Sperimentazioni sull'algoritmo K-SVD

Francesco Milizia
Università di Pisa, Dipartimento di Matematica
corso di Calcolo Scientifico, a.a. 2016/2017

21 febbraio 2017

Il lavoro svolto si basa sull'articolo "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation" di Michal Ahron, Michael Elad e Alfred Bruckstein.

1 Presentazione del problema

Il problema trattato nell'articolo nasce dalla seguente osservazione: molto spesso i segnali naturali sono combinazioni di un numero molto piccolo di segnali di "base", ovvero ammettono una *rappresentazione sparsa*. Con la parola "segnali" ci si può riferire ad esempio a suoni o immagini, che matematicamente possono essere tradotti in vettori $y \in \mathbb{R}^n$. Il fatto che i segnali di interesse si possono rappresentare in modo sparso si traduce con l'esistenza di una matrice $D \in \mathbb{R}^{n \times K}$ tale che per ogni osservazione y si abbia $y = Dx$ con $x \in \mathbb{R}^K$ avente un numero di componenti non nulle molto piccolo rispetto a K ed n. La matrice D, in questo contesto, viene chiamata *dizionario* e le sue colonne vengono dette *atomi*.

In realtà le osservazioni sono quasi sempre affette da errore, dunque il modello va leggermente modificato: si assume l'esistenza di un dizionario $D \in \mathbb{R}^{n \times K}$ tale che per ogni segnale osservato $y \in \mathbb{R}^n$ esiste un $x \in \mathbb{R}^K$ sparso tale che $\|y - Dx\|_2 \leq \epsilon$, dove ϵ è una tolleranza fissata.

Il problema affrontato nell'articolo è quello di progettare un algoritmo in grado di trovare un dizionario D che permetta rappresentazioni sparse di un insieme di segnali d'esempio forniti in input. La speranza è poi che tale dizionario consenta rappresentazioni sparse non solo degli esempi forniti, ma di tutte le osservazioni possibili.

Indicando con $y_1, \ldots, y_N$ i segnali d'esempio e ponendoli come colonne della matrice $Y \in \mathbb{R}^{n \times N}$ uno dei modi di formulare il problema è il seguente:

$$\min_{D,X} \left\{ \|Y - DX\|_F^2 \right\} \quad \text{soggetto a} \quad \|x_i\|_0 \leq T_0 \quad \text{per } i = 1, \ldots, N$$

dove con x_i si indica l'$i-$esima colonna della matrice $X \in \mathbb{R}^{K \times N}$, con la notazione $\|x_i\|_0$ si intende il numero di componenti non nulle del vettore x_i e infine T_0 è un intero che indica quanto si desidera che siano sparse le rappresentazioni.

2 Breve descrizione dell'algoritmo

L'algoritmo proposto è iterativo e ad ogni passo ha come obiettivo quello di modificare le matrici D e X in modo da far decrescere la quantità $\|Y - DX\|_F^2$, a condizione che ognuna delle colonne di X continui sempre ad avere un numero di componenti non nulle non superiore a T_0. Dopo aver inizializzato (ad esempio in modo casuale) la matrice D, è utile normalizzarne le colonne per semplificare alcune operazioni nel seguito. Quindi si procede con le iterazioni, ognuna delle quali consta di due parti:

- Fissato D si cerca X che minimizzi $\|Y - DX\|_F^2$.

  Questa fase può essere scomposta in N problemi indipendenti: per $i = 1, \ldots, N$ si cerca x_i che minimizzi $\|y_i - Dx_i\|_2^2$ e tale che $\|x_i\|_0 \leq T_0$, ovvero si cerca una rappresentazione sparsa del segnale y_i. Questo problema non ha una soluzione esatta di veloce esecuzione, quindi si deve ricorrere a delle strategie euristiche. Una possibilità è descritta al punto 2.1.

- Si modifica D per far descescere $\|Y - DX\|_F^2$.

  Gli autori dell'articolo propongono di aggiornare una alla volta le colonne di D, come descritto al punto 2.2.

2.1 Algoritmo OMP per la codifica sparsa di y

Dato $y \in \mathbb{R}^n$ e $D \in \mathbb{R}^{n \times K}$ si cerca $x \in \mathbb{R}^K$ che minimizzi la norma 2 del residuo $y - Dx$ e che abbia al più T_0 componenti non nulle. L'algoritmo OMP (orthogonal matching pursuit) opera come segue:

- Si inizializza il residuo $r_0 = y$ e l'insieme di atomi "attivati" $S_0 = \emptyset$.

