copper Documentation

Release 0.1.0

Kenneth Lyons

Mar 10, 2017

Contents

`copper` is a small infrastructure for processing data in a pipeline style. You create pipeline blocks, then connect them up with an efficient (but still readable) syntax. It was originally created for flexibly creating pipelines in real-time signal processing applications, but it can be useful in offline applications as well.

CHAPTER 1

User Guide

In copper, data processing is implemented through a *Pipeline*. A pipeline is a series of processing routines for transforming raw input data (e.g. electrophysiological data such as EMG) into useful output, such as the velocity of a cursor on the screen. These routines can usually be broken down into blocks which have common functionality.

Common Blocks

The typical picture for an electrophysiological signal processing pipeline looks something like:

Each block in this example is really a *type* of processing block, and the actual processing involved in each can vary. copper implements some of the common cases, but creating custom blocks and connecting them together in a pipeline structure is simple. Also, the picture above shows a simple series structure, where each block takes input only from the block before it. More complex structures are sometimes convenient or necessary, and some complexity is supported.

Windowing

Windowing involves specifying a time window over which the rest of the pipeline will operate. That is, a windower keeps track of the current input data and optionally some data from the past, concatentating the two and passing it along. This is useful for calculating statistics over a sufficient sample size while updating the pipeline output at a rapid rate, achieved by overlapping windows. In an offline processing context (i.e. processing static recordings), windowing also specifies how much data to read in on each iteration through the recording.

Windowing is handled by a *Windower*.

Conditioning

Raw data conditioning (or pre-processing) usually involves things like filtering and normalization. Usually the output of a conditioning block does not fundamentally change the representation of the input.

Feature Extraction

Features are statistics computed on a window of input data. Generally, they should represent the information contained in the raw input in a compact way. For example, you might take 100 samples of data from six channels of EMG and calculate the root-mean-square value of each channel during that 100-sample window of time. This results in an array of length 6 which represents the amplitude of each channel in the high-dimensional raw data. A feature extractor is just a collection of features to compute from the input.

Features in copper are classes that take all of their parameters in `__init__` and perform their operation on the input in a `compute` method.

Features are typically used by adding a handful of them to a *FeatureExtractor* and putting that extractor in a *Pipeline*.

Intent Recognition

Intent recognition is the prediction or estimation of what the user intends to do based on the signals generated. An example would be a large signal sensed at the group of extensor muscles in the forearm for communicating "wrist extension." Sometimes this mapping can be specified a priori, but most of the time we rely on machine learning techniques to infer this mapping from training data.

Connecting Blocks

The *core* module is a small infrastructure for processing data in a pipeline style. You create or use the built-in *PipelineBlock* objects, then connect them up with an efficient (but still readable) syntax with a *Pipeline*.

The syntax for expressing pipeline structure is based on lists and tuples. Lists hold elements that are connected in series:

```
[a, b]:

    -a-b-
```

The input is whatever `a` takes, and the output is whatever `b` outputs. Tuples hold elements that are connected in parallel:

```
(a, b):

     -a-
    -
     -b-
```

The input goes to *both* `a` and `b`, and the output is whatever `a` and `b` output in a list. If we connect another element in series with a parallel block, it must be a block that handles multiple inputs:

```
[(a, b), c]:

     -a-
    -   c-
     -b-
```

The bottom line is: pipeline blocks **accept** input types and they **specify** the output types. You are responsible for ensuring that pipeline blocks can be connected as specified.

Sometimes, you might want to pass the output of a block to some block structure *and* somewhere downstream. To handle this case, there is a *`PassthroughPipeline`* that you can use as a block within another pipeline:

```
passthrough pipeline p ← (b, c):

     ---
     -b- |
    -
     -c-

[a, p, d]:
                    ---
                    -b- |
    -a-pd-  →  -a-   d-
                    -c-
```

The pass-through pipeline places its own output(s) after its input, so the input is accesible on the other side. There are cases where this type of structure is possible with a list/tuple expression, but sometimes the pass-through pipeline as a block is needed. The above example is one of those cases.

Implementing Pipeline Blocks

Pipeline blocks are simple to implement. It is only expected that you implement a `process()` method which takes one argument (`data`) and returns something. For multi-input blocks, you'll probably want to just expand the inputs right off the bat (e.g. `in_a, in_b = data`). Usually, the output is some processed form of the input data:

```
import copper

class FooBlock(copper.PipelineBlock):
    def process(self, data):
```

```
        return data + 1

class BarBlock(copper.PipelineBlock):
    def process(self, data):
        return 2 * data
```

With some blocks implemented, the list/tuple syntax described above is used for specifying how they are connected:

```
a = FooBlock()
b = BarBlock()
p = copper.Pipeline([a, b])
```

Now, you just give the pipeline input and get its output:

```
input = 3
result = p.process(input)
```

In this case, the result would be `2 * (input + 1) == 8`.

