# Fireworks Documentation

*Release 0.2.8*

**Saad Khan**

**Jan 24, 2019**

# CONTENTS

Tensor algebra frameworks such as TensorFlow and PyTorch have made developing neural network based machine learning models surprisingly easy in the past few years. The flowgraph architecture that these frameworks are built around makes it simple to apply algebraic transforms such as gradients to tensor valued datasets with GPU optimization, such as in neural networks.

While these frameworks offer building blocks for constructing models, we often want tools to combine those blocks in reusable manners. Libraries such as Keras and Gluon are built on top of these computation frameworks to offer abstractions specific to certain types of neural networks that can be stacked together in layers. The Ignite library for pytorch takes a more hands-on approach. It provides an 'engine' class that has event methods corresponding to the stages of a machine learning training process (before training, before epoch, before step, during step, after step, etc.). The developer writes functions for what should happen during each of these events (what happens during a training step, at the end of an epoch, etc.), and the engine then makes sure those functions are called at the correct times. On the other extreme, there are completely hands-off machine learning frameworks such as arya.ai and Google autoML that allow one to drag and drop elements and data to construct and train a neural network model.

Which of these approaches makes the most sense? For a researcher, control and flexibility are of paramount importance. Models should be easy to construct but allow one to step in wherever additional control is needed. This is particularly important because in research, one often wants to design models that cannot be expressed using simpler frameworks. For this reason, I prefer the combination of PyTorch + Ignite for deep learning.

However, these tools do not satisfy all of the needs of a deep learning pipeline. A pain point in all deep learning frameworks is data entry; one has to take data in its original form and morph it into the form that the model expects (tensors, TFRecords, etc.). This process can include acquiring the data from a database or an API call, formatting it, applying preprocessing transforms such as mapping text to vectors based on a vocabulary, and preparing mini batches for training. In each of these steps, one must be cognizant of memory, storage, bandwidth, and compute limitations. For example, if a dataset is too big to fit in memory, then it's journey must be streamed or broken into chunks. In practice, dealing with these issues takes more time than developing the actual models. Hence, we are in a strange position where it's easy to construct a bidirectional LSTM RNN with attention, but it's hard to load in a corpus of text from a database to train a classifier with that RNN.

This is where Fireworks comes in. Fireworks is a python-first framework for performing the data processing steps of machine learning in a modular and reusable manner. It does this by moving data between objects called 'Sources'. A Source can represent a file, a database, or a transform. Each Source has a set of input Sources and is itself an input to other Sources, and as data flows from Source to Source, transforms are applied one at a time, creating a graph of data flow. Because each Source is independent, they can be stacked and reused in the future. Moreover, because Sources are aware of their inputs, they can also call methods on their inputs, and this enables lazy evaluation of data transformations. Lastly, the means of communication between Sources is represented by a Message object. A Message

is essentially a (python) dict of arrays, lists, vectors, tensors, etc. It generalizes the functionality of a pandas dataframe to include the ability to store pytorch tensors. This makes it easy to adapt traditional ML pipelines that use pandas and sklearn, because Messages behave like dataframes. As a result, Fireworks is useful for any data processing task, not just deep learning. It can be used for interacting with a database, constructing financial models with pandas, and so much more.

CHAPTER

ONE

# OVERVIEW

Fireworks consists of a number of modules that are designed to work together to facilitate an aspect of deep learning and data processing.

**Message**

> "A dictionary of vectors and tensors". This class standardizes the means of communication between sources. Standardizing the means of communication makes it easier to write models that are reusable, because the inputs and outputs are always the same format.

**Source**

A Class that abstracts data access and transformation. Sources can be linked together to form a graph, allowing one to modularly construct a data pipeline.

**MessageCache**

This class addresses the particular challenge of dealing with datasets that won't fit in memory. A MessageCache behaves like a python cache that supports insertions, deletions, and accessions, except the underlying data structure is a message. This enables one to hold a portion of a larger dataset in memory while virtually representing the entire dataset as a message.

**Hyperparameter Optimization**

This module takes an approach similar to Ignite, except for hyperparameter optimization. It provides a class called Factory that has methods corresponding to the events in a hyperparameter training process (train, evaluate, decide on new parameters, etc.) that can be provided by the developer. In addition, training runs are treated as independent processes, enabling one to spawn multiple training runs simultaneously to evaluate multiple hyperparameters at once.

**Relational Database Integration**

Fireworks.database has Sources that can read from and write to a database in the middle of a pipeline. Because it is based on python SQLalchemy library, it can be used to incorporate almost any relational database into a data analysis workflow. Collecting and Storing Experimental Runs / Metrics In order to make machine learning research reproducible, we have to be able to store metadata and outputs associated with experiments. This module implements an Experiment class that creates a folder and can generate file handles and SQLite tables residing in that folder to save information to. It can also store user defined metadata, all in a given experiment's folder. This folder can be reloaded at any time in order to access the results of that experiment, regenerate plots, perform additional analyses, and so on.

**Not Yet Implemented / Roadmap Objectives**

(An experiment is a single run of get data - preprocess - train - evaluate - hyperparams - test)

**Plotting**

Generating plots is the primary means for analyzing and communicating the results of an experiment. We want to generate plots in such a way that we can go back later on and change the formatting (color scheme, etc.) or generate new plots from the data. In order to do this, plots must be generated dynamically rather than as static images. In addition, we want to create a robust means for displaying plots using a dashboard framework such as Plotly Dash.

Tools such as Visdom are great for displaying live metrics from an experiment, but they are not designed to present hundreds of plots at once or to display those plots in a pleasing manner (such as with dropdown menus). My goal is to create a dashboard that can display all information associated with a chosen experiment. It should include an SQLite browser, a plotly dashboard, a Visdom instance, and a Tensorboard instance. This should allow one to have all of the common visualization tools for machine learning in a single place.

**Dry Runs**

Certain steps in a data processing pipeline can be time consuming one-time operations that make it annoying to repeatedly start an experiment over. We want to set up means to 'dry run' an experiment in order to identify and fix bugs before running it on the full dataset. Additionally, we want to be able to set up checkpoints that enable one to continue an experiment after pausing it or loading it from a database.

**Distributed and Parallel Data Processing**

The idea of representing the data processing pipeline as a graph can naturally scale to parallel and distributed environments. In order to do make this happen, we need to write a scheduler that can call methods on Sources in parallel and add support for asynchronous method calls on Sources. For distributed processing, we have to write a tool for containerizing Sources and having them communicate over a container orchestration framework. Argo is a framework for Kubernetes we can use that is designed for this task.

