Signatory

Release 1.1.4

Patrick Kidger

Nov 18, 2019

DOCUMENTATION

Differentiable computations of the signature and logsignature transforms, on both CPU and GPU.

The Signatory project is hosted on GitHub.

CHAPTER ONE

INTRODUCTION

This is the documentation for the Signatory package, which provides facilities for calculating the signature and logsignature transforms of streams of data.

If you want to get started on using the signature transform in your code then check out *Simple example* for a simple demonstration.

If you want to know more about the mathematics of the signature transform and how to use it then see *What are signatures?* for a very brief introduction. Further links to papers discussing the subject in more detail can also be found there.

If you have any comments or queries about signatures or about this package (for example, bug reports or feature requests) then open an issue on GitHub.

CHAPTER

TWO

INSTALLATION

Available for Python 2.7, Python 3.5, Python 3.6, Python 3.7 and Linux, Mac, Windows. Requires PyTorch 1.2.0 or 1.3.0.

Install via:

```
pip install signatory==<SIGNATORY_VERSION>.<TORCH_VERSION>
```

where <SIGNATORY_VERSION> is the version of Signatory you would like to download (the most recent version is 1.1.4) and <TORCH_VERSION> is the version of PyTorch you are using.

Example

For example, if you are using PyTorch 1.3.0 and want Signatory 1.1.4, then you should run:

```
pip install signatory==1.1.4.1.3.0
```

Yes, this looks a bit odd. This is needed to work around limitations of PyTorch and pip.

Take care **not** to run `pip install signatory`, as this will likely download the wrong version.

After installation, just `import signatory` inside Python.

If you have any problems with installation then check the FAQ. If that doesn't help then feel free to open an issue.

2.1 Install from source

For most use-cases, the prebuilt binaries available as described above should be sufficient. However installing from source is also perfectly feasible, and usually not too tricky.

You'll need to have a C++ compiler installed and known to `pip`, and furthermore this must be the same compiler that PyTorch uses. (This is `msvc` on Windows, `gcc` on Linux, and `clang` on Macs.) You must have already installed PyTorch. (You don't have to compile PyTorch itself from source, though!)

Then run **either**

```
pip install signatory==<SIGNATORY_VERSION>.<TORCH_VERSION> --no-binary signatory
```

(where <SIGNATORY_VERSION> and <TORCH_VERSION> are as above.)

or

```
git clone https://github.com/patrick-kidger/signatory.git
cd signatory
python setup.py install
```

If you chose the first option then you'll get just the files necessary to run Signatory.

If you choose the second option then tests, benchmarking code, and code to build the documentation will also be provided. Subsequent to this,

- Tests can be run, see `python command.py test --help`.
  This requires installing iisignature and pytest.
- Speed and memory benchmarks can be performed, see `python command.py benchmark --help`.
  This requires installing iisignature, esig, and memory profiler.
- Documentation can be built via `python command.py docs`.
  This requires installing Sphinx, sphinx_rtd_theme and py2annotate.

Note: If on Linux then the commands stated above should probably work.

If on Windows then it is probably first necessary to run a command of the form

```
"C:/Program Files (x86)/Microsoft Visual Studio/2017/Enterprise/VC/Auxiliary/Build/
↪vcvars64.bat"
```

(the exact command will depend on your operating system and version of Visual Studio).

If on a Mac then the installation command should instead look like either

```
MACOSX_DEPLOYMENT_TARGET=10.9 CC=clang CXX=clang++ pip install signatory==<SIGNATORY_
↪VERSION>.<TORCH_VERSION> --no-binary signatory
```

or

```
MACOSX_DEPLOYMENT_TARGET=10.9 CC=clang CXX=clang++ python setup.py install
```

depending on the choice of installation method.

A helpful point of reference for getting this to work might be the official build scripts for Signatory.

CHAPTER

THREE

LIBRARY API

For quick reference these are a list of all provided functions, grouped by which reference page they are on.

Signatures

`signatory.signature`	Applies the signature transform to a stream of data.
`signatory.Signature`	`torch.nn.Module` wrapper around the `signatory.signature()` function.
`signatory.signature_channels`	Computes the number of output channels from a signature call.
`signatory.extract_signature_term`	Extracts a particular term from a signature.
`signatory.signature_combine`	Combines two signatures into a single signature.
`signatory.multi_signature_combine`	Combines multiple signatures into a single signature.

Logsignatures

`signatory.logsignature`	Applies the logsignature transform to a stream of data.
`signatory.LogSignature`	`torch.nn.Module` wrapper around the `signatory.logsignature()` function.
`signatory.logsignature_channels`	Computes the number of output channels from a logsignature call with `mode in ("words", "brackets")`.
`signatory.signature_to_logsignature`	Calculates the logsignature corresponding to a signature.
`signatory.SignatureToLogSignature`	`torch.nn.Module` wrapper around the `signatory.signature_to_logsignature()` function.

Path

`signatory.Path`	Calculates signatures and logsignatures on intervals of an input path.

Utilities

`signatory.max_parallelism`	Gets or sets the maximum amount of parallelism used in Signatory's computations.

Continued on next page

Table 4 – continued from previous page

`signatory.Augment`	Augmenting a stream of data before feeding it into a signature is often useful; the hope is to obtain higher-order information in the signature.
`signatory.all_words`	Computes the collection of all words up to length `depth` in an alphabet of size `channels`.
`signatory.lyndon_words`	Computes the collection of all Lyndon words up to length `depth` in an alphabet of size `channels`.
`signatory.lyndon_brackets`	Computes the collection of all Lyndon words, in their standard bracketing, up to length `depth` in an alphabet of size `channels`.

3.1 Signatures

At the heart of the package is the `signatory.signature()` function.

Note: It comes with quite a lot of optional arguments, but most of them won't need to be used for most use cases. See *Simple example* for a straightforward look at how to use it.

`signatory.`**`signature`**(*path: torch.Tensor*, *depth: int*, *stream: bool = False*, *basepoint: Union[bool, torch.Tensor] = False*, *inverse: bool = False*, *initial: Union[None, torch.Tensor] = None*) → torch.Tensor

Applies the signature transform to a stream of data.

The input `path` is expected to be a three-dimensional tensor, with dimensions (N, L, C), where N is the batch size, L is the length of the input sequence, and C denotes the number of channels. Thus each batch element is interpreted as a stream of data $(x_1, \ldots, x_L)$, where each $x_i \in \mathbb{R}^C$.