Per $i = 1, 2, \ldots$ finché il residuo r_i non scende (in norma 2) sotto una certa soglia fissata o non si raggiunge il numero massimo T_0 di atomi attivi, si eseguono i passi:

- Si trova l'atomo d_j più "allineato" con il residuo, ovvero che massimizzi $|r_{i-1}^T d_j|$, e si aggiunge agli atomi attivati: $S_i = S_{i-1} \cup \{j\}$.

- Si considera la matrice D_{S_i} ottenuta da D tenendo solo le colonne i cui indici sono in S_i e si calcola il vettore x_{S_i} che minimizza $\|D_{S_i} x_{S_i} - y\|_2$. Quindi il nuovo residuo sarà $r_i = D_{S_i} x_{S_i} - y$.

Si restituisce il vettore x che si ottiene dall'ultimo x_{S_i} calcolato mettendo degli zeri in corrispondenza degli indici non attivati.

2.2 Aggiornamento del dizionario

Come accennato prima, si aggiornano una alla volta le colonne di D. Si supponga di voler aggiornare la $k-$esima colonna. Indicando con d_j e X^j rispettivamente le colonne di D e le righe di X, la quantità che si vuole minimizzare si può riscrivere come segue:

$$\|Y - DX\|_F^2 = \left\| Y - \sum_{j=1}^{K} d_j X^j \right\|_F^2 = \left\| \left(Y - \sum_{j \neq k} d_j X^j \right) - d_k X^k \right\|_F^2$$

L'idea è trovare una approssimazione di rango 1 per la matrice $E = Y - \sum_{j \neq k} d_j X^j$, prestando però attenzione alla condizione di sparsità che X deve conservare. Allora si costruisce l'insieme $\omega_k = \{m \text{ tale che } X^k_m \neq 0\}$, ovvero si selezionano i segnali che al momento usano l'atomo k, e al posto di E si considera la matrice E_{ω_k} ottenuta tenendo solo le colonne il cui indice è presente in ω_k. A questo punto si calcola la decomposizione a valori singolari: $E_{\omega_k} = U\Sigma V^T$ e si ottiene l'approssimazione ottimale di rango 1 per E_{ω_k}, ovvero $\sigma_1 u_1 v_1^T$. Di conseguenza si aggiornano $d_k = u_1$ e $X^k_{\omega_k} = \sigma_1 v_1^T$, dove con $X^k_{\omega_k}$ si indica il vettore ottenuto dalla riga X^k tenendo solo le componenti il cui indice è nell'insieme ω_k. Queste ultime erano già diverse da 0, dunque la sparsità di X viene preservata.

La quantità $\|Y - DX\|_F$ in questa fase dell'algoritmo non può che decrescere, vista l'ottimalità delle approssimazioni di rango 1 ottenute tramite le K SVD effettuate (una per ogni atomo del dizionario).

3 Sperimentazioni effettuate

Tutte le function e gli script che ho utilizzato nella sperimentazione sono stati implementati in MATLAB. Le due function principali sono:

```
function x = orthogonal_matching_pursuit(A, b, T0, tol)
function [D, err] = ksvd_step(D, Y, T0, mp_tol)
```

La prima implementa l'algoritmo OMP descritto al punto 2.1, mentre la seconda implementa una iterazione dell'algoritmo K-SVD. I codici, che contengono una descrizione degli argomenti e dei valori restituiti dalle funzioni, sono disponibili in fondo al documento.

3.1 Esperimenti su dati fittizi

Come fatto dagli autori dell'articolo ho innanzitutto verificato il funzionamento dell'algoritmo K-SVD su dati fittizi. Si crea un dizionario $D \in \mathbb{R}^{n \times K}$ contenente K atomi ognuno di lunghezza n scegliendo in modo casuale i coefficienti con distribuzione uniforme nell'intervallo $(-1, 1)$. Poi si generano N segnali, ognuno ottenuto tramite una combinazione lineare di T_0 atomi scelti a caso dal dizionario. Dopo di che si inizializza, sempre casualmente, un dizionario delle stesse dimensioni di D e si effettua su di esso un certo numero di iterazioni K-SVD, imponendo all'algoritmo di matching pursuit di trovare soluzioni con al massimo T_0 componenti non nulle.

Un modo per verificare la bontà del dizionario $\tilde{D}$ ottenuto al termine delle iterazioni è controllare quanti dei "veri" atomi (ovvero quelli che costituiscono D) sono presenti a meno di piccole variazioni anche in $\tilde{D}$. Per fare questo controllo è sufficiente effettuare il prodotto $\tilde{D}^T D$ e contare quante colonne contengono un elemento con valore assoluto maggiore di $1 - tol$, dove tol è una tolleranza fissata.