**True Graph-based Pipelines**

Fireworks at present only supports DAG shaped pipelines rooted at the end. This means that while multiple sources can feed into one source, feeding one source into multiple output sources does not do anything useful. Loops and branches would break everything, because there is no code for handling those scenarios right now. Additionally, Sources are only aware of their inputs, not their outputs. While this simplifies the framework, it only enables communication in one direction.

**Dynamic Optimization of Data Pipeline**

Many sources have to occasionally perform time consuming O(1) tasks (such as precomputing indices corresponding to minibatches). Ideally, these tasks should be performed asynchronously, and the timing of when to perform them should be communicated by downstream sources. Adding the ability to communicate such timings would allow the pipeline to dynamically optimize itself in creative ways. For example, a CachingSource could prefetch elements into its cache that are expected to be called in the future to speed up its operation.

**Static Performance Optimization**

Right now, the focus is on establishing the interface and abstractions associated with Fireworks. There are many places where operations can be optimized using better algorithms, cython implementations of important code sections, and eliminating redundant code.

CHAPTER

TWO

# CONTENTS

## 2.1 Project

### 2.1.1 History

Saad Khan began working on Fireworks in June 2018. It was open sourced in October 2018.

### 2.1.2 Committers

- @smk508 (Saad Khan)

### 2.1.3 Resources

- PyTorch
- Ignite

### 2.1.4 Roadmap

- Add examples and flesh out documentation
- Full test coverage
- Performance improvements
- Parallel/asynchronous execution support

## 2.2 License

## 2.3 Installation

### 2.3.1 Install from PyPi

You will be able to install Fireworks using pip (Note: this doesn't work yet.):

```
pip install pytorch-fireworks
```

It is recommended to do this from within a virtual environment.

### 2.3.2 Install from source

For development purposes, you can download the source repository and install the latest version directly.

```
git clone https://github.com/smk508/Fireworks.git
cd Fireworks
pip install .
```

## 2.4 Tutorial

### 2.4.1 Basic Operations

### 2.4.2 Chaining Sources

### 2.4.3 Using Databases

### 2.4.4 Using Experiments

### 2.4.5 Hyperparameter Optimization

## 2.5 API Reference

### 2.5.1 Messages

Most data processing workflows have the same basic architecture and only differ in the type of data and how those inputs are formatted. Minor differences in this formatting can make almost identical code non-reusable. To address this issue, this framework insists on using a single data structure to pass information between components - the Message object. A Message consists of two components: a Pandas DataFrame and a TensorMessage. The former is a very general purpose structure that benefits from all of the features of the popular Pandas library - a DataFrame is essentially a dictionary of arrays. However, in the context of pytorch deep learning, we cannot use DataFrames for everything

because we cannot store tensor objects inside a dataframe (Pandas breaks down tensors into unit sized tensors and stores those units as objects as opposed to storing them as one entity). The TensorMessage emulates the structure and methods of DataFrames, except it only stores pytorch tensors (in the future, tensor's in other frameworks could be supported). Because of this, it also attempts to autoconvert inputs to tensors. With this combined structure, one could store metadata in the dataframe and example/label pairs in the TensorMessage.

**class** Fireworks.core.message.**Message**(**args*, *metadata=None*, *length=None*, ***kwargs*)

Bases: object

A Message is a class for representing data in a way that can be consumed by different analysis pipelines in python. It does this by representing the data as a dictionary of arrays. This is the same approach that pandas takes, but Messages are specifically designed to be used with pytorch, in which the core data structure is a tensor, which cannot be mixed into a pandas dataframe. Hence, a Message has two elements: a TensorMessage which specifically stores tensor objects and a dataframe which can be used for handling any other type of array data. With this structure, you could put your training data in as a TensorMessage and any associated metadata as the df and lump all of that into a message. All of the existing df methods can be run on the metadata, and any pytorch operations can be performed on the tensor part.

Messages support operations such as appending one message to another, joining two messages along keys, applying maps to the message's values, and equality checks.

Additionally, messages behave as much as possible like dicts. In many scenarios, a dict will be able to substitute for a message and vice-versa. For example, for preparing batches to feed into an RNN to classify DNA sequences, one could create a Message like this:

```
my_message = Message({
'embedded_sequences': torch.Tensor([...]),
'labels': torch.LongTensor([...]),
'raw_sequences': ['TCGA...',...,],
...})
```

The Message constructor will parse this dictionary and store the labels and embedded sequences inside a TensorMessage and the raw_sequences and other metadata in the dataframe.

Now we can access elements of this Message:

```
my_message[0]
my_message[2:10]
my_message['labels']
len(my_message)
```

We can also move tensors to the GPU and back:

```
my_message.cpu()
my_message.cuda(device_num=1) # Defaults to 0 for device number
my_message.cuda(keys=['labels']) # Can specify only certain columns to move if␣
↪desired
```

**check_length**()

Checks that lengths of the internal tensor_message and dataframe are the same and equalto self.len If one of the two is empty (length 0) then, that is fine.

**columns**

Returns names of tensors in TensorMessage

**keys**()

Returns names of tensors in TensorMessage Note: This is the same as self.columns

**index**

Returns index for internal tensors

**append**(*other*)
Compines messages together. Should initialize other if not a message already.

**Parameters** **other** – The message to append to self. Must have the same keys as self so that in the resulting Message, every column continues to have the same length as needed.

**merge**(*other*)
Combines messages horizontally by producing a message with the keys/values of both.

**Parameters** **other** – The message to merge with self. Must have different keys and the same length as self to ensure length consistencies. Alternatively, if either self or other have an empty TensorMessage or df, then they can be merged together safely as long as the resulting Message has a consistent length. For example:

```
message_a = Message({'a': [1,2,3]}) # This is essentially a␣
↪DataFrame
message_b = Message({'b': torch.Tensor([1,2,3])}) # This is␣
↪essentially a TensorMessage
message_c = Message_a.merge(message_b) # This works
```

**Returns** The concatenated Message containing columns from self and other.

**Return type** message

**map**(*mapping*)
Applies function mapping to message. If mapping is a dict, then maps will be applied to the correspondign keys as columns, leaving columns not present in mapping untouched. In otherwords, mapping would be a dict of column_name:functions specifying the mappings.

**Parameters** **mapping** – Can either be a dict mapping column names to functions that should be applied to those columns, or a single function. In the latter case, the mapping function will be applied to every column.