Let $f = (f_1, \ldots, f_C) \colon [0, 1] \to \mathbb{R}^C$, be the unique continuous piecewise linear path such that $f(\frac{i-1}{N-1}) = x_i$. Then and the signature transform of depth `depth` is computed, defined by

$$\mathrm{Sig}(\text{path}) = \left(\left(\,\underset{0<t_1<\cdots<t_k<1}{\int\cdots\int} \prod_{j=1}^{k} \frac{\mathrm{d}f_{i_j}}{\mathrm{d}t}(t_j)\mathrm{d}t_1\cdots\mathrm{d}t_k\right)_{1\leq i_1,\ldots,i_k\leq C}\right)_{1\leq k\leq \text{depth}}.$$

This gives a tensor of shape

$$(N, C + C^2 + \cdots + C^{\text{depth}}).$$

Parameters

- **`path`** (`torch.Tensor`) – The batch of input paths to apply the signature transform to.
- **`depth`** (*`int`*) – The depth to truncate the signature at.
- **`stream`** (*`bool, optional`*) – Defaults to False. If False then the usual signature transform of the whole path is computed. If True then the signatures of all paths $(x_1, \ldots, x_j)$, for $j = 2, \ldots, L$, are returned. (Or $j = 1, \ldots, L$ is `basepoint` is passed, see below.)
- **`basepoint`** (bool or `torch.Tensor`, optional) – Defaults to False. If `basepoint` is True then an additional point $x_0 = 0 \in \mathbb{R}^C$ is prepended to the path before the signature transform is applied. (If this is False then the signature transform is invariant to translations of the path, which may or may not be desirable. Setting this to True removes this invariance.) Alternatively it may be a `torch.Tensor` specifying the value of x_0, in which case it should have shape (N, C).

- **inverse** (*bool, optional*) – Defaults to False. If True then it is in fact the inverse signature that is computed. That is, we flip the input path along its stream dimension before computing the signature. If `stream` is True then each sub-path is the same as before, and are each individually flipped along their stream dimensions, and kept in the same order with respect to each other. (But without the extra computational overhead of actually doing all of these flips.) From a machine learning perspective it does not particularly matter whether the signature or the inverse signature is computed - both represent essentially the same information as each other.
- **initial** (None or `torch.Tensor`, optional) – Defaults to None. If it is a `torch.Tensor` then it must be of size $(N, C + C^2 + ... + C^{\text{depth}})$, corresponding to the signature of another path. Then this signature is pre-tensor-multiplied on to the signature of `path`. For a more thorough explanation, see *this example*. (The appropriate modifications are made if `inverse=True` or if `basepoint`.)

Returns

A `torch.Tensor`. Given an input `torch.Tensor` of shape (N, L, C), and input arguments `depth`, `basepoint`, `stream`, then the return value is, in pseudocode:

```
if stream:
    if basepoint is True or isinstance(basepoint, torch.Tensor):
        return torch.Tensor of shape (N, L, C + C^2 + ... + C^depth)
    else:
        return torch.Tensor of shape (N, L - 1, C + C^2 + ... + C^
↪depth)
else:
    return torch.Tensor of shape (N, C + C^2 + ... + C^depth)
```

Note that the number of output channels may be calculated via the convenience function `signatory.signature_channels()`.

class signatory.**Signature**(*depth: int*, *stream: bool = False*, *inverse: bool = False*, ***kwargs: Any*)

`torch.nn.Module` wrapper around the `signatory.signature()` function.

Parameters

- **depth** (*int*) – as `signatory.signature()`.
- **stream** (*bool, optional*) – as `signatory.signature()`.
- **inverse** (*bool, optional*) – as `signatory.signature()`.

forward(*path: torch.Tensor*, *basepoint: Union[bool, torch.Tensor] = False*, *initial: Union[None, torch.Tensor] = None*) → torch.Tensor

The forward operation.

Parameters

- **path** (`torch.Tensor`) – As `signatory.signature()`.
- **basepoint** (bool or `torch.Tensor`, optional) – As `signatory.signature()`.
- **initial** (None or `torch.Tensor`, optional) – As `signatory.signature()`.

Returns As `signatory.signature()`.

signatory.**signature_channels**(*channels: int*, *depth: int*) → int

Computes the number of output channels from a signature call. Specifically, it computes

$$\text{channels} + \text{channels}^2 + \cdots + \text{channels}^{\text{depth}}.$$

Parameters

- **channels** (*int*) – The number of channels in the input; that is, the dimension of the space that the input path resides in.
- **depth** (*int*) – The depth of the signature that is being computed.

Returns An int specifying the number of channels in the signature of the path.

signatory.**extract_signature_term**(*sigtensor: torch.Tensor*, *channels: int*, *depth: int*) → torch.Tensor

Extracts a particular term from a signature.

The signature to depth d of a batch of paths in $\mathbb{R}^C$ is a tensor with $C + C^2 + \cdots + C^d$ channels. (See `signatory.signature()`.) This function extracts the `depth` term of that, returning a tensor with just C^{depth} channels.

Parameters

- **sigtensor** (`torch.Tensor`) – The signature to extract the term from. Should be a result from the `signatory.signature()` function.
- **channels** (*int*) – The number of input channels C.
- **depth** (*int*) – The depth of the term to be extracted from the signature.

Returns The `torch.Tensor` corresponding to the `depth` term of the signature.

signatory.**signature_combine**(*sigtensor1: torch.Tensor*, *sigtensor2: torch.Tensor*, *input_channels: int*, *depth: int*, *inverse: bool = False*) → torch.Tensor

Combines two signatures into a single signature.

Usage is most clear by example. See *Combining signatures*.

See also `signatory.multi_signature_combine()` for a more general version.

Parameters

- **sigtensor1** (`torch.Tensor`) – The signature of a path, as returned by `signatory.signature()`. This should be a two-dimensional tensor.
- **sigtensor2** (`torch.Tensor`) – The signature of a second path, as returned by `signatory.signature()`, with the same shape as `sigtensor1`. Note that when the signature of the second path was created, it should have been called with `basepoint` set to the final value of the path that created `sigtensor1`. (See *Combining signatures*.)
- **input_channels** (*int*) – The number of channels in the two paths that were used to compute `sigtensor1` and `sigtensor2`. This must be the same for both `sigtensor1` and `sigtensor2`.
- **depth** (*int*) – The depth that `sigtensor1` and `sigtensor2` have been calculated to. This must be the same for both `sigtensor1` and `sigtensor2`.
- **inverse** (*bool, optional*) – Defaults to False. Whether `sigtensor1` and `sigtensor2` were created with `inverse=True`. This must be the same for both `sigtensor1` and `sigtensor2`.

Returns Let `path1` be the path whose signature is `sigtensor1`. Let `path2` be the path whose signature is `sigtensor2`. Then this function returns the signature of the concatenation of `path1` and `path2` along their stream dimension.

