ScatNet Implementation Document

June 10, 2013

1 Introduction

2 The Scattering Transform

The scattering transform is implemented in ScatNet using the `scat` function. This function can be used to calculate several types of scattering transforms by varying the linear operators supplied to it.

2.1 The `scat` Function

The `scat` function takes as input a signal `x` and a set of linear operators `Wop` and outputs a scattering transform `S` and intermediate modulus coefficients `U`. A call to the `scat` thus looks like

```
[S, U] = scat(x, Wop);
```

Often, the modulus coefficients `U` are not necessary and so can be left out.

Outputs `S` and `U` are cell arrays, each element corresponding to a layer of the scattering transform. The format of these layers is described in the next subsection.

Each element of the `Wop` is a function handle, with signature

```
[A, V] = Wop{m+1}(X);
```

Note that operators are indexed starting at $m = 0$, so an offset of 1 is necessary for compatibility with MATLAB. Here, `X`, `A` and `V` are all in the network layer format described in the next subsection. The linear operator `Wop{m+1}` transforms the signals in the `X` into two new layers: an invariant layer `A` (average) and a covariant layer `V` (variations). In the case of a wavelet transform, `A` corresponds to the averaging of the lowpass filter ϕ while `V` consists of the wavelet coefficients obtained by convolving with ψ_j.

Initializing `U1` using the input signal, the following loop is then executed in `scat`

```
for m = 0:numel(Wop)-1
    if (m < numel(Wop)-1)
        [S{m+1}, V] = Wop{m+1}(U{m+1});
        U{m+2} = modulus_layer(V);
    else
        S{m+1} = Wop{m+1}(U{m+1});
    end
end
```

For each intermediate layer `U{m+1}`, we thus apply the linear operator `Wop{m+1}`, assigning the invariant output to the mth-order scattering layer `S{m+1}`, and computing the modulus of the covariant part to obtain the $(m+1)$th order intermediate coefficients `+2Um`. For the last layer, we do not need the intermediate coefficients, and so only compute the scattering coefficients.

2.2 Network Layers `S{m+1}` and Linear Operators `Wop{m+1}`

As mentioned earlier, each element of `S` and `U` are in the network layer format. These consist of two fields, `signal` and `meta`. The former is a cell array of signals and the latter is a structure containing information related to each signal.

Specifically, each of the fields in `meta` is an array with the same number of columns as the length of `signal`. For example, the `meta.resolution` field

has one row, indicating the extent the signal has been subsampled with respect to the original length. In the case of wavelet transforms used as linear operators, an important field is `meta.j`, which has a variable number of rows, describing the scales of the wavelets used to compute the coefficient. For a first-order coefficient $|x \star \psi_{j_1}| \star \phi(t)$, this will simply be one row containing j_1. For a second-order coefficient $||x \star \psi_{j_1}| \star \psi_{j_2}| \star \phi(t)$, the first row will contain j_1 while the second will contain j_2.

Let us consider the example output `S` from the `scat` function using wavelet transforms as linear operators. In this case, `S{m+1}` will contain the mth-order scattering coefficients, with `S{m+1}.signal{p}` being the pth signal among these. Its scales $(j_1, j_2, \ldots, j_m)$ are specified in `S{m+1}.meta.j(:,p)`.

As mentioned earlier, the `Wop{m+1}` function handles take as an input a network layer and outputs two layers, one invariant and one covariant. Any function with these inputs and outputs can be used as an element of `Wop`. Two basic functions can be used for this purpose: `wavelet_layer_1d` and `wavelet_layer_2d`, which define wavelet transforms on network layers. For example, supposing we have defined a filter bank `filters` (see next section), we can create an accompanying 1D wavelet transform by defining

```
Wop{m+1} = @(U)(wavelet_layer_1d(U, ...
    filters));
```

This has to be done for each layer of the scattering transform.

3 Creating Operators

Linear operators can be constructed manually as indicated in the previous section by defining filters and function handles. However, factory functions presented in this section simplify this process.

