Assignment 1: PCA

What you will learn

- Working with images using scikit-image
- PCA using scikit-learn
- Practical applications of PCA

Setup

- Download [Anaconda Python 3.6 (https://www.anaconda.com/download/)](https://www.anaconda.com/download/) for consistent environment.
- If you use pip environment then make sure your code is compatible with versions of libraries provided within Anaconda's Python 3.6 distribution.

Submission

- Do not change any variable/function names.
- Just add your own code and don't change existing code
- Save this file and rename it to be **studentid_lastname.ipynb** (student id (underscore) last name.ipynb) where your student id is all numbers.
- Export your .ipynb file to PDF (File > Download as > PDF via Latex). **Please don't leave this step for final minutes**.
- Submit both the notebook and PDF files.
- If you happen to use any external library not included in Anaconda (mention in **Submission Notes** section below)

Submission Notes

(Please write any notes here that you think I should know during marking)

[NO MARKS] PCA Warming Up (MUST READ)

I'm adding some code to illustrate examples of PCA using sklearn library.

Let's create some random `5d` data

In [1]:
```
import numpy as np
from sklearn.decomposition import PCA

# 100 points of 5d data
data = np.random.rand(100, 5)
```

Lets convert this `5d` data to `2d` using PCA`

In [2]:
```
# n_components=2 because I want to convert 5d data to 2d (dimensionality
 reduction)
pca = PCA(n_components=2)
pca.fit(data)
```

Out[2]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=Non
e,
  svd_solver='auto', tol=0.0, whiten=False)
```

In code above when we call `fit`, it populates two things in `pca`:

1. `mean_`
2. `components_`

In [3]:
```
# mean of the input data (per dimension) used to zeroying the mean
pca.mean_
```

Out[3]:
```
array([0.54778449, 0.54280291, 0.55500368, 0.50007919, 0.46390401])
```

In [4]:
```
# basis vectors
pca.components_
```

Out[4]:
```
array([[-0.4676832 , -0.31501925, -0.05788727, -0.55570976,  0.6081702
3],
       [-0.54636213,  0.34014575,  0.39129009,  0.57528807,  0.3189435
6]])
```

Now we are ready to transform `5d` from `data` into `2d` using following code

In [5]:
```
data_to_reduce = data[:10]
reduced_data = np.dot(data_to_reduce - pca.mean_, pca.components_.T)

# reduced data from 5d to 2d
reduced_data.shape
```

Out[5]:
```
(10, 2)
```

You can accomplish the same using `transform` function provided in `pca`

In [6]:
```
pca.transform(data_to_reduce).shape
```

Out[6]:
```
(10, 2)
```

Time for inverse transform or changing `2d` data back to `5d`

Compression --> Decompression

In [7]:
```
decompressed_data = np.dot(reduced_data, pca.components_)+pca.mean_
decompressed_data
```

Out[7]:
```
array([[0.11268137, 0.5292387 , 0.68228227, 0.466384  , 0.87516734],
       [0.7603341 , 0.48098921, 0.4484759 , 0.39819963, 0.30084041],
       [0.55407687, 0.67735918, 0.64023356, 0.73288934, 0.38366914],
       [0.82195761, 0.454136  , 0.41180261, 0.35321804, 0.25850254],
       [0.47939664, 0.63278575, 0.63470282, 0.65405938, 0.47761484],
       [0.38519453, 0.42707774, 0.53085565, 0.29615115, 0.67876708],
       [0.55501762, 0.39380899, 0.45618786, 0.24262316, 0.5395701 ],
       [0.93990411, 0.33789436, 0.29958842, 0.15500011, 0.21332105],
       [0.45572109, 0.51196045, 0.56380734, 0.44458237, 0.566389  ],
       [0.68642325, 0.81049071, 0.68511959, 0.96620297, 0.18724737]])
```

In [8]:
```
# same can we accomplished using inverse_transform
pca.inverse_transform(pca.transform(data_to_reduce))
```

Out[8]:
```
array([[0.11268137, 0.5292387 , 0.68228227, 0.466384  , 0.87516734],
       [0.7603341 , 0.48098921, 0.4484759 , 0.39819963, 0.30084041],
       [0.55407687, 0.67735918, 0.64023356, 0.73288934, 0.38366914],
       [0.82195761, 0.454136  , 0.41180261, 0.35321804, 0.25850254],
       [0.47939664, 0.63278575, 0.63470282, 0.65405938, 0.47761484],
       [0.38519453, 0.42707774, 0.53085565, 0.29615115, 0.67876708],
       [0.55501762, 0.39380899, 0.45618786, 0.24262316, 0.5395701 ],
       [0.93990411, 0.33789436, 0.29958842, 0.15500011, 0.21332105],
       [0.45572109, 0.51196045, 0.56380734, 0.44458237, 0.566389  ],
       [0.68642325, 0.81049071, 0.68511959, 0.96620297, 0.18724737]])
```

In [9]:
```
# Lets find compression decompression error (absolute mean error)
np.sum(np.abs(data_to_reduce - decompressed_data))/data_to_reduce.size
```

Out[9]:
```
0.13884886747264605
```

Questions [8 marks]

Answer the questions below as follows:

1) What is 2+2

- 4
- 5
- 6

In [10]:
```
ans1 = 4
ans1
```

Out[10]:
```
4
```

2) (2 marks) For a n-D data, you can ALWAYS reconstruct the data with 0\% error if all `n` PCAs are used.

- True
- False

In [11]:
```
ans2 = True
ans2
```

Out[11]:
```
True
```

3) (2 marks) From the 2nd tutorial, we ran PCA alorithm on faces. We called the extracted PCs--Eigenfaces. What is the value of a dot product between arbitary two eigen faces?

In [12]:
```
ans3 = 0
ans3
```

Out[12]:
```
0
```

4) (4 marks) Using the probablity, find the expected value for the function below:

Note: Lets say for a coin toss, head = 0 and tail = 1. Then the expected value for a coin-toss will be `p(x=tail)*1 + p(x=head)*0 = 1/2*1 = 0.5`.

In [13]:
```
import operator as op
from functools import reduce