**Returns** A Message with the column:value pairs produced by the mapping.

**Return type** message

**tensors**(*keys=None*)
Return tensors associated with message as a tensormessage. If keys are specified, returns tensors associated with those keys, performing conversions as needed.

**Parameters** **keys** – Keys to get. Default = None, in which case all tensors are returned as a TensorMessage. If columns corresponding to requested keys are not tensors, they will be converted.

**Returns** A TensorMessage containing the tensors requested.

**Return type** tensors (*TensorMessage*)

**dataframe**(*keys=None*)
Returns message as a dataframe. If keys are specified, only returns those keys as a dataframe.

**Parameters** **keys** – Keys to get. Default = None, in which case all non-tensors are returned as a DataFrame. If columns corresponding to requested keys are tensors, they will be converted (to np.arrays).

**Returns** A DataFrame containing the columns requested.

**Return type** df (pd.DataFrame)

**to_dataframe**(*keys=None*)
Returns message with columns indicated by keys converted to DataFrame. If keys is None, all tensors are converted.

**Parameters** **keys** – Keys to get. Default = None, in which case all tensors are mapped to DataFrame.

**Returns** A Message in which the desired columns are DataFrames.

**Return type** message

**to_tensors**(*keys=None*)
Returns message with columns indicated by keys converted to Tensors. If keys is None, all columns are converted.

**Parameters** **keys** – Keys to get. Default = None, in which case all columns are mapped to Tensor.

**Returns** A Message in which the desired columns are Tensors.

**Return type** message

**permute**(*index*)
Reorders elements of message based on index.

**Parameters** **index** – A valid index for the message.

**Returns**

**A new Message with the elements arranged according to the input index.** For example, :: message_a = Message({'a':[1,2,3]}) message_b = message_a.permute([2,1,0]) message_c = Message({'a': [3,2,1]}) message_b == message_c

The last statement will evaluate to True

**Return type** message

**cpu**(*keys=None*)
Moves tensors to system memory. Can specify which ones to move by specifying keys.

**Parameters** **keys** – Keys to move to system memory. Default = None, meaning all columns are moved.

**Returns** Moved message

**Return type** message (*Message*)

**cuda**(*device=0*, *keys=None*)
Moves tensors to gpu with given device number. Can specify which ones to move by specifying keys.

**Parameters**

- **device** (*int*) – CUDA device number to use. Default = 0.
- **keys** – Keys to move to GPU. Default = None, meaning all columns are moved.

**Returns** Moved message

**Return type** message (*Message*)

**class** Fireworks.core.message.**TensorMessage**(*message_dict=None*, *map_dict=None*)
Bases: object

A TensorMessage is a class for representing data meant for consumption by pytorch as a dictionary of tensors.

**keys**(**args*, ***kwargs*)

**columns**
Returns names of tensors in TensorMessage

**index**
Returns index for internal tensors

**append**(*other*)
Note if the target message has additional keys, those will be dropped. The target message must also have every key present in this message in order to avoid an value error due to length differences.

**merge**(*other*)
Combines self with other into a message with the keys of both. self and other must have distinct keys.

**permute**(*index*)
Rearranges elements of TensorMessage to align with new index.

**cuda**(*device=0*, *keys=None*)
Moves all tensors in TensorMessage to cuda device specified by device number. Specify keys to limit the transformation to specific keys only.

**cpu**(*keys=None*)
Moves all tensors in TensorMessage to cpu. Specify keys to limit the transformation to specific keys only.

Fireworks.core.message.**compute_length**(*of_this*)
Of_this is a dict of listlikes. This function computes the length of that object, which is the length of all of the listlikes, which are assumed to be equal. This also implicitly checks for the lengths to be equal, which is necessary for Message/TensorMessage.

Fireworks.core.message.**extract_tensors**(*from_this*)
Given a dict from_this, returns two dicts, one containing all of the key/value pairs corresponding to tensors in from_this, and the other containing the remaining pairs.

Fireworks.core.message.**slice_to_list**(*s*)
Converts a slice object to a list of indices

Fireworks.core.message.**complement**(*indices*, *n*)
Given an index, returns all indices between 0 and n that are not in the index.

Fireworks.core.message.**cat**(*list_of_args*)
Concatenates messages in list_of_args into one message.

Fireworks.core.message.**merge**(*list_of_args*)
Merges messages in list_of_args into one message with all the keys combined.

### 2.5.2 Pipes

With a uniform data structure for information transfer established, we can create functions and classes that are reusable because of the standardized I/O expectations. A Pipe object represents some transformation that is applied to data as it flows through a pipeline. For example, a pipeline could begin with a source that reads from the database, followed by one that cache those reads in memory, then one that applies embedding transformations to create tensors, and so on.

These transformations are represented as classes rather than functions because we sometimes want to be able to apply transformations in a just-in-time or ahead-of-time manner, or have the transformations be dependent on some upstream or downstream aspect of the pipeline. For example, the Pipe that creates minibatches for training can convert its inputs to tensors and move them to GPU as a minibatch is created, using the tensor-conversion method implemented by an upstream Pipe. Or a Pipe that caches its inputs can prefetch objects to improve overall performance, and so on.

**class** Fireworks.core.pipe.**Pipe**(*input=None*, **args*, ***kwargs*)
Bases: abc.ABC

The core data structure in fireworks. A Pipe can take Pipes as inputs, and its outputs can be streamed to other Pipes. All communication is done via Message objects. Method calls are deferred to input Pipes recursively until a Pipe that implements the method is reached.

This is made possible with a recursive function call method. Any Pipe can use this method to call a method on its inputs; this will recursively loop until reaching a Pipe that implements the method and return those outputs (as a Message) or raise an error if there are none. For example, we can do something like this:

```
reader = pipe_for_reading_from_some_dataset(...)
cache = CachingPipe(reader, type='LRU')
embedder = CreateEmbeddingsPipe(cache})
loader = CreateMinibatchesPipe(embedder})

loader.reset()
for batch in loader:
        # Code for training
```

Under the hood, the code for loader.__next__() can choose to recursively call a to_tensor() method which is implemented by embedder. Index queries and other magic methods can also be implemented recursively, and this enables a degree of commutativity when stacking Pipes together (changing the order of Pipes is often allowed because of the pass-through nature of recursive calls).

Note that in order for this to work well, there must be some consistency among method names. If a Pipe expects 'to_tensor' to convert batches to tensor format, then an upstream Pipe must have a method with that name, and this should remain consistent across projects to maintain reusability. Lastly, the format for specifying inputs to a Pipe is a dictionary of Pipes. The keys in this dictionary can provide information for the Pipe to use or be ignored completely.

