ScatNet : a MATLAB Toolbox for Scattering Networks

Laurent Sifre, Joakim Andén, Michel Kapoko,
Édouard Oyallon, and Vincent Lostanlen

October 11, 2013

1 Introduction

1.1 Overview

ScatNet is a MATLAB scientific toolbox which provides tools for the analysis and classification of digital signals, such as sounds, images or time series. Mostly owing to its properties of group invariance and stability to deformations, it has shown to achieve state-of-the art results in the challenges of music genre recognition, image, texture classification, and fetal heart rate characterization. Its core feature relies on the construction of a *scattering network*, i.e. a stack of signal processing layers of increasing width. Each layer consist in the association of a linear filter bank `Wop` with a non-linear operator, namely the complex modulus. The *scattering transform* of an input signal `x` is defined as the set of all paths that `x` might take from layer to layer. In this sense, the architecture of a scattering network closely resembles a convolutional deep network.

1.2 Getting started

1.3 Outline

In this document, the objects in ScatNet are first reviewed from the general to the specific : Sections 2, 3 and 4 respectively cover the architectures of the scattering operator, of each layer in the network, and of each filter in a layer. Afterwards, Section 5 presents some utilities that facilitate the manipulation and formatting of scattering transforms, for the purposes of visualization and discriminative learning. Section 6 is dedicated to the display tools of ScatNet, both for one-dimensional and two-dimensional input signals. Finally, Section 7 thoroughly describes the classification pipeline of affine space and support vector classifiers for scattering representations.

2 Scattering transform

2.1 Prototype

Assuming that the cell array of linear operators `Wop` is already built, the computation of the scattering transform of some input signal `x` is a fast operation, supported by the function `scat`. Interestingly, the array `x` may either be of dimensions:

1. $(N \times 1)$, for a time series or an audio file of length N,
2. $(N_1 \times N_2)$, for an image of height N_1 and width N_2, or
3. $(N_1 \times 1 \times N_3)$, for a set of N_3 audio files, each of length N_1.

In all cases, the network of scattering coefficients is obtained throught the single command :

```
S = scat(x, Wop);
```

`S` is a cell array whose length `M+1` is the same as `Wop`. The integer `M` is the maximal order of the scattering transform, and is noted $\overline{m}$ in academic papers. From a theoretical point of view, the order m ranges from 0 to $\overline{m}$; but since MATLAB is one-based, the corresponding layer index must range from `1` to `M`, hence the need of an offset of `1` for the variable `m`.

Each layer S_m of scattering coefficients proceeds from an averaging of modulus coefficients U_{m-1} computed at the previous layer. These modulus coefficients can be separately obtained with `scat`, as an optional second output argument :

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

2.2 Implementation of the scattering transform

Each operator `Wop{1+m}` performs two actions, leading to separate outputs :

1. an energy *averaging* according to the largest scale, by means of a low-pass filter ϕ, and
2. an energy *scattering* along all scales, by means of band-pass filters ψ_j.

After initializing `U{1+0}` to `x`, `scat` executes the following loop :

```
for m = 0:M-1
    [S{1+m}, V] = Wop{1+m}(U{1+m});
    U{1+(m+1)} = modulus_layer(V);
end
S{1+M} = Wop{1+M)}(U{1+(M-1)});
```

The function `modulus_layer` computes the modulus of all signals in `V` while preserving its layer structure.

At the last layer, it is not necessary to perform any scattering, since the coefficients will not be further processed. Therefore, `Wop{1+M}` is merely an averaging operator, with a single output argument.

2.3 Format of a scattering layer

The layer format, in which every `S{1+m}` and `U{1+m}` are encoded, consists of two fields, namely `signal` and `meta`. The former is a cell array of real-valued signals, while the latter is a structure containing meta-information. In the sequel, `S{1+m}.meta` is an array, whose number of columns is equal to the number of signals within the layer.