Danger: There is a subtle bug which can occur when using this function incautiously. Make sure that `sigtensor2` is created with an appropriate `basepoint`, see *Combining signatures*.

If this is not done then the return value of this function will be essentially meaningless numbers.

`signatory.multi_signature_combine` (*sigtensors: List[torch.Tensor], input_channels: int, depth: int, inverse: bool = False*) → torch.Tensor

Combines multiple signatures into a single signature.

See also `signatory.signature_combine()` for a simpler version.

Parameters

- **sigtensors** (list of `torch.Tensor`) – Signature of multiple paths, all of the same shape. They should all be two-dimensional tensors.
- **input_channels** (*int*) – As `signatory.signature_combine()`.
- **depth** (*int*) – As `signatory.signature_combine()`.
- **inverse** (*bool, optional*) – As `signatory.signature_combine()`.

Returns Let `sigtensors` be a list of tensors, call them sigtensor_i for $i = 0, 1, \ldots, k$. Let path_i be the path whose signature is sigtensor_i. Then this function returns the signature of the concatenation of path_i along their stream dimension.

> **Danger:** Make sure that each element of `sigtensors` is created with an appropriate `basepoint`, as with `signatory.signature_combine()`.

3.2 Logsignatures

`signatory.logsignature` (*path: torch.Tensor, depth: int, stream: bool = False, basepoint: Union[bool, torch.Tensor] = False, inverse: bool = False, mode: str = 'words'*) → torch.Tensor

Applies the logsignature transform to a stream of data.

The `modes` argument determines how the logsignature is represented.

Note that if performing many logsignature calculations for the same depth and size of input, then you will see a performance boost (at the cost of using a little extra memory) by using `signatory.LogSignature` instead of `signatory.logsignature()`.

Parameters

- **path** (`torch.Tensor`) – as `signatory.signature()`.
- **depth** (*int*) – as `signatory.signature()`.
- **stream** (*bool, optional*) – as `signatory.signature()`.
- **basepoint** (bool or `torch.Tensor`, optional) – as `signatory.signature()`.
- **inverse** (*bool, optional*) – as `signatory.signature()`.
- **mode** (*str, optional*) – Defaults to `"words"`. How the output should be presented. Valid values are `"words"`, `"brackets"`, or `"expand"`. Precisely what each of these options mean is described in the "Returns" section below. For machine learning applications, `"words"` is the appropriate choice. The other two options are mostly only interesting for mathematicians.

Returns

A `torch.Tensor`, of almost the same shape as the tensor returned from `signatory.signature()` called with the same arguments.

If `mode == "expand"` then it will be exactly the same shape as the returned tensor from *signatory.signature()*.

If `mode in ("brackets", "words")` then the channel dimension will instead be of size `signatory.logsignature_channels(path.size(-1), depth)`. (Where `path.size(-1)` is the number of input channels.)

The different modes correspond to different mathematical representations of the logsignature.

Tip: If you haven't studied tensor algebras and free Lie algebras, and none of the following explanation makes sense to you, then you probably want to leave `mode` on its default value of `"words"` and it will all be fine!

If `mode == "expand"` then the logsignature is presented as a member of the tensor algebra; the numbers returned correspond to the coefficients of all words in the tensor algebra.

If `mode == "brackets"` then the logsignature is presented in terms of the coefficients of the Lyndon basis of the free Lie algebra.

If `mode == "words"` then the logsignature is presented in terms of the coefficients of a particular computationally efficient basis of the free Lie algebra (that is not a Hall basis). Every basis element is given as a sum of Lyndon brackets. When each bracket is expanded out and the sum computed, the sum will contain precisely one Lyndon word (and some collection of non-Lyndon words). Moreover every Lyndon word is represented uniquely in this way. We identify these basis elements with each corresponding Lyndon word. This is natural as the coefficients in this basis are found just by extracting the coefficients of all Lyndon words from the tensor algebra representation of the logsignature.

In all cases, the ordering corresponds to the ordering on words given by first ordering the words by length, and then ordering each length class lexicographically.

class signatory.**LogSignature**(*depth: int*, *stream: bool = False*, *inverse: bool = False*, *mode: str = 'words'*, ***kwargs: Any*)

`torch.nn.Module` wrapper around the *signatory.logsignature()* function.

This `torch.nn.Module` performs certain optimisations to allow it to calculate multiple logsignatures faster than multiple calls to *signatory.logsignature()*.

Specifically, these optimisations will apply if this `torch.nn.Module` is called with an input `path` with the same number of channels as the last input `path` it was called with, as is likely to be very common in machine learning set-ups. For larger depths or numbers of channels, this speedup will be substantial.

Parameters

- **depth** (*int*) – as *signatory.logsignature()*.
- **stream** (*bool, optional*) – as *signatory.logsignature()*.
- **inverse** (*bool, optional*) – as *signatory.logsignature()*.
- **mode** (*str, optional*) – as *signatory.logsignature()*.

forward(*path: torch.Tensor*, *basepoint: Union[bool, torch.Tensor] = False*) → torch.Tensor

The forward operation.

Parameters

- **path** (`torch.Tensor`) – As *signatory.logsignature()*.
- **basepoint** (*bool or torch.Tensor, optional*) – As *signatory.logsignature()*.

Returns As `signatory.logsignature()`.

prepare(*in_channels: int*) → None

Prepares for computing logsignatures for paths of the specified number of channels. This will be done anyway automatically whenever this `torch.nn.Module` is called, if it hasn't been called already; this method simply allows to have it done earlier, for example when benchmarking.

Parameters **in_channels** (*int*) – The number of input channels of the path that this instance will subsequently be called with. (corresponding to `path.size(-1)`.)

signatory.**logsignature_channels**(*in_channels: int*, *depth: int*) → int

Computes the number of output channels from a logsignature call with `mode in ("words", "brackets")`.

Parameters

- **in_channels** (*int*) – The number of channels in the input; that is, the dimension of the space that the input path resides in. If calling `signatory.logsignature()` with argument `path` then `in_channels` should be equal to `path.size(-1)`.
- **depth** (*int*) – The depth of the signature that is being computed.

Returns An `int` specifying the number of channels in the logsignature of the path.

signatory.**signature_to_logsignature**(*signature: torch.Tensor*, *channels: int*, *depth: int*, *stream: bool = False*, *mode: str = 'words'*) → torch.Tensor

Calculates the logsignature corresponding to a signature.

Parameters

- **signature** (`torch.Tensor`) – The result of a call to `signatory.signature()`.
- **channels** (*int*) – The number of input channels of the `path` that `signatory.signature()` was called with.
- **depth** (*int*) – The value of `depth` that `signatory.signature()` was called with.
- **stream** (*bool, optional*) – Defaults to False. The value of `stream` that `signatory.signature()` was called with.
- **mode** (*str, optional*) – Defaults to `"words"`. As `signatory.logsignature()`.