def func():
    arr = [49, 8, 48, 15, 47, 4, 16, 23, 43, 44, 42, 45, 46]
    np.random.shuffle(arr)
    print(arr[0:6])
    return min(arr[0:6])

def ncr(n, r):
    r = min(r, n-r)
    numer = reduce(op.mul, range(n, n-r, -1), 1)
    denom = reduce(op.mul, range(1, r+1), 1)
    return numer / denom

# Sorted: [4, 8, 15, 16, 23, 42, 43, 44, 45, 46, 47, 48, 49]
# ie. 4 is picked as min in the total set of 13
# only 12 more numbers higher than 4 exist and only need to choose 5 mor
e numbers
# because the array is length 6 and one num (the min) was already chosen

# additionally, only need account up to 44 because if 44 is the min the
 5 numbers above it
# would create the rest of the subset of six numbers

# expected value is calculated by taking the value of the discrete varia
ble chosen
# and multiplying it by the probability of it being chosen
top = 4*ncr(12,5)+8*ncr(11,5)+15*ncr(10,5)+16*ncr(9,5)+23*ncr(8,5)+42*nc
r(7,5)+43*ncr(6,5)+44*ncr(5,5)
bot = ncr(13, 6)
res = top / bot
res
```

Out[13]: 8.818181818181818

Programming Tasks [92 marks]

Task 1: Building an Image Compression Algorithm

In this section you will build you own compression algorithm for images using PCA.

STEP 1: Read the image

1. Use `imread` function from `skimage.io` to read `leena.jpg`.
2. Show the image using `show_image` function provided to you

In [14]:

```
import matplotlib.pyplot as plt
# make matplotlib to show plots inline
%matplotlib inline

from skimage.io import imread

def show_image(img):
    if len(img.shape) == 2:
        plt.imshow(img, cmap='Greys_r')
    else:
        plt.imshow(img)
    plt.axis('off')
    plt.title('Image Dimension: {0},{1}'.format(img.shape[0], img.shape[
1]), fontsize=20)

# !!ADD CODE HERE!!
image = imread('leena.jpg')
#
show_image(image)
```

Image Dimension: 350,780

STEP 2: Compressing an image using PCA

Image is generally made of 3 (or 4) channels (RGB), we will build a compression algorithm that applies one channel at a time to compress multi-channel image.

In order to compress entire image you will compress all the channels within the image one by one and then serialize compressed channels and any auxiliary data required for decompression (for ex. principle components (components_), means (means_), original image size).

In order to decompress, you will deserialize the data, then uncompress the compressed channel one by one and stack them up to rebuild the uncompressed version of the original image.

Compression Strategy using PCA

- The above image you have read is of size `350 x 780` , i.e. width is `780` and height is `350`
- Patch the image into `10x10` patches yielding `35x78=2730` patches in total.
- Flatten each patch `(10x10)` into `100` dimensional vector.
- Now (for each channel) you will have `2730` number of `100-d` vectors.
- Apply PCA on these vectors.
- Reduce the dimensionality of `100-d` vector to `5-d` .

I've given you two functions `patchify` and `depatchify` .

`patchify` creates `100-d` vectors from all the patches from a given image and `depatchify` combines these `100-d` patches back to the image of the given size.

Please read through code below and figure out how these two functions work.

NOTE: `convert_to_cf` is important function to note (in cell below). It converts channel last format image to channel first format. Images that you read through `imread` function returns array of shape `X x Y x 3` channel is last axis, it is easier if channel were first axis then to extract any channel you can do use first indexer.

In [15]:
```
from sklearn.decomposition import PCA
import numpy as np


