1st mini project - ICA

Sebastien Duc

October 29, 2011

1 Single IC

1.1 Code

The code is in python, it was made to work on `python 2.7`. To plot the results, the package matplotlib was used. To use the program type `python ica.py mixsquarecos`. You can also choose which function will be used by the algorithm by typing `python ica.py mixsquarecos func` where func must be `exp`, `tanh` or `ycube`. By default `tanh` is used.

Basically there are two main python files. The first one is `fastICA.py` which contains all the functions used for the algorithm.

```
# this function implements the algorithm fastICA
# fastICA runs on data using function g
#(and dg is the derivative of g) it returns the
# estimate of one of the initial signal
def fastICA(data,g,dg):
    data = center(data)
    data = mix.whiten(data)
    w = random_w(data.shape[1])

    # epsilon is used to know if w converges
    epsilon = 0
    nbsteps = 0

    # max nb of steps. Used to test different functons ( used because
    # the algorithm might diverge)
    MAX = 100
    while(not converge(epsilon,1e-20) and nbsteps <= MAX):
        w_temp = newton_step(w,data,g,dg)
        w_temp /= np.linalg.norm(w_temp)
        epsilon = np.dot(w,w_temp)
        w = w_temp
        nbsteps += 1

    # if algorithm diverged , the nbsteps is negative
    if nbsteps > MAX:
        nbsteps = -1

    return np.dot(data,w),np.dot(data,np.array([w[1],-w[0]])),nbsteps

# center the data by substracting its mean
# return the data centered
def center(data):
    return data - data.mean(axis=0)

# return a random vector of the size of the number
# of input channels
def random_w(n_input_channel):
    w = np.random.rand(n_input_channel)
    return w/np.linalg.norm(w)

# it converges when w and w_new point in the same
# direction i.e. when the inner product is one
def converge(epsilon,lim):
    return abs(epsilon - 1) < lim

# implements the step 4 of the algorithm g
# is a function and dg its derivative
def newton_step(w,data,g,dg):
    y = np.dot(data,w)
    u = np.zeros(data.shape[1])
    v = 0
    for i in range(data.shape[0]):
        u += data[i]*g(y[i])
        v += dg(y[i])
    return (u - v*w)/data.shape[0]
```

The second is `ica.py` which first mix the signals, apply fastICA on it and then plot the result.

```
def square_cos_ica(g,dg):
    data = mix.mixsquarecos()
    x = f.fastICA(data,g,dg)
    return x

# g and its derivative used for the algorithm
g1 = lambda x: m.tanh(x)
dg1 = lambda x: 1 - m.tanh(x)**2

g2 = lambda x: x*m.exp(-(x**2)/2)
dg2 = lambda x: -m.exp(-(x**2)/2)*(x**2 - 1)

g3 = lambda y: y**3
dg3 = lambda y: 3*y**2


n = 200 # samples to plot
def plot(sig):
    pl.plot(np.arange(n), sig[:n])
    pl.show()

# run the algorithm and plot the result
sig1 = square_cos_ica(g1,dg1)
plot(sig1)
```

1.2 convergence measurement

We can say that the algorithm converges when w and w_{new} point in the same direction where w_{new} is the new vector found by the algorithm whereas w is the one from the previous step. Note that this is equivlent to say that their inner product is 1 since both are normalized. The following line was used, where epsilon is the inner product and lim is chosen number which should be very small

```
abs(epsilon -1) < lim
```

1.3 Comparison of g functions

First of all, when we run fastICA with $g(y) = y^3$, the algorithm diverges. At each step, $w_{new} = -w$.

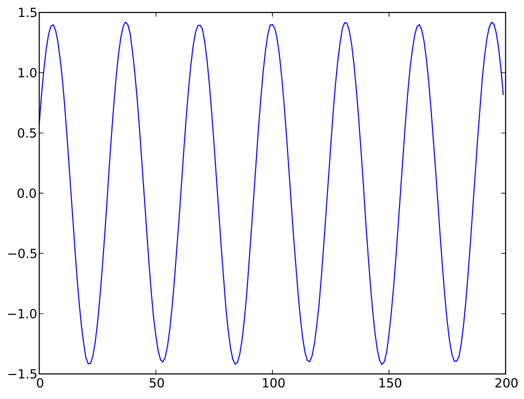
With the other functions, namely tanh and $y\exp(-y^2/2)$, the convergence is good and fast. But there is still a difference between them. To compare them we used a loop that run the algorithm n times. Each time the algorithm is ran, we compute the number of steps before convergence and we compute the mean over the n iterations. For the tests, we have set $\text{lim} = 10^{-20}$. We conclude that tanh converges almost surely and in 5 steps in mean. For $y\exp(-y^2/2)$ the algorithm diverges sometimes but otherwise it converges quiet fast. It converges in 4 steps in mean, which is a bit faster. Note that to run a test, type `python ica.py convergence`.

