noggin Documentation

Release 0.10.1+23.g0f1ef42

Ryan Soklaski

Oct 11, 2022

Contents:

Noggin is a simple Python tool for 'live' logging and plotting measurements during an experiment. Although Noggin can be used in a general context, it is designed around the train/test and batch/epoch paradigm for training a machine learning model.

Noggin's primary features are its abilities to:

- Log batch-level and epoch-level measurements by name
- Seamlessly update a 'live' plot of your measurements, embedded within a Jupyter notebook
- Organize your measurements into a data set of arrays with labeled axes, via xarray
- Save and load your measurements & live-plot session: resume your experiment later without a hitch

CHAPTER 1

A Simple Example of Using Your Noggin

Here is a sneak peak of what it looks like to use Noggin to record and plot data during an experiment. The following code is meant to be run in a Jupyter notebook.

```
%matplotlib notebook
import numpy as np
from noggin import create_plot
metrics = ["accuracy", "loss"]
plotter, fig, ax = create_plot(metrics)

for i, x in enumerate(np.linspace(0, 10, 100)):
    # record and plot batch-level metrics
    x += np.random.rand(1)*5
    batch_metrics = {"accuracy": x**2, "loss": 1/x**.5}
    plotter.set_train_batch(batch_metrics, batch_size=1, plot=True)

    # record training epoch
    if i%10 == 0 and i > 0:
        plotter.set_train_epoch()

        # cue test-evaluation of model
        for x in np.linspace(0, 10, 5):
            x += (np.random.rand(1) - 0.5)*5
            test_metrics = {"accuracy": x**2}
            plotter.set_test_batch(test_metrics, batch_size=1)
        plotter.set_test_epoch()
plotter.plot()  # ensures final data gets plotted
```

1.1 Noggin

Noggin is a simple Python tool for 'live' logging and plotting measurements during experiments. Although Noggin can be used in a general context, it is designed around the train/test and batch/epoch paradigm for training a machine

learning model.

Noggin's primary features are its abilities to:

- Log batch-level and epoch-level measurements by name
- Seamlessly update a 'live' plot of your measurements, embedded within a Jupyter notebook
- Organize your measurements into a data set of arrays with labeled axes, via xarray
- Save and load your measurements & live-plot session: resume your experiment later without a hitch

1.1.1 A Simple Example of Using Your Noggin

Here is a sneak peak of what it looks like to use Noggin to record and plot data during an experiment. The following code is meant to be run in a Jupyter notebook.

```
%matplotlib notebook
import numpy as np
from noggin import create_plot
metrics = ["accuracy", "loss"]
plotter, fig, ax = create_plot(metrics)

for i, x in enumerate(np.linspace(0, 10, 100)):
    # record and plot batch-level metrics
    x += np.random.rand(1)*5
    batch_metrics = {"accuracy": x**2, "loss": 1/x**.5}
    plotter.set_train_batch(batch_metrics, batch_size=1, plot=True)

    # record training epoch
    if i%10 == 0 and i > 0:
        plotter.set_train_epoch()

        # cue test-evaluation of model
        for x in np.linspace(0, 10, 5):
            x += (np.random.rand(1) - 0.5)*5
            test_metrics = {"accuracy": x**2}
            plotter.set_test_batch(test_metrics, batch_size=1)
        plotter.set_test_epoch()
plotter.plot()  # ensures final data gets plotted
```

1.2 Installing Noggin

Noggin requires: numpy, matplotlib, and xarray. You can install Noggin using pip:

```
pip install noggin
```

You can instead install Noggin from its source code. Clone this repository and navigate to the Noggin directory, then run:

```
python setup.py install
```

1.3 A Typical Workflow Using Noggin

Here, we will create a simple mock-up of an experiment in which we use Noggin to record and plot our measurements. We will exercise the critical features that this library provides us with. The following demo is intended to be conducted in a Jupyter notebook.

Please note that, if you don't need to visualize your data as you collect it, you can use *LiveLogger* to record your measurements in a nearly-identical manner.