Per iniziare ho posto $n = 20$, $K = 50$, $N = 1500$, $T_0 = 3$ e ho fissato ad 80 il numero di iterazioni (sono gli stessi parametri usati dagli autori dell'articolo). Nella quasi totalità degli esperimenti effettuati con questi parametri il numero degli atomi correttamente trovati dall'algoritmo (ponendo $tol = 0.01$) è stato superiore a 40, e in alcuni casi addirittura tutti i 50 atomi sono stati individuati.

Figura 1: Per ogni atomo d_i di D (il dizionario da cui sono stati generati i segnali) riporta sull'asse y il massimo $|\tilde{d}_j^T d_i|$ al variare di $\tilde{d}_j$ tra gli atomi di $\tilde{D}$, dopo 80 iterazioni. Per chiarezza, questi valori sono riportati in ordine crescente. La linea rossa indica la soglia entro la quale viene riconosciuta una corrispondenza. In questo esempio il numero di atomi trovati è 46 su 50.

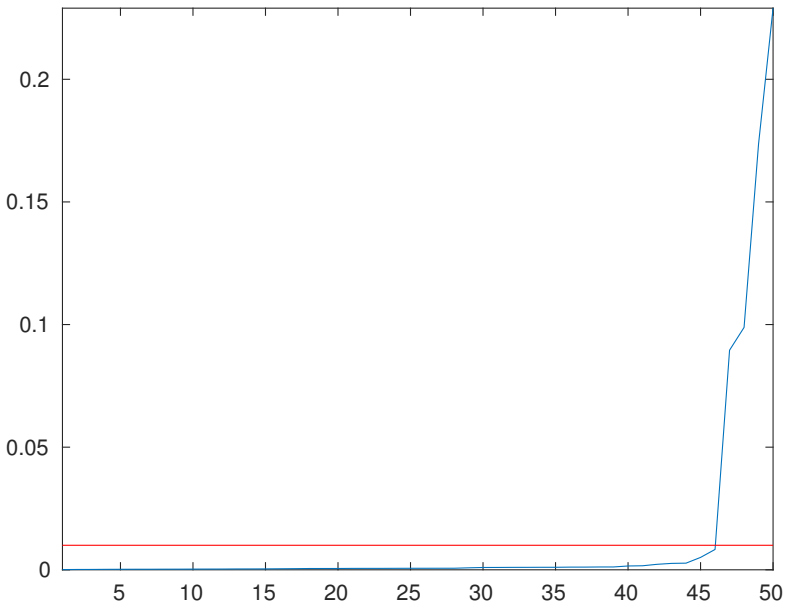
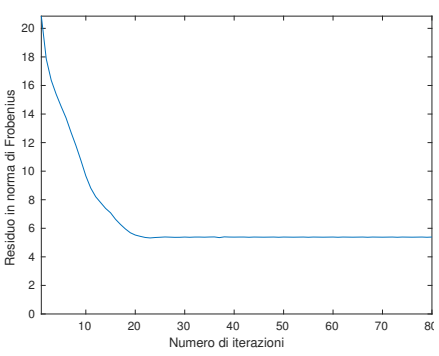
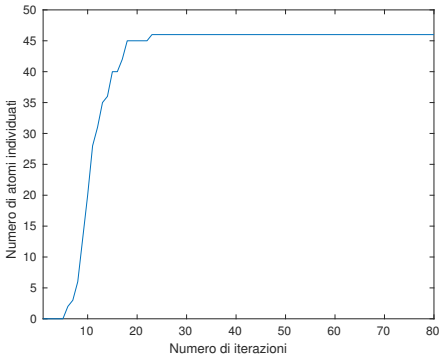


Figura 2: Esempio di grafici in cui si visualizza l'andamento del residuo e del numero di atomi individuati nel corso delle 80 iterazioni.