**name = 'base_pipe'**

**recursive_call**(*attribute*, **args*, *ignore_first=True*, ***kwargs*)

Recursively calls method/attribute on input until reaching an upstream Pipe that implements the method and returns the response as a message (empty if response is None). Recursive calls enable a stack of Pipes to behave as one entity; any method implemented by any component can be accessed recursively.

**Parameters**

- **attribute** – The name of the attribute/method to call.
- **args** – The arguments if this is a recursive method call.
- **ignore_first** – If True, then ignore whether or not the target attribute is implemented by self. This can be useful if a Pipe implements a method and wants to use an upstream call of the same method as well.
- **kwargs** – The kwargs is this is a recursive method call.

**Returns** A dictionary mapping the name of each input Pipe to the response that was returned.

**Return type** Responses (dict)

**class** Fireworks.core.pipe.**HookedPassThroughPipe**(*input=None*, **args*, ***kwargs*)

Bases: *Fireworks.core.pipe.Pipe*

This Pipe has hooks which can be implemented by subclasses to modify the behavior of passed through calls.

**name = 'Hooked-passthrough Pipe'**

**class** Fireworks.core.cache.**MessageCache**(*max_size*)

Bases: abc.ABC

A message cache stores parts of a larger method and supports retrievals and insertions based on index. The use case for a MessageCache is for storing parts of a large dataset in memory. The MessageCache can keep track of which elements and which indices are present in memory at a given time and allow for updates and retrievals.

**insert**(*index*, *message*)
Inserts message into cache along with the desired indices. This method should be called by __setitem__ as needed to perform the insertion.

**Parameters**

- **index** – The index to insert into. Can be an int, slice, or list of integer indices.
- **message** – The Message to insert. Should have the same length as the provided idnex.

**delete**(*index*)
Deletes elements in the message corresponding to index. This method should be called by __setitem__ or __delitem__ as needed.

**Parameters** **index** – The index to insert into. Can be an int, slice, or list of integer indices.

**sort**()
Rearranges internal cache indices to be in sorted order.

**search**(***kwargs*)

**size**

**class** Fireworks.core.cache.**UnlimitedCache**
Bases: *Fireworks.core.cache.MessageCache*

This is a basic implementation of a MessageCache that simply appends new elements and never clears memory internally

**class** Fireworks.core.cache.**BufferedCache**(*max_size*)
Bases: *Fireworks.core.cache.MessageCache*

This implements a setitem method that assumes that when the cache is full, elements must be deleted until it is max_size - buffer_size in length. The deletion method, _free, must be implemented by a subclass.

**init_buffer**(*buffer_size=0*)

**free**(*n*)

**class** Fireworks.core.cache.**RRCache**(*max_size*)
Bases: *Fireworks.core.cache.BufferedCache*

**free**(*n*)

**class** Fireworks.core.cache.**RankingCache**(*max_size*)
Bases: *Fireworks.core.cache.MessageCache*

Implements a free method that deletes elements based on a ranking function.

**init_rank_dict**()

**free**(*n*, *x=0*)

**on_update_existing**(*index*, *message*)

**on_add_new**(*index*, *message*)

**on_delete**(*index*)

**on_getitem**(*index*)

**class** Fireworks.core.cache.**LRUCache**(**args*, *buffer_size=0*, ***kwargs*)
Bases: *Fireworks.core.cache.RankingCache*, *Fireworks.core.cache.BufferedCache*

Implements a Least Recently Used cache. Items are deleted in descending order of how recently they were accessed. A call to __getitem__ or __setitem__ counts as accessing an element.

**update_rank**(*index*)

**on_update_existing**(*index*, *message*)

**on_add_new**(*index*, *message*)

**on_delete**(*index*)

**on_getitem**(*index*)

**class** Fireworks.core.cache.**LFUCache**(**args*, *buffer_size=0*, ***kwargs*)
Bases: *Fireworks.core.cache.RankingCache*, *Fireworks.core.cache.BufferedCache*

Implements a Least Frequently Used cache. Items are deleted in increasing order of how frequently they are accessed. A call to __getitem__ or __setitem__ counts as accessing an element.

**update_rank**(*index*)

**on_update_existing**(*index*, *message*)

**on_add_new**(*index*, *message*)

**on_delete**(*index*)

**on_getitem**(*index*)

Fireworks.core.cache.**pointer_adjustment_function**(*index*)
Given an index, returns a function that that takes an integer as input and returns how many elements of the index the number is greater than. This is used for readjusting pointers after a deletion. For example, if you delete index 2, then every index greater than 2 must slide down 1 but index 0 and 1 do not more.

Fireworks.core.cache.**index_to_list**(*index*)
Converts an index to a list.

Fireworks.core.cache.**slice_to_list**(*s*)
Converts a slice object to a list of indices

Fireworks.core.cache.**get_indices**(*values*, *listlike*)
Returns the indices in litlike that match elements in values

**class** Fireworks.toolbox.pipes.**BioSeqPipe**(*path*, *input=None*, *filetype='fasta'*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Class for representing biosequence data. Specifically, this class can read biological data files (such as fasta) and iterate throug them as a Pipe. This can serve as the first Pipe in a pipeline for analyzing genomic data.

**name = 'BioSeqPipe'**

**reset**()
Resets the iterator to the beginning of the file. Specifically, it calls SeqIO.parse again as if reinstantiating the object.

**class** Fireworks.toolbox.pipes.**LoopingPipe**(*input*, **args*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

This Pipe can take any iterator and make it appear to be indexable by iterating through the input as needed to reach any given index.

The input Pipe must implement __next__ and reset (to be repeatable), and this will simulate __getitem__ by repeatedly looping through the iterator as needed.

For example, say we have a Pipe that iterates through the lines of a FASTA file:

```
fasta = BioSeqPipe('genes.fasta')
```

This Pipe can only iterate through the file in one direciton. If we want to access arbitrary elements, we can do this:

```
clock = LoopingPipe(inputs=fasta)
clock[10]
clock[2:6]
len(clock)
```

All of these actions are now possible. Note that this is in general an expensive process, because the Pipe has to iterate one at a time to get to the index it needs. In practice, this Pipe should pipe its output to a CachingPipe that can store values in memory. This approach enables you to process datasets that don't entirely fit in memory; you can stream in what you need and cache portions. From the perspective of the downstream Pipes, every element of the dataset is accessible as if it were in memory.