1.3.1 Recording and Plotting Data During an Experiment

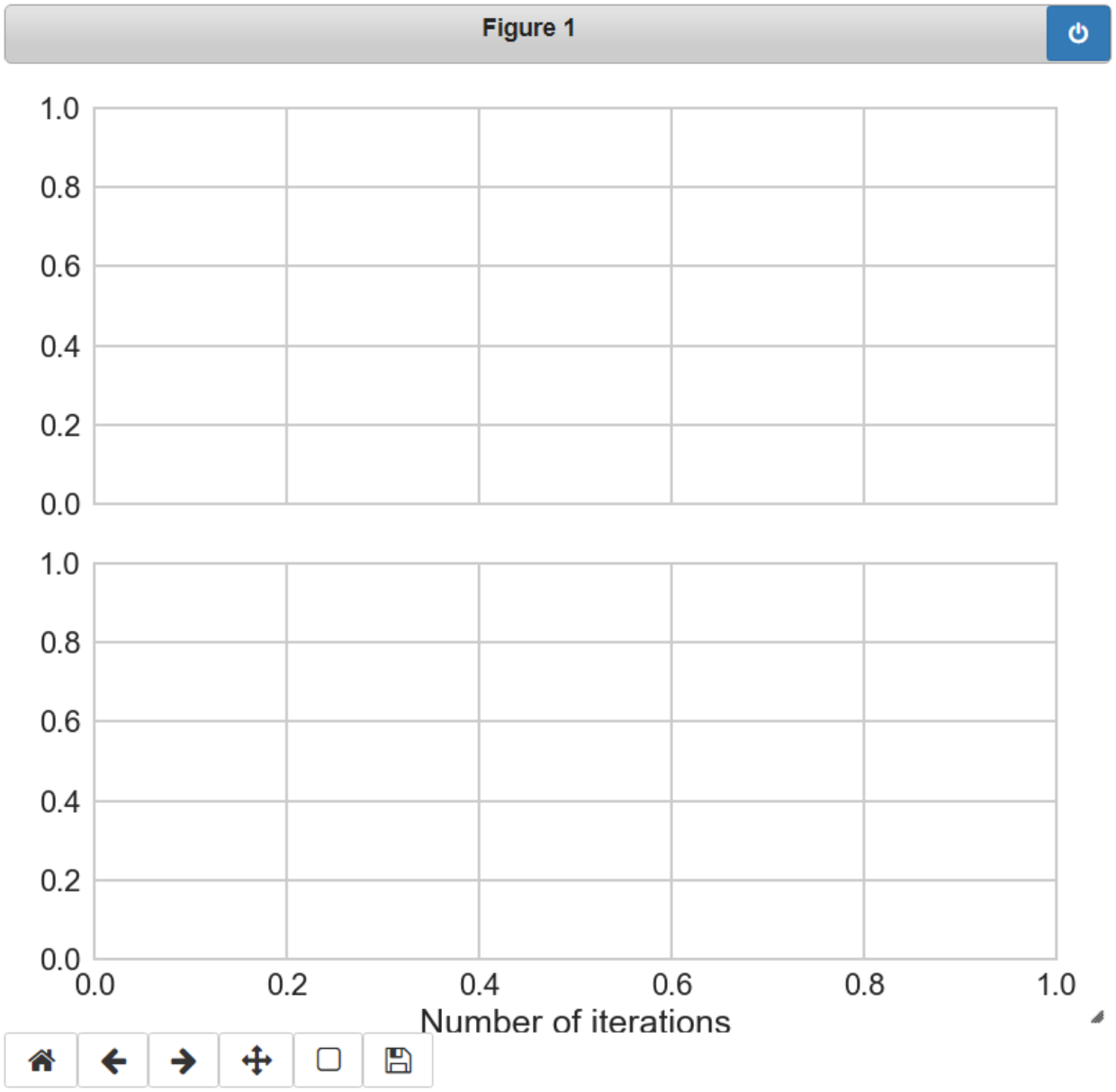
To begin, let's make up some functions to represent a data loader and a model that we are training:

```
"""
Defining mock data-loader and model-training functions for this simple demo.
The details here are not important, other than the fact that
`training_loop` returns a tuple of two floats.
"""

from time import sleep
from typing import Tuple

import numpy as np
np.random.seed(0)

def batch_loader(num_batches):
    """Simulates loading batches of data of varying sizes"""
    for i in np.linspace(-10, 10, num_batches):
        batch_size = np.random.randint(1, 10)
        yield batch_size * [i]

def training_loop(batch) -> Tuple[float, float]:
    """Simulates data processing Takes ~10ms to process a batch.
    Returns a 'loss' and 'accuracy'"""
    sleep(0.01)
    x = np.mean(batch)
    x += np.random.rand(1)*5 # add some noise
    return np.exp(-x / 5), 1 / (1 + np.exp(-x))
```

Noggin's operation is centered around metrics: the various measurements that we want to record and visualize. Here, we will be interested in measuring the *accuracy* of our model - on both training data and validation data - along with the training *loss*. In general, we can work with any variety and number of metrics in Noggin; a metric boils down to being any scalar value.

Let's create a live-plot for these two metrics; the resulting empty plot pane will automatically update as we proceed to make measurements during our experiment. In order to permit live-plotting in a Jupyter notebook, we need to enable the appropriate plotting backend: this is done by invoking the 'cell-magic' `%matplotlib notebook` (Note: one typically has to run this command twice before it will take effect - this seems to be a minor bug in Jupyter).

```
%matplotlib notebook
from noggin import create_plot

metrics = ['loss', 'accuracy']
plotter, fig, axes = create_plot(metrics)
```



There are two sets of axes in this figure, one for each of the metrics that we passed to *create_plot()*. Regarding the objects that this returned:

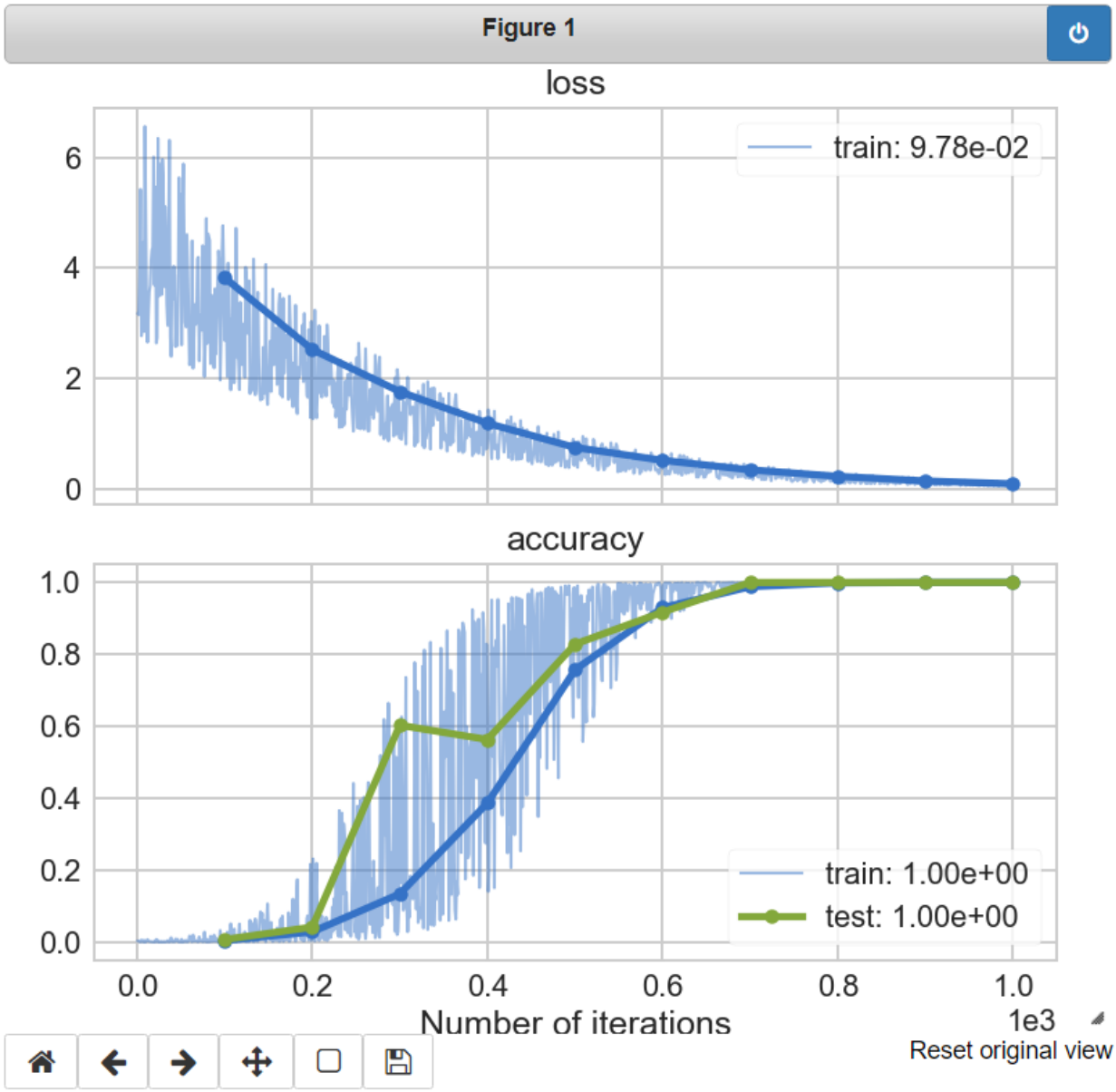
- `plotter` is an instance of *LivePlot*; it will be responsible for logging and plotting our measurements.
- `fig` and `axes` are the standard matplotlib figure and axes objects that are produced when one invokes `matplotlib.pyplot.subplots`; these can be used to affect and save the plot as you would with any matplotlib plot.

Without further ado, let's run our mock-experiment.

We will be passing batches of training data to our model-training function, recording the training-loss (i.e. the training objective) and the accuracy of our model. An 'epoch' will represent one hundred training batches. Here we will plot the average model accuracy for that epoch. We will also, at each epoch, measure the accuracy of our model on a set of test data. This is the standard affair for training a machine learning model.

```
# logging and plotting measurements during an experiment
for nbatch, batch in enumerate(batch_loader(1000)):
    loss, train_accuracy = training_loop(batch)
    recorded_metrics = dict(loss=loss, accuracy=train_accuracy)
    plotter.set_train_batch(recorded_metrics,
                            batch_size=len(batch))
    if (nbatch + 1) % 100 == 0:
        # record epoch-level statistics
        for test_cnt in range(10):
            # Measure model-accuracy on a validation set
            _, test_accuracy = training_loop(batch)
            plotter.set_test_batch(dict(accuracy=test_accuracy),
                                   batch_size=len(batch))
        plotter.set_train_epoch()
        plotter.set_test_epoch()
# make sure any "straggler" data gets plotted
plotter.plot()
```

As this experiment runs our plot pane will draw batch-level data with thin, semi-transparent lines. The epoch-level data will appear in bold, with each marker indicated. The most-recent epoch value for a metric will be recorded in the plot's legend. Please note that the x-axis, the number of batch iterations, *is indicated using scientific notation*. Once the experiment is complete our plot will look as follows:



There are a number of ways that you can customize your live plot; these are detailed elsewhere in the Noggin documentation. You can control:

- the figure-size of the plot and axis-grid layout for your metrics
- the plot colors across metrics and train/test splits
- the rate at which the plot is updated
- the maximum number of batches to be included in the plot
- whether or not you want to plot the batch-level data at all

1.3.2 Accessing Your Data

There are two ways to access the data that you recorded during your experiment: via xarray datasets or via dictionaries. It is recommended that you make keen use of the xarrays and their ability to handle data-alignment, missing data, and

many other features.

via xarray Datasets

The metrics that we recorded during our experiment are recorded as so-called 'data-variables' in an xarray dataset, which can be accessed via *to_xarray()*. And iteration-count serves as the coordinate that uniquely indexes these metrics.

```
# accessing train-metrics as an xarray dataset
>>> train_batch, train_epoch = plotter.to_xarray('train')
>>> train_batch
<xarray.Dataset>
Dimensions:     (iterations: 1000)
Coordinates:
  * iterations  (iterations) int32 1 2 3 4 5 6 7 ... 995 996 997 998 999 1000
Data variables:
    loss        (iterations) float64 3.176 3.154 3.842 ... 0.1056 0.06601 0.1135
    accuracy    (iterations) float64 0.003083 0.003193 0.001193 ... 1.0 1.0 1.0

>>> train_epoch
<xarray.Dataset>
Dimensions:     (iterations: 10)
Coordinates:
  * iterations  (iterations) int32 100 200 300 400 500 600 700 800 900 1000
Data variables:
loss        (iterations) float64 3.825 2.526 1.764 ... 0.2331 0.1495 0.09778
accuracy    (iterations) float64 0.00388 0.02844 0.1339 ... 0.9998 1.0
```