(a) Norma di Frobenius del residuo

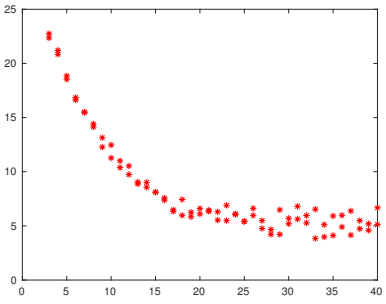


(b) Atomi individuati

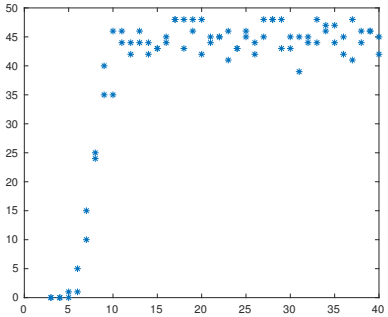
Nell'articolo non viene dato un criterio d'arresto per fermare le iterazioni. Idealmente ciò andrebbe fatto quando non ci si aspetta più di individuare ulteriori atomi del dizionario D da cui provengono i segnali, quindi in questi esperimenti basati su dati fittizi si potrebbe considerare di fermare le iterazioni quando il numero di atomi trovati si stabilizza. Però questo criterio sarebbe chiaramente inutilizzabile nelle applicazioni, poiché il dizionario da cui provengono i segnali è proprio ciò che si sta cercando di calcolare ed è completamente sconosciuto. Quindi ho pensato di tenere traccia dopo ogni iterazione sia della norma di Frobenius del residuo $\|Y - DX\|_F$ che del numero di atomi correttamente individuati fino a quel punto, con la speranza che questi due indicatori fossero correlati in qualche modo. Come si vede dai grafici riportati in figura 2, e da come è emerso in tutti gli esperimenti condotti, ho notato che vi è una fase iniziale (dopo le primissime iterazioni in cui non vengono individuati atomi) in cui il residuo cala abbastanza rapidamente e contemporaneamente cresce il numero di atomi individuati. Invece dopo un numero di iterazioni compreso a seconda dei casi tra 20 e 60 il grafico del residuo si appiattisce e il numero di atomi individuati non cambia. Inoltre questa variazione di andamento avviene nei due grafici in modo pressoché simultaneo. Dunque la seguente condizione d'arresto mi è sembrata adatta allo scopo: l'algoritmo si ferma quando la media degli ultimi 5 residui è maggiore della media dei 5 residui precedenti moltiplicata per un certo coefficiente α. In questo modo le iterazioni dovrebbero fermarsi quando non c'è più molta speranza di individuare ulteriori atomi. Dalle prove condotte per aggiustare il parametro α la scelta $\alpha = 0.999$ sembra funzionare abbastanza bene.

Facendo variare i parametri T_0 e K ho osservato che se questi vengono aumentati si rendono necessarie più iterazioni per avere risultati accettabili, inoltre si deve aumentare il numero di esempi N. In particolare variando T_0 anche di poche unità le performance cambiano notevolmente. Ad esempio, ponendo $N = 2000$, $K = 60$ e $T_0 = 5$ in media i primi atomi sono stati individuati dopo 100/120 iterazioni (ma comunque entro le 200 iterazioni quasi tutti gli atomi venivano individuati).

Figura 3: Risultati ottenuti facendo variare il rapporto segnale-rumore (SNR). Ogni punto colorato corrisponde ad una esecuzione dello script `test_ksvd`. Sull'asse delle ascisse è riportato il valore del SNR in decibel, mentre sull'asse delle ordinate è riportato il residuo finale (a sinistra) e il numero di atomi individuati (a destra). Valori bassi del rapporto segnale-rumore corrispondono a dati molto perturbati.



(a) SNR - residuo finale



(b) SNR - atomi individuati

Successivamente ho provato a far lavorare l'algoritmo K-SVD fornendogli in input segnali perturbati da rumore casuale, facendo variare il cosiddetto SNR (rapporto segnale-rumore, come descritto in `https://en.wikipedia.org/wiki/Signal-to-noise_ratio`. L'algoritmo sembra lavorare senza problemi per un SNR superiore a 10 decibel (come osservato anche nell'articolo), mentre se si scende sotto gli 8 decibel le prestazioni calano rapidamente. In figura 3 si possono vedere i risultati di queste prove, in cui sono stati utilizzati i parametri $N = 1500$, $K = 50$, $n = 20$, $T_0 = 3$.

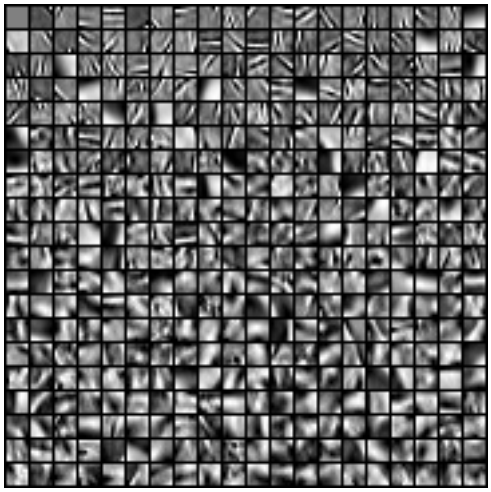
Gli script utilizzati per eseguire gli esperimenti descritti si trovano nei file `test_ksvd.m` e `test_snr.m`.