Example

```
import signatory
import torch
batch, stream, channels = 8, 8, 8
depth = 3
path = torch.rand(batch, stream, channels)
signature = signatory.signature(path, depth)
logsignature = signatory.signature_to_logsignature(signature, channels, depth)
```

Returns A `torch.Tensor` representing the logsignature corresponding to the given signature. See `signatory.logsignature()`.

class signatory.**SignatureToLogSignature**(*channels: int*, *depth: int*, *stream: bool = False*, *mode: str = 'words'*, ***kwargs: Any*)

`torch.nn.Module` wrapper around the `signatory.signature_to_logsignature()` function.

Calling this `torch.nn.Module` on an input `signature` with the same number of channels as the last `signature` it was called with will be faster than multiple calls to the *`signatory.signature_to_logsignature()`* function, in the same way that *`signatory.LogSignature`* will be faster than *`signatory.logsignature()`*.

Parameters

- **`channels`** (*int*) – as *`signatory.signature_to_logsignature()`*.
- **`depth`** (*int*) – as *`signatory.signature_to_logsignature()`*.
- **`stream`** (*bool, optional*) – as *`signatory.signature_to_logsignature()`*.
- **`mode`** (*str, optional*) – as *`signatory.signature_to_logsignature()`*.

`forward`(*signature: torch.Tensor*) → torch.Tensor

The forward operation.

Parameters **`signature`** (`torch.Tensor`) – As *`signatory.signature_to_logsignature()`*.

Returns As *`signatory.signature_to_logsignature()`*.

3.3 Path

`class` `signatory.`**`Path`**(*path: torch.Tensor, depth: int, basepoint: Union[bool, torch.Tensor] = False*)

Calculates signatures and logsignatures on intervals of an input path.

By doing some precomputation, it can rapidly calculate the signature or logsignature over any slice of the input path. This is particularly useful if you need the signature or logsignature of a path over many different intervals: using this class will be much faster than computing the signature or logsignature of each sub-path each time.

Parameters

- **`path`** (*`torch.Tensor`*) – As *`signatory.signature()`*.
- **`depth`** (*int*) – As *`signatory.signature()`*.
- **`basepoint`** (*bool or `torch.Tensor`, optional*) – As *`signatory.signature()`*.

`signature`(*start: Optional[int] = None, end: Optional[int] = None*) → torch.Tensor

Returns the signature on a particular interval.

Parameters

- **`start`** (*int or None, optional*) – Defaults to the start of the path. The start point of the interval to calculate the signature on.
- **`end`** (*int or None, optional*) – Defaults to the end of the path. The end point of the interval to calculate the signature on.

Returns

The signature on the interval `[start, end]`.

In the simplest case, when *`path`* and *`depth`* are the arguments that this class was initialised with (and `basepoint` was not passed), then this function returns a value equal to `signatory.signature(path[start:end], depth)`.

In general, let `p = torch.cat(self.path, dim=1)`, so that it is all given paths (including those *`path`* from both initialistion and *`signatory.Path.update()`*, and any `basepoint`) concatenated together. Then this function will return a value equal to `signatory.signature(p[start:end], depth)`.

logsignature (*start: Optional[int] = None*, *end: Optional[int] = None*, *mode: str = 'words'*) → torch.Tensor
Returns the logsignature on a particular interval.

Parameters

- **start** (*int or None, optional*) – As *signatory.Path.signature()*.
- **end** (*int or None, optional*) – As *signatory.Path.signature()*.
- **mode** (*str, optional*) – As *signatory.logsignature()*.

Returns The logsignature on the interval `[start, end]`. See the documentation for *signatory.Path.signature()*.

update (*path: torch.Tensor*) → None
Concatenates the given path onto the path already stored.

This means that the signature of the new overall path can now be asked for via *signatory.Path.signature()*. Furthermore this will be dramatically faster than concatenating the two paths together and then creating a new Path object: the 'concatenation' occurs implicitly, without actually involving any recomputation or reallocation of memory.

Parameters path (*torch.Tensor*) – The path to concatenate on. As *signatory.signature()*.

property path
The path(s) that this Path was created with.

property depth
The depth that Path has calculated the signature to.

size (*index: Optional[int] = None*) → Union[int, torch.Size]
The size of the input path. As `torch.Tensor.size()`.

Parameters index (*int or None, optional*) – As `torch.Tensor.size()`.

Returns As `torch.Tensor.size()`.

property shape
The shape of the input path. As `torch.Tensor.shape`.

channels () → int
The number of channels of the input stream.

signature_size (*index: Optional[int] = None*) → Union[int, torch.Size]
The size of the signature of the path. As `torch.Tensor.size()`.

Parameters index (*int or None, optional*) – As `torch.Tensor.size()`.

Returns As `torch.Tensor.size()`.

property signature_shape
The shape of the signature of the path. As `torch.Tensor.shape`.

signature_channels () → int
The number of signature channels; as *signatory.signature_channels()*.

logsignature_size (*index: Optional[int] = None*) → Union[int, torch.Size]
The size of the logsignature of the path. As `torch.Tensor.size()`.

Parameters index (*int or None, optional*) – As `torch.Tensor.size()`.

Returns As `torch.Tensor.size()`.

property logsignature_shape
The shape of the logsignature of the path. As `torch.Tensor.shape`.

logsignature_channels() → int
The number of logsignature channels; as `signatory.logsignature_channels()`.

> **Warning:** If repeatedly making forward and backward passes (for example when training a neural network) and you have a learnt layer before the `signatory.Path`, then make sure to construct a new `signatory.Path` object for each forward pass.
>
> Reusing the same object between forward passes will mean that signatures aren't computed using the latest information, as the internal buffers will still correspond to the data passed in when the `signatory.Path` object was first constructed.

3.4 Utilities

The following miscellaneous operations are provided as a convenience.

---

signatory.**max_parallelism**(*value: Optional[int] = None*) → int
Gets or sets the maximum amount of parallelism used in Signatory's computations. Higher values will typically result in quicker computations but will use more memory.

Calling without arguments will return the current value. Passing a value of 1 will disable parallelism. Passing `-1`, `math.inf`, `np.inf` or `float('inf')` will enable unlimited parallelism.

