Lab_5

April 7, 2022

1 LAB 5: Reconstructing galaxy images with compressed sensing

In this lab, you will implement a **compressed sensing** method, namely the **iterative soft thresholding algorithm (ISTA)**, and use it to reconstruct **DESI Legacy Imaging Surveys (DECals)** images of spiral galaxies (see Lab 3) from sparse and noisy measurements. ISTA iteratively minimizes the **Lasso optimization problem** that we considered in the lecture. We will provide a brief introduction of the ideas behind ISTA as we go. Some of the code has already been filled in for you; simply add your code (and you answers) wherever you find a "**TODO**" tag. If you prefer working with Python scripts rather than a Jupyter notebook, you can of course copy-paste the content of this notebook to a Python script (or simply run the command `jupyter nbconvert --to script [NOTEBOOK].ipynb`).

1.1 Loading, inspecting, and preparing the data

First, let us import the modules that we need. If you have not installed `h5py`, `PyWavelets` and `tqdm` yet, you can do so with

```
conda install h5py  # if you are using Anaconda
pip install h5py  # if you are using pip

conda install pywavelets  # if you are using Anaconda
pip install PyWavelets  # if you are using pip

conda install -c conda-forge tqdm  # if you are using Anaconda
pip install tqdm  # if you are using pip
```

[1]:
```
import os
import numpy as np
from scipy import fftpack
from scipy import linalg
import h5py
import pywt
from tqdm import tqdm
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['image.cmap'] = 'turbo'
np.random.seed(1111)
```

TODO: Load the data from 'DECals_galaxies.hdf5' (same data as in Lab 3). The data has shape 256 x 256 x 256 x 3, namely 256 spiral galaxy images of size 256 x 256 pixels with 3 channels (g,

r, and z band).

```
[ ]: # Load the data
     ...
```

TODO: Select your favourite galaxy image and take the average over the three channels (index 1 is a nice one for example). We will only need a single image in this lab, but you can of course re-run the Jupyter notebook for another galaxy image.

```
[ ]: X_orig = ...
```

TODO: Plot the galaxy image $\mathbf{X}_{orig}$.

```
[ ]: # Plot the galaxy image
     ...
```

```
[ ]: # Store the shape (2D and flattened as 1D)
     shape = X_orig.shape
     D = np.prod(shape)
```

```
[ ]: # Print the minimum and maximum pixels values
     print(X_orig.min(), X_orig.max())
```

TODO: Add some Gaussian (artificial) measurement noise to the data (independently drawn in each pixel), with zero mean and standard deviation $\sigma = 10$.

```
[ ]: X_noisy_raw = ...
```

TODO: We need to make sure the pixel values stay in the range [0, 255]. Create a function that clips pixel values outside the range `[low, high]` and sets them to the boundary values `low` and `high`, respectively.

```
[ ]: def clip_vals(x, low, high):
         ...
```

TODO: Apply this function to the noisy data.

```
[ ]: X_noisy = ...
```

TODO: Plot the noisy galaxy image $\mathbf{X}_{noisy}$.

```
[ ]: # Plot of the noisy image
     ...
```

We will now randomly subsample pixel values in the noisy galaxy image $\mathbf{X}_{noisy}$, yielding an image $\mathbf{Y}$ containing much fewer values (and zeros everywhere else) and then try to reconstruct the original image $\mathbf{X}_{orig}$ by iteratively solving a regularized minimization problem as considered in the lecture.

TODO: Generate a new image $\mathbf{Y}$ by randomly selecting 5% of the pixel values in $\mathbf{X}_{noisy}$ that will be kept and setting the remaining 95% of the pixel values in $\mathbf{Y}$ to 0 (in such a way that each

pixel in $\mathbf{X}_{noisy}$ has the same probability of being kept). You can do this for example by defining a random mask (array containing booleans True/False with the same shape as $\mathbf{X}_{noisy}$) that is `True` whenever a value shall be kept and `False` else. Also print out the number of pixels in $\mathbf{X}_{noisy}$ and the number of subsampled pixels in $\mathbf{Y}$.