Post-Process Hooks

Sometimes, it's useful to be able to hook into some block in the pipeline to retrieve its data in the middle of a run through the pipeline. For instance, let's say you have a simple pipeline:

```
[a, b]:

    -a-b-
```

You run some data through the pipeline to get the result from block `b`, but you also want to run some function with the output of `a`. `PipelineBlock` takes a `hooks` keword argument which takes a list of functions to execute after the block's `process` method finishes. To use hooks, make sure your custom block calls the parent `PipelineBlock` `__init__` method. For example:

```
import copper

class FooBlock(copper.PipelineBlock):
    def __init__(self, hooks=None):
        super(FooBlock, self).__init__(hooks=hooks)

    def process(self, data):
        return data + 1

class BarBlock(copper.PipelineBlock):
    def process(self, data):
        return 2 * data

def foo_hook(data):
    print("FooBlock output is %d".format(data))

a = FooBlock(hooks=[foo_hook])
b = BarBlock()

p = copper.Pipeline([a, b])
result = p.process(3)
```

Now, the call to `process` on the pipeline will input 3 to block `a`, block `a` will add 1 then print `FooBlock output is 4`, and then 4 will be passed to block `b`, which will return 8.

CHAPTER 2

API

These are the modules/subpackages which constitute copper.

core

Base classes for pipelines and pipeline blocks.

class copper.core.**PipelineBlock**(*name=None*, *hooks=None*)
Base class for all blocks in copper.

Notes

Blocks should take their parameters in __init__ and provide at least the process method for taking in data and returning some result.

process(*data*)
Process input data and produce a result.

Subclasses must implement this method, otherwise it shouldn't really be a PipelineBlock.

clear()
Clear the state of the block.

Some blocks don't keep stateful attributes, so clear does nothing by default.

class copper.core.**Pipeline**(*blocks*, *name=None*)
Feedforward arrangement of blocks for processing data.

A *Pipeline* contains a set of :class:'PipelineBlock's which operate on data to produce a final output.

To create a pipeline, the following two rules are needed: blocks in a list processed in series, and blocks in a tuple are processed in parallel.

Blocks that are arranged to take multiple inputs should expect to take the corresponding number of inputs in the order they are given. It is up to the user constructing the pipeline to make sure that the arrangement of blocks makes sense.

Parameters **blocks** (*container*) – The blocks in the pipline, with lists processed in series and tuples processed in parallel.

named_blocks
dict – Dictionary of blocks in the pipeline. Keys are the names given to the blocks in the pipeline and values are the block objects.

process(*data*)
Calls the `process` method of each block in the pipeline, passing the outputs around as specified in the block structure.

Parameters **data** (*object*) – The input to the first block(s) in the pipeline. The type/format doesn't matter to copper, as long as the blocks you define accept it.

Returns **out** – The data output by the `process` method of the last block(s) in the pipeline.

Return type object

clear()
Calls the `clear` method on each block in the pipeline. The effect depends on the blocks themselves.

class copper.core.**PassthroughPipeline**(*blocks*, *expand_output=True*, *name=None*)
Convenience block for passing input along to output.

A passthrough pipeline block is useful when you want to process some data then provided both the processed output as well as the original input to another block downstream.

class copper.core.**CallablePipelineBlock**(*processor*, *name=None*, *hooks=None*)
A *PipelineBlock* that does not require persistent attributes.

Many *PipelineBlock* implementations don't require attributes to update on successive calls to the *process* method, but instead are essentially a function that can be called repeatedly. This class is for conveniently creating such a block.

Parameters

- **processor** (*callable(data)*) – Function that gets called when the block's *process* method is called. Should take a single input and return output which is compatible with whatever is connected to the block.
- **name** (*str, optional, default=None*) – Name of the block. By default, the name of the *processor* function is used.
- **hooks** (*list, optional, default=None*) – List of callables (callbacks) to run when after the block's *process* method is called.

common

Common processing tasks.

class copper.common.**Windower**(*length*)
Windows incoming data to a specific length.

Takes new input data and combines with past data to maintain a sliding window with optional overlap. The window length is specified directly, so the overlap depends on the length of the input.

The input length may change on each iteration, but the `Windower` must be cleared before the number of channels can change.

Parameters **length** (*int*) – Total number of samples to output on each iteration. This must be at least as large as the number of samples input to the windower on each iteration.

