TISC docucmentation

Roger Grosse

September 15, 2007

Contents

1 About

This is the code which implements the algorithms described in "Shift-invariant sparse coding for audio classification." Because some of the algorithms take time to implement, we are providing this subset of our code base which implements the core inference routines. We also provide example code which uses SISC to learn a set of shift-invariant basis functions from Olshausen and Field's Sparsenet collection of natural images. We hope our code is modular enough that it will be easy to plug in your own data into our inference code.

Because the code is written in MATLAB and C, we expect it to be portable between machines. However, we should note that we have only run the code on 64-bit dual core AMD Opteron(tm) processors running Linux, and with MATLAB 7.4, and we can't guarantee that everything will run smoothly on other systems.

In this documentation, for simplicity, I describe all of the processing in terms of images, but nothing here is specific to images. In section 2.2, we briefly describe how to apply the code to other kinds of data.

Note: although all of the code here is our own, we have previously worked with Bruno Olshausen's SPARSENET package. There may be a few unintentional similarities in the overall organization of the code. SPARSENET can be found at `http://redwood.berkeley.edu/bruno/sparsenet/`.

2 Running the code

First, you need to run `make` to compile all of the C code. Our makefile is written for Linux, but if you are running on other platforms, you can compile each of the files individually (from the MATLAB command window) with

```
mex -g -largeArrayDims file.c sisc_lib.C
```

The makefile is set by default with the `-g` flag so that the C functions perform consistency checks on the parameters you pass in. If you wish to turn on code optimization, simply remove the `-g` flag. (MATLAB enables optimization by default.)

To run the demo, start MATLAB from this directory and type `run_sisc`. This will learn a set of 16 basis functions from the SPARSENET images, and should take roughly 45 minutes with code optimization turned off.

2.1 Notation

For convenience, I list the notations I will use in this paper.

- M and N, the number of rows and columns in each image.
- $\bar{M}$ and $\bar{N}$, the number of rows and columns in a basis function.
- C, the number of channels (e.g. colors) in the data.
- m, the number of images
- n, the number of basis functions
- $X^{(i)}$, the i^{th} image.
- $S^{(i,j)}$, the computed coefficients for the j^{th} basis function for the i^{th} image.
- $A^{(j)}$, the j^{th} basis function.

2.2 Overview

Let $X^{(i)}$ be image i, $A^{(j)}$ be the jth basis, and $S^{(i,j)}$ be the activation matrix for basis j in image i. SISC minimizes the following objective function:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} \|X^{(i)} - \sum_{j=1}^{n} A^{(j)} * S^{(i,j)}\|_2^2 + \beta \sum_{i,j} \|S^{(i,j)}\|_1 \\ \text{subject to} \quad & \|a_i\|_2^2 \leq c, \forall i, \end{aligned}$$

with respect to A and S.

Each image X is stored as an $M \times N \times C$ array[1], where M is the number of rows, N the number of columns, and C the number of "channels." In the case of images, channels could represent color or stereo, so for the black and white Sparsenet images, there is only a single channel. The algorithm is shift-invariant with respect to the first two dimensions, but not the third. The set of all images will, naturally, be stored as an $M \times N \times C \times m$ array. (The code currently only handles equal-sized images.)

The coefficient array S for a single image is an $M \times N \times n$ array (where n is the number of basis functions). All of the entries which cause a basis function to fall off the side of the image (e.g. a row larger than $M - \bar{M} + 1$ or a column larger than $N - \bar{N} + 1$) are constrained to always be zero. The basis functions are an $\bar{M} \times \bar{N} \times C \times n$ array.

Although I describe everything here in terms of images, remember that there is nothing here that is specific to vision. For processing raw time-series audio, we take M to be the length of the signal, and $N = C = 1$. For processing spectrograms, since we do not intend the algorithm to be shift-invariant in frequency, we take M to be the length of the signal (in spectrogram windows) and C to be the number of frequencies, and let $N = 1$. In other words, each frequency is represented as a channel.