Any user-defined function with one input and two outputs in the previous format may become an element of the cell array `Wop`, through an instruction of this kind :

```
1 Wop{1+m} = @mth_user_defined_operator;
```

Most often, however, it is not necessary to do so ; indeed, ScatNet provides efficient linear operators by default, especially adapted to the processing of images and sounds. Therefore, the whole cell array `Wop` can be defined in one line through built-in "factories", which rely on the theory of wavelet analysis. These factories are reviewed in the next section.

3 Wavelet factories

All five built-in factories are available in the `core` directory. After reviewin their respective implementations and prototypes, we document the customization of the scattering transform throught the `scat_opt` structure.

3.1 Scattering representations of audio signals and time series

The function `wavelet_1d` gives an efficient representation of one-dimensional signals, such as sounds or time series. It bears the following prototype :

```
1 [Wop,filters] = wavelet_factory_1d(size(x),filt_opt,scat_opt);
```

The input arguments `filt_opt` and `scat_opt` respectively gather the parameters of the wavelet filter banks and the scattering network ; note that both can be omitted at first glance. Section 4.2 explains how to create custom filter banks. As regards `scat_opt`, the parameter most subject to change is the

integer `scat_opt.M`, which represents the maximal order of the scattering transform, and equals 2 by default. The `scat_opt` structure is further specified in Section 3.4.

The first output argument is a cell array of function handles, which fit the required format in the prototype of `scat` (see Section 2.1). The second output is not mandatory for the processing of `x`, as it is only useful for the visualisation of the filter bank themselves : its architecture is reviewed in Section 4.1.

3.2 Translation-invariant representations of images ??

For its part, `wavelet_factory_2d` provides a translation-invariant representation of images. Its prototype is exactly the same as in the one-dimensional case (see Section 3.1), and all previous remarks hold :

```
[Wop,filters] = wavelet_factory_2d(size(x),filt_opt,scat_opt);
```

In addition, ScatNet provides another wavelet factory for images, called `wavelet_factory_2d_pyramid`, and relying on the cascade algorithm. While pursuing the same goal as `wavelet_factory_2d`, it happens to be computationally quicker, and doesn't require to be given the size of the signal `x` as a first input ; hence the following prototype :

```
[Wop,filters] = ...
    wavelet_factory_2d_pyramid(size(x),filt_opt,scat_opt);
```

In spite of these assets, `wavelet_factory_2d_pyramid` is not as customizable as its counterpart, especially in terms of anti-aliasing and numerical approximations. As a matter of fact, these two functions mainly differ in the format of `filters` : while `wavelet_factory_2d`, like `wavelet_1d`, stores its filters in the Fourier domain, `wavelet_factory_2d_pyramid` uses their spatial support.

3.3 Roto-translation invariant representations of images

Finally, the function `wavelet_factory_3d` enables a scattering representation of images which is invariant to both translations and rotations. This original feature dramatically improves the performance of texture recognition in comparison with the translation-invariant representation provided by `wavelet_factory_2d`.

```
[Wop,filters,filters_rot] = wavelet_factory_3d(size(x), ...
    filt_opt, filt_rot_opt,scat_opt);
```

With `wavelet_factory_3d`, at the first order, the energy is scattered along the vertical and horizontal dimensions of the image, similarly to `wavelet_factory_2d`. At higher orders, however, `wavelet_factory_3d` computes a one-dimensional wavelet transform along the orientations of the previous coefficients. Therefore,

for a scattering transform of order 2, each coefficient bank bears three indices vertical scale, horizontal scale, and angle hence the "3d" denomination.

Whereas the structure `filt_opt` contains the parameters of the filter bank at the first order, `filt_rot_opt` is related to the roto-translation wavelets of higher orders.

