lim** – Threshold for low intensity patches

Returns

CHAPTER 7

Working with VTK

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, image processing, and visualization. It consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods, as well as advanced modeling techniques such as implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. VTK has an extensive information visualization framework and a suite of 3D interaction widgets. The toolkit supports parallel processing and integrates with various databases on GUI toolkits such as Qt and Tk. VTK is cross-platform and runs on Linux, Windows, Mac, and Unix platforms. VTK is part of Kitware's collection of commercially supported open-source platforms for software development.

7.1 Installing

VTK version 8.0.0 and later is available on PyPi for python versions 2.7, 3.4, 3.5 and 3.6

```
$ pip install vtk
```

On Windows only python 3.5 and 3.6 are supported. VTK can also be installed with anaconda. Available versions are at https://anaconda.org/conda-forge/vtk/files.

If you want to build VTK yourself, download it from https://www.vtk.org. When building toggle the flag *VTK_WRAP_PYTHON* to generate the wrapping files. Detailed instructions can bee seen here

7.1.1 Wrappers

Reading and writing VTK image files to and from numpy array requires a lot of boilerplate code. The classes below, `VTKImage` and `VTKInformation` wraps reading and writing `vti` files, ie images of type `vtkImageData`. To read an image, write: `image = VTKImage.read('path.vti')`. If `image` will be modified you have to save its attributes: `info = image.information()`.

`VTKImage` is a subclass of `numpy.ndarray` and can be used as any normal numpy array.

Finally the image can be written to disk

```
VTKImage.write_vti('new_path.vti', image, info)
```

class VTKImage

Numpy ndarray wrapper for VTK images. This class holds meta data about the image such that reading and writing to file will keep the correct attributes

```
>>> import numpy as np
>>> import dictlearn as dl
>>> # volume is a numpy array
>>> volume = VTKImage.read('path/to/volume.vti')
>>> assert isinstance(volume, np.ndarray)
>>> prod = np.dot(volume[:10, :10], np.random.rand(10, 5))
>>> assert prod.shape == (10, 5)
>>> patches = dl.Patches(volume)
>>> patch_generator = patches.create_batch_and_reconstruct(10000):
>>> for batch, reconstruct in patch_generator:
>>>     # Handle batch
>>>     reconstruct(batch)
```

Write 'patches.reconstructed' to disk with the same attributes as 'path/to/volume.vti'

```
>>> volume.write('path/to/volume_new.vti', patches.volume)
```

information(*self*)

Get image meta data. See VTKInformation

static from_array(*array*, *info=None*)

Create a VTKImage from a numpy array

static from_image_data(*image_data*, *name=None*)

Crate VTKImage from vtkImageData

Parameters

- **image_data** – vtk.vtkImageData instance
- **name** – Name of point array to extract. Defaults to array at index 0

Returns VTKImage

static read(*path*, *name=None*)

Read a vti image.

Parameters

- **path** – Path to file
- **name** – Name or index of array. If 'name' is None then array at index 0 is returned

Returns VTKImage instance

write(*self*, *path*, *array=None*)

Write data (array or self) to 'vti' file. This file is written with self.extent, self.origin, self.spacing and self.dtype. If the instance is created with VTKImage.read() these attributes are copied from the read file, otherwise the default values are used:

- **extent = [0, self.dimensions[0] - 1,** 0, self.dimensions[1] - 1, 0, self.dimensions[2] - 1]
- origin = [0, 0, 0]
- spacing = [1, 1, 1]
- dtype = np.float64

Parameters

- **path** – Filename, where to save
- **array** – Optional, if array is None 'self' is written to file. If array is not None then array is written to file

Returns True if writing successful

static write_vti(*path*, *array*, *info=None*, *extent=None*, *origin=None*, *spacing=None*, *use_array_type=True*, *name='ImageScalars'*)

Write 'array' to 'path' as vti file

Parameters

- **path** – Where to write
- **array** – Data to write, ndarray with array.ndim == 3
- **info** – Optional instance of VTKInformation, overwrites extent, origin and spacing.
- **extent** – Data extent, array like, len(extent) == 6. Default [0, array.shape[0] - 1,0, array.shape[1] - 1, 0, array.shape[2] - 1]
- **origin** – Data origin, default [0, 0, 0]
- **spacing** – Spacing between voxels, default [1, 1, 1]
- **use_array_type** – Only used if info is not None. If this is False the image is saved with the data type given by info, otherwise array.dtype is used
- **name** – Name of scalar array