Each metric can be easily accessed as an attribute of this dataset; this returns an individual xarray `DataArray` for that metric:

```
  # accessing the data array for 'accuracy'
  >>> train_batch.accuracy  # or `train_batch['accuracy']
  <xarray.DataArray 'accuracy' (iterations: 1000)>
  array([0.003083, 0.003193, 0.001193, ..., 0.999987, 0.999999, 0.999981])
  Coordinates:
* iterations  (iterations) int32 1 2 3 4 5 6 7 ... 995 996 997 998 999 1000
```

xarray's data structures are powerful and highly-convenient. They provide a natural means for aligning batch-level and epoch-level measurements using iteration count. Furthermore, they handle missing data gracefully.

Towards this end, if you run multiple iterations of an experiment, then you can use *concat_experiments()* to combine your data sets along a new 'experiments' axis. This will gracefully accommodate combining experiments that were run for differing numbers of iterations, and will permit you to seamlessly compute statistics across them.

via Dictionaries

You can access your recorded metrics as dictionaries via `train_metrics()` and `test_metrics()`.

The structure of the resulting dictionary is:

```
'<metric-name>' -> {"batch_data":   array,
                    "epoch_data":   array,
                    "epoch_domain": array,
                    ...}
```

```
>>> plotter.train_metrics['accuracy']['batch_data']
array([3.08328619e-03, 3.19260208e-03, ..., 9.99981201e-01])
```

1.3.3 Saving and Resuming Your Experiment

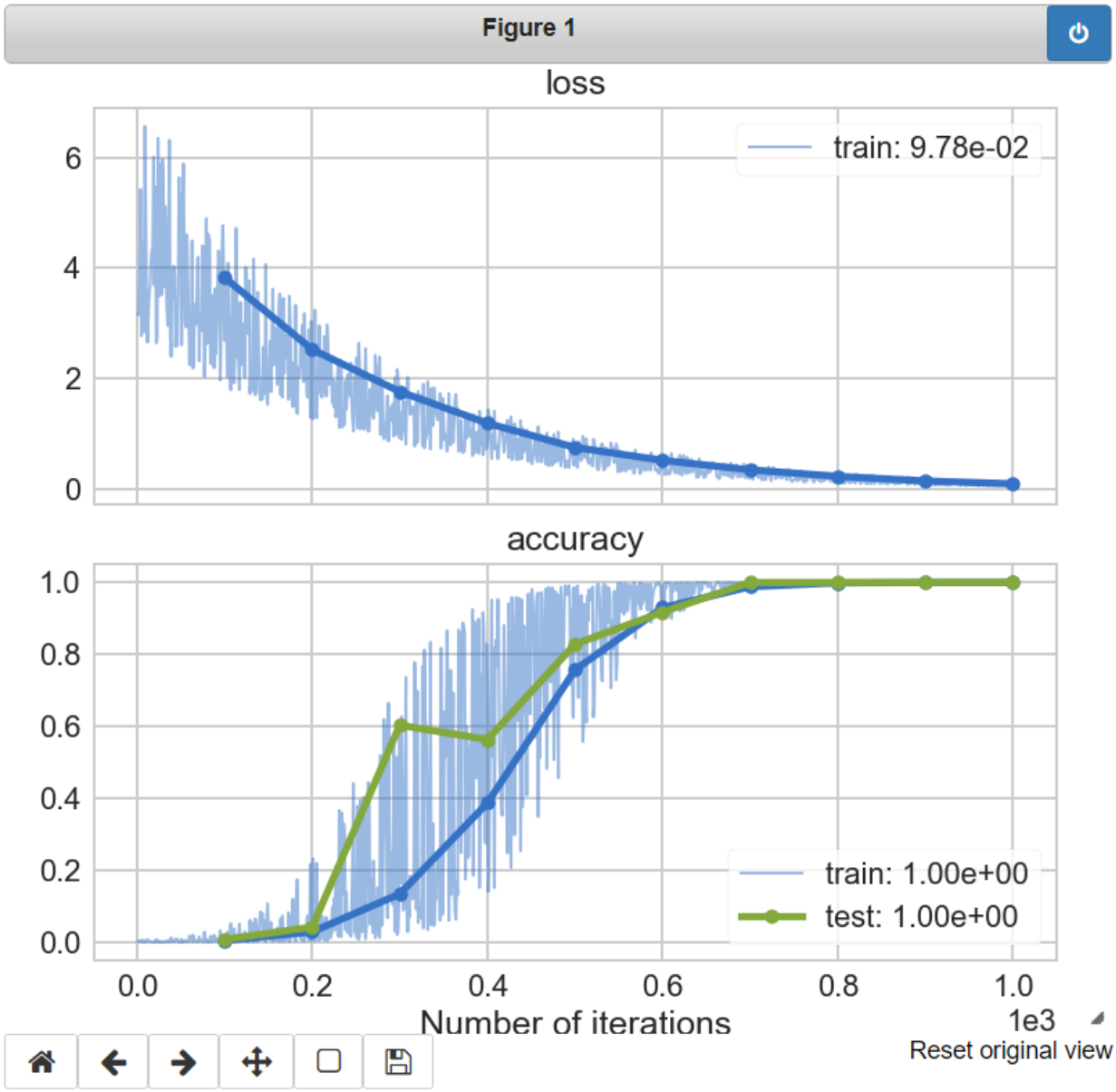
Instances of Noggin's *LivePlot* and *LiveLogger* classes can both be converted to dictionaries, which can then be "pickled" - saving them for later use.