NOTE: In the lecture, we defined the subsampled vector $\mathbf{y}$ to have dimension $S \ll D$. Since we are dealing with images here, it will be convenient to use the same shape for the subsampled picture $\mathbf{Y}$ as for the original picture $\mathbf{X}_{orig}$, so the subsampled picture can be plotted (which would not be possible with a flattened 1D vector of length S). The information about the pixels contained in the subsampled set is provided in the mask, and $\mathbf{Y} = 0$ wherever `mask == False`.

[ ]:
```
# Generate subsampled picture Y
mask = ...
Y = ...
```

[ ]:
```
# Plot of the subsampled image
...
```

1.2 Discrete cosine transform and discrete wavelet transform

The basis classes DCT and DWT are ready to be used - no need to do anything here.

[2]:
```
class DCT:
    """Discrete cosine transform"""

    def __init__(self, shape):
        """
        Initialize discrete cosine transform.
        :param shape: image shape
        """
        self.shape = shape

    def forward(self, image):
        """
        Forward DCT
        :param image: 2D array
        :return cosine-transformed image
        """
        image_t = fftpack.dct(fftpack.dct(image, norm='ortho', axis=0),␣
↪norm='ortho', axis=1)
        return image_t.reshape(-1)

    def inverse(self, image_t):
        """
        Inverse DCT
        :param image_t: cosine-transformed image
        :return image in real space
        """
```

```
        image_t = image_t.reshape(self.shape)
        return fftpack.idct(fftpack.idct(image_t, norm='ortho', axis=0),␣
↪norm='ortho', axis=1)
```

[3]:
```
class DWT:
    """Discrete wavelet transform"""

    def __init__(self, shape, wavelet='db4', level=3, do_weighting=True):
        """
        Initialize discrete wavelet transform.
        :param shape: image shape
        :param wavelet: name of the wavelet
        :param level: level of hierarchical wavelet analysis
        :param do_weighting: if True: weight deeper levels more
        """
        self.shape = shape
        self.wavelet = wavelet
        self.level = level
        self.mat_shapes = []

        # Weight the coefficients
        if do_weighting is False:
            self.weights = np.ones(3 * self.level + 1)
        else:
            weights = np.linspace(1, .2, self.level)
            weights = np.repeat(weights, 3)
            weights = np.hstack([[20], weights])  # insert first coefficient
            self.weights = weights

    def forward(self, image):
        """
        Forward DWT
        :param image: 2D array
        :return list with all wavelet coefficients (for all levels)
        """
        coeffs = pywt.wavedec2(image, wavelet=self.wavelet, level=self.level)
        # format: [cAn, (cHn, cVn, cDn), ...,(cH1, cV1, cD1)] , n=level

        # Transform to a list of arrays
        mat_list = [coeffs[0]]
        mat_list += [c_mat for c in coeffs[1:] for c_mat in c]

        # Store all matrix shapes
        self.mat_shapes = list(map(np.shape, mat_list))

        # Store coefficent matrices as list of arrays and apply weighting
```

```
        vec_list = [np.asarray(mat_list[j]).flatten() * self.weights[j] for j␣
↪in range(3 * self.level + 1)]

        # Return a long list with all coefficients
        return np.concatenate(vec_list)

    def inverse(self, wavelet_vector):
        """
        Inverse DWT
        :param wavelet_vector: long list with all wavelet coefficients (output␣
↪of forward)
        :return reconstructed image
        """
        # forward must be called at least once before calling inv
        if len(self.mat_shapes) == 0:
            print("Call 'forward' first to set the shapes of coefficient␣
↪matrices!")
            return []

        # Define a list with the matrix shapes
        vec_shapes = list(map(np.prod, self.mat_shapes))

        # Split up the 1D wavelet array into a list of arrays (one for each␣
↪coefficient)
        split_indices = np.cumsum(vec_shapes)
        vec_list_weighted = np.split(wavelet_vector, split_indices)

        # Revert weighting
        vec_list = [vec_list_weighted[j] / self.weights[j] for j in range(3 *␣
↪self.level + 1)]

        # Transform back to level-wise format
        coeffs = [np.reshape(vec_list[0], self.mat_shapes[0])]
        for j in range(self.level):
            triple = vec_list[3 * j + 1:3 * (j + 1) + 1]
            triple = [np.reshape(triple[i], self.mat_shapes[1 + 3 * j + i]) for␣
↪i in range(3)]
            coeffs = coeffs + [tuple(triple)]

        # Return the reconstructed image
        return pywt.waverec2(coeffs, wavelet=self.wavelet)
```