See also:

copper.common.Ensure2D Ensure input to the windower is 2D.

Examples

Basic use of a windower:

```
>>> import copper
>>> import numpy as np
>>> win = copper.Windower(4)
>>> win.process(np.array([[1, 2], [3, 4]]))
array([[ 0.,  0.,  1.,  2.],
       [ 0.,  0.,  3.,  4.]])
>>> win.process(np.array([[7, 8], [5, 6]]))
array([[ 1.,  2.,  7.,  8.],
       [ 3.,  4.,  5.,  6.]])
>>> win.clear()
>>> win.process(np.array([[1, 2], [3, 4]]))
array([[ 0.,  0.,  1.,  2.],
       [ 0.,  0.,  3.,  4.]])
```

If your data is 1-dimensional (shape `(n_samples,)`), use an *Ensure2D* block in front of the *Windower*:

```
>>> win = copper.Windower(4)
>>> p = copper.Pipeline([copper.Ensure2D(), win])
>>> p.process(np.array([1, 2]))
array([[ 0.,  0.,  1.,  2.]])
```

clear()
Clear the buffer containing previous input data.

process(*data*)
Add new data to the end of the window.

Parameters **data** (*array, shape (n_channels, n_samples)*) – Input data. `n_samples` must be less than or equal to the windower `length`.

Returns **out** – Output window with the input data at the end.

Return type array, shape (n_channels, length)

class copper.common.**Centerer**(*name=None*, *hooks=None*)
Centers data by subtracting out its mean.

process(*data*)
Center each row of the input.

Parameters **data** (*array, shape (n_channels, n_samples)*) – Input data.

Returns **out** – Input data that's been centered.

Return type array, shape (n_channels, n_samples)

class copper.common.**Filter**(*b*, *a=1*, *overlap=0*)
Filters incoming data with a time domain filter.

This filter implementation takes filter coefficients that are designed by the user – it merely applies the filter to the input, remembering the final inputs/outputs from the previous update and using them as initial conditions for the current update.

Parameters

- **b** (*ndarray*) – Numerator polynomial coefficients of the filter.
- **a** (*ndarray, optional*) – Denominator polynomial coefficients of the filter. Default is 1, meaning the filter is FIR.
- **overlap** (*int, optional*) – Number of samples overlapping in consecutive inputs. Needed for correct filter initial conditions in each filtering operation. Default is 0, meaning the final inputs/outputs of the previous update are used.

See also:

copper.common.Ensure2D Ensure input to the filter is 2D.

Examples

Design a filter using scipy and use the coefficients:

```
>>> import copper
>>> import numpy as np
>>> from scipy.signal import butter
>>> b, a = butter(4, 100/1000/2)
>>> f = copper.Filter(b, a)
>>> f.process(np.random.randn(1, 5))
array([...
```

Use a filter in combination with a *Windower*, making sure to account for overlapping data in consecutive filtering operations. Here, we'll use a window of length 5 and pass in 3 samples at a time, so there will be an overlap of 2 samples. The overlapping samples in each output will agree:

```
>>> w = copper.Windower(5)
>>> f = copper.Filter(b, a, overlap=2)
>>> p = copper.Pipeline([w, f])
>>> out1 = p.process(np.random.randn(1, 3))
>>> out2 = p.process(np.random.randn(1, 3))
>>> out1[:, -2:] == out2[:, :2]
array([[ True,  True]], dtype=bool)
```

clear()
Clears the filter initial conditions.

Clearing the initial conditions is important when starting a new recording if `overlap` is nonzero.

process(*data*)
Applies the filter to the input.

Parameters **data** (*ndarray, shape (n_channels, n_samples)*) – Input signals.

class copper.common.**FeatureExtractor**(*features*, *hooks=None*)
Computes multiple features from the input, concatenating the results.

Each feature should be able to take in the same data and output a 1D array, so overall output of the FeatureExtractor can be a single 1D array.

This block isn't strictly necessary, since you could just apply multiple feature blocks in parallel and the result of each will be passed to the next block. However, the block following feature computation typically expects the input to be a single array (or row) per data sample.

Parameters **features** (*list*) – List of (name, feature) tuples (i.e. implementing a `compute` method).

named_features
dict – Dictionary of features accessed by name.

feature_indices
dict – Dictionary of (start, stop) tuples indicating the bounds of each feature, accessed by name. Will be empty until after data is first passed through.

clear()
Clears the output array.

This should be called if the input is going to change form in some way (i.e. the shape of the input array changes).

process(*data*)
Run data through the list of features and concatenates the results.

The first pass (after a `clear` call) will be a little slow since the extractor needs to allocate the output array.