---

class signatory.**Augment**(*in_channels: int, layer_sizes: Tuple[int, ...], kernel_size: int, stride: int = 1, padding: int = 0, dilation: int = 1, bias: bool = True, activation: Callable[[torch.Tensor], torch.Tensor] = <function relu>, include_original: bool = True, include_time: bool = True, **kwargs: Any*)
Augmenting a stream of data before feeding it into a signature is often useful; the hope is to obtain higher-order information in the signature. One way to do this is in a data-dependent way is to apply a feedforward neural network to sections of the stream, so as to obtain another stream; on this stream the signature is then applied; that is what this `torch.nn.Module` does.

Thus this `torch.nn.Module` is essentially unrelated to signatures, but is provided as it is often useful in the same context. As described in Deep Signature Transforms – Bonnier et al. 2019, it is often advantageous to augment a path before taking the signature.

The input path is expected to be a three-dimensional tensor, with dimensions (N, L, C), where N is the batch size, L is the length of the input sequence, and C denotes the number of channels. Thus each batch element is interpreted as a stream of data $(x_1, \ldots, x_L)$, where each $x_i \in \mathbb{R}^C$.

Then this stream may be 'augmented' via some function

$$\phi \colon \mathbb{R}^{C \times k} \to \mathbb{R}^{\widehat{C}}$$

giving a stream of data

$$\left(\phi(x_1, ...x_k), \ldots, \phi(x_{n-k+1}, \ldots, x_n)\right),$$

which is essentially a three-dimensional tensor with dimensions $(N, L - k + 1, \widehat{C})$.

Thus this essentially operates as a one dimensional convolution, except that a whole network is swept across the input, rather than just a single convolutional layer.

Both the original stream and time can be specifically included in the augmentation. (This usually tends to give better empirical results.) For example, if both `include_original` is True and `include_time` is True, then each $\phi(x_i, ...x_{k+i-1})$ is of the form

$$\left(\frac{i}{T}, x_i, \varphi(x_i, ...x_{k+i-1})\right).$$

where T is a constant appropriately chosen so that the first entry moves between 0 and 1 as i varies. (Specifically, $T = L - k + 1 + 2 \times \text{padding}$.)

Parameters

- **in_channels** (*int*) – Number of channels C in the input stream.
- **layer_sizes** (*tuple of int*) – Specifies the sizes of the layers of the feedforward neural network to apply to the stream. The final value of this tuple specifies the number of channels in the augmented stream, corresponding to the value $\widehat{C}$ in the preceding discussion.
- **kernel_size** (*int*) – Specifies the size of the kernel to slide over the stream, corresponding to the value k in the preceding discussion.
- **stride** (*int, optional*) – Defaults to 1. How far to move along the input stream before re-applying the feedforward neural network. Thus the output stream is given by

$$(\phi(x_1, \ldots, x_k), \phi(x_{1+\text{stride}}, \ldots, x_{k+2\times\text{stride}}), \phi(x_{1+2\times\text{stride}}, \ldots, x_{k+2\times\text{stride}}), \ldots)$$

- **padding** (*int, optional*) – Defaults to 0. How much zero padding to add to either end of the the input stream before sweeping the feedforward neural network along it.
- **dilation** (*int, optional*) – The spacing between input elements given to the feedforward neural network. Defaults to 1. Harder to describe; see the equivalent argument for `torch.nn.Conv1d`.
- **bias** (*bool, optional*) – Defaults to True. Whether to use biases in the neural network.
- **activation** (*callable, optional*) – Defaults to ReLU. The activation function to use in the feedforward neural network.
- **include_original** (*bool, optional*) – Defaults to True. Whether or not to include the original stream (pre-augmentation) in the augmented stream.
- **include_time** (*bool, optional*) – Defaults to True. Whether or not to also augment the stream with a 'time' value. These are values in $[0, 1]$ corresponding to how far along the stream dimension the element is.

Note: Thus the resulting stream of data has shape $(N, L,$

```
out_channels = layer_sizes[-1]
if include_original:
    out_channels += in_channels
if include_time:
    out_channels += 1
```

forward(*x: torch.Tensor*) → torch.Tensor

The forward operation.

Parameters **x** (*torch.Tensor*) – The path to augment.

Returns The augmented path.

`signatory.`**all_words**(*channels: int*, *depth: int*) $\to$ List[List[int]]

Computes the collection of all words up to length `depth` in an alphabet of size `channels`. Each letter is represented by an integer i in the range $0 \leq i <$ channels.

Signatures may be thought of as a sum of coefficients of words. This gives the words in the order that they correspond to the values returned by *signatory.signature()*.

Logsignatures may be thought of as a sum of coefficients of words. This gives the words in the order that they correspond to the values returned by *signatory.logsignature()* with `mode="expand"`.

Parameters

- **channels** (*int*) – The size of the alphabet.
- **depth** (*int*) – The maximum word length.

Returns A list of lists of integers. Each sub-list corresponds to one word. The words are ordered by length, and then ordered lexicographically within each length class.

`signatory.`**lyndon_words**(*channels: int*, *depth: int*) $\to$ List[List[int]]

Computes the collection of all Lyndon words up to length `depth` in an alphabet of size `channels`. Each letter is represented by an integer i in the range $0 \leq i <$ channels.

Logsignatures may be thought of as a sum of coefficients of Lyndon words. This gives the words in the order that they correspond to the values returned by *signatory.logsignature()* with `mode="words"`.

Parameters

- **channels** (*int*) – The size of the alphabet.
- **depth** (*int*) – The maximum word length.

Returns A list of lists of integers. Each sub-list corresponds to one Lyndon word. The words are ordered by length, and then ordered lexicographically within each length class.

`signatory.`**lyndon_brackets**(*channels: int*, *depth: int*) $\to$ List[Union[int, List]]

Computes the collection of all Lyndon words, in their standard bracketing, up to length `depth` in an alphabet of size `channels`. Each letter is represented by an integer i in the range $0 \leq i <$ channels.

Logsignatures may be thought of as a sum of coefficients of Lyndon brackets. This gives the brackets in the order that they correspond to the values returned by *signatory.logsignature()* with `mode="brackets"`.

Parameters

- **channels** (*int*) – The size of the alphabet.
- **depth** (*int*) – The maximum word length.

Returns A list. Each element corresponds to a single Lyndon word with its standard bracketing. The words are ordered by length, and then ordered lexicographically within each length class.

CHAPTER

FOUR

EXAMPLES

4.1 Simple example

Here's a very simple example on using *signatory.signature()*.

```
import torch
import signatory
# Create a tensor of shape (2, 10, 5)
# Recall that the order of dimensions is (batch, stream, channel)
path = torch.rand(2, 10, 5)
# Take the signature to depth 3
sig = signatory.signature(path, 3)
# sig is of shape (2, 155)
```

In this example,`path` is a three dimensional tensor,and the returned tensor is two dimensional. The first dimension of `path` corresponds to the batch dimension,and indeed we can see that this dimension is also in the shape of `sig`.

