Independent Component Analysis

In this exercise, you will implement an ICA algorithm similar to the FastICA method described in the paper "*A. Hyvärinen and E. Oja. 2000. Independent component analysis: algorithms and applications*" linked from ISIS, and apply it to model the independent components of a distribution of image patches.

In [1]:

```
import numpy as np
import matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import sklearn
import sklearn.datasets
import sklearn.feature_extraction.image
import utils
```

As a first step, we take a sample image, extract a collection of (8×8) patches from it and plot them.

In [2]:

```
I = sklearn.datasets.load_sample_image('china.jpg')
X = sklearn.feature_extraction.image.extract_patches_2d(I, (8,8), max_patches=10000, random_state=0
)
utils.showimage(I)
utils.showpatches(X)
```

As a starting point, the patches we have extracted are flattened to appear as abstract input vectors of $8 \times 8 \times 3 = 192$ dimensions. The input data is then centered and standardized.

In [3]:

```
X = X.reshape(len(X),-1)
X = X - X.mean(axis=0)
X = X / X.std()
```

In [4]:

```
X.shape
```

Out[4]:

```
(10000, 192)
```

Whitening (10 P)

A precondition for applying the ICA procedure is that the input data has variance 1 under any projection. This can be achieved by whitening, which is a transformation $\mathrm{W}: \mathbb{R}^d \to \mathbb{R}^d$ with $z = \mathrm{W}(x)$ such that $\mathrm{E}[zz^\top] = I$.

A simple procedure for whitening a collection of data points $x_1, \dots, x_N$ (assumed to be centered) first computes the PCA components $u_1, \dots, u_d$ of the data and then applies the following three consecutive steps:

1. project the data on the PCA components i.e. $p_{n,i} = x_n^\top u_i$.
2. divide the projected data by the standard deviation in PCA space, i.e. $\tilde{p}_{n,i} = p_{n,i} / \mathrm{std}(p_{:,i})$
3. backproject to the input space $z_n = \sum_i \tilde{p}_{n,i} u_i$.

Task:

- **Implement this whitening procedure, in particular, write a function that receives the input data matrix and returns the matrix containing all whitened data points.**

For efficiency, your whitening procedure should be implemented in matrix form.

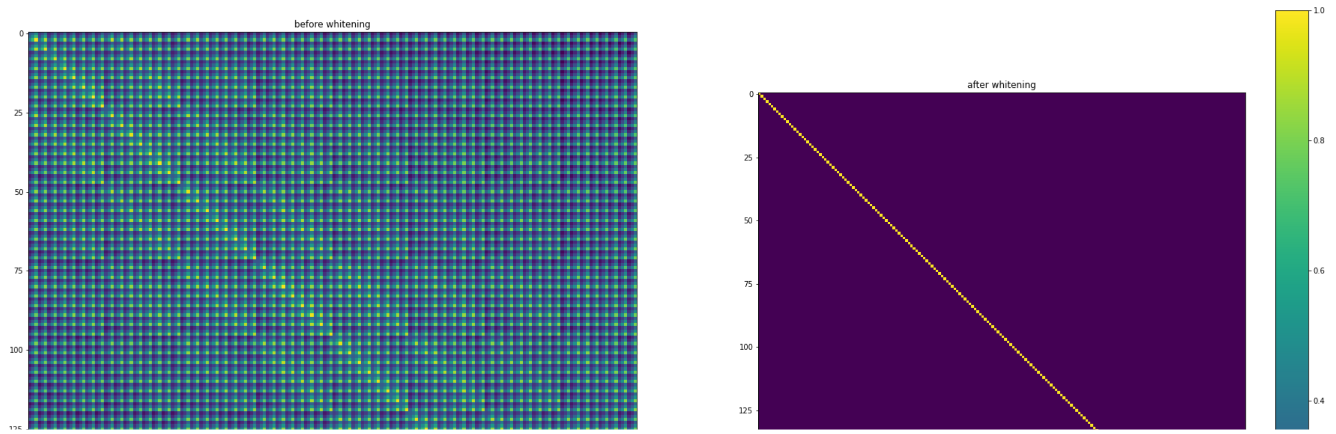
In [5]:

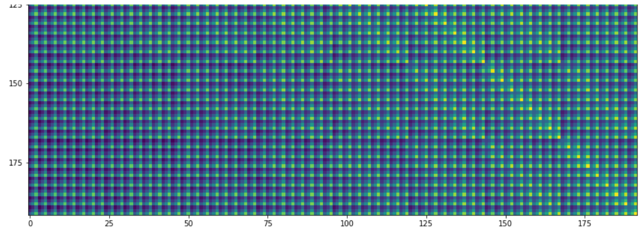
```
def whitening(X):
    N,D = X.shape
    cov =  (1/N)*X.T.dot(X)

    #U contains eigenvectors of cov as columnvectors
    #S contains eigenvalues in descending order
    U, S, _ = np.linalg.svd(cov)

    res =  X @ U @ np.linalg.inv(np.sqrt(np.diag(S))) @ U.T

    return res


Z = whitening(X)
```

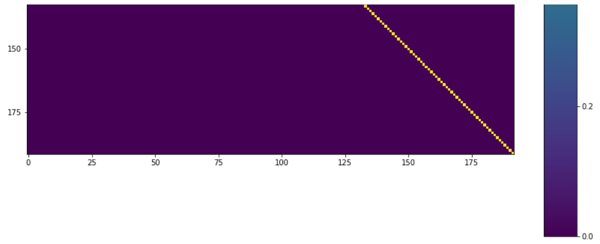
The code below verifies graphically that whitening has removed correlations between the different input dimensions:

In [6]:

```
f = plt.figure(figsize=(32,16))
p = f.add_subplot(1,2,1); p.set_title('before whitening')
p.imshow(np.dot(X.T,X)/len(X))
p = f.add_subplot(1,2,2); p.set_title('after whitening')
im = p.imshow(np.dot(Z.T,Z)/len(Z))
f.colorbar(im, ax=p)
plt.show()
```



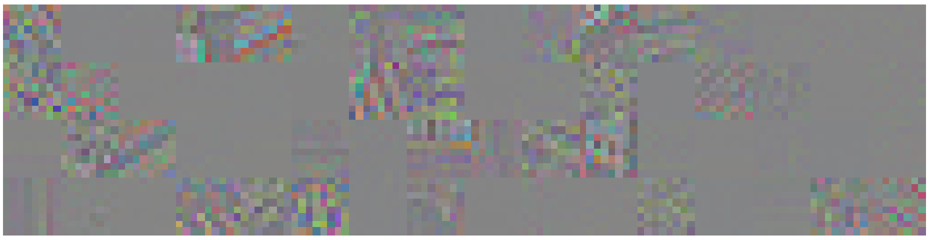




Finally, to get visual picture of what will enter into our ICA algorithm, the whitened data can be visualized in the same way as the original input data.

In [7]:

```
utils.showpatches(X)
utils.showpatches(Z)
```



We observe that all high constrasts and spatial correlations have been removed after whitening. Remaining patterns include high-frequency textures and oriented edges of different colors.

Implementing ICA (20 P)

We now would like to learn $h = 64$ independent components of the distribution of whitened image patches. For this, we adopt a procedure similar to the FastICA procedure described in the paper above. In particular, we start with random weights $w_1, \dots, w_h \in \mathrm{R}^d$ and iterate multiple times the sequence of operations: <

1. $\forall_{i=1}^{d}\ w_i = \mathrm{E}[x \cdot g(w_i^\top x)] - w_i \cdot \mathrm{E}[g'(w_i^\top x)]$
2. $w_1, \dots, w_h = \text{decorrelate}\{w_1, \dots, w_h\}$

where $\mathrm{E}[\,\cdot\,]$ denotes the expectation with the data distribution.

The first step increases non-Gaussianity of the projected data. Here, we will make use of the nonquadratic function $G(x) = \frac{1}{a}\log\cosh(ax)$ with $a = 1.5$. This function admits as a derivative the function $g(x) = \tanh(ax)$, and as a double derivative the function $g'(x) = a \cdot (1 - \tanh^2(ax))$.

The second step enforces that the learned projections are decorrelated, i.e.\ $w_i^\top w_j = 1_{i=j}$. It will be implemented by calling in an appropriate manner the whitening procedure which we have already implemented to decorrelate the different input dimensions.

This procedure minimizes the non-Gaussianity of the projected data as measured by the objective:

$$J(w) = \sum_{i=1}^{h} (\mathrm{E}[G(w_i^\top x)] - \mathrm{E}[G(\varepsilon)])^2 \qquad \text{where} \quad \varepsilon \sim \mathrm{N}(0, 1).$$

Task:

- **Implement the ICA procedure described above, run it for 200 iterations, and print the value of the objective function every 25 iterations.**

In order to keep the learning procedure computationally affordable, the code must be parallelized, in particular, make use of numpy matrix multiplications instead of loops whenever it is possible.