**name = 'LoopingPipe'**

**check_input**()
Checks input to determine if it implements __next__ and reset methods.

**reset**()
Calls reset on input Pipes and sets position to 0.

**compute_length**()
Step forward as far as the inputs will allow and compute length.

Length is also calculated implicitly as items are accessed.

Note: If the inputs are infinite, then this will go on forever.

**step_forward**(*n*)
Steps forward through inputs until position = n and then returns that value.

This also updates the internal length variable if the iterator ends due to this method call.

**class** Fireworks.toolbox.pipes.**CachingPipe**(*input*, **args*, *cache_size=100*, *buffer_size=0*, *cache_type='LRU'*, *infinite=False*, ***kwargs*)

Bases: *Fireworks.core.pipe.Pipe*

This Pipe can be used to dynamically cache elements from upstream Pipes. Whenever data is requested by index, this Pipe will intercept the request and add that message alongside the index to its internal cache. This can be useful for dealing with datasets that don't fit in memory or are streamed in. You can cache portions of the dataset as you use them. By combining this with a LoopingPipe, you can create the illusion of making the entire dataset available to downstream Pipes regardless of the type and size of the original data.

More specifically, fiven input Pipes that implement __getitem__, will store all calls to __getitem__ into an internal cache and therafter __getitem__ calls will either access from the cache or trigger __getitem__ calls on the input and an update to the cache.

For example,

```
fasta = BioSeqPipe(path='genes.fasta')
clock = LoopingPipe(inputs=fasta)
cache = CachingPipe(inputs=clock, cache_size=100) # cache_size is optional;␣
↪default=100
```

Will set up a pipeline that reads lines from a FASTA file and acessess and caches elements as requests are made

```
cache[20:40] # This will be cached
cache[25:30] # This will read from the cache
cache[44] # This will update the cache
```

(continues on next page)

(continued from previous page)

```
cache[40:140] # This will fill the cache, flushing out old elements
cache[25:30] # This will read from the dataset and update the cache again
```

**init_cache**(**args*, ***kwargs*)
This should initialize a cache object called self.cache

**check_input**()
Checks inputs to determine if they implement __getitem__.

**compute_length**()
Step forward as far as the inputs will allow and compute length. Note: If the inputs are infinite, then this will go on forever.

**class** Fireworks.toolbox.pipes.**Title2LabelPipe**(*title*, *input*, **args*, *labels_column='labels'*, ***kwargs*)
Bases: *Fireworks.core.pipe.HookedPassThroughPipe*

This Pipe takes one Pipe as input and inserts a column called 'label' containing the provided title of the input Pipe to to all outputs.

**insert_labels**(*message*)

**class** Fireworks.toolbox.pipes.**LabelerPipe**(*input*, *labels*, **args*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

This Pipe implements a to_tensor function that converts labels contained in messages to tensors based on an internal labels dict.

**to_tensor**(*batch*, *labels_column='labels'*)

**class** Fireworks.toolbox.pipes.**RepeaterPipe**(*input*, **args*, *repetitions=10*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Given an input Pipe that is iterable, enables repeat iteration.

**reset**()

**class** Fireworks.toolbox.pipes.**ShufflerPipe**(*input*, **args*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Given input Pipes that implement __getitem__ and __len__, will shuffle the indices so that iterating through the Pipe or calling __getitem__ will return different values.

**check_input**()
Check inputs to see if they implement __getitem__ and __len__

**shuffle**(*order=None*)

**reset**()
Triggers a shuffle on reset.

**class** Fireworks.toolbox.pipes.**IndexMapperPipe**(*input_indices*, *output_indices*, **args*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Given input Pipes that implement __getitem__, returns a Pipe that maps indices in input_indices to output_indices via __getitem__

**check_input**()

**class** Fireworks.toolbox.pipes.**BatchingPipe**(**args*, *batch_size=5*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Generates minibatches.

**reset**()

### 2.5.3 Junctions

Whereas Pipes are designed to have one input, Junctions can have multiple inputs, called components. Since there is no unambiguous way to define how recursive method calls would work in this situation, it is the responsibility of each Junction to have built-in logic for how to aggregate its components in order to respond to method calls from downstream sources. This provides a way to construct more complex computation graphs.

**class** Fireworks.core.junction.**Junction**(**args*, *components=None*, ***kwargs*)
Bases: object

A junction can take pipes as inputs, and its outputs can be piped to other pipes. All communication is done via Message objects.

Unlike Pipes, junctions do not automatically have recursive method calling. This is because they have multiple input sources, which would result in ambiguity. Instead, junctions are meant to act as bridges between multiple pipes in order to enable complex workflows which require more than a linear pipeline.

**class** Fireworks.toolbox.junctions.**HubJunction**(**args*, ***kwargs*)
Bases: *Fireworks.core.junction.Junction*

This junction takes multiple sources implementing __next__ as input and implements a new __next__ method that samples its input sources.

**check_inputs**()

**reset**()

**sample_inputs**()
Returns the key associated with an input source that should be stepped through next.

**class** Fireworks.toolbox.junctions.**RandomHubJunction**(**args*, ***kwargs*)
Bases: *Fireworks.toolbox.junctions.HubJunction*

HubJunction that randomly chooses inputs to step through.

**sample_inputs**()
Returns the key associated with an input source that should be stepped through next.

**class** Fireworks.toolbox.junctions.**ClockworkHubJunction**(**args*, ***kwargs*)
Bases: *Fireworks.toolbox.junctions.HubJunction*

HubJunction that iterates through input sources one at a time.

**reset**()

**sample_inputs**()
Loops through inputs until finding one that is available.

**class** Fireworks.toolbox.junctions.**SwitchJunction**(**args*, ***kwargs*)
Bases: *Fireworks.core.junction.Junction*

This junction has an internal switch that determines which of it's components all method calls will be routed to.