Let's convert `plotter` to a dictionary using *to_dict()* and save it:

```
# converting `plotter` to a dictionary and pickling it
import pickle

with open('plotter.pkl', 'wb') as f:
    pickle.dump(plotter.to_dict(), f, protocol=-1)
```

We can now easily load our pickled plotter and recreate our plot as we left it, via *from_dict()*

```
# loading the pickled plotter and recreating the plot
from noggin import LivePlot

with open('plotter.pkl', 'rb') as f:
    loaded_dict = pickle.load(f)
    loaded_plotter = LivePlot.from_dict(loaded_dict)

fig, ax = loaded_plotter.plot_objects
loaded_plotter.plot()
```



We can now resume recording measurements in our experiment just as we were doing earlier; our metrics will be logged and plotted just as before!

1.4 Logging Data with Noggin

Noggin's core role in an experiment is to log your measurements and store them in an organized, accessible manner. `LiveLogger` is responsible for facilitating this. This class is meant to serve as a drop-in replacement for `LivePlot`, and is useful for running experiments where you do not need a live visualization of your data.

Batch-level measurements can be logged for both train and test splits of data, and an epoch can be marked in order to compute the mean-value of each metric over that epoch.

Let's record measurements for two batches of data and mark an epoch. Note that we need to provide the logger the measurements by name, via a dictionary.

```
>>> from noggin import LiveLogger
>>> logger = LiveLogger()
>>> logger.set_train_batch(dict(metric_a=2., metric_b=1.), batch_size=10)
>>> logger.set_train_batch(dict(metric_a=0., metric_b=2.), batch_size=4)
>>> logger.set_train_epoch()  # compute the mean statistics
>>> logger
LiveLogger(metric_a, metric_b)
number of training batches set: 2
number of training epochs set: 1
number of testing batches set: 0
number of testing epochs set: 0
```

Accessing our logged batch-level and epoch-level data works the same way as when working with an instance of *LivePlot*:

```
# accessing the logged data as xarrays.
>>> batch_array, epoch_array = logger.to_xarray("train")
>>> batch_array
<xarray.Dataset>
Dimensions:     (iterations: 2)
Coordinates:
  * iterations  (iterations) int32 1 2
Data variables:
    metric_a    (iterations) float64 2.0 0.0
    metric_b    (iterations) float64 1.0 2.0
>>> epoch_array
<xarray.Dataset>
Dimensions:     (iterations: 1)
Coordinates:
  * iterations  (iterations) int32 2
Data variables:
    metric_a    (iterations) float64 1.429
    metric_b    (iterations) float64 1.286)
```

Note that the epoch-level measurements are aligned with the batch-level measurements along the 'iterations' coordinate-axis. (E.g. we can see that our first epoch was recorded at batch-iteration 2).

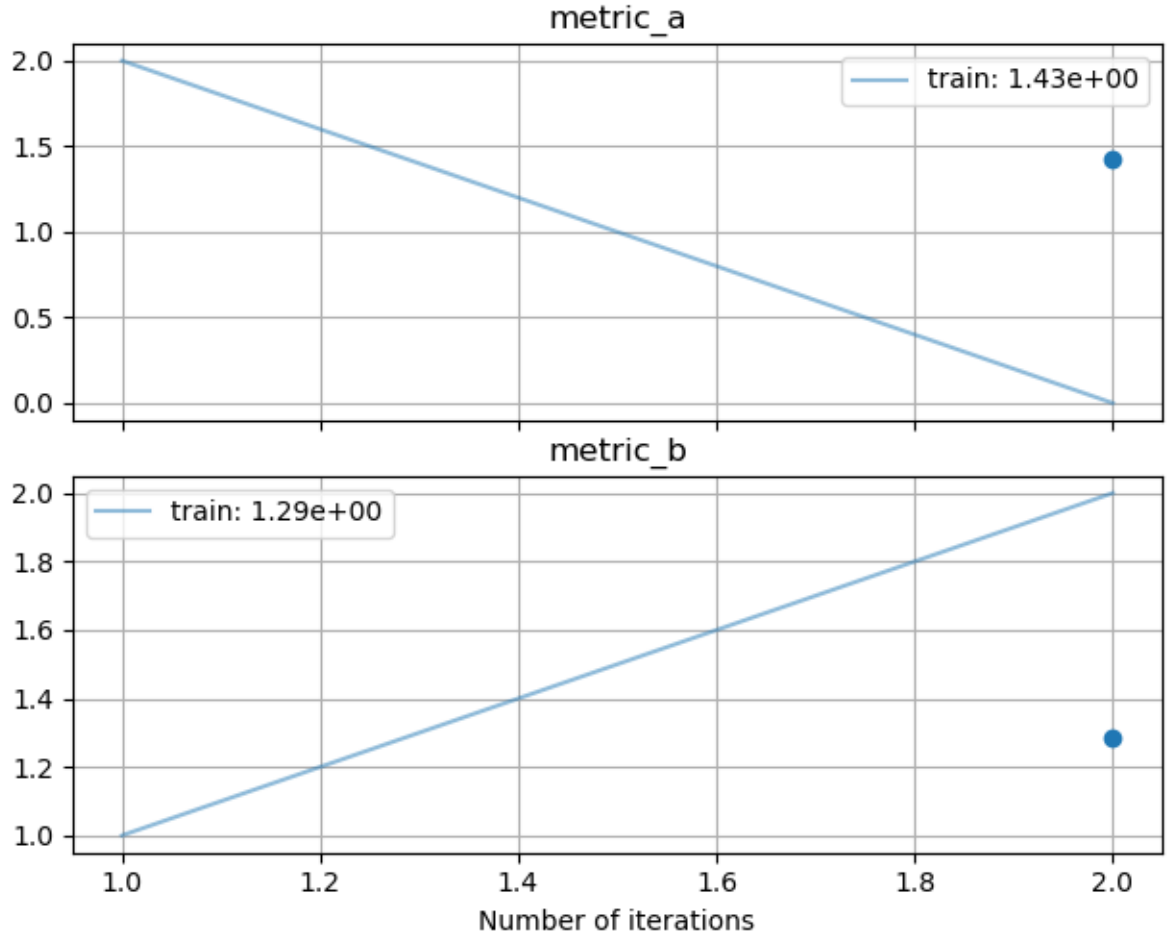
Additionally, *saving and loading your logger* is as simple as converting your logger to a dictionary, and pickling it:

```
import pickle

# converting `logger` to a dictionary and pickling it
with open('logger.pkl', 'wb') as f:
    pickle.dump(logger.to_dict(), f, protocol=-1)

# loading the logger
with open('logger.pkl', 'rb') as f:
    loaded_dict = pickle.load(f)
    loaded_logger = LiveLogger.from_dict(loaded_dict)
```