3.2 Esperimenti sul restauro di immagini

In questo caso i "segnali" consistono in blocchetti di dimensione 8×8 pixel prelevati dalle immagini prese in esame. In particolare, come gli autori dell'articolo, ho testato l'efficacia dell'algoritmo K-SVD nel problema della ricostruzione di un'immagine a cui sono stati cancellati alcuni pixel. Per poter effettuare gli esperimenti ho utilizzato le 400 immagini che si possono scaricare dal sito `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`. Si tratta di foto in scala di grigi di dimensioni 112×92 pixel, ognuna delle quali ritrae il volto di una persona. La function che si trova nel file `picture_dictionary.m` preleva automaticamente i segnali d'esempio dalle immagini ed utilizza l'algoritmo K-SVD per costruire il dizionario.

Come suggerito dagli autori dell'articolo, la function inizializza il dizionario fissando il primo atomo uguale ad un vettore con tutte le componenti uguali tra loro, e ponendo tutti gli altri atomi uguali a vettori con media 0. Quindi rimuove da ogni segnale la proiezione sul primo atomo ed esegue le iterazioni

Figura 4: Esempio di dizionario ottenuto tramite K-SVD prendendo come segnali d'esempio 7800 blocchi di dimensione 8×8. Per ottenerlo sono state effettuate 107 iterazioni K-SVD. Ogni quadratino rappresenta uno degli atomi del dizionario.



K-SVD non sull'intero dizionario ma solo sugli atomi a media 0, lasciando il primo invariato nel corso di tutto l'algoritmo.

Volendo poi testare il dizionario su una delle foto scaricate, riportata in figura 5a, prima di prelevare i blocchi d'esempio ho rimosso dal database le 10 immagini che ritraggono l'individuo protagonista della suddetta foto. Quindi ho fatto prelevare 20 blocchetti in posizioni casuali da ognuna delle 390 immagini rimaste, per un totale di 7800 esempi. Il dizionario ottenuto in questo modo, con parametri $K = 400$ e $T_0 = 10$, è riportato in figura 4.

Per testare il dizionario ho implementato le seguenti function, nel cui codice (disponibile in allegato) vengono descritti più precisamente i parametri in input:

- La function `delete_pixels` prende in input un'immagine A e un numero $p \in [0, 1]$ e restituisce un'immagine ottenuta da A cancellando (ponendo uguali a 0) alcuni pixel: in particolare ogni pixel ha probabilità $1 - p$ di essere cancellato.

- La function `restore_picture` prende in input un dizionario e un'immagine danneggiata che prova a ricostruire utilizzando l'algoritmo OMP.

  La ricostruzione di un singolo blocchetto avviene invocando OMP sul segnale "ridotto" che si ottiene dal blocchetto rimuovendo i pixel corrotti e sulla matrice $\tilde{D}$ che si ottiene dal dizionario D rimuovendo le righe corrispondenti ai pixel corrotti. Così si ottiene la parte "non danneggiata" del blocchetto come combinazione di alcune colonne di $\tilde{D}$. Il blocco restaurato si ottiene usando la stessa combinazione, ma sulle colonne di D.

  Nell'invocare la function è possibile specificare su "quanti" blocchi eseguire la ricostruzione. Ad esempio si può scegliere di considerare solo blocchi non sovrapposti, oppure al contrario di considerare tutti i blocchi 8×8 presenti nell'immagine. Nel caso di blocchi sovrapposti l'immagine finale

Figura 5: Ricostruzione di un volto con dei pixel mancanti.

(a) Immagine originale.

(b) Immagine a cui il 50% circa dei pixel sono stati cancellati.

(c) Immagine ricostruita tramite OMP, trattando separatamente blocchi non sovrapposti.

(d) Immagine ricostruita applicando OMP su tutti i blocchi 8×8 e facendo una media pesata delle ricostruzioni.

viene calcolata facendo una media tra le singole ricostruzioni dei blocchi, pesate in base al numero di pixel cancellati presenti nel blocco prima della ricostruzione (dando più peso alle ricostruzioni che partono da blocchi meno danneggiati). È anche possibile specificare una soglia per l'algoritmo OMP sotto la quale le iterazioni si fermano prima del raggiungimento del massimo numero consentito di componenti non nulle, come già spiegato al punto 2.1. Se si incrementa il valore di questa soglia le immagini risultanti tendono ad essere un po' sfocate, ma scegliendo il valore corretto è possibile ridurre il rumore dell'immagine (figura 8).

Le immagini che si ottengono eseguendo `delete_pixels` e `restore_picture` a partire dalla foto 5a ponendo uguale a 0.5 la probabilità di cancellare ciascun pixel sono riportate in 5b, 5c e 5d.

Ho provato a testare il funzionamento dell'algoritmo su un database di immagini non ristretto a foto di volti umani, ottenendo ancora risultati molto buoni. Ho utilizzato il database contenente 100 immagini scaricabile dalla pagina `http://mmlab.ie.cuhk.edu.hk/projects/FSRCNN.html`.