**route**
Returns the component to route method calls to based on the internal switch.

### 2.5.4 Models

Models are a data structure for representing mathematical models that can be stacked together, incorporated into pipelines, and have their parameters trained using PyTorch. These Models don't have to be neural networks or even machine learning models; they can represent any function that you want. The goal of the Models class is to decouple the parameterization of a model from its computation. By doing this, those parameters can be swapped in and out as needed, while the computation logic is contained in the code itself. This structure makes it easy to save and load models. For example, if a Model computes y = m*x+b, the parameters m and b can be provided during initialization, they can be learned using gradient descent, or loaded in from a database. Models function like Junctions with respect to their parameters, which are called components. These components can be PyTorch Parameters, PyTorch Modules, or some other object that has whatever methods/attributes the Model requires. Models function like Pipes with respect to their arguments. Hence, you can insert a Model inside a Pipeline. Models also function like PyTorch Modules with respect to computation and training. Hence, once you have created a Model, you can train it using a method like gradient descent. PyTorch will keep track of gradients and Parameters inside your Models automatically. You can also freeze and unfreeze components of a Model using the freeze/unfreeze methods.

```
m = LinearModel(components={'m': [1.]}) # Initialize model for y = m*x+b with m = 1.
print(m.required_components) # This will return ['m', 'b']. A model can optionally␣
↪have initialization logic for components not provided
# For example, the y-intercept b can have a default initialization if not provided␣
↪here.
print(m.components) # This should return a dict containing both m and b. The model␣
↪should have initialized a y-intercept and automatically added that to it's␣
↪components dict.
f = NonlinearModel(input=m) # Initialize a model that represents some nonlinearity␣
↪and give it m as an input.
result = f(x) # Evaluates f(m(x)) on argument message x. Because m is an input of f,␣
↪m will be called first and pipe its output to f.
```

**class** Fireworks.core.model.**Model**(*components={}*, **args*, *input=None*, *skip_module_init=False*, ***kwargs*)

Bases: torch.nn.modules.module.Module, *Fireworks.core.pipe.HookedPassThroughPipe*, *Fireworks.core.junction.Junction*, abc.ABC

Represents a statistical model which has a set of components, and a means for converting inputs into outputs. The model functions like a Pipe with respect to the input/output stream, and it functions like a Junction with respect to the parameterization. components can be provided via multiple different sources in this way, providing flexibility in model configuration. Models can also provide components for other Models, enabling one to create complex graphs of Models that can be trained simultaneously or individually.

**init_default_components**()
This method can optionally be implemented in order for the model to provide a default initialization for some or all of its required components.

**update_components**(*components=None*)

**check_components**(*components=None*)
Checks to see if the provided components dict provides all necessary params for this model to run.

**required_components**
This should be overridden by a subclass in order to specify components that should be provided during initialization. Otherwise, this will default to just return the components already present within the Model.

**forward**(*message*)
Represents a forward pass application of the model to an input. Must be implemented by a subclass. This should return a Message.

**freeze**(*components=None*)
Freezes the given components of the model (or all of them if none are specified) in order to prevent gradient

updates. This means setting requires_grad to false for specified components so that these components are not updated during training.

**unfreeze**(*components=None*)
Unfreezes the given components of the model (or all of them if none are specified) in order to prevent gradient updates. This means setting requires_grad to true for specified components so that these components are updated during training.

Fireworks.core.model.**freeze_module**(*module*, *parameters=None*, *submodules=None*)
Recursively freezes the parameters in a PyTorch module.

Fireworks.core.model.**unfreeze_module**(*module*, *parameters=None*, *submodules=None*)
Recursively unfreezes the parameters in a PyTorch module.

Fireworks.core.model.**model_from_module**(*module_class*)
Given the class definition for a pytorch module, returns a model that encapsulates that module.

### 2.5.5 Database

This module contains methods and classes for ingesting and reading data to/from a database. A user can specify a schema and stream messages from a source into a relational database. You can also create a source that streams data from a database based on a query. Because this module is built using SQLalchemy, it inherits all of the capabilities of that library, such as the ability to interface with many different relational databases and very precise control over schema and access. There are two sources: A TableSource implements methods for writing a Message to a table, and a DBSource is an iterable that produces Messages as it loops through a database query.

**TableSource**

A TableSource is initialized with an SQLalchemy table, and SQLalchemy engine, and an optional list of columns that the TableSource will write to in the table. By specifying columns, you can choose to use only a subset of the columns in a table (for example, if there are auto-incrementing ID columns that don't need to explicitly written). In addition to methods for standard relational database actions such as rollback, commit, etc., the TableSource has an insert method that takes a Message object, converts it into a format that can be written to the database and then performs the insert. It also has a query method that takes the same arguments that the query function in SQLalchemy takes (or does a SELECT * query by default) and returns a DBSource object corresponding to that query.

**DBSource**

This Source is initialized with an SQLalchemy query and iterates through the results of that query. It converts the outputs to Messages as it does so, enabling one to easily incorporate database queries into a Source pipeline.

**class** Fireworks.extensions.database.**TablePipe**(*table*, *engine*, *columns=None*, *input=None*, ***kwargs*)
Bases: *Fireworks.core.pipe.Pipe*

Represents an SQLalchemy Table while having the functionality of a Pipe.

**init_db**()
Initializes metadata for internal table. This ensures that the table exists in the database.

**commit**()
Commits transactions to database.

**rollback**()
Rollbacks transactions in current session that haven't been committed yet.

**insert**(*batch*)
Inserts the contents of batch message into the database using self.table object NOTE: Only the dataframe components of a message will be inserted.

**Parameters** **batch** (Message) – A message to be inserted. The columns and types must be consistent with the database schema.

**query**(*entities=None*, **args*, ***kwargs*)
Queries the database and generates a DBPipe corresponding to the result.

**Parameters**

- **entities** – A list of column names
- **args** – Optional positional arguments for the SQLalchemy query function
- **kwargs** – Optional keyword arguments for the SQLalchemy query function

**Returns** A DBPipe object that can iterate through the results of the query.

**Return type** dbpipe (*DBPipe*)

**delete**(*column_name*, *values*)

**update**(*filter_column*, *batch*)

**upsert**(*batch*)
Performs an upsert into the database. This is equivalent to performing an update + insert (ie. if value is not present, insert it, otherwise update the existing value.)

**Parameters** **batch** (Message) – The message to upsert.

**make_row**(*row*)
Converts a Message or dict mapping columns to values into a table object that can be inserted into an SQLalchemy database.

**Parameters** **row** – row in Message or dict form to convert.

**Returns** row converted to table form.

**Return type** table

**make_row_dict**(*row*)
Converts a 1-row Message into a dict of atomic (non-listlike) elements. This can be used for the bulk_insert_mappings method of an SQLalchemy session, which skips table instantiation and takes dictionaries as arguments instead.

**Parameters** **row** – row in Message or dict form to convert.

**Returns** row converted to table form.

**Return type** table

Fireworks.extensions.database.**create_table**(*name*, *columns*, *primary_key=None*, *Base=None*)
Creates a table given a dict of column names to data types. This is an easy way to quickly create a schema for a data pipeline.