Returns True if write successful

print(*self*)

Print image information

copy(*self*, *order='C'*)

Return a copy of the image

Parameters **order** – {'C', 'F', 'A', 'K'}, optional Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

class VTKInformation(*path=None*, *reader=None*)

Holds image metadata

- datatype, VTK datatype, int
- bounds, bounds of the geometry, size 6
- center, center of the geometry, size 3
- dimensions, size of the geometry, size 6
- **extent, six integers - give the index of the first and last** point in each direction
- origin,
- spacing,

Parameters

- **path** – Path to vtk image
- **reader** – Instance of a vtk image reader

vtp_to_vti(*surface*, *information*, *invalue=1*, *outvalue=0*, *flip=None*)

Convert a closed surface to ImageData using vtkPolyDataToImageStencil. All points on or inside the takes 'invalue' while all point outside the surface takes 'outvalue'

Parameters

- **surface** – Path to surface file
- **information** – Information about the volume to create. Either an instance of VTKInformation or path to a vti file. If this is a path to an image, its attributes are copied to the converted image
- **invalue** – Value of points inside or of the surface
- **outvalue** – Value of points outside the surface.
- **flip** – Flip around an axis, options: 'x', 'y', 'z' or None to keep as is

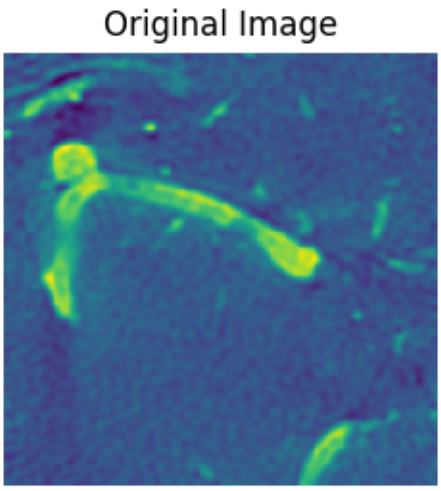
Returns An instance of VTKImage

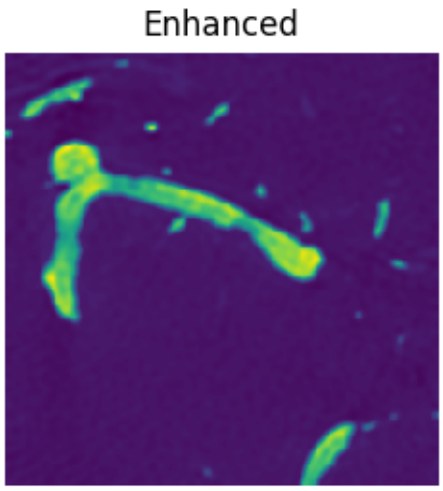
CHAPTER 8

Examples

8.1 Vessel Enhancement

```
import sys
import dictlearn as dl
import matplotlib.pyplot as plt


image = dl.imread('images/vessel.png')
patches = dl.Patches(image, size=4)
labels = dl.detection.smallest_cluster(patches.patches.T, 2, True)

# Adjust alpha to change the weight for the enhanced image
if len(sys.argv) == 2:
    alpha = float(sys.argv[1])
else:
    alpha = 0.2

vessels = patches.patches * labels
new = alpha*patches.patches + (1 - alpha)*vessels
enhanced = patches.reconstruct(new)

plt.subplot(121)
plt.imshow(image)
plt.axis('off')
plt.title('Original Image')

plt.subplot(122)
plt.imshow(enhanced)
plt.axis('off')
plt.title('Enhanced')

plt.show()
```





8.2 Denoise

```
import os
import matplotlib.pyplot as plt
import dictlearn as dl

base_dir = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))


plt.rcParams['image.cmap'] = 'bone'

image1 = os.path.join(base_dir, 'images/lena_noisy512.png')
image2 = os.path.join(base_dir, 'images/lena512.png')
noisy_image = dl.imread(image1).astype(float)
clean_image = dl.imread(image2).astype(float)


denoise = dl.Denoise(noisy_image, patch_size=11, method='online')
# method='batch' for ksvd

denoise.train(iters=1000, n_nonzero=10, n_atoms=256, n_threads=2, verbose=True)
denoised_odl = denoise.denoise(sigma=33, n_threads=2)
```

(continues on next page)