1.3 Some background on ISTA

In order to reconstruct the image $\mathbf{X}_{orig}$, consider again the **Lasso optimization problem** in the lecture notes for the task of compressed sensing

$$\theta^\star = \text{argmin}_\theta \ \|\mathbf{y} - \Phi\theta\|_2^2 + \lambda\|\theta\|_1 =: \text{argmin}_\theta \ F(\theta) + \lambda G(\theta),$$

where $\mathbf{y} = \text{flatten}(\mathbf{Y})$ (or, equivalently, the quantities can be left as matrices, in which case the Euclidean vector norm is replaced by the Frobenius norm). Recall that

$$\Phi = \mathbf{C}\Psi,$$

where Ψ transforms the basis and $\mathbf{C}$ is the measurement matrix (corresponding to going from the full image to the subsampled image). Note that the vector $\theta \in \mathbb{R}^D$ characterizes the coefficients in the *sparse* basis (discrete cosine or wavelet), and Ψ is therefore the **inverse** transform that transforms the coordinates back to the pixel space (i.e. the inverse cosine transform and inverse wavelet transform, respectively). On the other hand, $\Psi^\top$ is the **forward** transform. The l^1 norm acts as a proxy for the l^0 pseudo-norm and promotes sparsity of the coefficient vector θ. Note that $F(\theta)$ is smooth and strictly convex in θ; however, $G(\theta)$ is not differentiable wherever a component $\theta_d = 0$ for $d = 1, \dots, D$. One possible way to solve the minimization problem above would be to apply the **subgradient method**, which extends the gradient descent method to non-differentiable problems. Specifically, one iteratively updates θ as

$$\theta_{t+1} = \theta_t - \eta\left(2\Phi^\top(\Phi\theta_t - \mathbf{y}) + \lambda\mathbf{z}\right).$$

Here, $t = 1, 2, \dots$ denotes the iteration step, $\eta > 0$ is the step size, the first term in the brackets is simply $\nabla F(\theta)$, and $\mathbf{z} = (z_d)_{d=1}^D \in \mathbb{R}^D$ is the subdifferential of the l^1 norm of $_t$ (written as $\mathbf{z} = \partial\|\theta_t\|_1$), i.e.

$$z_d = \begin{cases} 1, & (\theta_t)_d > 0, \\ -1, & (\theta_t)_d < 0, \\ [-1, 1], & (\theta_t)_d = 0. \end{cases}$$

Note that at the points of non-differentiability, z_d is *set-valued*, and any line that touches or lies below the absolute value function (i.e. has a slope between -1 and 1) is a subderivative. However, the non-smoothness of the l^1 norm yields very slow convergence of order $\mathcal{O}(1/\sqrt{t})$ with respect to the iteration step t for the subgradient method, for which reason we will implement a more elaborate method in this lab session.

A significantly better method is given by the Proximal gradient method, which in the case of our Lasso objective is known as the **iterative soft thresholding algorithm (ISTA)**. The following brief introduction is quite mathematical, but don't worry too much about the details - the important take-away is the resulting iterative scheme that can be implemented by following the **implementation recipe** below. For more background on ISTA and extensions thereof, see e.g. Beck & Teboulle 2009 or this video.

For a convex (but not necessarily differentiable) function $G : \mathbb{R}^D \to \mathbb{R}$ (such as $G(\theta) = \|\theta\|_1$ in our case), one can define the **proximal operator** $\text{prox}_R : \mathbb{R}^D \to \mathbb{R}^D$ as

$$\text{prox}_G(\mathbf{u}) = \text{argmin}_\theta \ G(\theta) + \frac{1}{2}\|\mathbf{u} - \theta\|_2^2.$$