In [8]:

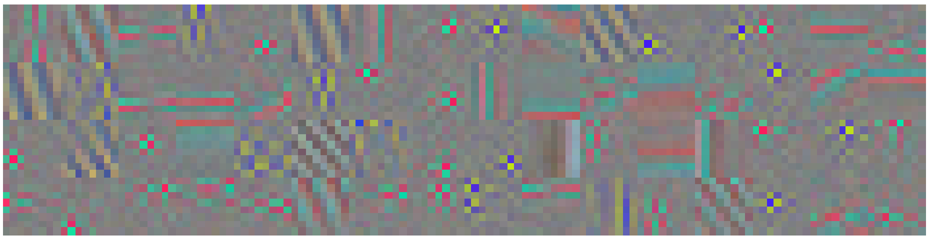
```
h = 64
a = 1.5
N, D = X.shape

G = lambda x: (1/1.5) * np.log(np.cosh(1.5*x))
G_epsilon = np.mean(G(np.random.normal(size=10000)))
J = lambda x: np.sum(np.square(np.mean(G(x),axis=0) - G_epsilon))

g = lambda x: np.tanh(1.5*x)
g_prime = lambda x: 1.5 * (1. - np.square(np.tanh(1.5*x)))

W = np.random.rand(D,h)
Z = whitening(X)

for i in range(201):

    b = (1/N) * Z.T @ g(Z.dot(W))
    c = np.mean(g_prime(Z.dot(W)),axis=0,keepdims=True) * W
    W = b - c

    W = W - np.mean(W,axis=0)
    W = whitening(W)

    W = W / np.linalg.norm(W,axis=0,keepdims=True)

    if i % 25 == 0: print(f'it: {i} J(W)= {J(Z.dot(W))}')

print("DONE")
```

```
it: 0 J(W)= 0.5431926578890748
it: 25 J(W)= 1.6225750324696397
it: 50 J(W)= 1.919874807256956
it: 75 J(W)= 2.1009345076114183
it: 100 J(W)= 2.173390373913928
it: 125 J(W)= 2.2055887141972534
it: 150 J(W)= 2.2247052687986892
it: 175 J(W)= 2.2335458485086366
it: 200 J(W)= 2.239438215219879
DONE
```

Because the learned ICA components are in a space of same dimensions as the input data, they can also be visualized as image patches.

In [9]:

```
W = W.T
utils.showpatches(W)
```

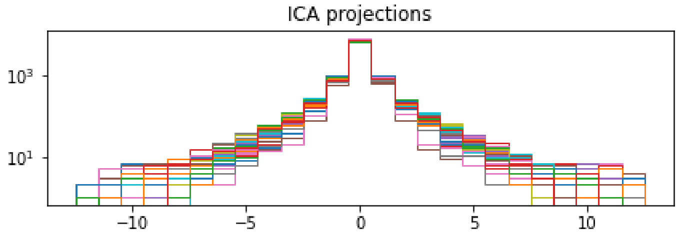


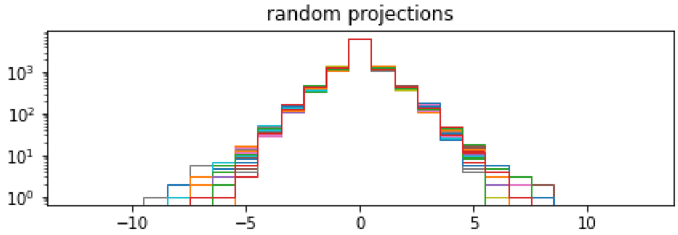
We observe that an interesting decomposition appears, composed of frequency filters, edges filters and localized texture filters. The decomposition further aligns on specific directions of the RGB space, specifically yellow/blue and red/cyan.

To verify that strongly non-Gaussian components have been learned, we build a histogram of projections on the various ICA components and compare it to histograms for random projections.

In [10]:

```
import numpy
plt.figure(figsize=(7,2))
for i in range(64):
    plt.hist(numpy.dot(Z,W[i]),bins=numpy.linspace(-12.5,12.5,26),histtype='step',log=True)
plt.title('ICA projections')
plt.show()

plt.figure(figsize=(7,2))
for i in range(64):
    R = numpy.random.mtrand.RandomState(i).normal(0,1,Z.shape[1])
    plt.hist(numpy.dot(Z,R/(R**2).sum()**.5),bins=numpy.linspace(-12.5,12.5,26),histtype='step',log
=True)
plt.title('random projections')
plt.show()
```





We observe that the ICA projections have much heavier tails. This is a typical characteristic of independent components of a data distribution.