In figura 7 ripeto l'esperimento della ricostruzione di un'immagine a cui mancano dei pixel, mentre in figura 8 provo a ridurre il rumore presente nell'immagine. In particolare l'immagine 8a è stata ottenuta tramite il comando:
`rumorosa = uint8(double(foto) + randn(size(foto))*10);`

Figura 6: Immagine originale

Figura 7: Ricostruzione di un'immagine a cui mancano dei pixel.

(a) Immagine a cui manca circa il 50% dei pixel.

(b) Immagine ricostruita.

Figura 8: Ricostruzione di un'immagine affetta da rumore.

(a) Immagine a cui è stato aggiunto rumore.

(b) Immagine ricostruita.

dove `foto` è l'immagine originale. Dopo di che con il comando
`restauro = restore_picture(rumorosa, D, [8 8], [1,1], 10, 100);`
ho ottenuto la 8b.

```
function x = orthogonal_matching_pursuit(D, y, T0, tol)
% function x = ORTHOGONAL_MATCHING_PURSUIT(D, y, T0, tol)
% Trova x con al massimo T0 componenti non nulle che minimizza la norma 2 di Dx-y.
% Se il residuo ha norma 2 minore di tol prima che vengano attivate T0 componenti viene restituito il risultato trovato fino a quel punto.
% --input--
% D:    matrice N*M con colonne normalizzate
% y:    vettore colonna di lunghezza N
% T0:   massimo numero di componenti non nulle nella soluzione
% tol:  numero floating point
% --output--
% x:    vettore colonna di lunghezza M

    M = size(D,2);
    it = 0;
    r = y;
    x = zeros(M,1);
    active = zeros(1,T0);
    tol = max(tol, 1e-10);

    while(it < T0 && norm(r,2) > tol)
        it = it + 1;

        d = D' * r;
        [~, active(it)] = max(abs(d));

        x(active(1:it)) = D(:,active(1:it)) \ y;

        r = y - D(:,active(1:it))*x(active(1:it));
    end
end
```

```
function [D, err] = ksvd_step(D, Y, T0)
% function [D, err] = KSVD_STEP(D, Y, T0)
% Dato un dizionario D e un set di segnali d'esempio Y, esegue un passo dell'algoritmo KSVD.
% --input--
% D:    matrice n*K contenente un atomo in ogni colonna. Gli atomi devono essere già normalizzati.
% Y:    matrice n*N contenente un segnale in ogni colonna
% T0:   intero che specifica la sparsità desiderata
% --output--
% D:    dizionario ottenuto dopo l'iterazione KSVD
% err:  norma di Frobenius di DX-Y dopo l'iterazione

    N = size(Y,2);
    K = size(D,2);

    X = zeros(K,N);

    for j=1:N
```

```
        X(:,j) = orthogonal_matching_pursuit(D, Y(:,j), T0, 0);
    end

    E = Y - D*X;
    for k=1:K
        w = find(X(k,:));
        if(~isempty(w))
            E(:,w) = E(:,w) + D(:,k)*X(k,w);
            [U,S,V] = svd(E(:,w));
            D(:,k) = U(:,1);
            X(k,w) = S(1,1)*V(:,1)';
            E(:,w) = E(:,w) - D(:,k)*X(k,w);
        end
    end

    err = norm(E,'fro');
end
```

```
% Prova l'algoritmo KSVD su dati fittizi.
n = 20;     % Lunghezza dei segnali
K = 50;     % Numero di atomi nel dizionario
N = 1500;   % Numero di esempi
T0 = 3;     % Massimo numero di componenti non nulle
SNR = 30;   % Rapporto segnale-rumore (in dB)

max_iterations = 300; % Massimo numero di iterazioni KSVD
matching_tol = 0.01;  % Tolleranza entro la quale viene
    riconosciuta una corrispondenza tra un vero atomo e uno
    del dizionario allenato

fprintf('Dimensioni del dizionario: %d*%d\n', n, K);
fprintf('Numero di esempi: %d\n', N);
fprintf('T0 = %d\n', T0);
fprintf('SNR = %d dB\n', SNR);
fprintf('Massimo numero di iterazioni: %d\n', max_iterations);
fprintf('Soglia entro la quale viene riconosciuto un atomo: %f
    \n', matching_tol);

% Creo un dizionario casuale e normalizzo le colonne
D = rand(n,K)*2-1;
D = D/diag(sqrt(sum(D.^2)));

% Imposto casualmente i coefficienti e il supporto dei segnali
positions = randi(K,T0,N);
coeffs = rand(T0,N);

% Creo i segnali nelle colonne della matrice Y