Likewise the translation-invariant case, ScatNet provides an alternative built-in factory for `wavelet_3d`, which relies on the cascade algorithm (see Section **??**). All previous remarks hold.

```
[Wop,filters,filters_rot] = ...
    wavelet_factory_3d_pyramid(filt_opt,filt_rot_opt,scat_opt);
```

Contrary to what their name might suggest, neither `wavelet_3d` nor `wavelet_3d_pyramid` are adapted to an input array `x` of size $(N_1 \times N_2 \times N_3)$. Up to now, there is no built-in factory for the processing of three-dimensional signals, such as video clips or voxel-based measurements. Nevertheless, it is not difficult to derive a custom pipeline that matches the geometric dimension of the data, in a similar fashion to the available factories.

3.4 Customized wavelet factory

The structure `scat_opt` may contain up to the four following fields sorted here by order

- `scat_opt.M` : the maximal order of the scattering transform (see Section 2.1). This value must be a positive integer. When set to `1`, the scattering transform is merely the modulus of a wavelet transform. Equal to `2` by default whatever be the chosen built-in factory. Increasing `scat_opt.M` marginally improves classification results, at a great computational cost.
- `scat_opt.oversampling`: the log_2 of the desired oversampoling factor. By default equal to `1`, which means that .
- `scat_opt.psi_mask`
- `scat_opt.x_resolution`

If `scat_opt` is an empty data structure or is omitted in the call to the wavelet factory, all the previous fields are set to their default values, whatever be the chosen built-in factory.

4 Filter banks

Filters bank corresponds to predefined sets of filters. The nature of the used filters is highly important since it determines the result of the scattering transform. Filters bank can be created using the `wavelet_factory` set of functions, and have a similar skeleton in 1D, 2D or 3D.

4.1 Architecture

In each of this case, `filters` have the same architecture. The first field corresponds to a low-pass filter, the second to the band-pass filters and a field `meta` contains the information for the whole bank-filters. The band-pass filters are stored as cells.

4.2 Options

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.

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)`.

4.3 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`. Note that the previous prototype, `T_to_J(T,Q,B,phi_bw_multiplier)` is no longer supported in this version.

4.4 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.

4.5 1D Selesnick Filters

Selesnick wavelet filters are compactly supported complex wavelets, whose real and imaginary parts have 4 vanishing moments and are nearly Hilbert transform pairs.

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

- `filt_opt.J`: The number of wavelet scales.

The maximal bandwidth is specified here by 2^J.

The code for the Selesnick wavelet filters is an independent package that has been included in ScatNet. The standalone package, including more documentation, can be found at:
`http://eeweb.poly.edu/iselesni/WaveletSoftware/dt1D.html`.

4.6 2D Morlet Filters

The Gabor wavelets consist of Gaussian envelopes modulated by complex exponentials to cover the en- tire 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.

A 2D Morlet filter bank consists in a gaussian window

$$\phi_J(u) = 2^{-2j/Q}\phi(2^{-j/Q}(u,v)) \tag{1}$$

and in dilated and rotated Morlet filters.

$$\psi_{j,\theta}(u) = 2^{-2j/Q}\psi(r_\theta 2^{-j/Q}(u,v)) \tag{2}$$

with

$$j \in [0, J-1] \tag{3}$$
$$\pi^{-1}L^{-1}\theta \in [0, L-1] \tag{4}$$

The full filter bank can be obtained by calling `morlet_filter_bank_2d`.

```
size_in = [128, 128];
filt_opt.L = 6;
filt_opt.Q = 1;
filt_opt.J = 3;
filters = morlet_filter_bank_2d(size_in, filt_opt);
```

This example builds a `128x128` filter bank with the following options :

- `filt_opt.Q` the number of wavelets per octave. By default 1.
- `filt_opt.J` the number of wavelet scales. By default 4.
- `filt_opt.L` the number of wavelet orientations. By default 8.

The mother Morlet filter is

$$\psi(u,v) = e^{\frac{u^2+s^2v^2}{2\sigma^2}}(K - e^{iu\xi}) \tag{5}$$

Its parameters are

- σ the spread of the gaussian envelope
- ξ the frequency of the oscillatory exponential
- s the eccentricity of the elliptical gaussian enveloppe

Setting `Q` and `L` influences the default parameters of the mother Morlet filter σ, ξ and s. For example, increasing L also increases the angular resolution by decreasing ξ. This adaptive behavior can be overridden by manually setting σ, ξ, s in the `filt_opt` structure.