By reading `run_sisc.m` and `run_batch.m`, you can probably get a good feel for how to use the code for your own data. I also describe the key functions in the following sections.

2.3 Solving for the coefficients

The most useful thing to be able to do is probably to compute the coefficients S for a given image. The function which does this is `get_responses`:

```
[S, coef_stats] = get_responses(X, A, beta, coef_pars,
                                patch_num, S_init);
```

The first three arguments are what you'd expect. `X` is the image, `A` is the basis functions, and `beta` is the sparsity penalty.

`coef_pars` is a data structure containing all of the other required parameters, most of which you probably want to ignore. You can get the default

[1]Here I use the term array rather than matrix, since we do not use operations such as matrix multiplication. However, arrays and matrices are a single data type in MATLAB.

settings with `default_coef_pars`, as shown in the example below. For more information on parameter settings, run `help default_coef_pars`.

The last two arguments are optional. `patch_num` is the patch number, used only for printing, and `S_init` is an initial value for `S`. (The default is all zeros.)

The function returns `S`, the activations, as well as `coef_stats`, a bunch of statistics such as the time elapsed and the objective function after each iteration. `S` will be a (full) 3-dimensional matrix, where the first two dimensions are the (m, n) coordinates and the third is the basis ID.

2.3.1 Example

To reconstruct a single image using the default parameters:

```
coef_pars = default_coef_pars(struct);
S = get_responses(X, A, beta, coef_pars);
```

2.3.2 Parameters

There are two ways you can set the optional parameters: by passing fields to `default_coef_pars` and by setting the fields directly. There are three options which can be passed as fields to `default_coef_pars`: `exact`, `verbosity`, and `tile`. `exact`, which is set to `true` by default, specifies that `FS_EXACT` should be used to compute the coefficients. (The alternative is a coordinate descent procedure which will not generally find the exact solution, but may sometimes work faster for large problem sizes.) `verbosity` is what the name describes; higher values produce more output. Finally, if `tile` is set to `true`, that means that `FS_WINDOW` wraps around `FS_EXACT` as described in the paper. If it is `false`, as is the default setting, `FS_EXACT` is used directly. We recommend setting `tile` to false for learning basis functions, since it is best to use small image excerpts anyway. However, if you want to reconstruct a large image from a fixed set of basis functions, it pays to set `tile` to `true`.

The other parameters can be set with `pars.foo = bar`. Here are some you might use:

- `coeff_iter` gives the maximum number of iterations to run before the algorithm gives up. (It'll stop earlier if it finds the exact answer.) 20 is usually pretty reasonable, but you can decrease this if it's taking too long.
- `num_coords` is the number of variables to add to the active set in each iteration of `FS_EXACT`. To achieve good performance, you might want to try a few different values of this, since the best value might depend on the data set or the system you are running on.

2.3.3 What if it's too slow?

Try raising the sparsity penalty. This will mean fewer nonzero activations, and therefore less to solve for.

2.4 Getting the reconstruction from A and S

Once you have `S`, you can get the reconstruction of the image with

```
rec = reconstruction(S,A);
```

To compute the objective function as well as each of the components, use:

```
err = X - rec;
fres = sum(sum(sum(err.^2)));
fspars = beta*sum(sum(sum(abs(S))));
fobj = fres + fspars;
```

2.5 Learning bases

The best way to solve for the bases is probably to call

```
[A, lambda] = get_bases(X_all, S_all, basis_M, basis_N, verbosity, lambda);
```

`X_all` is a set of images of size `M` by `N` by `num`images+. `S_all` is the set of all activations for all of the patches. It must be a sparse matrix, and since MATLAB requires that all sparse matrices are 2-dimensional, each column contains all of the activations for one image. You can get the sparse representation with

```
S_all(:,i) = sparse(reshape(S, M * N * num_bases, 1));
```

To retrieve the full array of activations for a single patch from the sparse `S_all`, you can use

```
S = reshape(full(S_all(:,i)), M, N, num_bases));
```

The other parameters are straightforward: `basis_M` and `basis_N` are the dimensions of the bases, `verbosity` is the amount of printed output, and `lambda` is an optional parameter specifying the initial value for the dual problem. The return value `A` is as described in Section 2.3.