1.4 Results

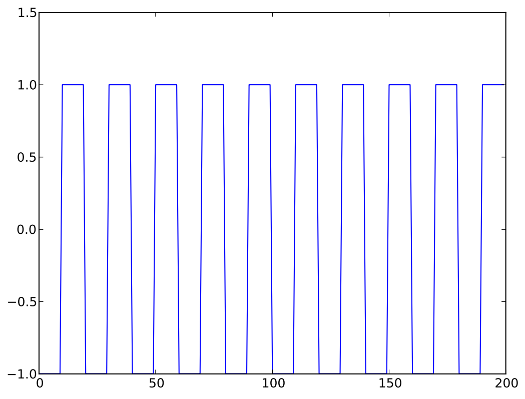
When using functions tanh and $y\exp(-y^2/2)$, the results are very convincing. To find the other component, we exploited the fact that the dimension is two. W simply multiplied the data with an orthogonal vector of the w we got with the algorithm. In the code the change is very simple. In the function `fastICA` of file `fastICA.py`, we just changed the return value to

```
return np.dot(data,w),np.dot(data,np.array([w[1],-w[0]]))
```

In Figure 1 you can see the plots of the recovered cos and square singals. Note that only the first 200 samples are plotted.



(a) Recovered cos



(b) Recovered square

Figure 1: Results of part 1

2 All ICs

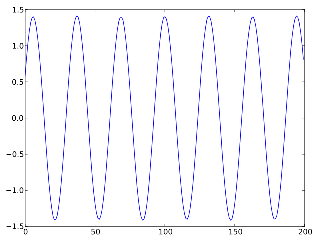
2.1 Code

To use the program type `python ica.py mixsouds`. As before you can also choose function g. The estimate of the signals that are output by the program are saved in directory `output`. For this part the same python files were used. First in `fastICA.py` two functions were added. `sym_decorrelation` and `fastICAall` which is as function `fastICA` but adapted to retrieve multiple sources.

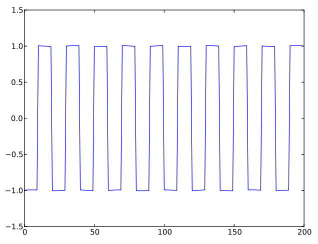
```
# this function implements the algorithm fastICA
# for all components it returns the unmixed signals
def fastICAall(data,g,dg):
    nIC = data.shape[1]

    data = center(data)
    data = mix.whiten(data)
    w = np.array([random_w(nIC) for i in range(nIC)])

    epsilon = 0
    w_temp = np.zeros_like(w)
    while(not converge(epsilon)):
        print("Epsilon is currently ", epsilon)
        for i in range(nIC):
            w_temp[i] = newton_step(w[i],data,g,dg)
            w_temp[i] /= np.linalg.norm(w_temp[i])
        W = sym_decorrelation(w_temp)
        epsilon = max(abs(np.diag(np.dot(W, w.T))))
        w = np.copy(W)

    return np.array([np.dot(data,w[i])
                     for i in range(nIC)])

# symmetric decorrelation
def sym_decorrelation(W):
    K = np.dot(W, W.T)
    s, u = np.linalg.eigh(K)
    # u (resp. s) contains the eigenvectors (resp.
    # square roots of the eigenvalues)
    u, W = [np.asmatrix(e) for e in (u, W)]
    W = (u * np.diag(1.0/np.sqrt(s)) * u.T) * W
    return np.asarray(W)
```

Then in file `ica.py` we added function `mixsounds_ica` to run fast ICA on the mixed sounds. We also added lines at the end to store the results in `.vaw` files

```
def mixsounds_ica(g, dg):
    data = mix.mixsounds()
```



(a) Recovered cos



(b) Recovered square

Figure 2: Plots when applying the algorithm to recover all components on mixsquarecos

```
    X = f.fastICAall(data,g,dg)
    return X

sigs = mixsounds_ica(g1,dg1)
# write in a file
if 'output' not in os.listdir('.'):
    os.mkdir('output')
for i in range(sigs.shape[0]):
    wavwrite('output/unmixedsound'+`i`+
             '.wav',8000,sigs[i])
```

Then to recover the initial sounds, we had to work a little more since the output of fastICA is an array of floats with wrong scale beacause of whitening step. We know that we must represent the arrray in bytes, or more simply with integers from 0 to 255. We added the following lines at the end of function `fastICAall` to have a proper scale.