(continued from previous page)

```
denosied_ksvd = dl.ksvd_denoise(noisy_image, patch_size=11, n_atoms=256, sigma=33,
                                verbose=True, n_threads=4)


plt.subplot(221)
plt.imshow(clean_image)
plt.title('Original')
plt.axis('off')

plt.subplot(222)
plt.imshow(noisy_image)
plt.title('Noisy, psnr = {:2f}'.format(dl.utils.psnr(clean_image, noisy_image, 255)))
plt.axis('off')

plt.subplot(223)
plt.imshow(denoised_odl)
plt.title('ODL, psnr = {:.2f}'.format(dl.utils.psnr(clean_image, denoised_odl, 255)))
plt.axis('off')

plt.subplot(224)
plt.imshow(denosied_ksvd)
plt.title('K-SVD, psnr = {:.2f}'.format(dl.utils.psnr(clean_image, denosied_ksvd,␣
↪255)))
plt.axis('off')
plt.show()
```

8.3 Inpaint

```
import os
import numpy as np
import matplotlib.pyplot as plt
import dictlearn as dl

base_dir = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))

plt.rcParams['image.cmap'] = 'bone'


house = os.path.join(base_dir, 'images/test/house.png')
lena = os.path.join(base_dir, 'images/test/lena.png')
text_mask = os.path.join(base_dir, 'images/test/TextMask256.png')

house = dl.imread(house).astype(float)
lena = dl.imread(lena).astype(float)
text_mask = dl.imread(text_mask).astype(bool)

keep = 0.3  # Keep 30% of the original data
random_mask = np.random.rand(*lena.shape) < keep

# We now have two images, and two masks - we'll apply both masks to
# both images and see how the structure of the image affect the result
plt.subplot(221)
plt.imshow(house)
```

(continues on next page)

(continued from previous page)

```
plt.axis('off')

plt.subplot(222)
plt.imshow(lena)
plt.axis('off')

plt.subplot(223)
plt.imshow(text_mask)
plt.axis('off')

plt.subplot(224)
plt.imshow(random_mask)
plt.axis('off')
plt.figure()

# Corrupt the images
house_text = house*text_mask
house_rnd = house*random_mask
lena_text = lena*text_mask
lena_rnd = lena*random_mask

plt.suptitle('Corrupted images')
plt.subplot(221)
plt.imshow(house_text)
plt.axis('off')

plt.subplot(222)
plt.imshow(lena_text)
plt.axis('off')

plt.subplot(223)
plt.imshow(house_rnd)
plt.axis('off')

plt.subplot(224)
plt.imshow(lena_rnd)
plt.axis('off')
plt.figure()

iters = 10


def create_callback(original_image):
    def print_iter(image_estimate, iteration):
        psnr = dl.utils.psnr(original_image, image_estimate, 255)
        print('Iter %d, PSNR = %.2f' % (iteration + 1, psnr))

    return print_iter


inpaint = dl.Inpaint(house_text, text_mask)
house_text_inpainted = inpaint.inpaint(callback=create_callback(house))

inpaint = dl.Inpaint(lena_text, text_mask)
lena_text_inpainted = inpaint.inpaint(callback=create_callback(lena))

inpaint = dl.Inpaint(house_rnd, random_mask)
```

(continues on next page)

(continued from previous page)

```
house_rnd_inpainted = inpaint.inpaint(callback=create_callback(house))

inpaint = dl.Inpaint(lena_rnd, random_mask)
lena_rnd_inpainted = inpaint.inpaint(callback=create_callback(lena))

plt.suptitle('Each of these are the cleaned version of img in same spot as prev plot')
plt.subplot(221)
plt.imshow(house_text_inpainted)
plt.title('PSNR = {:.3f}'.format(dl.utils.psnr(house, house_text_inpainted, 255)))
plt.axis('off')

plt.subplot(222)
plt.imshow(lena_text_inpainted)
plt.title('PSNR = {:.3f}'.format(dl.utils.psnr(lena, lena_text_inpainted, 255)))
plt.axis('off')

plt.subplot(223)
plt.imshow(house_rnd_inpainted)
plt.title('PSNR = {:.3f}'.format(dl.utils.psnr(house, house_rnd_inpainted, 255)))
plt.axis('off')

plt.subplot(224)
plt.imshow(lena_rnd_inpainted)
plt.title('PSNR = {:.3f}'.format(dl.utils.psnr(lena, lena_rnd_inpainted, 255)))
plt.axis('off')
plt.show()
```

CHAPTER 9

Indices and tables

- genindex
- search

Index

T

V

W