Exercises
Computational Intelligence Lab
SS 2011

Machine Learning Laboratory
Dept. of Computer Science, ETH Zürich
Prof. Dr. Joachim M. Buhmann
Web http://ml2.inf.ethz.ch/courses/cil

Series 11, May 11th, 2011 (Sparse Coding)

Problem 1 (Inpainting using Sparse Coding):

In this final assignment, we will tackle the problem of **inpainting**. Inpainting is the task of robustly inferring missing information of an image based on the still observable parts of it. Typical real-world examples of the problem include removal of scratches in old photos, removal of overlaid text and scaling-up images. Inpainting can be also viewed as an interpolation problem and has attracted much interest in the recent years.

We begin with first setting up the environment on your local machine for sending your compression solution to the submission system, then we go step by step through a simple inpainting pipeline algorithm which utilizes sparse coding to infer the missing pixel information.

Submission system environment setup:

1. The web page for the image compression application can be found here:

   http://cil.inf.ethz.ch/applications/inpainting.

2. Download the provided templates for the functions needed, the evaluation script and the training data into a local folder.

3. Unzip the training images into the *same* folder.

The functions we provide fall into two categories:

1. **Helper functions** which we provide to help you perform certain tasks :
   - `haarTrans.m` : computes the normalized Haar Wavelet basis of a given size.
   - `overDCTdict.m` : computes the overcomplete DCT dictionary of a given size.
   - `random_mask.m` : returns a random mask of missing pixels for the inpainting.
2. **Template functions** which you have to fill in order to perform the inpainting:
   - `my_im2col.m` : extracts patches from an image and transform each patch to an 1D signal.
   - `my_col2im.m` : transforms a set of patches back to an image.
   - `inPainting.m` : the main function performing the inpainting. Takes as input the original image `I` and the mask denoting the missing pixels and returns the reconstructed image `I_rec`.
   - `buildDictionary.m` : builds a dictionary of your choice.
   - `sparseCoding.m` : implements the matching pursuit to perform the sparse coding.

To submit your solution to the online evaluation system, we require you to prepare a ".zip" file containing all the above functions and everything else you implement to support these functions.

Your submission is evaluated according to the following criteria:

1. Approximation quality: mean squared error of the reconstructed image `I_rec` w.r.t. the original image `I`.
2. Run time of the inpainting algorithm.

In order to be able to debug and optimize your routines before submission, we provide you the evaluation platform for your code in `EvaluateInpainting.m` : . For it to work, all given and implemented matlab functions and all the training images should be in one folder.

Images:

All images used in this assignment (training and test set) are grayscale .png images consisting of 512×512 pixels.

Inpainting using Sparse Coding:

In the following we are going to describe the basic algorithm to successfully attack the inpainting problem providing also programming hints and insights on the implementation part. The main function that performs the inpainting is

```
function I_rec = inPainting(I_mask, mask)
```

which takes as input the image with the missing information `I_mask` and the mask indicating the missing pixels and returns the reconstructed image having filled-in the missing holes. To successfully perform this operation the following steps need to be implemented:

1. Divide the image `I_mask` and the `mask` into non-overlapping patches of size `neib`. For each patch translate the 2D patch into an 1D signal of dimension $d = neib \times neib$. Every patch is now a column vector of dimension d, thus you can put together all l patches extracted from the image and construct the observations matrix $\mathbf{X}^{d \times l}$. This can be done with the matlab function `im2col` but since the image toolbox is not supported in the evaluation server you have to make your own implementation:

   ```
   X = my_im2col(I_mask, neib);
   M = my_im2col(mask, neib);
   ```

2. Next step is to construct the fixed overcomplete dictionary. To this end you have to implement function:

   ```
   U = buildDictionary(d)
   ```

   You can use your own dictionaries. However we provide you with functions to compute the overcomplete DCT dictionary (`D = overDCTdict(d,L)`) and the normalized Haar wavelet basis (`H=haarTrans(d)`).

3. Having the dictionary you can now perform the most important step of the pipeline, namely sparse coding of the known parts of the image using the fixed dictionary:

   ```
   Z = sparseCoding(U, X, M, sigma, rc_min);
   ```

   As a first solution to this problem we propose the algorithm by *Elad et al (2005)* discussed in the lecture. The algorithm consists of the following steps (the notation follows the one used in the lectures):

   (a) Define diagonal masking matrix $\mathbf{M}$, $m_{d,d} = 1$ if pixel d is known, $m_{d,d} = 0$ if pixel d is missing.

   (b) Sparse coding of known parts in overcomplete dictionary $\mathbf{U}$:

   $$\begin{aligned} \mathbf{z}^* &= \min_{\mathbf{z}} \|\mathbf{z}\|_0 \\ \text{s.t.} \quad & \|\mathbf{M}(\mathbf{x} - \mathbf{U}\mathbf{z})\|_2 < \sigma \end{aligned} \tag{1}$$

   We already know that the Matching Pursuit (MP) algorithm solves the following problem:

   $$\begin{aligned} \mathbf{z}^* &= \arg\min_{\mathbf{z}} \|\mathbf{z}\|_0 \\ \text{s.t.} \quad & \|\mathbf{x} - \mathbf{U}\mathbf{z}\|_2 < \sigma \end{aligned} \tag{2}$$

   Comparing equations (1) and (2) we observe that we can modify MP to solve our original problem, if we only consider the known parts of the image and the reconstruction that are defined by the mask $\mathbf{M}$.

   Therefore in function `sparseCoding()` we can implement MP as described in the lecture slides keeping in mind that we don't use the complete matrices and vectors but the subsets defined by the known pixels in $\mathbf{M}$.