3.1 `wavelet_factory_1d`

The `wavelet_factory_1d` function takes a signal size along with a number of parameters and returns a cell array of linear operators corresponding to wavelet transforms. Its signature is given by

```
[Wop, filters] = wavelet_factory_1d(N, ...
    filt_opt, scat_opt);
```

where `N` is the size of the signal, `filt_opt` are the filter parameters, `scat_opt` are other parameters for the wavelet transforms. Specifically, `scat_opt.M` is the maximal order of the scattering transform (i.e. how many layers/linear operators to create). The function outputs the wavelet transforms `Wop` and the filters used to define them `filters`. To calculate the scattering, only the former are necessary.

Default filter options are obtained by calling `default_filter_options`, which takes as input a type of filter and an averaging scale. The type of filter is one of `'audio'`, `'dyadic'` or `'image'`, corresponding to audio signals, other (less oscillatory) 1D signals, and images, respectively. The averaging scale determines the size of the largest wavelet and consequently that of the lowpass filter ϕ. For more filter options, see next section.

The `scat_opt` supports the following options:

- `scat_opt.oversampling`: The extent to which the final scattering output is oversampled. Specifically, the sampling rate is `2^scat_opt.oversampling` times the critical sampling rate. By default equal to `1`.

To specify default values, `scat_opt` can be left empty.

Using the above information, we can create a second-order scattering transform for an audio signal of length 65536 with an averaging scale of 4096

```
filt_opt = ...
    default_filter_options('audio', 4096);
scat_opt.M = 2;

Wop = wavelet_factory_1d(65536, ...
    filt_opt, scat_opt);
```

The scattering transform of a signal `x` is then obtained by calling `scat(x, Wop)`.

3.2 `wavelet_factory_2d`

The `wavelet_factory_2d` function takes a `size_in` vector of image size and optional parameters wrapped in an `options` structure and outputs a cell array of linear operators corresponding to successive wavelet transforms. Its signature is given by

```
[Wop, filters] = ...
    wavelet_factory_2d(size_in, options);
```

4 Defining Filters

As described in the previous section, filter parameters are specified in the `filt_opt` structure passed to `wavelet_factory_1d` or `wavelet_factory_2d`. Depending on the type of filter used, this structure can contain different parameters, but the following can always be specified:

- `filt_opt.filter_type`: The wavelet type, such as 'morlet_1d', 'morlet_2d', 'spline_1d', for example (default `'morlet_1d'` for 1D signals, `'morlet_2d'` for 2D signals).
- `filt_opt.precision`: The numeric precision of the filters. Either `'double'` or `'single'` (default `'double'`).

Since the `filter_type` field determines the type of filter bank to create, it determines what other options can be specified. The rest of this section covers the different types of filters and their parameters.

For 1D signals, it is often useful to specify different filter parameters for different orders. This is done by using arrays instead of scalar values for parameters in `filt_opt`. In the case of numeric parameters, this means that they are regular arrays, whereas for string parameters, such as `filter_type`, they are cell arrays.

For example, setting `filt_opt.filter_type ... = {'gabor_1d','spline_1d'}`, means that the first-order filter bank should be a Gabor filter bank whereas the second should be a spline wavelet filter bank. If more filter banks are needed than parameters are available in `filt_opt`, the last element in each array is repeated as necessary.

4.1 1D Morlet/Gabor Filters

The Gabor wavelets consist of Gaussian envelopes modulated by complex exponentials to cover the entire frequency spectrum. Morlet wavelets are derived from these by subtracting the envelope multiplied by a constant such that the integral of the filter equals zero. As they are closely related, the Gabor and Morlet wavelet have identical sets of options.

These are obtained by setting `filt_opt.filter_type` to `'morlet_1d'` or `'gabor_1d'`, respectively. The Morlet filters are the default filters for 1D signals.

The Morlet/Gabor filter bank has the following options:

- `filt_opt.Q`: The number of wavelets per octave. By default 1.
- `filt_opt.J`: The number of wavelet scales.
- `filt_opt.B`: The reciprocal octave bandwidth of the wavelets. By default `filt_opt.Q`.
- `filt_opt.sigma_psi`: The standard deviation of the mother wavelet in space. By default calculated from `filt_opt.B`.
- `filt_opt.sigma_phi`: The standard deviation of the scaling function in space. By default calculated from `filt_opt.B`.

The maximal wavelet bandwidth (in space) is determined by $2^{J/Q}$ times the bandwidth of the mother wavelet, which is proportional to `sigma_psi`. If `sigma_psi` is smaller than a certain threshold, a number of constant-bandwidth filters are added, linearly spaced, to cover the low frequencies.