**Parameters**

- **columns** (*dict*) – Dict mapping column names to SQLalchemy types.
- **primary_key** – The column that should be the primary key of the table. If unspecified, a new auto-incrementing column called 'id' will be added as the primary key. SQLalchemy requires that all tables have a primary key, and this ensures that every row is always uniquely identifiable.
- **Base** – An optional argument that can be provided to specify the Base class that the new table class will inherit from. By default, this will be set to an instance of declarative_base from SQLalchemy.

**Returns** A table class specifying the schema for the database table.

**Return type** simpletable (sqlalchemy.ext.declarative.api.DeclarativeMeta)

**class** Fireworks.extensions.database.**DBPipe**(*table*, *engine*, *query=None*, *columns_and_types=None*)

Bases: *Fireworks.core.pipe.Pipe*

Pipe that can iterate through the output of a database query.

**reset**(*entities=None*, **args*, ***kwargs*)
Resets DBPipe by reperforming the query, so that it is now at the beginning of the query.

**filter**(*column_name*, *predicate*, **args*, ***kwargs*)
Applies an sqlalchemy filter to query.

**all**()
Returns the results of the query as a single Message object.

**delete**()

**update**(*batch*)
Updates the contents of this DBPipe by replacing them with batch

**Parameters** **batch** – A Message

Fireworks.extensions.database.**parse_columns**(*object*, *ignore_id=True*)
Returns the names of columns in a table or query object

**Parameters**

- **table** (*sqlalchemy.ext.declarative.api.DeclarativeMeta*) –
- **ignore_id** (*bool*) – If True, ignore the 'id' column, which is a default primary key added by the create_table function.

**Returns** A list of columns names in the sqlalchemy object.

**Return type** columns (list)

Fireworks.extensions.database.**parse_columns_and_types**(*object*, *ignore_id=True*)
Returns column names and types in a object or query object as a dict

**Parameters**

- **object** – An SQLalchemy table or Query object
- **ignore_id** (*bool*) – If True, ignore the 'id' column, which is a default primary key added by the create_table function.

**Returns** A dict mapping column names to their SQLalchemy type.

**Return type** columns_and_types (dict)

Fireworks.extensions.database.**convert**(*value*, *sqltype*)
Converts a given value to a value that SQLalchemy can read.

Fireworks.extensions.database.**to_message**(*row*, *columns_and_types=None*)
Converts a database query result produced by SQLalchemy into a Message

**Parameters**

- **row** – A row from the query.
- **columns_and_types** (*dict*) – If unspecified, this will be inferred. Otherwise, you can specify the columns to parse, for example, if you only want to extract some columns.

**Returns** Message representation of input.

**Return type** message

Fireworks.extensions.database.**cast**(*value*)
Converts values to basic types (ie. np.int64 to int)

**Parameters** **value** – The object to be cast.

**Returns** The cast object.

Fireworks.extensions.database.**reflect_table**(*table_name*, *engine*)
Gets the table with the given name from the sqlalchemy engine.

**Parameters**

- **table_name** (*str*) – Name of the table to extract.
- **engine** (*sqlalchemy.engine.base.Engine*) – Engine to extract from.

**Returns** The extracted table, which can be now be used to read from the database.

**Return type** table (sqlalchemy.ext.declarative.api.DeclarativeMeta)

### 2.5.6 Experiment

The Experiment module offers a way to save data from individual runs of a model. This makes it convenient to compare results from different experiments and to replicate those experiments.

```
exp = Experiment('name', 'db_path', 'description')
```

will create a folder named db_path/name containing a sqlite file called name.sqlite. You can now save any objects to that folder using

```
with exp.open('filename') as f:
    f.save(...)
```

This will create a file handle f to the desired filename in the folder. You can also use exp.get_engine('name') or exp.get_session('name') to get an SQLalchemy session/engine object with the given name that you can then use to save/load data. Combined with Fireworks.db, you can save any data in Message format relatively easily.

Fireworks.extensions.experiment.**load_experiment**(*experiment_path*)
Returns an experiment object corresponding to the database in the given path.

**Parameters** **experiment_path** (*str*) – Path to the experiment folder.

**Returns** An Experiment object loaded using the files in the given folder path.

**Return type** experiment (*Experiment*)

**class** Fireworks.extensions.experiment.**Experiment**(*experiment_name*, *db_path*, *description=None*, *load=False*)
Bases: object

**load_experiment**(*path=None*, *experiment_name=None*)
Loads in parameters associated with this experiment from a directory.

**Parameters**

- **path** (*str*) – Path to the experiment folder.
- **experiment_name** (*str*) – Name to set this experiment to.

**create_dir**()
Creates a folder in db_path directory corresponding to this Experiment.

**load_metadata**()
Loads metadata from experiment folder by reading the metadata table.

**init_metadata**()
Initializes metadata table. This is a necessary action whenever using an SQLalchemy table for the first time and is idempotent, so calling this method multiple times does not produce side-effects.

**get_engine**(*name*)
Creates an engine corresponding to a database with the given name. In particular, this creates a file called {name}.sqlite in this experiment's save directory, and makes an engine to connect to it.

**Parameters** **name** – Name of engine to create. This will also be the name of the file that is created.

**Returns** The new engine. You can also reach this engine now by calling self.engines[name]

**Return type** engine

**get_session**(*name*)
Creates an SQLalchemy session corresponding to the engine with the given name that can be used to interact with the database.

**Parameters** **name** – Name of engine corresponding to session. The engine will be created if one with that name does not already exist.

**Returns** A session created from the chosen engine.

**Return type** session

**open**(*filename*, **args*, *string_only=False*)
Returns a handle to a file with the given filename inside this experiment's directory. If string_only is true, then this instead returns a string with the path to create the file. If the a file with 'filename' is already present in the directory, this will raise an error.

**Parameters**

- **filename** (*str*) – Name of file.
- **args** – Additional positional args for the open function.
- **string_only** (*bool*) – If true, will return the path to the file rather than the file handle. This can be useful if you want to create the file using some other library.

**Returns**

**If string_only is True, the path to the file. Otherwise, the opened file handle. Note: You can use this method i**
with statement to auto-close the file.

**Return type** file

Fireworks.extensions.experiment.**filter_columns**(*message*, *columns=None*)
Returns only the given columns of message or everything if columns is None. If tensor columns are requested, they are converted to ndarray first.