The proximal operator can be viewed as a projection, and the rough idea is to replace the non-smooth function G by a local smooth approximation. One can show that $\theta^\star$ is a minimizer of the problem

$$F(\theta) + G(\theta) \to \min.$$

if and only if it satisfies the following **fixed-point equation**:

$$\theta^\star = \text{prox}_{\gamma G}\left(\theta^\star - \gamma \nabla F(\theta^\star)\right),$$

where $\gamma > 0$ can be any positive real number. Here, $F : \mathbb{R}^D \to \mathbb{R}$ needs to differentiable with Lipschitz-continuous gradient and convex. These requirements are satisfied for

$$F(\theta) = \|\mathbf{y} - \Phi\theta\|_2^2$$

in our Lasso optimization problem. An alternative characterization of the proximal operator can be obtained by extending the idea of "extreme point $\Leftrightarrow$ gradient vanishes" for convex functions to the subdifferential:

$$\begin{aligned}
&\mathbf{u} = \text{prox}_G(\theta) && (1)\\
\Leftrightarrow\ &\mathbf{0}^\top \in \partial\left(G(\mathbf{u}) + \frac{1}{2}\|\mathbf{u} - \theta\|_2^2\right) && (2)\\
\Leftrightarrow\ &\mathbf{0}^\top \in \partial G(\mathbf{u}) + \mathbf{u} - \theta && (3)\\
\Leftrightarrow\ &\theta - \mathbf{u} \in \partial G(\mathbf{u}). && (4)
\end{aligned}$$

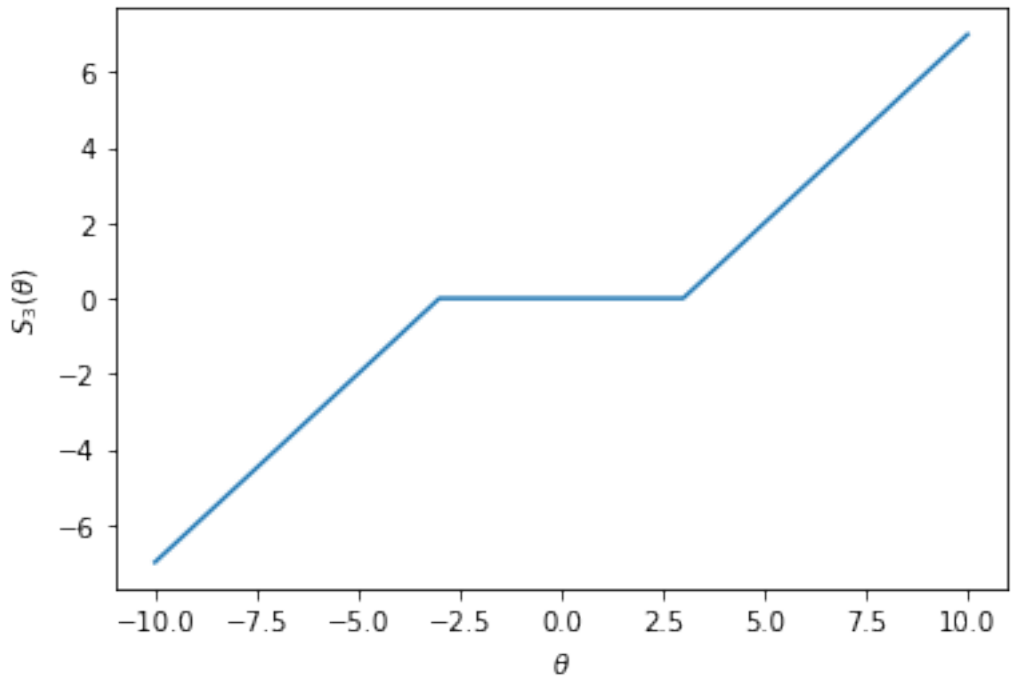
For Lasso, we are interested in the specific choice $G(\theta) = \|\theta\|_1$. We have already computed the subdifferential $\partial\|\theta\|_1$ of the l^1 norm above. Therefore, we can now compute $\text{prox}_{\gamma G}$ that is required in the fixed-point equation above:

$$\left(\text{prox}_{\gamma G}(\theta)\right)_d = \begin{cases} \theta_d - \gamma, & \theta_d > \gamma \\ 0, & |\theta_d| \leq \gamma \\ \theta_d + \gamma, & \theta_d < -\gamma. \end{cases}$$

This operator is known as the **soft thresholding operator**, denoted as $S_\gamma(\theta) := \text{prox}_{\gamma\|\cdot\|_1}(\theta)$. It looks as follows (shown here for $\gamma = 3$):