Again, we can specify different filter banks by setting `filt_opt.Q` and `filt_opt.J`, etc. to arrays instead of scalars. This is often useful if the nature of the signal is different at different orders, which is usually the case in audio. For this reason, the `default_filter_options` for `'audio'` specify `filt_opt.Q = [8 1]`, whereas for `'dyadic'`, `filt_opt.Q` is set to `1`. A higher frequency resolution is needed in audio for the first-order filter banks due to the highly oscillatory structure of the signals.

To calculate the appropriate `J` necessary for a given window size `T` and filter parameters `filt_opt`, there is the conversion function `T_to_J`, which is called using:

```
filt_opt.J = T_to_J(T, filt_opt);
```

This will ensure that the maximum wavelet scale, and thus the averaging scale of the lowpass filter, will be approximately `T`.

4.2 1D Spline Filters

The spline wavelet filter bank is an unitary filter bank as defined in [**?**]. As a result, the scattering transform defined using these filters has perfect energy conservation [**?**].

In addition to the parameters listed above, the spline filter bank has the following options:

- `filt_opt.J`: The number of wavelet scales.
- `filt_opt.spline_order`: The order of the splines. Only linear (spline order 1) and cubic (spline order 3) are supported.

The maximal bandwidth is specified here by 2^J.

4.3 2D Morlet/Gabor Filters

LAURENT

5 Manipulating, Formatting Scattering Coefficients

To facilitate the manipulation of the scattering coefficients, several functions are provided that perform different operations on the network layer format described above. In addition, we can convert the coefficients to an easier-to-manage array representation, concatenating scattering coefficients and separating out meta-data.

5.1 Post-processing

This section describes the different post-processing function that are available to apply to scattering coefficients.

5.1.1 `renorm_scat`

As described in [**?**], second-order scattering coefficients can be renormalized by dividing $Sx(t, j_1, j_2)$ by $Sx(t, j_1)$, which decorrelates the second order from the first. A similar procedure is available for higher orders.

In ScatNet, this transformation is available through the `renorm_scat` function. Its signature is as follows

```
S_renorm = renorm_scat(S, epsilon);
```

where `S` is the scattering transform to be renormalized, and `epsilon` is a regularization constant that prevents the fraction from diverging when the first order becomes zero.

5.1.2 `log_scat`

For several signals, it is sometimes necessary to compute the logarithm of scattering coefficients. This is achieved in ScatNet using the `log_scat` function. Its signature is

```
S_log = log_scat(S, epsilon);
```

where `S` is the scattering transform whose logarithm we want to compute, and `epsilon` is a regularization constant that prevents the logarithm from going to negative infinity when the scattering coefficients approach zero.

5.1.3 `average_scat`

5.2 Formatting

In order to use the scattering coefficients for classification and other tasks, it is often useful to convert

the representation into a purely numeric format, separating out the metadata. This is the role of the format_scat function, whose signature is

```
[S_table, meta] = format_scat(S, fmt);
```

where S is the scattering transform, fmt is the type of output we want, S_table is the formatted output, and meta contain the metadata from S associated with the signals in S_table.

There are three different formats available for fmt. The first is 'raw', which leaves the representation intact, setting S_table = S. The second is 'order_table', which creates a two-dimensional table for each scattering order, and sets S_table to the cell array of these tables, with meta being a cell array of the associated S{m+1}.meta. Finally, 'table' (the default), concatenates the tables of all orders and does the same for the meta fields. In the case where meta fields have different number of rows for different orders, the smaller fields are extended with -1. Note that this last format is only possible if scattering coefficients of different orders are of the same resolution, which is most often the case.

6 Display

6.1 1D

6.1.1 scattergram

The scattergram function displays the scattering coefficients as a function of time and the last filter index with the remaining filter indices fixed. For first-order coefficients, this means that it is shown as a function of time and j_1. For second-order coefficients, it is shown as a function of time and j_2 for a fixed j_1, and so on.

For one display, a pair consisting of a scattering layer along with a scale prefix is needed. We can specify multiple displays by specifying multiple of these pairs, which will then be displayed one on top of the other.