The second dimension of `path` corresponds to the 'stream' dimension,whilst the third dimension corresponds to channels. Mathematically speaking,that means that each batch element of `path` is interpreted as a sequence of points $x_1, \ldots, x_{10}$,with each $x_i \in \mathbb{R}^5$.

The output `sig` has batch dimension of size 2,just like the input. Its other dimension is of size 155. This is the number of terms in the depth-3 signature of a path with 5 channels. (This can also be computed with the helper function *signatory.signature_channels()*.)

4.2 Online computation of signatures

Suppose we have the signature of a stream of data $x_1, \ldots, x_{1000}$. Subsequently some more data arrives,say $x_{1001}, \ldots, x_{1007}$. It is possible to calculate the signature of the whole stream of data $x_1, \ldots, x_{1007}$ with just this information. It is not necessary to compute the signature of the whole path from the beginning!

In code,this problem can be solved like this:

```
import torch
import signatory

# Generate a path X
# Recall that the order of dimensions is (batch, stream, channel)
X = torch.rand(1, 1000, 5)
```

(continues on next page)

(continued from previous page)

```
# Calculate its signature to depth 3
sig_X = signatory.signature(X, 3)

# Generate some more data for the path
Y = torch.rand(1, 7, 5)
# Calculate the signature of the overall path
final_X = X[:, -1, :]
sig_XY = signatory.signature(Y, 3, basepoint=final_X, initial=sig_X)

# This is equivalent to
XY = torch.cat([X, Y], dim=1)
sig_XY = signatory.signature(XY, 3)
```

As can be seen,two pieces of information need to be provided: the final value of `X` along the stream dimension,and the signature of `X`. But not `X` itself.

The first method (using the `initial` argument) will be much quicker than the second (simpler) method. The first method efficiently uses just the new information `Y`,whilst the second method unnecessarily iterates over all of the old information `X`.

In particular note that we only needed the last value of `X`. If memory efficiency is a concern,then by using the first method we can discard the other 999 terms of `X` without an issue!

Note: If the signature of `Y` on its own was also of interest,then it is possible to compute this first,and then combine it with `sig_X` to compute `sig_XY`. See *Combining signatures*.

4.3 Combining signatures

Suppose we have two paths,and want to combine their signatures. That is,we know the signatures of the two paths,and would like to know the signature of the two paths concatenated together. This can be done with the *`signatory.signature_combine()`* function.

```
import torch
import signatory

depth = 3
input_channels = 5
path1 = torch.rand(1, 10, input_channels)
path2 = torch.rand(1, 5, input_channels)
sig_path1 = signatory.signature(path1, depth)
sig_path2 = signatory.signature(path2, depth,
                                basepoint=path1[:, -1, :])

### OPTION 1: efficient, using signature_combine
sig_combined = signatory.signature_combine(sig_path1, sig_path2,
                                           input_channels, depth)

### OPTION 2: inefficient, without using signature_combine
path_combined = torch.cat([path1, path2], dim=1)
sig_combined = signatory.signature(path_combined, depth)
```

(continues on next page)

(continued from previous page)

```
### Both options will produce the same value for sig_combined
```

Danger: Note in particular that the end of `path1` is used as the `basepoint` when calculating `sig_path2` in Option 1. It is important that `path2` starts from the same place that `path1` finishes. Otherwise there will be a jump between the end of `path1` and the start of `path2` which the signature will not see.

If it is known that `path1[:, -1, :] == path2[:, 0, :]`,so that in fact `path1` does finish where `path2` starts,then only in this case can the use of `basepoint` safely be skipped. (And if `basepoint` is set to this value then it will not change the result.)

With Option 2 it is clearest what is being computed. However this is also going to be slower: the signature of `path1` is already known,but Option 2 does not use this information at all,and instead performs a lot of unnecessary computation. Furthermore its calculation requires holding all of `path1` in memory,instead of just `path1[:, -1, :]`.

Note how with Option 1,once `sig_path1` has been computed,then the only thing that must now be held in memory is `sig_path1` and `path1[:, -1, :]`. This means that the amount of memory required is independent of the length of `path1`. Thus if `path` is very long,or can grow to arbitrary length as time goes by,then the use of this option (over Option 2) is crucial.

Tip: Combining signatures in this way is the most sensible way to do things if the signature of `path2` is actually desirable information on its own.

However if only the signature of the combined path is of interest,then this can be computed even more efficiently by

```
sig_path1 = signatory.signature(path1, depth)
sig_combined = signatory.signature(path2, depth,
                                   basepoint=path1[:, -1, :],
                                   initial=sig_path1)
```

For further examples of this nature,see *Online computation of signatures*.

4.4 Signatures on intervals

The basic *`signatory.signature()`* function computes the signature of a whole stream of data. Sometimes we have a whole stream of data,and then want to compute the signature of just the data sitting in some subinterval.

Naively,we could just slice it:

```
import torch
import signatory
# WARNING! THIS IS SLOW AND INEFFICIENT CODE
path = torch.rand(1, 1000, 5)
sig1 = signatory.signature(path[:, :40, :], 3)
```

(continues on next page)

(continued from previous page)

```
sig2 = signatory.signature(path[:, 300:600, :], 3)
sig3 = signatory.signature(path[:, 400:990, :], 3)
sig4 = signatory.signature(path[:, 700:, :], 3)
sig5 = signatory.signature(path, 3)
```

However in this scenario it is possible to be much more efficient by doing some precomputation,which can then allow for computing such signatures very rapidly. This is done by the `signatory.Path` class.

```
import torch
import signatory

path = torch.rand(1, 1000, 5)
path_class = signatory.Path(path, 3)
sig1 = path_class.signature(0, 40)
sig2 = path_class.signature(300, 600)
sig3 = path_class.signature(400, 990)
sig4 = path_class.signature(700, None)
sig5 = path_class.signature()
```

In fact,the `signatory.Path` class supports adding data to it as well:

```
import torch
import signatory

path1 = torch.rand(1, 1000, 5)
path_class = signatory.Path(path1, 3)
# path_class is considering a path of length 1000
# calculate signatures as normal
sig1 = path_class.signature(40, None)
sig2 = path_class.signature(500, 600)
# more data arrives
path2 = torch.rand(1, 200, 5)
path_class.update(path2)
# path_class is now considering a path of length 1200
sig3 = path_class.signature(900, 1150)
```

Note: To be able to compute signatures over intervals like this,then of course `signatory.Path` must hold information about the whole stream of data in memory.

If only the signature of the whole path is of interest then the main `signatory.signature()` function will work fine.