Parameters **data** (*array, shape (n_channels, n_samples)*) – Input data. Must be appropriate for all features.

Returns **out**

Return type array, shape (n_features,)

class copper.common.**Estimator**(*estimator*)
A pipeline block wrapper around scikit-learn's idea of an estimator.

An estimator is an object that can be trained with some data (`fit`) and, once trained, can output predictions from novel inputs. A common use-case for this block is to utilize a scikit-learn pipeline in the context of a axopy pipeline.

Parameters **estimator** (*object*) – An object implementing the scikit-learn Estimator interface (i.e. implementing `fit` and `predict` methods).

process(*data*)
Calls the estimator's `predict` method and returns the result.

class copper.common.**Transformer**(*transformer*, *hooks=None*)
A pipeline block wrapper around scikit-learn's idea of a transformer.

A transformer is trained with some data (`fit`) and, once trained, can output projections of the input data to some other space. A common example is projecting data in high-dimensional space to a lower-dimensional space using principal components analysis.

Parameters **transformer** (*object*) – An object implementing the scikit-learn Transformer interface (i.e. implementing `fit` and `transform` methods).

process(*data*)
Calls the transformer's `transform` method and returns the result.

class copper.common.**Ensure2D**(*orientation='row'*)
Transforms an array to ensure it has 2 dimensions.

Input with shape `(n,)` can be made to have shape `(n, 1)` or `(1, n)`.

Parameters **orientation** (*{'row', 'col'}, optional*) – Orientation of the output. If 'row', the output will have shape `(1, n)`, meaning the output is a row vector. This is the default behavior, useful when the data is something like samples of a 1-channel signal. If 'col', the output will have shape `(n, 1)`, meaning the output is a column vector.

Examples

Output row data:

```
>>> import numpy as np
>>> import copper
>>> block = copper.Ensure2D()
>>> block.process(np.array([1, 2, 3]))
array([[1, 2, 3]])
```

Output column data:

```
>>> block = copper.Ensure2D(orientation='col')
>>> block.process(np.array([1, 2, 3]))
array([[1],
       [2],
       [3]])
```

process (*data*)

Make sure data is 2-dimensional.

If the input already has two dimensions, it is unaffected.

Parameters **data** (*array, shape (n,)*) – Input data.

Returns **out** – Output data, with shape specified by `orientation`.

Return type array, shape (1, n) or (n, 1)

sources

Data streams for processing with a pipeline.

copper.sources.**segment** (*data*, *length*, *overlap=0*)

Generate segments of an array.

Each segment is of a specified length and optional overlap with the previous segment. Only segments of the specified length are retrieved (if segments don't fit evenly into the data).

Parameters

- **data** (*array, shape (n_channels, n_samples)*) – Data to segment.
- **length** (*int*) – Number of samples to retrieve in each chunk.
- **overlap** (*int, optional*) – Number of overlapping samples in consecutive chunks.

Yields **segment** (*array (n_channels, length)*) – Segment of the input array.

Examples

Segment a 2-channel recording:

```
>>> import numpy as np
>>> from copper.sources import segment
>>> x = np.arange(8).reshape(2, 4)
>>> x
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> seg = segment(x, 2)
>>> next(seg)
array([[0, 1],
       [4, 5]])
>>> next(seg)
array([[2, 3],
       [6, 7]])
```

Consecutive segments with overlapping samples agree:

```
>>> seg = segment(x, 3, overlap=2)
>>> next(seg)
array([[0, 1, 2],
       [4, 5, 6]])
>>> next(seg)
array([[1, 2, 3],
       [5, 6, 7]])
```

copper.sources.**segment_indices**(*n*, *length*, *overlap=0*)

Generate indices to segment an array.

Each segment is of a specified length with optional overlap with the previous segment. Only segments of the specified length are retrieved if they don't fit evenly into the the total length. The indices returned are meant to be used for slicing, e.g. `data[:, from:to]`.

Parameters

- **n** (*int*) – Number of samples to segment up.
- **length** (*int*) – Length of each segment.
- **overlap** (*int, optional*) – Number of overlapping samples in consecutive segments.

Yields

- **from** (*int*) – Index of the beginning of the segment with respect to the input array.
- **to** (*int*) – Index of the end of the segement with respect to the input array.

Examples

Basic usage – segment a 6-sample recording into segments of length 2:

```
>>> import numpy as np
>>> from copper.sources import segment_indices
>>> list(segment_indices(6, 2))
[(0, 2), (2, 4), (4, 6)]
```

Overlapping segments:

```
>>> list(segment_indices(11, 5, overlap=2))
[(0, 5), (3, 8), (6, 11)]
```

Python Module Index

C

Index