Its signature is the following

```
img = scattergram(S{m1+1}, jprefix_1, ...
      S{m2+1}, jprefix_2, ...);
```

where S is the scattering transform, m1 is the order of the first display, jprefix_1 is the first scale prefix, and so on. Note that is m1 = 1 the layer is of first order so no scale prefix is necessary and we set jprefix_1 to [].

Instead of a scattering output S, we can also specify intermediate modulus coefficients U.

6.2 2D

LAURENT

7 Classification

ScatNet contains a classification pipeline for affine space and support vector classifiers. These let the user specify a feature transformation for computing features in batch form, then set up a training/testing mechanism for evaluating results.

7.1 Batch Computation

To compute scattering coefficients for a database of signals, we first define a source. A source is a set of files, each file containing a number of objects to be classified. Each object has a class and a location within the file.

The source can be created using create_src(directory, extract_objects_fun) (or one of its wrappers, such as gtzan_src, phone_src, uiuc_src, etc). Given a directory, this function recursively traverses it, looking for '.jpg', '.wav', or '.au' files. For each such file, it calls extract_objects_fun to determine its constituent objects and their classes. To add a new database, it suffices to write the corresponding extract_objects_fun function.

The extract_objects_fun function has the following signature

```
[objects, classes] = ...
    extract_objects_fun(file);
```

Given a file, it returns a structure array `objects` of the objects it contains and a cell array of their class names `classes`. The structure of the `objects` variable is described below.

Most often, a file will only contain one object, so `objects` is a structure of size one, but this can vary from database to database (for example in speech recognition a file corresponds to a recording of a phrase, which can contain different phones to be classified).

The objects structure has the fields `objects.u1` and `objects.u2` which correspond to the lower-left and upper-right corners (start- and endpoints) of the object in a 2D (or 1D) signal. These bounds are inclusive. For the case where a file only contains one object, the bounds will simply be the bounds of the signal. If desired, additional fields can be specified that will be passed on and saved in the source structure.

Given a directory and this function, `create_src` returns a source structure `src`, which has the following fields

- `src.classes`: A cell array of the class names.
- `src.files`: A cell array of the file names.
- `src.objects`: A struct array of the objects returned by applying `objects_fun` to each filename, but with two supplementary fields: `src.objects.ind`, corresponding to the index of its parent filename in `src.files` and `src.objects.class`, corresponding to the index of its class in `src.classes`.

The scattering coefficients of the objects in `src` can now be calculated using the function `prepare_database`. This function takes for input the `src` structure, a function handle `feature_fun`, which calculates the feature representation, and an options structure. If we want to calculate multiple feature representations and concatenate the results, their feature functions can be assembled into a cell array and given instead of `feature_fun`.

A "feature function" takes as input one or more signals and returns the their feature vectors. Its signature is thus

```
features = feature_fun(signal);
```

When `signal` is a 1D signal, `features` should be a 2D matrix (potentially just a column vector) where the first dimension is the feature index and the second dimension is a time index. The `signal` argument can also be a collection of 1D signals, arranged as a columns of a 2D matrix. In this case, the output `features` is a 3D array, with the first two dimensions being the same, but the third dimension corresponding to the signal index in `signal`.

Similarly, for `signal` being a 2D signal, `features` is a 2D matrix with first dimension corresponding to feature dimension and the second dimension corresponding to a space index. Again, multiple 2D signals can be given to `feature_fun` by specifying `signal` to be a 3D array, with signals arranged along the third dimension. The output `features` will then also be a 3D array as in the 1D case.

Note that a feature function is not required to output only one feature vector for each signal. That is, it can leave a time/space index in the output feature representation. If more than one feature vector is associated to a single signal object, the classifier will classify each feature vector separately and aggregate the results, either through averaging the approximation error (for the affine space classifier) or through voting (for the SVM classifier). If this is not desired, the user has to make sure that the feature function only returns feature vectors with one time index per signal.