- `filt_opt.sigma_phi` the spread of ϕ
- `filt_opt.sigma_psi` the spread σ in (5)
- `filt_opt.xi_psi` the frequency ξ in (5)
- `filt_opt.slant_psi` the eccentricity s in (5)

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 metadata.

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

```
1 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

```
1 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

For several classification problems, it is sometimes useful to average features in time as it reduces the complexity of the problem and smoothes the distribution of the features. This is achieved in ScatNet using the `average_scat` function. Its signature is

```
1   S_av =  average_scat(S,T,step,window_fun);
```

where `S` is the scattering transform we want to average, `T` is the length of the window with which to average, `step` is the stepping of the successive windows (which default value is $T/2$), and finally `window_fun` represents the windowing function to use for averaging(which default value is $@hanning_standalone$).

5.1.4 aggregate_scat

Some signals (mostly audio signals) have a somewhat obvious and particular structure(for e.g the attack sustain and release phases of a muscical note) which we might want to provide as a cue to classifiers. In that case, it would totally make sense to concatenate features computed over a certain number of successive

time frames. In ScatNet, the function `aggregate_scat` empowers you with such an ability. Its signature is:

```
S_ag = aggregate_scat(S,N);
```

where `S` is the scattering transform we want to aggregate and `N` is the length of the window on which we wish to aggregaate the features.

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

The `image_scat` opens one figure per layer and display all the path of a layer in the corresponding figure. Its signature is

```
Sx = scat(x, Wop);
% display scattering coefficients
image_scat(Sx)
```

To display all the coefficients of a particular layer, one can use

```
m = 2;
image_scat_layer(Sx{m+1})
```

This stacks all paths of the layer next to each other, renormalize them and overlay their meta information. Both these feature can be turned off using

```
m = 2;
image_scat_layer(S{m+1}, 0, 0);
```

Paths can also be arranged in a specific order along rows and column. For example, the following arranges scattering coefficient of second order by lexicographic order on j along rows and lexicographic order on θ along columns.

```
var_y{1}.name = 'j';
var_y{1}.index = 1;
var_y{2}.name = 'j';
var_y{2}.index = 2;
var_x{1}.name = 'theta';
var_x{1}.index = 1;
var_x{2}.name = 'theta';
var_x{2}.index = 2;
m = 2;
image_scat_layer_order(S{3},var_x, var_y,1);
```

The spatial coefficients of all filter of a filter bank can be displayed using

```
display_filter_bank_2d(filters);
```

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

```
1 [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);

feature_fun = @(x)(format_scat( ...
    log_scat(renorm_scat(scat(x, Wop)))));

```

```
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:

```
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:

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

To calculate the classification error, the classif_err is used:

```
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.search_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);
```

8 Separable time and frequency scattering

For temporal signals, the regular scattering transform as implemented in scat using operators Wop from wavelet_factory_1d calculates a representation that is locally invariant to time-shifts and stable to time-warping deformation. However, for many tasks, this is not sufficient. Often a additional invariance to frequency transposition and stability to frequency warping is needed. For example, this is the case for speaker-independent phone segment identification, where a single phone can be pronounced at different pitches, and the formant peaks can be deformed in frequency.

A simple way of creating such an invariant representation out of the scattering transform is to average along the log-frequency axis λ_1. Since a frequency transposition corresponds to a translation along log-frequency, this will create the desired invariance and stability properties. However, it also loses a great amount of information on frequential structure.

To remedy this problem, we apply frequential scattering transform along the log-frequency axis of the original temporal scattering transform. Since the frequency axis is not entirely sampled on a logarithmic scale, but is linear at low frequencies, this is not strictly the same as scattering along log-frequency, but the difference is only visible on the linear regime of the filter bank, which is often below audible frequencies in many applications.