def patchify(img, ps=(10, 10)):
    patches = []
    h, w = img.shape
    for i in range(0, h, ps[0]):
        for j in range(0, w, ps[1]):
            patches.append(img[i:i+ps[0], j:j+ps[1]].ravel())
    return np.array(patches)


def depatchify(patches, patch_size=(10, 10), img_size=(350, 780)):
    # normalize
    patches[patches > 255.] = 255.
    patches[patches < 0.] = 0.

    # convert to unint8
    patches = patches.astype('uint8')
    rec_img = np.zeros(img_size, dtype='uint8')
    ph, pw = patch_size

    h, w = img_size
    x = 0
    for i in range(0, h, ph):
        for j in range(0, w, pw):
            rec_img[i:i+ph, j:j+pw] = patches[x].reshape((ph, pw))
            x += 1

    return rec_img

def convert_to_cf(img_cl):
    # convert image to channel first
    img_cf = np.swapaxes(img_cl.T, 1, 2)
    return img_cf

image=imread('leena.jpg')
img_cf = convert_to_cf(image)

# I'll patchify each channel and depatchify them
# Then I'll stack them together to create original image back
ch1 = depatchify(patchify(img_cf[0])) # first channel
ch2 = depatchify(patchify(img_cf[1])) # second channel
ch3 = depatchify(patchify(img_cf[2])) # third channel

# combine them now
rec_img = np.dstack((ch1, ch2, ch3))
plt.figure(figsize=(12, 12))
show_image(rec_img)
```

STEP 3: Compressing single channel of an image (FILL TWO FUNCTIONS BELOW)

Now you are familiar with how `patchify` and `depatchify` work. Do the following:

1. Write a function `compress` that will take one of the channel of the image as input and outputs compressed channel and auxiliary data required for decompressing. Return type of this function should be dictionary.
2. Write another function `decompress` that will take whatever dictionary data you returned from previous `compress` function and decompresses it into image channel (that was compressed).
3. PCA compression is lossy compression algorithm -- means you will loose the information during decompression.
4. I should be able to call two of your functions like so `decompress(compress(img_ch[0]))` to compress and decompress the given image's channel.

Compress: Pseudo code

- `Patchify` the given image's channel (input).
- Run `PCA` on the patches (you may use sklearn's implementation) to reduce them to `5d` vectors.
- You will need `basis vectors` or `principal components` and `mean` in order to reconstruct the data back to `100d`
- Return dictionary: `{'compressed_patches': 5d vectors, 'aux_data': principal components/basis vectors/means/final size of image}`
- By converting `100d` to `5d`, you reduce size by `20` times.

Decompress: Pseudo code

- Input is dictionary as returned by your `compress` function.
- Use `5d` vectors and do inverse PCA (check tutorial 2) and convert them back to `100d` vectors.
- Use `depatchify` function to convert these reconstructed `100d` vectors into an image channel

Tip: Good code is always modular and easy to read.

In [16]:
```
# COMPLETE FOLLOWING FUNCTIONS
from sklearn.decomposition import PCA

def compress(img_ch, n_components=5):
    """
    Inputs
        img_ch: one of the channel of given image
        n_components: number of components returned by PCA
    Returns
        comp_data: Some data structure (may be dict.) that represents co
mpressed form of given input
        along with auxiliary data required for decompression (components
_, mean_ and shape of input image)
    """
    # patchify img_ch
    patches = patchify(img_ch)
    # ADD CODE HERE!!