Y = zeros(n,N);
for i=1:N
    Y(:,i) = D(:,positions(:,i))*coeffs(:,i);

    s2 = sum(Y(:,i).^2)/n/(10^(SNR/10));
    Y(:,i) = Y(:,i) + randn(n,1)*sqrt(s2);      % Perturbo Y
end
```

```
34
35 tic;
36 % Inizializzo il dizionario da allenare
37 learned_D = rand(n,K)*2-1;
38 learned_D = learned_D/diag(sqrt(sum(learned_D.^2)));
39
40 residuals = zeros(1,max_iterations);
41 matches = zeros(1,max_iterations);
42
43 it = 0;
44 while(it < max_iterations && (it <= 10 || sum(residuals(it-4:
       it))/5 < 0.999*sum(residuals(it-9:it-5))/5))
45   it = it + 1;
46   [learned_D, residuals(it)] = ksvd_step(learned_D, Y, T0);
47   matches(it) = sum(min(1-abs(learned_D'*D))<matching_tol);
48 end
49
50 fprintf('Tempo impiegato: %.3f\n',toc);
51 fprintf('Iterazioni: %d\n', it);
52 fprintf('Atomi del vero dizionario individuati: %d\n\n',
       matches(it));
53
54 figure;
55 plot(sort(min(1-abs(learned_D'*D))));
56 hold on;
57 plot(matching_tol*ones(1,K), 'r');
58 axis([1 K 0 inf]);
59 hold off;
60
61 figure;
62 plot(matches(1:it));
63 axis([1 it 0 K]);
64 xlabel('Numero di iterazioni');
65 ylabel('Numero di atomi individuati');
66
67 figure;
68 plot(residuals(1:it));
69 axis([1 it 0 inf]);
70 xlabel('Numero di iterazioni');
71 ylabel('Residuo in norma di Frobenius');
```

```
1 % Prova a variare il parametro SNR nel TEST_KSVD.
2 % Prima di lanciare questo script è necessario commentare la
      riga nel file 'test_ksvd' che imposta il parametro SNR.
3 % Si consiglia inoltre di commentare le ultime righe del
      suddetto file che plottano i grafici al termine delle
      iterazioni KSVD.
4
5 SNR1 = 3;  % Il più basso SNR da provare
6 SNR2 = 40; % Il più alto SNR da provare
7 prove = 2; % Quante esecuzioni per ogni valore del SNR
8
9 totali = prove*(SNR2-SNR1+1);
10 risultati = zeros(3,prove);
```

```
11
12 b = 0;
13 for SNR = SNR1:SNR2
14    for a=1:prove
15       b = b+1;
16       test_ksvd;
17       risultati(1,b) = SNR;
18       risultati(2,b) = matches(it);
19       risultati(3,b) = residuals(it);
20    end
21 end
22
23 figure;
24 plot(risultati(1,:), risultati(2,:), '*');
25 figure;
26 plot(risultati(1,:), risultati(3,:), 'r*');
```

```
1  function learned_D = picture_dictionary(pic_reg, K, T0, exmp,
       block_size, max_iterations)
2  % function learned_D = PICTURE_DICTIONARY(pic_reg, K, T0, exmp
       , block_size, max_iterations)
3  % Usa l'algoritmo KSVD per costruire un dizionario che
       permetta rappresentazioni sparse di 'blocchi' costituenti
       le immagini.
4  %
5  % pic_reg: espressione regolare per i file delle immagini (es.
        'miacartella/*.jpg')
6  % K: numero di atomi nel dizionario da produrre
7  % T0: massima norma 0 che si desidera nelle rappresentazioni
8  % exmp: numero di blocchi d'esempio estratti da ogni immagine
9  % block_size: dimensioni [altezza, larghezza] dei blocchi
10 % max_iterations: massimo numero di iterazioni da effettuare
11
12   bh = block_size(1);
13   bw = block_size(2);
14   n = bh*bw;
15
16   % Trovo i nomi delle immagini
17   files = dir(pic_reg);
18   pic_names = cell(length(files));
19   pic_num = 0;
20   for i = 1:length(files)
21     if(files(i).name ~= string('.') && files(i).name ~= string
         ('..'))
22       pic_num = pic_num + 1;
23       pic_names{pic_num} = [files(i).folder '/' files(i).name
           ];
24     end
25   end
26   N = exmp*pic_num;
27   fprintf('File trovati nella directory specificata: %d\n',
       pic_num);
28
29   % Prelevo i segnali e li inserisco nelle colonne di Y
```

```
  Y = zeros(n,N);
  for i=1:pic_num
    P = imread(pic_names{i});
    if(size(P,3) > 1)
      P = rgb2gray(P);
    end
    h = size(P,1);
    w = size(P,2);