Having computed a temporal scattering transform of a signal x using

```
filt_opt.Q = [8 1];
filt_opt.J = T_to_J(1024, filt_opt);

scat_opt.M = 2;

Wop = wavelet_factory_1d(size(x), filt_opt, scat_opt);

S = scat(x, Wop);
```

we now wish to calculate a scattering transform along the `j(1,:)` index of S, for all orders, which yields the desired invariance property.

To do so, we must first construct a one-dimensional wavelet operator defined on the `j(1,:)` axis. This is done as in the temporal case, but the signal length now corresponds to the maximum number of `j(1,:)` samples, which can be found by calling `length(S2.signal)`, which returns the number of first-order coefficients. We thus have

```
Nfreq = length(S{2}.signal);

freq_filt_opt.J = ceil(log2(Nfreq));
freq_scat_opt.M = 1;

Wop_freq = wavelet_factory_1d(Nfreq, freq_filt_opt, freq_scat_opt);
```

The options structures `freq_filt_opt` and `freq_scat_opt` have the same parameters and function as their temporal counterparts `filt_opt` and `scat_opt`.

Note that `freq_filt_opt.J` now controls the degree of transposition invariance. Since `freq_filt_opt.Q = 1` by default, we have that the width of the averaging filter ϕ is given by 2^J. Setting `freq_filt_opt.J` to `ceil(log2(Nfreq))` thus guarantees total transposition invariance. The degree of invariance necessary depends on the task at hand. If only one octave of invariance is needed, we can specify

```
freq_filt_opt.J = filt_opt.Q;
```

where `filt_opt.Q` is the number of wavelets per octave in the orignal, temporal, filter bank.

Finally, we calculate the frequential scattering transform by calling the `scat_freq` function.

```
[S_freq, U_freq] = scat_freq(S, Wop_freq);
```

Just like `S`, `S_freq` is a cell array of scattering layers, each layer corresponding to the original temporal scattering order. However, each temporal scattering layer now contains the coefficients of the frequential scattering transform applied to that layer. That is, `S_freq2` contains all the frequential scattering coefficients of `S2`, while `S_freq3` is the transform of `S3`.

The identity of the coefficients is given by the meta fields. For example `meta.fr_order` gives the order of the frequential scattering transform, while `meta.fr_j` gives the frequential scale, corresponding to the "quefrency" of the frequential filter. For example,

```
coeff = S{2}.signal(S{2}.meta.fr_order==0);
```

extracts the zeroth-order frequential scattering coefficients of the first-order temproal coefficients. From the definition of the transform, these are simply the

original temporal scattering coefficients averaged along log-frequency. The temporal scale of these coefficients is still stored in `meta.j`, so we can write

```
j = S{2}.meta.j(:,S{2}.meta.fr_order==0);
```

to obtain the scales `j` corresponding to `coeff`. Note that since the scattering transform subsamples to the critical bandwidth, the coefficients have been subsampled along frequency, which is reflected in `j`.

We can also extract the first-order frequential coefficients and the corresponding temporal scales `j` for a particular frequency scale `fr_j`, using the following code

```
coeff = S{2}.signal(S{2}.meta.fr_order==1 & ...
    S{2}.meta.fr_j==0);
j = S{2}.meta.j(:,S{2}.meta.fr_order==1 & ...
    S{2}.meta.fr_j==0);
```

And finally, we can extract the same frequential coefficients, but for the second temporal order scattering coefficients. In this case, we also need to specify `meta.j(2,:)`. The code is then

```
coeff = S{3}.signal(S{3}.meta.fr_order==1 & ...
    S{3}.meta.fr_j==0 & ...
    S{3}.j(2,:)==9);
j = S{3}.meta.j(:,S{3}.meta.fr_order==1 & ...
    S{3}.meta.fr_j==0 & ...
    S{3}.j(2,:)==9);
```