1.4.1 Converting a Logger to a Plotter

It is easy to visualize your logged data and to convert your logger to an instance of *LivePlot*, thanks to *plot_logger()*:

```
from noggin import plot_logger
plotter, fig, ax = plot_logger(logger)
plotter.show()
```



This gives us access to the matplotlib figure and axes objects for our plot, and `plotter` is the instance of *`LivePlot`* that stores our logged data. `plotter.max_fraction_spent_plotting` will be 0 by default, but you can increase this value and proceed to use `plotter` to visualize your measurements in realtime.

1.5 Documentation for Noggin

1.5.1 Documentation for noggin.logger

`LiveMetric`(name)	Holds the relevant data for a train/test metric for live plotting.

noggin.logger.LiveMetric

class noggin.logger.**LiveMetric**(*name: str*)

Holds the relevant data for a train/test metric for live plotting.

Attributes

batch_data Batch-level measurements of the metric.

batch_domain Array of iteration-counts at which the metric was recorded.

epoch_data Epoch-level measurements of the metrics.

epoch_domain Array of iteration-counts at which an epoch was set for this metric.

name Name of the metric.

Methods

add_datapoint(value, weighting)	Record a batch-level measurement of the metric.
from_dict(metrics_dict, numpy.ndarray])	The inverse of `LiveMetric.to_dict`.
set_epoch_datapoint(x)	Mark the present iteration as an epoch, and compute the mean value of the metric since the past epoch.
to_dict()	Returns the batch data, epoch domain, and epoch data in a dictionary.

noggin.logger.LiveMetric.add_datapoint

`LiveMetric.`**`add_datapoint`**(*value: numbers.Real*, *weighting: numbers.Real = 1.0*)

Record a batch-level measurement of the metric.

Parameters

value [Real] The recorded value.

weighting [Real] The weight with which this recorded value will contribute to the epoch-level mean.

noggin.logger.LiveMetric.from_dict

classmethod `LiveMetric.`**`from_dict`**(*metrics_dict: Dict[str, numpy.ndarray]*)

The inverse of `LiveMetric.to_dict`. Given a dictionary of live-metric data, constructs an instance of *LiveMetric*.

Parameters

metrics_dict: Dict[str, ndarray] Stores the state of the live-metric instance being created.

Returns

noggin.LiveMetric

Notes

The encoded dictionary stores:

```
'batch_data' -> ndarray, shape-(N,)
'epoch_data' -> ndarray, shape-(M,)
'epoch_domain' -> ndarray, shape-(M,)
'cnt_since_epoch' -> int
'total_weighting' -> float
```

(continues on next page)

(continued from previous page)

```
'running_weighted_sum' -> float
'name' -> str
```

noggin.logger.LiveMetric.set_epoch_datapoint

LiveMetric.**set_epoch_datapoint**(*x: Optional[numbers.Real] = None*)

Mark the present iteration as an epoch, and compute the mean value of the metric since the past epoch.

Parameters

x [Optional[Real]] Specify the domain-value to be set for this data point.

noggin.logger.LiveMetric.to_dict

LiveMetric.**to_dict**() → Dict[str, numpy.ndarray]

Returns the batch data, epoch domain, and epoch data in a dictionary.

Additionally, running statistics are included in order to preserve the state of the metric.

Returns

Dict[str, ndarray]

Notes

The encoded dictionary stores:

```
'batch_data' -> ndarray, shape=(N,)
'epoch_data' -> ndarray, shape=(M,)
'epoch_domain' -> ndarray, shape=(M,)
'cnt_since_epoch' -> int
'total_weighting' -> float
'running_weighted_sum' -> float
'name' -> str
```

__init__(*name: str*)

Parameters

name [str]

Raises

TypeError Invalid metric name (must be string)

Methods

__init__(name)	**Parameters**
add_datapoint(value, weighting)	Record a batch-level measurement of the metric.
from_dict(metrics_dict, numpy.ndarray])	The inverse of LiveMetric.to_dict.

Continued on next page

Table 3 – continued from previous page

set_epoch_datapoint(x)	Mark the present iteration as an epoch, and compute the mean value of the metric since the past epoch.
to_dict()	Returns the batch data, epoch domain, and epoch data in a dictionary.