    # 1) Get PCA components and means
    pca = PCA(n_components)
    pca.fit(patches)

    # 2) Compress the patches
    y = pca.transform(patches)

    # 3) Return data
    aux_data = [
        pca.components_, # basis vecs
        pca.mean_, # mean
        img_ch.shape # shape of image
    ]
    #
    return {'y': y, 'aux_data': aux_data }
    pass

def decompress(comp_data):
    """
    Inputs
        comp_data: data structure that is returned by `compress` functio
n
    Returns
        img_ch: decompressed form of channel compressed and contained in
side `comp_data` data structure
    """

    # 1) Get compressed data, y, and the aux_data
    y = comp_data['y']
    components, means, img_size = comp_data['aux_data']

    # 2) recontruct the patches to 100d using aux_data and inverse PCA
    rec_patches = np.dot(y, components) + means

    # 3) depatchify and return img_ch
    img_ch = depatchify(rec_patches)
    return img_ch
```

```
# Red channel
# visualize your compression and decompression
img_ch = img_cf[0]
plt.figure(figsize=(10, 10))
show_image(img_ch)

plt.figure(figsize=(10, 10))
show_image(decompress(compress(img_ch, n_components=5)))
```

STEP 3: Compress and decompress entire image (FILL TWO MORE FUNCTIONS)

Write a `compress_image` function that:

- takes `channel last` (regular image read from `imread`) representation of an image
- convert `channel last` to `channel first` representation using `convert_to_cf` function defined previously
- compresses each of the channel using `compress` function
- outputs a one dictionary that contains all auxiliary data required to reconstruct/decompress entire image back.

Similarly, write `decompress_image` function that:

- decompresses the image compressed by `compress_image` function
- return `channel last` image

NOTE: You would patchify each channel and implement PCA on each channel thus for decompression you need `components_` , `mean_` for each channel and you need shape of input image as well.

In [17]:
```
def compress_image(img, n_components=5):
    """
    Inputs:
        img_cf: `Channel last` image data
    """
    img_cf = convert_to_cf(img)
    return {
        '1': compress(img_cf[0], n_components),
        '2': compress(img_cf[1], n_components),
        '3': compress(img_cf[2], n_components)
    }

def decompress_image(comp_img):
    """
    Returns:
        img_rec: Decompressed `channel last` image
    """
    ch1 = decompress(comp_img['1'])
    ch2 = decompress(comp_img['2'])
    ch3 = decompress(comp_img['3'])
    return np.dstack((ch1,ch2,ch3))

plt.figure(figsize=(10, 10))
show_image(
    decompress_image(
        compress_image(image)
    )
)
```

STEP 4: Serialization and de-serialization

You can easily (de)serialize dictionary using `np.save` and `np.load` (see example below)

`compress_and_serialize` should:

- Read image given by `inp_path`
- Use `compress_image` to compress it
- Serialize the compressed data using `np.save` to a file specified by `out_path`

NOTE: All the data required for deserialization must be saved to a one single file only.

`deserialize_and_decompress` should:

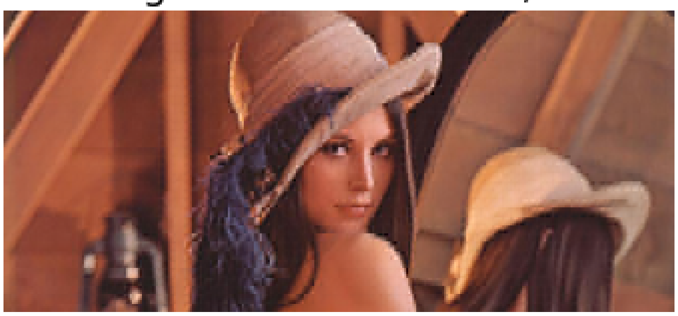
- Read the file specified by `inp_path` using `np.load`
- Use `decompress_image` function to decompress it
- Return image (channel last as returned by `decompress_image`) function

In [18]:
```
# EXAMPLE OF SAVING AND LOADING DICTIONARY OBJECT TO/FROM FILESYSTEM
d = {'a': [1, 2, 3], 'b': [4, 5, 6, 7, 8]}
np.save('example.npy', d)
ds = np.load('example.npy').item()

ds
```

Out[18]:
```
{'a': [1, 2, 3], 'b': [4, 5, 6, 7, 8]}
```

In [19]:
```
import os
# COMPLETE THESE TWO FUNCTIONS
def compress_and_serialize(inp_path='leena.jpg', out_path='output.bin'):
    image = imread(inp_path)
    comp_img = compress_image(image)
    np.save(out_path, comp_img)
    # rename the np
    os.rename(out_path+".npy", out_path)

def deserialize_and_decompress(inp_path='output.bin'):
    compressed = np.load(inp_path).item()
    return decompress_image(compressed)