If basis learning takes too long, it's probably because the images are too large. The dual learning algorithm scales badly in the size of the images (as opposed to the size of the bases), so you might consider using smaller images.

2.6 Running one batch

If you want to compute all the activations for one batch, and then possibly solve for the bases, you can use `run_batch`:

```
[A, stats, lambda, s_all] = run_batch(X_all, A, pars, coef_pars,
       batch_num, learn, ones(num_bases,1), s_all);
```

`X_all` is the set of images, as in Section 2.5. `A` is the set of bases, as in Section 2.3. `pars` and `coef_pars` are the parameters described in Sections 2.7.1 and 2.3.2, respectively. `batch_num` is the batch ID, only used for printing. Set `learn` to `true` if you want it to learn bases. Finally, you can pass the optional parameter `s_all`, which is the initial value for all of the activations, as described in Section 2.5.

2.7 Running the vision code

The great big function that does everything is `run_tisc`, and you can call it with the default parameters using

```
pars = default_pars(struct);
coef_pars = default_coef_pars(struct);
run_tisc(pars, coef_pars);
```

The two parameter sets are described in Sections 2.7.1 and 2.3.2, respectively.

I should mention that, although the code can take images of any size, I found that in practice it's faster to learn bases by splitting the data up into patches of, say, 80 pixels on a side.

After every batch, the basis functions are displayed, and the objective function is plotted with respect to number of patches and running time. To reduce bias, the objective function is computed on a separate "test batch" which does not overlap the training batches.

2.7.1 Parameters

These are the parameters for running the TISC code in its entirety. As with the coefficient parameters, you can get the default parameters with the `default_pars` function, and then set particular fields of the result. Here are some fields you may find useful:

- `num_trials` The number of batches to run.
- `batch_size` The number of images in a batch.
- `patch_M` and `patch_N` The dimensions of an image (see Section 2.7).
- `beta` The sparsity penalty.
- `num_bases`
- `basis_M` and `basis_N`, the size of the bases.
- `display_bases_every` Display bases after this many batches.
- `save_bases_every` Save bases (as a `.m` file) after this many iterations. Zero if you do not want to save anything.
- `basedir` and `savename`: everything will be saved in `basedir/savename*`. Do not include the `.mat` extension in `savename`.
- `verbosity` The level of printed output.

3 Code files

- General
  - `run_batch.m` Wrapper function which solves for the activations for all of the images in a given batch, and possibly solves for the bases too.
  - `run_sisc.m` Main loop for learning bases from Sparsenet.
  - `default_pars.m` Gives the default parameters for running Sparsenet.
  - `sisc_lib.c` Various utility functions used by all of our C code.
- For computing activations
  - `get_responses.m` Wrapper for all of the other functions which compute activations.
  - `default_coef_pars.m` Gives the default parameters for computing activations.
  - `get_responses_mex.c` A C implementation of feature sign search. Only implements a subset of the features in `get_responses2.m`, and doesn't produce much by way of output.
  - `solve_fs.m` Performs feature sign search. Solves for x to minimize the following objective function:

    $$x^T Ax + b^T x + c + \beta \|x\|_1$$

    Note that A here has nothing to do with the basis set A.
- For learning basis functions
  - `get_bases.m` Wrapper function for solving for bases.
  - `basis_compute_CtC.C` Precomputes $\hat{S}_t^\star \hat{S}_t$ and $\hat{S}_t^\star \hat{x}_t$ as described in the paper.
  - `basis_dual_objective.c` Computes the objective function, gradient, and Hessian of the dual problem.
  - `basis_solve_mex.c` Solves for A to minimize the following objective function:

    $$\sum_i \|X^{(i)} - \sum_j A^{(j)} \star S^{(i,j)}\|_2^2 + \sum_j \lambda_j \|A^{(j)}\|_2^2,$$

    where λ should be the Lagrange multipliers solved for in the dual problem.