Attributes

batch_data	Batch-level measurements of the metric.
batch_domain	Array of iteration-counts at which the metric was recorded.
epoch_data	Epoch-level measurements of the metrics.
epoch_domain	Array of iteration-counts at which an epoch was set for this metric.
name	Name of the metric.

LiveLogger(*args, **kwargs)	Logs batch-level and epoch-level summary statistics of the training and testing metrics of a model during a session.

noggin.logger.LiveLogger

class noggin.logger.**LiveLogger**(**args*, ***kwargs*)

Logs batch-level and epoch-level summary statistics of the training and testing metrics of a model during a session.

Examples

A simple example in which we log two iterations of training batches, and set an epoch.

```
>>> from noggin import LiveLogger
>>> logger = LiveLogger()
>>> logger.set_train_batch(dict(metric_a=2., metric_b=1.), batch_size=10)
>>> logger.set_train_batch(dict(metric_a=0., metric_b=2.), batch_size=4)
>>> logger.set_train_epoch()  # compute the mean statistics
>>> logger
LiveLogger(metric_a, metric_b)
number of training batches set: 2
number of training epochs set: 1
number of testing batches set: 0
number of testing epochs set: 0
```

Accessing our logged batch-level and epoch-level data

```
>>> logger.to_xarray("train")
MetricArrays(batch=<xarray.Dataset>
Dimensions:     (iterations: 2)
Coordinates:
  * iterations  (iterations) int32 1 2
Data variables:
    metric_a    (iterations) float64 2.0 0.0
    metric_b    (iterations) float64 1.0 2.0,
```

(continues on next page)

(continued from previous page)

```
epoch=<xarray.Dataset>
Dimensions:     (iterations: 1)
Coordinates:
  * iterations  (iterations) int32 2
Data variables:
    metric_a    (iterations) float64 1.429
    metric_b    (iterations) float64 1.286)
```

Attributes

test_metrics The batch and epoch data for each test-metric.

train_metrics The batch and epoch data for each train-metric.

Methods

from_dict(logger_dict, Any])	Records the state of the logger in a dictionary.
set_test_batch(metrics, numbers.Real], ...)	Record batch-level measurements for test-metrics.
set_test_epoch()	Record an epoch for the test-metrics.
set_train_batch(metrics, numbers.Real], ...)	Record batch-level measurements for train-metrics.
set_train_epoch()	Record an epoch for the train-metrics.
to_dict()	Records the state of the logger in a dictionary.
to_xarray(train_or_test)	Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

noggin.logger.LiveLogger.from_dict

classmethod LiveLogger.**from_dict**(*logger_dict: Dict[str, Any]*)

Records the state of the logger in a dictionary.

This is the inverse of *to_dict()*

Parameters

logger_dict [Dict[str, Any]] The dictionary storing the state of the logger to be restored.

Returns

noggin.LiveLogger The restored logger.

Notes

This is a class-method, the syntax for invoking it is:

```
>>> LiveLogger.from_dict(logger_dict)
LiveLogger(metric_a, metric_b)
number of training batches set: 3
number of training epochs set: 1
number of testing batches set: 0
number of testing epochs set: 0
```

noggin.logger.LiveLogger.set_test_batch

LiveLogger.**set_test_batch**(*metrics: Dict[str, numbers.Real], batch_size: numbers.Integral*)
Record batch-level measurements for test-metrics.

Parameters

metrics [Dict[str, Real]] Mapping of metric-name to value. Only those metrics that were registered when initializing LivePlot will be recorded.

batch_size [Integral] The number of samples in the batch used to produce the metrics. Used to weight the metrics to produce epoch-level statistics.

noggin.logger.LiveLogger.set_test_epoch

LiveLogger.**set_test_epoch**()
Record an epoch for the test-metrics.

Computes epoch-level statistics based on the batches accumulated since the prior epoch.

noggin.logger.LiveLogger.set_train_batch

LiveLogger.**set_train_batch**(*metrics: Dict[str, numbers.Real], batch_size: numbers.Integral, **kwargs*)
Record batch-level measurements for train-metrics.

Parameters

metrics [Dict[str, Real]] Mapping of metric-name to value. Only those metrics that were registered when initializing LivePlot will be recorded.

batch_size [Integral] The number of samples in the batch used to produce the metrics. Used to weight the metrics to produce epoch-level statistics.

Notes

`**kwargs` is included in the signature only to facilitate a seamless drop-in replacement for *LivePlot*. It is not utilized here.

noggin.logger.LiveLogger.set_train_epoch

LiveLogger.**set_train_epoch**()
Record an epoch for the train-metrics.

Computes epoch-level statistics based on the batches accumulated since the prior epoch.

noggin.logger.LiveLogger.to_dict

LiveLogger.**to_dict**() → Dict[str, Any]
Records the state of the logger in a dictionary.

This is the inverse of *from_dict()*

Returns

Dict[str, Any]

Notes

To save your logger, use this method to convert it to a dictionary and then pickle the dictionary.

noggin.logger.LiveLogger.to_xarray

`LiveLogger.`**`to_xarray`**(*train_or_test: str*) → Tuple[xarray.core.dataset.Dataset, xarray.core.dataset.Dataset]

Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

Parameters

train_or_test [str] Either 'train' or 'test' - specifies which measurements to be returned

Returns

Tuple[xarray.Dataset, xarray.Dataset] The batch-level and epoch-level datasets. The metrics are reported as data variables in the dataset, and the coordinates corresponds to the batch-iteration count.