    for j=1:exmp
      y = randi(h-bh+1);
      x = randi(w-bw+1);
      sP = P(y:y+bh-1, x:x+bw-1);
      Y(:,(i-1)*exmp+j) = reshape(sP,n,1);
    end
  end

  Y = Y - ones(n,1)*mean(Y);

  % Inizializzo il dizionario da allenare
  learned_D = rand(n,K);
  learned_D = learned_D - ones(n,1)*mean(learned_D);
  learned_D(:,1) = ones(n,1);
  learned_D = learned_D/diag(sqrt(sum(learned_D.^2)));
  fprintf('Dizionario inizializzato casualmente.\n');

  % K-SVD
  tic;
  residuals = zeros(1,max_iterations);
  it = 0;
  while(it < max_iterations && (it <= 10 || sum(residuals(it
      -4:it))/5 < 0.999*sum(residuals(it-9:it-5))/5))
    it = it + 1;
    [learned_D(:,2:K), residuals(it)] = ksvd_step(learned_D
        (:,2:K), Y, T0-1);
    fprintf('Iterazione %d, residuo = %f\n', it, residuals(it)
        );
  end

  % Risultati
  fprintf('Tempo impiegato: %d\n',toc);
  fprintf('Iterazioni: %d\n', it);
end
```

```
function show_dictionary(D, block_size, table_size)
% Mostra i blocchetti (ovvero gli atomi) del dizionario D.
% Le intensità sono riscalate per prendere tutta la scala di
    grigi.
% table_size indica le dimensioni della tabella che viene
    disegnata, ad esempio con table_size=[20 15] l'immagine
    conterrà 300 blocchi disposti in 20 righe e 15 colonne

  for idx=1:size(D,2)
    low = min(D(:,idx))-eps;   % Evito di dividere per 0.
    high = max(D(:,idx))+eps;
```

```
    D(:,idx) = (D(:,idx)-low)/(high-low);
  end

  b1 = block_size(1)+1;
  b2 = block_size(2)+1;
  H = zeros(1+table_size(1)*b1, 1+table_size(2)*b2);
  for i=1:table_size(1)
    for j=1:table_size(2)
      idx = (i-1)*table_size(1)+j;
      H((i-1)*b1+2:i*b1, (j-1)*b2+2:j*b2) = reshape(D(:,idx),
          b1-1, b2-1);
    end
  end

  figure;
  imagesc(H);
  colormap(gray);
  axis equal
  axis off
end
```

```
function B = delete_pixels(A, p)
%function DELETE_PIXELS(A, p)
% Restituisce un'immagine B ottenuta da A cancellando (ovvero
    ponendo uguali a 0) una certa quantità di pixel.
% Più precisamente, ogni pixel ha probabilità p di non essere
    cancellato.
%   A: immagine da danneggiare
%   p: probabilità per ogni pixel di non essere cancellato

   M = rand(size(A)) > p;
   B = A.*uint8(M);
end
```

```
function B = restore_picture(picture, D, block_size, step, T0,
    mp_tol)
% function B = RESTORE_PICTURE(picture, D, block_size, step,
    T0, mp_tol)
% Prova a ricostruire un'immagine danneggiata.
%   picture: immagine in scala di grigi in cui alcuni pixel
    sono stati posti a 0
%   D: dizionario per la rappresentazione dei blocchetti
%   block_size: dimensioni [altezza, larghezza] dei blocchetti
%   step: quanti pixel [verticali, orizzontali] di distacco
    tra blocchetti vicini
%   T0: sparsità presunta delle rappresentazioni dei blocchi
%   mp_tol: parametro per l'algoritmo di matching pursuit

  h = size(picture,1);
  w = size(picture,2);
  bh = block_size(1);
  bw = block_size(2);
  B = zeros(size(picture));
```

```
16    C = zeros(size(picture));
17
18    for y=[1:step(1):h-bh+1 h-bh+1]
19      for x=[1:step(2):w-bw+1 w-bw+1]
20        v = reshape(picture(y:y+bh-1, x:x+bw-1), bh*bw, 1);
21        ind = find(v);
22        if(~isempty(ind))
23          tD = D(ind,:);
24          scale = sqrt(sum(tD.^2));
25          tD = tD/diag(scale);
26          c = orthogonal_matching_pursuit(tD, double(v(ind)), T0
                , mp_tol);
27          c = c./scale';
28          B(y:y+bh-1, x:x+bw-1) = B(y:y+bh-1, x:x+bw-1) + length
                (ind)*reshape(D*c, bh, bw);
29          C(y:y+bh-1, x:x+bw-1) = C(y:y+bh-1, x:x+bw-1) + length
                (ind)*ones(bh, bw);
30        end
31      end
32    end
33    B = B./C;
34    B = uint8(B);
35  end
```