compress_and_serialize()
show_image(deserialize_and_decompress())
```

Image Dimension: 350,780



STEP 5: What is size of output.bin file?

Did you end of in compressing anything? Conclude your experiments!

Conclusion

Write your conclusion here!!

- the compressed bytes is higher than the original image, but this is because output.bin has all the metadata
- the compressed image size should be compared instead of the .bin file

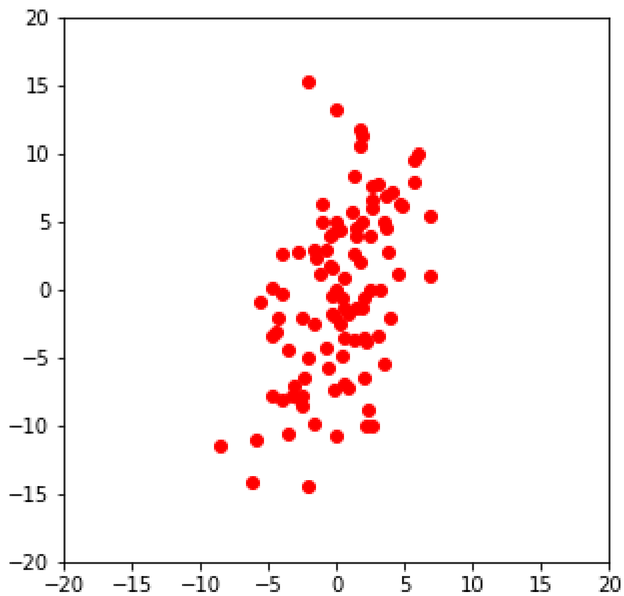
In [20]:
```
import os

print('Original', os.path.getsize('leena.jpg'), 'bytes')
print('Compressed', os.path.getsize('output.bin'), 'bytes')
```

```
Original 35740 bytes
Compressed 498280 bytes
```

Task 2: Rotation and Translation Invariance in PCA

STEP 1 (done already): Create normally distributed data

In [21]:
```
mean = [0, 0]
cov = [[10, 10], [10, 40]]  # diagonal covariance
x, y = np.random.multivariate_normal(mean, cov, 100).T
X = np.array(list(zip(x, y)))

def plot_data(X):
    plt.figure(figsize=(5, 5))
    plt.plot(X[:, 0], X[:, 1], 'ro')
    plt.xlim([-20, 20])
    plt.ylim([-20, 20])
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

plot_data(X)
```

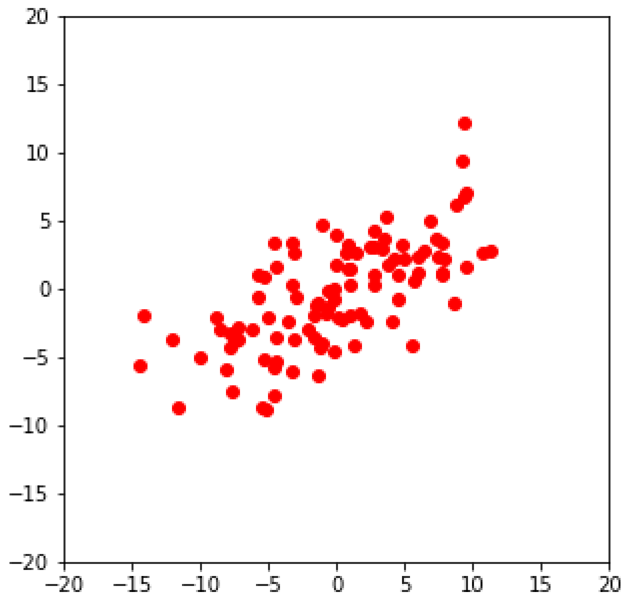


STEP 2: Rotate the data by 45 degrees and create new array X_rot

Write code to rotate X by 45

- Use rotation matrix
- Or individually rotate each point in `X` by 45 degrees
- Use `plot_data` function defined in cell above to visualize the rotated data

In [22]:
```
theta = np.radians(45)
# WRITE CODE HERE
# Code to rotate X
c, s = np.cos(theta), np.sin(theta)
R = np.array(((c,-s), (s, c)))
# populate new matrix X_rot
X_rot = X.dot(R)
plot_data(X_rot)
```



STEP 3 (done but need explanation): Perform pca with n_components=2

Do you see anything interesting?

Explain the code below and write down your observations (2-3 lines only).

Observations

- pca fit_transform of X is the same as X_rot
- pca is a rotational invariant

In [23]:

```
import pandas as pd

def visualize_components(X, X_rot):
    pca = PCA(n_components=2)

    pca.fit(X)
    df1 = pd.DataFrame(pca.fit_transform(X))
    df2 = pd.DataFrame(pca.fit_transform(X_rot))

    return pd.concat((df1, df2), axis=1)[:20]

visualize_components(X, X_rot)
```

Out[23]:

	0	1	0	1
0	5.504224	-3.355173	5.504224	-3.355173
1	-3.005763	-6.036108	-3.005763	-6.036108
2	-2.203019	3.725383	-2.203019	3.725383
3	-5.549313	-2.025345	-5.549313	-2.025345
4	-3.981244	1.758984	-3.981244	1.758984
5	7.939445	0.652354	7.939445	0.652354
6	-9.341564	-3.055980	-9.341564	-3.055980
7	0.929962	-0.591284	0.929962	-0.591284
8	0.677020	-3.966748	0.677020	-3.966748
9	-5.069111	0.140076	-5.069111	0.140076
10	-3.876856	-2.599697	-3.876856	-2.599697
11	-7.850321	-1.374726	-7.850321	-1.374726
12	0.706440	-1.476864	0.706440	-1.476864
13	2.391178	2.181455	2.391178	2.181455
14	-1.792900	1.169575	-1.792900	1.169575
15	4.287491	-1.385176	4.287491	-1.385176
16	2.883898	-2.829643	2.883898	-2.829643
17	-10.914111	1.405779	-10.914111	1.405779
18	-8.255857	-1.699187	-8.255857	-1.699187
19	-0.833272	-2.108724	-0.833272	-2.108724

STEP 4: Perform PCA again and find angle between principal components

- Perform PCA on X and X_rot with n_components=1
- It will give one basis vector for each of data (X and X_rot)
- Find the angle between these two basis vectors
- Explain your observations?

Observations

- the two basis vectors are 45 degrees to each other
- this is because the X_rot is 45 degrees rotation of the X
    - the original 45 degrees rotation is reflected in the basis vectors

In [24]:
```
import math
from numpy.linalg import norm

def angle_between(a,b):
    # COMPLETE THIS FUNCTION
    # CALCULATE ANGLE IN RAD BETWEEN TWO VECTORS
    dotprod = np.dot(a,b)
    lengthA = norm(a)
    lengthB = norm(b)
    return math.acos(dotprod / (lengthA*lengthB))

pca = PCA(n_components=1)
pca.fit(X)
c1 = pca.components_

pca.fit(X_rot)
c2 = pca.components_

np.rad2deg(angle_between(c1[0], c2[0]))
```

Out[24]:
```
45.0
```

(NO MARKS) STEP 5: Repeat these experiments for translation as well

There is not marks for this part. You can do this for your own learning.

Now translate every point in `X` by fixed `x` and `y` amount.

```
X = X + [1, 2]
```

like so and repeat all the above experiments in cell below and write down your observations:

In [25]:
```
# PERFORM EXPERIMENTS WITH TRANSLATION HERE
# NO MARKS FOR DOING THIS
```

TASK 3: Recovery of corrupted images using PCA

Check the code below.

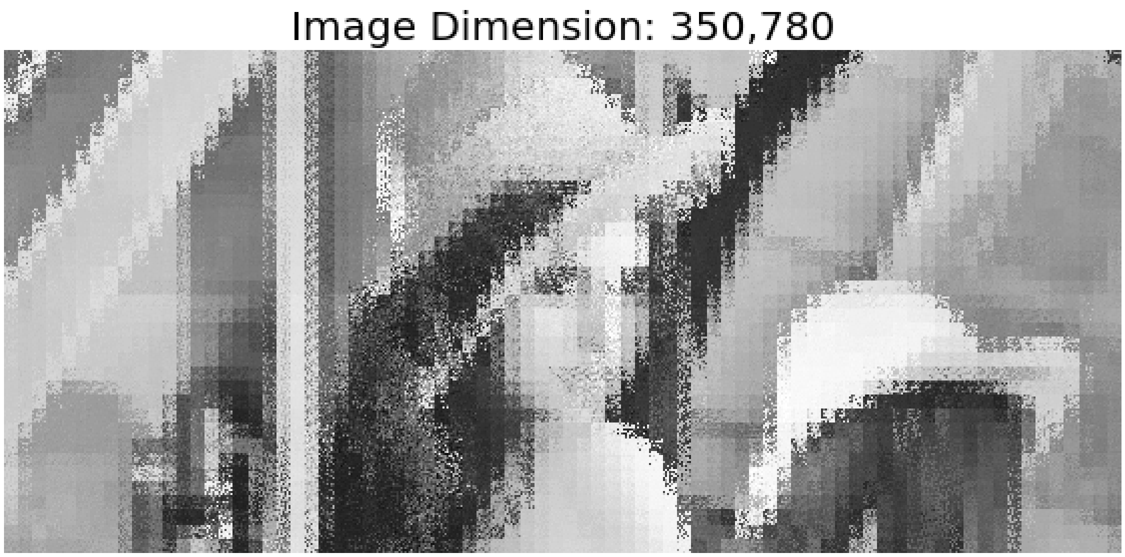
It corrupts the `leena.jpg` image that you worked on before.

Check very carefully what below code is doing on the image and answer:

Is it same as rotation of data points (like done before), if yes explain (just one liner)?

No, this corruption is random

In [26]:
```
# Code for corrupting the leena image
image = convert_to_cf(imread('leena.jpg'))[0]