[4]:
```
theta_vec_ = np.linspace(-10, 10, 101, endpoint=True)
gamma_ = 3.0
S_gamma_ = np.sign(theta_vec_) * (np.abs(theta_vec_) - gamma_) * ((np.
 ↪abs(theta_vec_) - gamma_) > 0)
plt.plot(theta_vec_, S_gamma_)
plt.xlabel(r"$\theta$")
plt.ylabel(r"$S_3(\theta)$")
```

[4]:
```
Text(0, 0.5, '$S_3(\\theta)$')
```



Now, we can go back to the fixed-point problem for Lasso, which becomes

$$\theta^\star = \text{prox}_{\gamma G}\left(\theta^\star - \gamma \nabla F\left(\theta^\star\right)\right) = S_\gamma\left(\theta^\star - \gamma \nabla F\left(\theta^\star\right)\right).$$

This yields the fixed-point iteration scheme

$$\theta_{t+1} = S_\gamma\left(\theta_t - \gamma \nabla F(\theta_t)\right) = S_\gamma\left(\theta_t - 2\gamma \Phi^\top\left(\Phi\theta_t - \mathbf{y}\right)\right).$$

Since we started off with a free regularization parameter $\lambda > 0$ weighting the importance of $G(\theta)$, we can choose a different constant for the $\nabla F(\theta)$ term, say η, from which we arrive at the final formulation of the ISTA method

$$\theta_{t+1} = S_\gamma\left(\theta_t - 2\eta \Phi^\top\left(\Phi\theta_t - \mathbf{y}\right)\right).$$

A more careful analysis actually shows that η is related to the Lipschitz constant of $\nabla F(\theta)$. We can also express the ISTA scheme in terms of the (flattened) image in pixel space $\mathbf{x}_t = \Psi\theta_t$ as

$$\begin{aligned}
\theta_{t+1} &= S_\gamma\left(\theta_t - 2\eta \Phi^\top\left(\Phi\theta_t - \mathbf{y}\right)\right) && (5)\\
&= S_\gamma\left(\theta_t - 2\eta \Psi^\top \mathbf{C}^\top\left(\mathbf{C}\Psi\theta_t - \mathbf{y}\right)\right) && (6)\\
&= S_\gamma\left(\Psi^\top \mathbf{x}_t - 2\eta \Psi^\top \mathbf{C}^\top\left(\mathbf{C}\Psi\theta_t - \mathbf{y}\right)\right) && (7)\\
&= \left(S_\gamma \circ \Psi^\top\right)\left(\mathbf{x}_t - 2\eta \mathbf{C}^\top\left(\mathbf{C}\mathbf{x}_t - \mathbf{y}\right)\right). && (8)
\end{aligned}$$

where we used in the third line that Ψ is orthogonal for the bases considered here and therefore $\Psi^\top = \Psi^{-1}$.

1.4 Implementation recipe

The last formulation is particularly suitable because we can directly implement it as follows: 1. Compute $\mathbf{C}\mathbf{x}_t - \mathbf{y}$, where $\mathbf{C}$ sets all the elements in $\mathbf{x}_t$ where `mask == False` to zero. 2. Compute $\mathbf{x}_t - 2\eta\mathbf{C}^\top(\mathbf{C}\mathbf{x}_t - \mathbf{y})$, where the adjoint measurement matrix $\mathbf{C}^\top$ increases the dimension from S to D (meaning that we go from a subsampled quantity to a "full" quantity by inserting zeros for all the pixels that were not sampled (i.e. where `mask == False`). If you neither reduce the dimensions of $\mathbf{y}$ nor those of $\mathbf{x}_t$ in Step 1 from the full length D to the number of sparse measurements S, but simply zero out the pixels that were not sampled, the shape of $\mathbf{C}\mathbf{x}_t - \mathbf{y}$ should already be correct, and $\mathbf{C}^\top$ can be ignored. 3. Apply the **forward** transform represented by the matrix $\Psi^\top$. For this purpose, the `ista_step` function should receive an object of class `DCT` or `DWT` as an input whose method `forward` should be called on the correct input, e.g. `trafo = DCT(shape)`, and inside the function call `trafo.forward(...)`. 4. Finally, apply the **soft threshold** S_γ to obtain θ_{t+1}. 5. Then, transform θ_{t+1} back to pixel space and return $\mathbf{x}_{t+1}$ (or possibly instead of a flattened matrix a 2D matrix $\mathbf{X}_{t+1}$) by applying the `inverse` method of `DCT` and `DWT`, respectively.