If the signature of a path for which data continues to arrive (analogous to the use of `signatory.Path.update()` above) is of interest,then see *Online computation of signatures*,which demonstrates how to efficiently use the `signatory.signature()` function in this way.

If the signature on adjacent disjoint intervals is required,and the signature on the union of these intervals is desired,then see *Combining signatures* for how to compute the signature on each of these intervals,and how to efficiently combine them to find the signature on larger intervals. This then avoids the overhead of the `signatory.Path` class.

4.5 Translation invariance

The signature is translation invariant. That is,given some stream of data $x_1, \ldots, x_n$ with $x_i \in \mathbb{R}^c$,and some $y \in \mathbb{R}^c$,then the signature of $x_1, \ldots, x_n$ is equal to the signature of $x_1 + y, \ldots, x_n + y$.

Sometimes this is desirable,sometimes it isn't. If it isn't desirable,then the simplest solution is to add a 'basepoint'. That is,add a point $0 \in \mathbb{R}^c$ to the start of the path. This will allow us to notice any translations,as the signature of $0, x_1, \ldots, x_n$ and the signature of $0, x_1 + y, \ldots, x_n + y$ will be different.

In code,this can be accomplished very easily by using the `basepoint` argument. Simply set it to `True` to add such a basepoint to the path before taking the signature:

```
import torch
import signatory
path = torch.rand(2, 10, 5)
sig = signatory.signature(path, 3, basepoint=True)
```

4.6 Using signatures in neural networks

In principle a simple augment-signature-linear model is enough to achieve universal approximation:

```
import signatory
import torch
from torch import nn


class SigNet(nn.Module):
    def __init__(self, in_channels, out_dimension, sig_depth):
        super(SigNet, self).__init__()
        self.augment = signatory.Augment(in_channels=in_channels,
                                         layer_sizes=(),
                                         kernel_size=1,
                                         include_original=True,
                                         include_time=True)
        self.signature = signatory.Signature(depth=sig_depth)
        # +1 because signatory.Augment is used to add time as well
        sig_channels = signatory.signature_channels(channels=in_channels + 1,
                                                    depth=sig_depth)
        self.linear = torch.nn.Linear(sig_channels,
                                      out_dimension)

    def forward(self, inp):
        # inp is a three dimensional tensor of shape (batch, stream, in_channels)
        x = self.augment(inp)
        if x.size(1) <= 1:
            raise RuntimeError("Given an input with too short a stream to take the
↪"
                               " signature")
        # x in a three dimensional tensor of shape (batch, stream, in_channels +␣
↪1),
        # as time has been added as a value
        y = self.signature(x, basepoint=True)
        # y is a two dimensional tensor of shape (batch, terms), corresponding to
```

(continues on next page)

(continued from previous page)

```
        # the terms of the signature
        z = self.linear(y)
        # z is a two dimensional tensor of shape (batch, out_dimension)
        return z
```

Whilst in principle this exhibits universal approximation,adding some learnt transformation before the signature transform tends to improve things. See Deep Signature Transforms – Bonnier et al. 2019. Thus we might improve our model:

```
import signatory
import torch
from torch import nn


class SigNet2(nn.Module):
    def __init__(self, in_channels, out_dimension, sig_depth):
        super(SigNet2, self).__init__()
        self.augment = signatory.Augment(in_channels=in_channels,
                                         layer_sizes=(8, 8, 2),
                                         kernel_size=4,
                                         include_original=True,
                                         include_time=True)
        self.signature = signatory.Signature(depth=sig_depth)
        # +3 because signatory.Augment is used to add time, and 2 other channels,
        # as well
        sig_channels = signatory.signature_channels(channels=in_channels + 3,
                                                    depth=sig_depth)
        self.linear = torch.nn.Linear(sig_channels,
                                      out_dimension)

    def forward(self, inp):
        # inp is a three dimensional tensor of shape (batch, stream, in_channels)
        x = self.augment(inp)
        if x.size(1) <= 1:
            raise RuntimeError("Given an input with too short a stream to take the
↪"
                               " signature")
        # x in a three dimensional tensor of shape (batch, stream, in_channels +␣
↪3)
        y = self.signature(x, basepoint=True)
        # y is a two dimensional tensor of shape (batch, sig_channels),
        # corresponding to the terms of the signature
        z = self.linear(y)
        # z is a two dimensional tensor of shape (batch, out_dimension)
        return z
```

The *signatory.Signature* layer can be used multiple times in a neural network. In this next example the first *signatory.Signature* layer is called with `stream` as True,so that the stream dimension is preserved. This means that the signatures of all intermediate streams are returned as well. So as we still have a stream dimension,it is reasonable to take the signature again.

```
import signatory
import torch
```

(continues on next page)

(continued from previous page)

```
from torch import nn


class SigNet3(nn.Module):
    def __init__(self, in_channels, out_dimension, sig_depth):
        super(SigNet3, self).__init__()
        self.augment1 = signatory.Augment(in_channels=in_channels,
                                          layer_sizes=(8, 8, 4),
                                          kernel_size=4,
                                          include_original=True,
                                          include_time=True)
        self.signature1 = signatory.Signature(depth=sig_depth,
                                              stream=True)

        # +5 because self.augment1 is used to add time, and 4 other
        # channels, as well
        sig_channels1 = signatory.signature_channels(channels=in_channels + 5,
                                                     depth=sig_depth)
        self.augment2 = signatory.Augment(in_channels=sig_channels1,
                                          layer_sizes=(8, 8, 4),
                                          kernel_size=4,
                                          include_original=False,
                                          include_time=False)
        self.signature2 = signatory.Signature(depth=sig_depth,
                                              stream=False)

        # 4 because that's the final layer size in self.augment2
        sig_channels2 = signatory.signature_channels(channels=4,
                                                     depth=sig_depth)
        self.linear = torch.nn.Linear(sig_channels2, out_dimension)

    def forward(self, inp):
        # inp is a three dimensional tensor of shape (batch, stream, in_channels)
        a = self.augment1(inp)
        if a.size(1) <= 1:
            raise RuntimeError("Given an input with too short a stream to take the
↪"
                               " signature")
        # a in a three dimensional tensor of shape (batch, stream, in_channels +␣
↪5)
        b = self.signature1(a, basepoint=True)
        # b is a three dimensional tensor of shape (batch, stream, sig_channels1)
        c = self.augment2(b)
        if c.size(1) <= 1:
            raise RuntimeError("Given an input with too short a stream to take the
↪"
                               " signature")
        # c is a three dimensional tensor of shape (batch, stream, 4)
        d = self.signature2(c, basepoint=True)
        # d is a two dimensional tensor of shape (batch, sig_channels2)
        e = self.linear(d)
        # e is a two dimensional tensor of shape (batch, out_dimension)
        return e
```

CHAPTER

FIVE

CITATION

If you found this library useful in your research,please consider citing

```
@misc{signatory,
    title={{Signatory: differentiable computations of the signature and
↪logsignature transforms, on both CPU and GPU}},
    author={Kidger, Patrick},
    note={\texttt{https://github.com/patrick-kidger/signatory}},
    year={2019}
}
```

CHAPTER

SIX

FAQ AND KNOWN ISSUES

If you have a question and don't find an answer here then do please open an issue.