**Parameters** **columns** – Columns to keep. Default = None, meaning return the Message as is.

**Returns** Message with the filetered columns.

**Return type** message

### 2.5.7 Factory

The Factory module contains a class with the same name that performs hyperparameter optimization by repeatedly spawning independent instances of a model, training and evaluating them, and recording their parameters. The design

of this module is based off of a 'blackboard architecture' in software engineering, in which multiple independent processes can read and write from a shared pool of information, the blackboard. In this case, the shared pool of information is the hyperparameters and their corresponding evaluation metrics. The factory class is able to use that information to choose new hyperparameters (based on a user supplied search algorithm) and repeat this process until a trigger to stop is raised.

**A factory class takes four arguments:**

- Trainer - A function that takes a dictionary of hyperparameters, trains a model and returns the trained model
- Metrics_dict - A dictionary of objects that compute metrics during model training or evaluation.
- Generator - A function that takes the computed metrics and parameters up to this point as arguments and generates a new set of metrics to

use for training. The generator represents the search strategy that you are using. - Eval_dataloader - A dataloader (an iterable that produces minibatches as Message objects) that represents the evaluation dataset.

After instantiated with these arguments and calling the run method, the factory will use its generator to generate hyperparameters, train models using those hyperparameters, and compute metrics by evaluating those models against the eval_dataloader. This will loop until something raises a StopHyperparameterOptimization flag.

Different subclasses of Factory have different means for storing metrics and parameters. The LocalMemoryFactory stores them in memory as the name implies. The SQLFactory stores them in a relational database table. Because of this, SQLFactory takes three additional initialization arguments:

- Params_table - An SQLalchemy table specifying the schema for storing parameters.
- Metrics_table - An SQLalchemy table specifying the schema for storing metrics.
- Engine - An SQLalchemy engine, representing the database connection.

Additionally, to reduce memory and network bandwidth usage, the SQLFactory table caches information in local memory while regularly syncing with the database.

Currently, all of these steps take place on a single thread, but in the future we will be able to automatically parallelize and distribute them.

**class** Fireworks.extensions.factory.**Factory**(*trainer*, *metrics_dict*, *generator*, *eval_dataloader*, **args*, ***kwargs*)

Bases: object

Base class for parallel hyperparameter optimization in pytorch using queues.

**get_connection**()

**run**()

**read**()

**write**(*params*, *metrics_dict*)

**after**(**args*, ***kwargs*)

**class** Fireworks.extensions.factory.**LocalMemoryFactory**(*trainer*, *metrics_dict*, *generator*, *eval_dataloader*, **args*, ***kwargs*)

Bases: *Fireworks.extensions.factory.Factory*

Factory that stores parameters in memory.

**get_connection**()

**read**()

**write**(*params*, *metrics_dict*)

**class** Fireworks.extensions.factory.**SQLFactory**(**args*, *params_table*, *metrics_tables*, *engine*, ***kwargs*)

Bases: *Fireworks.extensions.factory.Factory*

Factory that stores parameters in SQLalchemy database while caching them locally.

**get_connection**()

**write**(*params*, *metrics*)

**read**()

**read_db**()

**sync**()

Syncs local copy of metrics and params with db.

**after**()

### 2.5.8 Miscellaneous

Fireworks.toolbox.text.**character_tokenizer**(*sequence*)

Splits sequence into a list of characters.

Fireworks.toolbox.text.**space_tokenizer**(*sequence*)

Splits sequence based on spaces.

Fireworks.toolbox.text.**pad_sequence**(*sequence*, *max_length*, *embeddings_dict*, *pad_token='EOS'*)

Adds EOS tokens until sequence length is max_length.

Fireworks.toolbox.text.**pad**(*batch*, *embeddings_dict*, *pad_token='EOS'*)

Pads all embeddings in a batch to be the same length.

Fireworks.toolbox.text.**apply_embeddings**(*sequence*, *embeddings_dict*, *tokenizer*)

Decomposes sequence into tokens using tokenizer and then converts tokens to embeddings using embeddings_dict.

Fireworks.toolbox.text.**create_pretrained_embeddings**(*embeddings_file*)

Loads embeddings vectors from file into a dict.

Fireworks.toolbox.text.**load_embeddings**(*name='glove840b'*)

Loads serialized embeddings from pickle.

Fireworks.toolbox.text.**make_vocabulary**(*text*, *tokenizer=None*, *cutoff_rule=None*)

Converts an iterable of phrases into the set of unique tokens that are in the vocabulary.

Fireworks.toolbox.text.**make_indices**(*vocabulary*)

Constructs a dictionary of token names to indices from a vocabulary. Each index value corresponds to a one-hot vector.

Fireworks.toolbox.text.**too_big**(*dataset*, *start*, *end*, *dim=300*, *cutoff=620000*)

Calculates if a batch consisting of dataset[start:end] is too big based on cutoff.

Fireworks.utils.utils.**one_hot**

Fireworks.utils.utils.**index_to_list**(*index*)

Converts an index to a list.

Fireworks.utils.utils.**slice_to_list**(*s*)

Converts a slice object to a list of indices

Fireworks.utils.utils.**get_indices**(*values*, *listlike*)
Returns the indices in litlike that match elements in values

Fireworks.utils.utils.**slice_length**(*orange*)
Returns the length of the index corresponding to a slice. For example, slice(0,4,2) has a length of two.

Fireworks.toolbox.preprocessing.**train_test_split**(*pipe*, *test=0.2*)
Splits input pipe into a training pipe and a test pipe.

Fireworks.toolbox.preprocessing.**oversample**()

Fireworks.toolbox.preprocessing.**apply_noise**()

**class** Fireworks.toolbox.preprocessing.**Normalizer**(*components={}*, **args*, *input=None*, *skip_module_init=False*, ***kwargs*)
Bases: *Fireworks.core.model.Model*

Normalizes Data by Mean and Variance. Analogous to sklearn.preprocessing.Normalizer

**required_components = ['mean', 'variance']**

**init_default_components**()
This method can optionally be implemented in order for the model to provide a default initialization for some or all of its required components.

**forward**(*batch*)
Uses computed means and variances in order to transform the given batch.

**fit**(*dataset=None*, *continuamos=False*)

**reset**()

Fireworks.utils.events.**visdom_loss_handler**(*modules_dict*, *model_name*)
Attaches plots and metrics to trainer.

**exception** Fireworks.utils.exceptions.**EndHyperparameterOptimization**
Bases: RuntimeError

This exception can be raised to signal a factory to stop looping.

**exception** Fireworks.utils.exceptions.**ParameterizationError**
Bases: KeyError

This exception is raised to indicate that a Model is missing required parameters that it needs to function.

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## f

# INDEX

## F



## G



## H



## I



## J



## K



## L

## M



## N



## O



## P



## Q



## R

## S



## T



## U



## V



## W