Note: small letters derived from capital letters (e.g. $\mathbf{y}$ and $\mathbf{Y}$) stand for the flattened version of the image matrix. Alternatively, you can keep everything as 2D images throughout. Also, the `DCT` and `DWT` objects expect **images** as inputs, so if you flatten the images, make sure to reshape them to images again before this step.

Your task will be to implement this iteration scheme below. It can be shown that ISTA converges with order $\mathcal{O}(1/t)$, in comparison to $\mathcal{O}(1/\sqrt{t})$ for the subgradient descent introduced above. For completeness, let us mention that there is a modification of ISTA - known as fast ISTA (FISTA) - that even achieves a convergence order of $\mathcal{O}(1/t^2)$ by harnessing the idea of Nesterov momentum.

1.5 ISTA implementation

TODO: Write a function that implements the soft threshold S_γ.

[ ]:
```
def thresh(v, gamma):
    ...
```

TODO: Now, write a function `ista_step` that executes a single iteration step.

[ ]:
```
def ista_step(Psi, Xt, Y, Thresh, mask, eta=1.0):
    """
    Execute a single ISTA iteration step.
    :param Psi: object of class DCT / DWT
    :param Xt: reconstructed image X at current step t
    :param Y: subsampled image
    :param Thresh: threshold function
    :param mask: mask that is True at pixels subsampled in Y and False else
    :param eta: learning rate for the gradient term
    :return: reconstructed image X at next step t + 1
    """
    # 1. Compute C Xt - Y, where C Xt is identical to Xt at the subsampled␣
 ↪pixels and 0 everywhere else
    ...
```

```
    # 2. Compute update Xt - 2 eta C.T (C Xt - Y)
    ...

    # 3. Apply the forward transform
    ...

    # 4. Soft thresholding
    ...

    # 5. Convert the new theta to pixel space
    ...

    # Return reconstructed image at step t + 1
    ...
```

TODO: Now, reconstruct the galaxy image by iteratively running the ISTA scheme. Fill in the missing parts in the code below. We will use a **schedule** of soft thresholds γ, varying them as a function of the iteration step t, i.e. $\gamma = \gamma_t$. This leads to faster convergence.

[ ]:
```
# Set transformation
psi_op_dwt = DWT(shape)
psi_op_dct = DCT(shape)

# Set iteration settings
N_it = 5000  # number of iterations
print_freq = 500
eta = 0.75  # learning rate

# Schedule for the threshold values gamma (one value per iteration)
gammas = np.linspace(25, 4, N_it, endpoint=True)

# Initial guess: subsampled image Y
X_rec = Y

# Start iterating
# Choose either psi_op_dwt or psi_op_dct for now
for it in tqdm(range(N_it)):
    # Set the current soft threshold function
    ...
    # Perform an ISTA step
    ...
    # Clip the values of the reconstruction to [0, 255]
    ...

    if it % print_freq == 0:
        # Compute relative error and print
```

```
        rel_error = ...
        print("Relative error: {:3.3f}".format(rel_error))
```

TODO: Plot the original image $\mathbf{X}_{orig}$, the noisy image $\mathbf{X}_{noisy}$, the reconstructed image $\mathbf{X}_{rec}$, and the subsampled image $\mathbf{Y}$ next to each other.

[ ]:
```
# Make a plot
...
```

TODO: Finally, please answer the following questions: 1. Extend the method `ista_step` to also output the number of non-zero coefficients in θ_{t+1}. How many coefficients (and what percentage) of the converged solution $\theta^\star$ are non-zero? 2. How big is the relative error between the reconstruction and the original image? 3. How do the results for the wavelet basis and the discrete cosine basis compare? 4. What happens to the reconstruction error and the sparsity of $\theta^\star$ when you *alternate* between steps with the cosine basis and the wavelet basis?