6.1 Problems with importing or installing Signatory

- I get an `ImportError: DLL load failed: The specified procedure could not be found.` when I try to import Signatory.

This appears to be caused by using old versions of Python,e.g. `3.6.6` instead of `3.6.9`. Upgrading your version of Python seems to resolve the issue.

- I get an `Import Error: ... Symbol not found: ...` when I try to import Signatory.

This occurs when the version of PyTorch you have installed is different to the version of PyTorch that your copy of Signatory is compiled for. Make sure that you have specified the correct version of PyTorch when downloading Signatory; see *the installation instructions*.

6.2 All other issues

- What's the difference between Signatory and iisignature?

The essential difference (and the reason for Signatory's existence) is that iisignature is limited to the CPU,whilst Signatory is for both CPU and GPU. Signatory is also typically faster even on the CPU,thanks to parallelisation and algorithmic improvements. Other than that,iisignature is NumPy-based,whilst Signatory uses PyTorch. There are also a few differences in the provided functionality; each package provides some operations that the other doesn't.

- Exceptions messages aren't very helpful on a Mac.

This isn't an issue directly to do with Signatory. We use pybind11 to translate C++ exceptions to Python exceptions,and some part of this process breaks down when on a Mac. If you're trying to debug your code then the best (somewhat unhelpful) advice is to try running the problematic code on either Windows or Linux to check what the error message is.

CHAPTER

SEVEN

ADVICE ON USING SIGNATURES

7.1 What are signatures?

If you're reading this then it's probably because you already know what the signature transform is,and are looking to use it in your project. But in case you've stumbled across this and are curious what this 'signature' thing is...

The *signature transform* is a transformation that takes in a stream of data (often a time series),and returns a collection of statistics about that stream of data,called the *signature*. This collection of statistics determines the path essentially uniquely. Importantly,the signature is rich enough that every continuous function of the input stream may be approximated arbitrarily well by a linear function of its signature; the signature transform is what we call a *universal nonlinearity*. If you're doing machine learning then you probably understand why this is such a desirable property!

The definition of the signature transform can be a little bit intimidating -

Definition

Let $\mathbf{x} = (x_1, \ldots, x_n)$,where $x_i \in \mathbb{R}^d$. Linearly interpolate $\mathbf{x}$ into a path $f = (f^1, \ldots, f^d)\colon [0,1] \to \mathbb{R}^d$. The signature of $\mathbf{x}$ is defined as $\mathrm{Sig}(\mathbf{x}) = \mathrm{Sig}(f)$,where

$$\mathrm{Sig}(f) = \left(\left(\underset{0<t_1<\cdots<t_k<1}{\int \cdots \int} \prod_{j=1}^{k} \frac{\mathrm{d}f^{i_j}}{\mathrm{d}t}(t_j)\mathrm{d}t_1 \cdots \mathrm{d}t_k \right)_{1\leq i_1,\ldots,i_k\leq d} \right)_{k\geq 0}.$$

But if you're just using the signature transform then you don't need to worry about really understanding what all of that means – just how to use it. Computing it is somewhat nontrivial. Now if only someone had already written a *package to compute it for you...*

In principle the signature transform is quite similar to the Fourier transform: it is a transformation that can be applied to a stream of data which extracts certain information. The Fourier transform describes frequencies; meanwhile the signature most naturally describes *order* and *area*. The order of events,potentially in different channels,is a particularly easy thing to understand using the signature. Similarly various notions of area are also easy to understand.

Note: It turns out that order and area are actually in some sense the same concept. For a (very simplistic) example of this: consider the functions $f(x) = x(1-x)$ and $g(x) = x(x-1)$ for $x \in [0,1]$. Then the area of f is $\int_0^1 f(x)\mathrm{d}x = \frac{1}{6}$ whilst the area of g is $\int_0^1 g(x)\mathrm{d}x = \frac{-1}{6}$. Meanwhile,the graph of f goes *up* then *down*,whilst the graph of g goes *down* then *up*: the order of the ups and downs corresponds to the area.

Check out this for a primer on the use of the signature transform in machine learning,just as a feature transformation,and this for a more in-depth look at integrating the signature transform into neural networks.

7.2 Neural networks

The universal nonlinearity property (mentioned *here*) requires the whole,infinite,signature. This doesn't fit in your computer's memory. The solution is simple: truncate the signature to some finite collection of statistics,and then embed it within a nonlinear model,like a neural network. The signature transform now instead acts as a pooling function,doing a provably good job of extracting information.

Have a look at this for a more in-depth look at integrating it into neural neural networks.

As a general recommendation:

- The number of terms in signatures can grow rapidly with depth and number of channels,so experiment with what is an acceptable amount of work.
- Place small stream-preserving neural networks before the signature transform; these typically greatly enhance the power of the signature transform. This can be done easily with the `signatory.Augment` class.
- It's often worth augmenting the input stream with an extra 'time' dimension. This can be done easily with the `signatory.Augment` class. (Have a look at Appendix A of this for an understanding of what augmenting with time gives you,and when you may or may not want to do it.)

7.3 Kernels and Gaussian Processes

The signature may be used to define a universal kernel for sequentially ordered data.

See here for using signatures with kernels,and here for using signatures with Gaussian Processes.

7.4 Signatures vs. Logsignatures

Signatures can get quite large. This is in fact the whole point of them! They provide a way to linearise all possible functions of their input. In contrast logsignatures tend to be reasonably modestly sized.

If you know that you want to try and capture particularly high order interactions between your input channels then you may prefer to use logsignatures over signatures,as this will capture this the same information,but in a more information-dense way. This comes with a price though,as the logsignature is somewhat slower to compute than the signature.

Note that as the logsignature is computed by going via the signature,it is not more memory-efficient to compute the logsignature than the signature.

CHAPTER EIGHT

SOURCE CODE

The Signatory project is hosted on GitHub.

CHAPTER

NINE

ACKNOWLEDGEMENTS

The Python bindings for the C++ code were written with the aid of pybind11.

For NumPy-based CPU-only signature calculations,you may also be interested in the iisignature package. The notes accompanying the iisignature project greatly helped with the implementation of Signatory.

PYTHON MODULE INDEX

INDEX