def corrupt_image(img):
    # lets patchify R channel with patches of 35x78 patches
    patches = patchify(img)

    # Noise 1:
    # lets jumble pixels of patches now
    jumble_idx = list(range(len(patches[0])))
    np.random.shuffle(jumble_idx)

    jumbled_patches = np.array([patch[jumble_idx] for patch in patches])
    rec_jumbled_image = depatchify(jumbled_patches, img_size=img.shape)

    return rec_jumbled_image


cimage = corrupt_image(image)

plt.figure(figsize=(10,10))
show_image(image)

plt.figure(figsize=(10,10))
show_image(cimage)
```



Recovering from the corruption

Use the `compress` function you coded earlier to get compressed form of original leena image and corrupted leena image.

We will try to recover the corrupted leena given the compressed version of original leena.

Notice in code below, I replace `aux_data` of corrupted version with `aux_data` of original version.

Does the code below code work in recovering the original leena image back?

Explain how below code works and show it actually works?

Yes the code works in getting the original leena image back. It compresses the corrupted image and then replaces the corrupted image's aux_data with the original image's aux_data.

In [27]:
```
# This assuming your compress function returns dictionary which contains `aux_data`
# if you coded your compress in diferent way then change code below appropriately
# basically u need to replace auxiliary data of corrupted with original while keeping compressed (transformed) data same
d = compress(image, n_components=50)
d1 = compress(cimage, n_components=50)

plt.figure(figsize=(10, 10))
show_image(decompress(d1))