The `scat` function, combined with the `format_scat` function, fulfills the above criteria. As such, we can use it to define a feature function for `prepare_database`. For batch calculation of second-order audio scattering vectors for the GTZAN dataset, we have the following code.

```
N = 5*2^17;
src = gtzan_src;

filt_opt = ...
    default_filter_options('audio', 8192);
scat_opt.M = 2;

Wop = wavelet_factory_1d(N, filt_opt, ...
    scat_opt);
```

```
8
9  feature_fun = @(x)(format_scat( ...
10     log_scat(renorm_scat(scat(x, Wop)))));
11
12 database_opt = struct();
13
14 database = prepare_database(src, ...
       feature_fun, database_opt);
```

Observe the signal length `N`, which is equal to the length of the signals contained in `gtzan_src`.

The `database_opt` contains options given to `prepare_database`. These include:

- `database_opt.feature_sampling`: The factor by which the features should be downsampled. Useful for reducing computational complexity of the classifier training and testing (default `1` - no downsampling).
- `database_opt.file_normalize`: The norm to use when normalizing each file. Can be either `1`, `2`, `Inf` or `[]`, corresponding to $\mathbf{l}^1$, $\mathbf{l}^2$, $\mathbf{l}^\infty$ norms and no normalization, respectively (default `[]`).
- `database_opt.parallel`: If `1`, `prepare_database` will attempt to parallelize the code by using the `parfor` construct from the MATLAB Distributed Computing Toolbox. Note that if this is not available, performance may be worse since MATLAB will only use one core (default `1`).

The resulting `database` structure now contains three fields:

- `database.src`: The original `src` structure from which the features were computed.
- `database.features`: The feature vectors, arranged in a 2D matrix, each feature vector forming a column.
- `database.indices`: The indices corresponding to each object in `database.src`. Specifically, `database.features(:,database.indices{k})` contains the features vector(s) calculated from `database.src.objects(k)`. If there is one feature vector per object, each element of `database.indices` is a scalar.

7.2 Classifier Framework

Once the database of feature vectors is constructed, models can be trained and tested. To create a train/test partition, the function `create_partition` is available. Given a source `src` and a ratio, it partitions the objects so that each class is divided evenly into the training and testing set according to the ratio. For example, the following code defines a train/test partition with 80% of the objects in the training set:

```
1 [train_set, test_set] = ...
      create_partition(src, 0.8);
```

Here, `train_set` will contain the indices in `src.objects` that correspond to the training instances, while `test_set` will contain those corresponding to testing instances.

Alternatively, a partition can be defined manually by traversing the `src.objects` array and recording the desired indices.

Given a partition, a model can be trained using the appropriate training function. We will denote the training function by `train`, but in reality it will be either `affine_train` or `svm_train`. The training is then done by:

```
1 model = train(database, train_set, ...
      train_opt);
```

The structure `train_opt` contains various parameters for training the model, which will depend on the type of classifier used.

To test the model, the functions `affine_test` and `svm_test` are used. Here, we denote the testing function by `test`. The labels obtained for the testing instances specified by `test_set` are then obtained by:

```
1 labels = test(database, model, test_set);
```

To calculate the classification error, the `classif_err` is used:

```
1 err = classif_err(labels, test_set, src);
```

Note that the original src structure is necessary here to verify the class membership of the testing instances.

To calculate the errors over a range of training parameters in train_opt, the functions affine_param_search and svm_param_search are available. Denoting them both by param_search, they have the following signature

```
[err, param1, param2] = param_search( ...
    database, train_set, valid_set, ...
        train_opt);
```

Where database and train_set are the feature database and training set, respectively, valid_set is a validation set to calculate the errors on and train_opt contain the training parameters for the model. Multiple training parameters are specified by setting the field to an array instead of a scalar in train_opt. The classification error for different parameter combinations is then given in err, with the actual parameter values stored in param1, param2, etc.

The validation set can be taken to be test_set, but this would lead to overfitting of the parameters. For some tasks, a separate validation test is given for this purpose. Another alternative is to perform cross-validation on the training set train_set. This is obtained by setting test_set to []. By default, param_search will then split train_set into five parts, train on four and test on the remaining one, then shuffle and repeat. The number of folds can be determined by the train_opt.cv_folds option. In this case, err is a 2D matrix, with the parameter set index along the first dimension and the fold index along the second. To obtain the best parameter set, an average can then be performed along the fold index.

7.3 Affine Space Classifier

The affine classifier is defined by the functions affine_train and affine_test. The former takes as an option the number of dimensions, train_opt.dim used for the affine space model of each class.