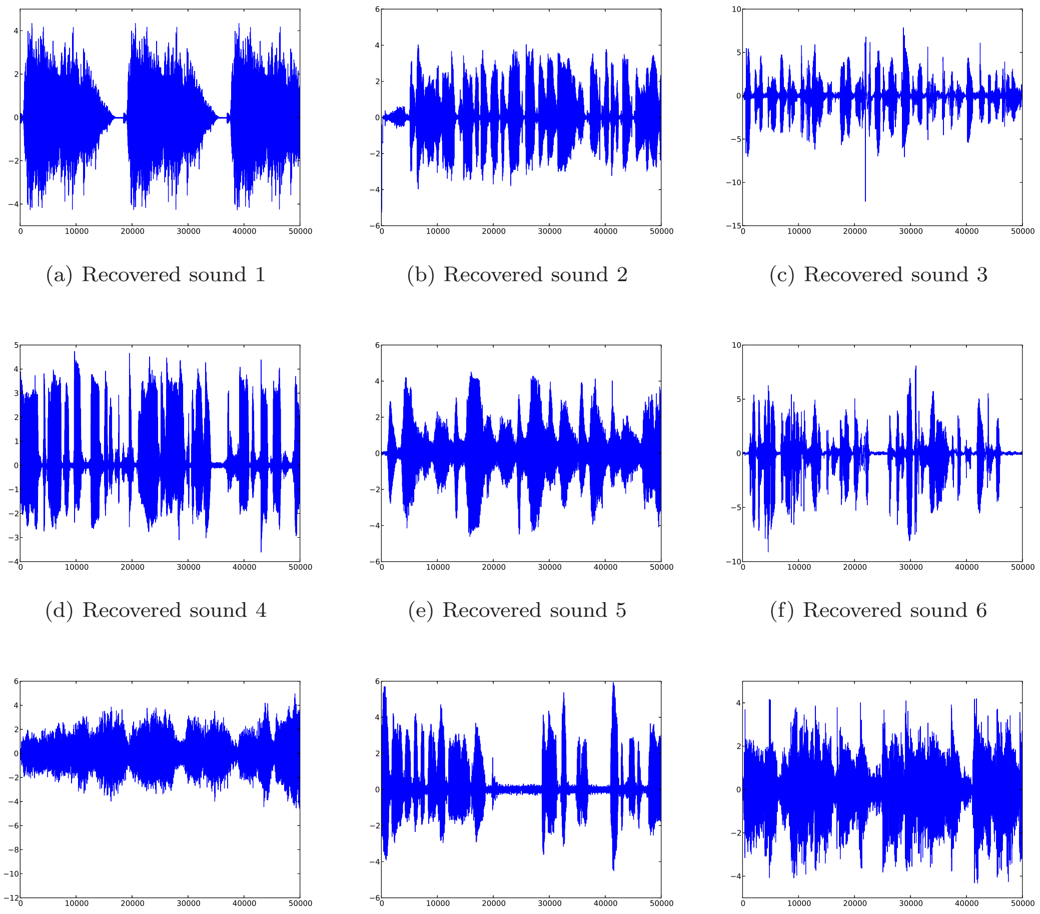
```
## recover unmixed signals
# first get the estimate of the data as before
dataest = np.array([np.dot(data,w[i]) for i in range(nIC)])
# the scale things so that it fitts between 0 and 255
dataest = dataest.T*(127)/np.max(np.abs(dataest), axis = 1)
return (dataest + 127).T
```

2.2 Symmetric orthogonalization

Symmetric orthogonalization is used to decorrelate the w_i so that not all of them converge to the same maximum value. This allows us to retrieve all the signals that are mixed.

2.3 Results

For the mixed square and cos, the algorithm was applied. You can see the plots in Figure 2. For the sounds, when using fastICA we get the following plots depicted in Figure 3. They are pretty close to the original sound but they are centered and whitened, thus don't have the same amplitude. Some of them are also sign reversed which is due to our convergence criterion because of its absolute value.



(a) Recovered sound 1 (b) Recovered sound 2 (c) Recovered sound 3

(d) Recovered sound 4 (e) Recovered sound 5 (f) Recovered sound 6

(g) Recovered sound 7 (h) Recovered sound 8 (i) Recovered sound 9

Figure 3: Results of part 2