   Programming Hint :
   When you code the main while loop of the MP algorithm, use the following logical expression to stay in the loop:

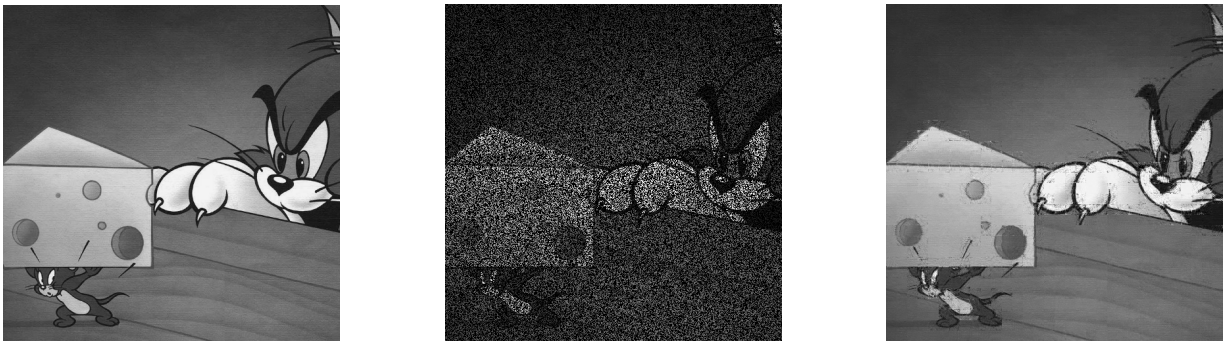
```
while (norm(residual) > sigma*norm(x)) && (rc_max > rc_min)
```

where `residual` is the current iteration residual, `rc_max` is the maximum absolute correlation to the current residual and `rc_min` is a minimum value of correlation to the residual we pass as parameter (e.g. `rc_min` = 0.01). The first term of the logical expression ensures that (1) is satisfied (where we normalize σ with the norm of the signal) and the second term ensures that you don't get into infinite loops when you have undercomplete dictionaries.

4. Finally we have to reconstruct the image. To achieve this we have to use the pixels we already know and for the missing ones use the reconstruction obtained from the sparse coding $\mathbf{z}^*$ in dictionary $\mathbf{U}$. In algebraic terms this can be written as: $\hat{\mathbf{x}} = \mathbf{M}\mathbf{x} + (\mathbf{I} - \mathbf{M})\,\mathbf{U}\mathbf{z}^*$

   Programming Hint :
   Make sure that after filling the missing pixels you stitch back together the sequence of 1D patches to reconstruct the image `I_rec` which you have to return from the `inPainting()` function. For this you need to reverse the transform you implemented in the `my_im2col` function.

Now, you have all the tools in order to implement the inpainting algorithm. Fig. 1 depicts the result of the inpainting algorithm presented here when applied to an image with 70% missing pixels.



(a) (b) (c)

Figure 1: Original image (left), masked with 70% missing pixels (middle) and inpainted reconstruction (right).

Possible extensions:

1. Instead of using only once sparse coding of the known parts in fixed dictionary $\mathbf{U}$, you can in a straightforward way implement the iterative EM variant of *Fadili et al (2007)* presented in the lecture. The key point in the variant is the alternation until convergence of the following two steps:

   - **(E-step)** Image reconstruction using the mask and the current estimate of the sparse coding $\mathbf{z}^t$:

   $$\mathbf{x}^t = \mathbf{M}\mathbf{x} + (\mathbf{I} - \mathbf{M})\,\mathbf{U}\mathbf{z}^\mathbf{t}$$

   - **(M-step)** Sparse coding of the **whole** currently reconstructed image $\mathbf{x}^t$:

   $$\begin{aligned} \mathbf{z}^t &= \min_{\mathbf{z}} \|\mathbf{z}\|_0 \\ \text{s.t.} &\quad \left\|\mathbf{x}^t - \mathbf{U}\mathbf{z}\right\|_2 < \sigma \end{aligned}$$

   Take some time to think how to initialize $\mathbf{z}^0$. Can you do better than starting from a random vector or a vector with zero coefficients?

2. We provide routines to work with the overcomplete DCT dictionary and/or the normalized Haar wavelet basis we encountered in Series 8. We encourage you to implement the dictionary learning algorithm presented in Lecture 12. Then you can use the dictionary learned on the training set for the sparse coding which will probably lead to better results in the inpainting task, as you will have atoms in the dictionary that are tailored to your training data and not a general purpose dictionary.

Update: In the syllabus webpage a basic template for the dictionary learning algorithm (`dictionary_learning.m`) is now provided. The algorithm can be used in order to learn offline a dictionary and then upload the learned dictionary to the submission server. Inside the `dictionary_learning.m` function a function `mp.m` is called, consisting of an implementation of the Matching Pursuit algorithm. Given that the main assignment has been already tackled, you should be able to readily provide the `mp.m` function.

We now assume that you have implemented the code for the dictionary learning and you are able to learn a dictionary $\mathbf{U}$ on the training set offline on your machine. In order to submit your code to the evaluation server you need to save matrix $\mathbf{U}$ under the filename `dictionary.mat` and include it in your zipped submission. If you look at the updated `EvaluateInpainting.m` function, dictionary $\mathbf{U}$ is declared as a global variable and then we check if the `dictionary.mat` file containing the dictionary $\mathbf{U}$ is supplied or not.

- If `dictionary.mat` is supplied in the .zip submission, it is loaded in the evaluation script and the loaded dictionary can be used in the `inPainting.m` function. It follows that U has to be declared as a global variable in the `inPainting.m` function and furthermore that the `buildDictionary.m` function is no longer needed as the dictionary is precomputed.
- If `dictionary.mat` is not submitted it means that the dictionary is not supplied, thus the implementation follows the guidelines of the main assignment.