The following code calculates the error for an affine space classifier dimension 40:

```
train_opt.dim = 40;
model = affine_train(database, ...
    train_set, train_opt);
labels = affine_test(database, model, ...
    test_set);
err = classif_err(labels, test_set, src);
```

For an affine space classifier, the model structure contains the dimension for which it is defined model.dim, the centers of the classes model.mu (a cell array) and the direction vectors defining the affine space model.v (a cell array).

In addition to the labels, affine_test can also output the approximation errors for each feature vector and object. This is done by the additional outputs obj_err and feature_err, as in the following call:

```
[labels, obj_err, feature_err] = ...
    affine_test(database, model, test_set);
```

Here, obj_err is a 2D matrix with the first dimension corresponding to the class index of the affine space the object was approximated using and the second dimension corresponding to the object index. Similarly, feature is a 2D matrix with first dimension being the class index and the second dimension corresponding to the feature vector index, arranged according to the objects they belong to. That is, the feature vectors of test_set(1) come first, then those of test_set(2), etc. Note that obj_err can be calculated from feature_err by averaging over the all the feature vectors for one object.

To find the optimal dimension of the affine space, affine_param_search is used, with a range of dimensions specified for train_opt.dim. Specifically, we call

```
[err, dim] = ...
    affine_param_search(database, ...
    train_set, valid_set, train_opt);
```

where database, train_set, valid_set, train_opt are as described in the previous subsection. The output dim is equal to train_opt.dim, and err contains

the errors for this range of dimensions.

7.4 Support Vector Classifier

NOTE: Requires the LIBSVM library, see `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

The support vector machine classifier is defined by the functions `svm_train` and `svm_test`. The training options consist of:

- `train_opt.kernel_type`: The kernel type used in the SVM. Can be `'linear'` for a linear kernel $u^T v$ or `'gaussian'` for a Gaussian kernel $e^{-\gamma\|u-v\|^2}$.
- `train_opt.C`: The slack factor C used for training the SVM.
- `train_opt.gamma`: The regularity constant γ used in the case of a Gaussian SVM kernel.

The following code calculates the error for an SVM classifier with a linear kernel and $C = 8$:

```
train_opt.kernel_type = 'linear';
train_opt.C = 8;
model = svm_train(database, train_set, ...
    train_opt);
labels = svm_test(database, model, ...
    test_set);
err = classif_err(labels, test_set, src);
```

If the feature vectors are very large, which can be the case for scattering coefficients, it is often useful to precalculate the kernel. This significantly speeds up training and testing, but at a cost of memory consumption. To precalculate the kernel, the function `svm_calc_kernel` is used on `database`, as in

```
database = svm_calc_kernel(database, ...
    kernel_type);
```

where `kernel_type` is either `'linear'` or `'gaussian'`. Afterwards, when `svm_train` is called, it can take advantage of this kernel if the same kernel type is specified in `train_opt.kernel_type`.

To determine the optimal parameters, a parameter search function `svm_param_search` is provided, with the same syntax as described above. Its signature is

```
[err, C, gamma] = ...
    svm_param_search(database, ...
    train_set, valid_set, train_opt);
```

where `database`, `train_set`, `valid_set`, `train_opt` are as described previously. We set the values of `C` and `gamma` that we wish to test in `train_opt`. All combinations of these parameters are then evaluated, with errors recorded in `err` and the particular values for each set stores in `C` and `gamma`.

Since parameters can vary over a large domain, and a fine-grained parameter search can be quite costly, there is also an adaptive parameter search function, `svm_adaptive_param_search`. This function calls `svm_param_search`, recenters the `C-gamma` parameter grid on the optimal parameter combination while increasing the grid resolution, then reapplies `svm_param_search`, and so on. This is done `train_opt.cv_depth` times, which by default equals 2.

If a linear kernel is used, we can extract the discriminant vector w and bias ρ from the model using the function `svm_extract_w`. The function takes as input the database `database` and the SVM model `model`. It outputs the ws corresponding to each pair of classes in the model, arranged in the order 1vs2,1vs3,...1vs$N$,2vs3,...,2vs$N$,...,$(N-1)$vs$N$, where N is the number of classes. The function is called as in:

```
[w,rho] = svm_extract_w(database, model);
```