d1['aux_data'] = d['aux_data']
plt.figure(figsize=(10, 10))
show_image(decompress(d1))
```

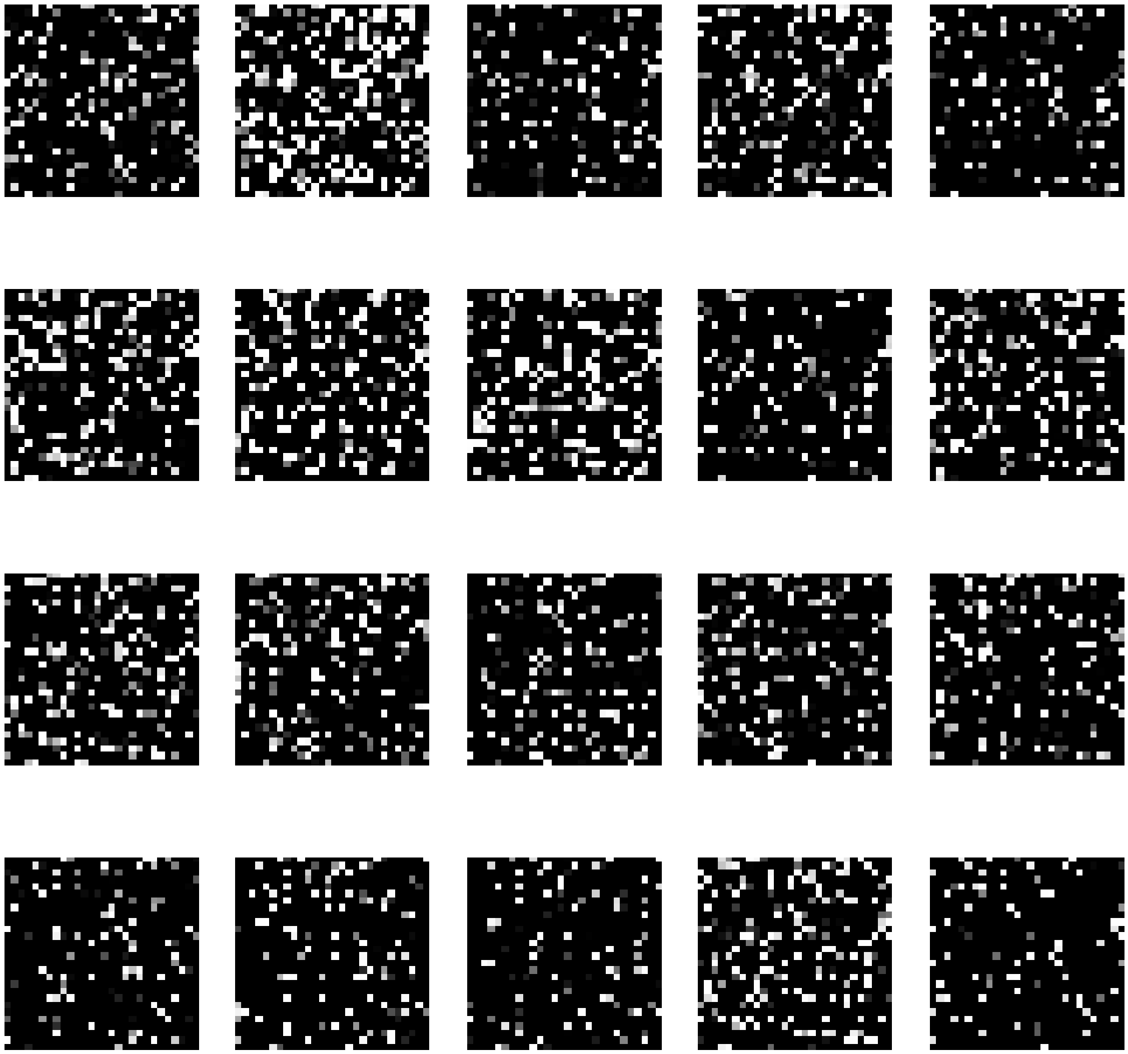
Task 4: Decrypting manuscript of the lost civilization

You belong to one of the advanced civilization, while exploring the universe you land on the planet Earth. However, there is no inhabitants on the planet anymore. While exploring Earth, you come across some damaged hard drive that contains various images. You suspect that these images form a manuscript of how "humans" use to write different digits in maths. You recovered all the data from the hard drive safely. You opened up your jupyter notebook (python being universal language and popular among alien species), you started plotting the images you just recovered.

In [28]:
```
rimages = np.load('recovered_images.npy')
rimages.shape
```

Out[28]: (20, 28, 28)

In [31]:
```
def plot_manuscript(images):
    for i, img in enumerate(images):
        plt.subplot(4, 5, i+1)
        plt.imshow(img, cmap='Greys_r')
        plt.axis('off')

plt.figure(figsize=(10, 10))
plot_manuscript(rimages)
```

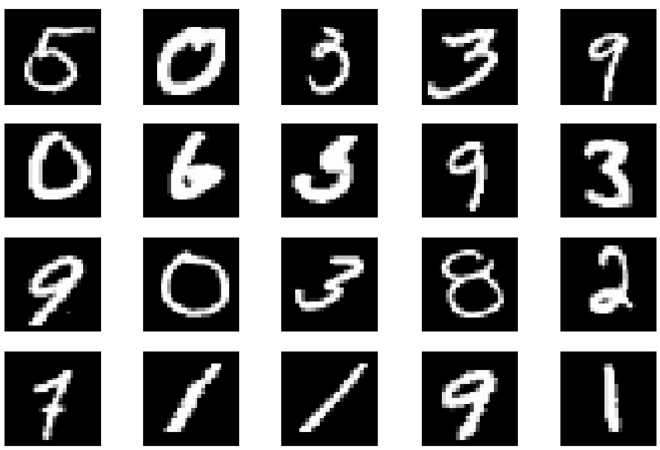


You being a smart alient, figured out that these images are encrypted and not in their original form. You also figured out that encryption is just fixed jumbling of the pixels of the images. Since you're unaware of the manuscript, you cannot decrypt the images just by themselves even if they follow the fixed jumbling pattern. However, fortunately you found the `principal components` and `mean` of the the original manuscript somewhere in the same harddisk. Now, your task is to recover all the 20 images and plot them nicely in cell below.

Use `plot_manuscript` function from above to plot the recovered manuscript.

In [34]:

```
original_components = np.load('original_pca.npy')
original_mean = np.load('original_mean.npy')

# Write your code here
# reshape to proper dimensions
rimages.shape = (20, 784)
# compress so dimensions match
pca = PCA(n_components = 20)
pca.fit(rimages)
transformed = pca.transform(rimages)

# decompress
res = np.dot(transformed, original_components) + original_mean

# set 20 images of 28x28 size
res.shape = (20, 28, 28)

# plot manuscript
plot_manuscript(res)
```



In [ ]:

```
```