Notes

The layout of the resulting data sets are:

```
Dimensions:     (iterations: num_iterations)
Coordinates:
  * iterations  (iterations) int64 1 2 3 ...
Data variables:
    metric0      (iterations) float64 val_0 val_1 ...
    metric1      (iterations) float64 val_0 val_1 ...
    ...
```

Each metric can be accessed as an attribute of the resulting data-set, e.g. `dataset.metric0`, or via the 'get-item' syntax, e.g. `dataset['metric0']`. This returns a data-array for that metric.

Data sets collected from multiple trials of an experiment can be combined using *`concat_experiments()`*.

`__init__`(**args*, ***kwargs*)

LiveLogger.__init__ does not utilize any input arguments, but accepts `*args, **kwargs` so that it can be used as a drop-in replacement for

`LivePlot`.

Methods

`__init__`(*args, **kwargs)	LiveLogger.__init__ does not utilize any input arguments, but accepts `*args, **kwargs` so that it can be used as a drop-in replacement for *`LivePlot`*.

Continued on next page

Table 7 – continued from previous page

from_dict(logger_dict, Any])	Records the state of the logger in a dictionary.
set_test_batch(metrics, numbers.Real], ...)	Record batch-level measurements for test-metrics.
set_test_epoch()	Record an epoch for the test-metrics.
set_train_batch(metrics, numbers.Real], ...)	Record batch-level measurements for train-metrics.
set_train_epoch()	Record an epoch for the train-metrics.
to_dict()	Records the state of the logger in a dictionary.
to_xarray(train_or_test)	Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

Attributes

`test_metrics`	The batch and epoch data for each test-metric.
`train_metrics`	The batch and epoch data for each train-metric.

1.5.2 Documentation for noggin.plotter

LivePlot(metrics, Sequence[str], Dict[str, ...)	Records and plots batch-level and epoch-level summary statistics of the training and testing metrics of a model during a session.

noggin.plotter.LivePlot

class noggin.plotter.**LivePlot**(*metrics: Union[str, Sequence[str], Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]], Dict[str, Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]]]], max_fraction_spent_plotting: float = 0.05, last_n_batches: Optional[int] = None, nrows: Optional[int] = None, ncols: int = 1, figsize: Optional[Tuple[int, int]] = None*)

Records and plots batch-level and epoch-level summary statistics of the training and testing metrics of a model during a session.

The rate at which the plot is updated is controlled by `max_fraction_spent_plotting`.

The maximum number of batches to be included in the plot is controlled by `last_n_batches`.

Notes

Live plotting is only supported for the 'nbAgg' backend (i.e. when the cell magic `%matplotlib notebook` is invoked in a jupyter notebook).

Attributes

figsize Returns the current size of the figure in inches.

last_n_batches The maximum number of batches to be plotted at any given time.

max_fraction_spent_plotting The maximum fraction of time spent plotting.

metric_colors The color associated with each of the train/test and batch/epoch-level metrics.

metrics A tuple of all the metric names

plot_objects The figure-instance of the plot, and the axis-instance for each metric.

test_metrics The batch and epoch data for each test-metric.

train_metrics The batch and epoch data for each train-metric.

Methods

from_dict(plotter_dict)	Records the state of the plotter in a dictionary.
plot(plot_batches)	Plot the logged data.
set_test_batch(metrics, numbers.Real], ...)	Record batch-level measurements for test-metrics.
set_test_epoch()	Record and plot an epoch for the test-metrics.
set_train_batch(metrics, numbers.Real], ...)	Record batch-level measurements for train-metrics, and (optionally) plot them.
set_train_epoch()	Record and plot an epoch for the train-metrics.
show()	Calls `matplotlib.pyplot.show()`.
to_dict()	Records the state of the plotter in a dictionary.
to_xarray(train_or_test)	Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

noggin.plotter.LivePlot.from_dict

classmethod LivePlot.**from_dict**(*plotter_dict*)

Records the state of the plotter in a dictionary.

This is the inverse of *to_dict()*

Parameters

plotter_dict [Dict[str, Any]] The dictionary storing the state of the logger to be restored.

Returns

noggin.LivePlot The restored plotter.

Notes

This is a class-method, the syntax for invoking it is:

```
>>> loaded_plotter = LivePlot.from_dict(plotter_dict)
```

To restore your plot from the loaded plotter, call:

```
>>> loaded_plotter.plot()
```

noggin.plotter.LivePlot.plot

LivePlot.**plot**(*plot_batches: bool = True*)

Plot the logged data.

This method can be used to 'force' a plot to be drawn, and should *not* be called repeatedly while logging data.

Instead, one should invoke `Liveplot.set_train_batch(plot=True)`, `Liveplot.set_train_epoch`, and `Liveplot.set_test_epoch`, which will adjust their plot-rates according to `Liveplot.max_fraction_spent_plotting`.

`LivePlot.plot` should be called at the end of a logging-loop to ensure that the logged data is plotted in its entirety. This can also be used to recreate a plot after deserializing a `LivePlot` instance.

Parameters

plot_batches [bool, optional (default=True)] If `True` include batch-level data in plot.

noggin.plotter.LivePlot.set_test_batch

`LivePlot.`**`set_test_batch`**(*metrics: Dict[str, numbers.Real], batch_size: numbers.Integral*)

Record batch-level measurements for test-metrics.

Parameters

metrics [Dict[str, Real]] Mapping of metric-name to value. Only those metrics that were registered when initializing LivePlot will be recorded.

batch_size [Integral] The number of samples in the batch used to produce the metrics. Used to weight the metrics to produce epoch-level statistics.

noggin.plotter.LivePlot.set_test_epoch

`LivePlot.`**`set_test_epoch`**()

Record and plot an epoch for the test-metrics.

Computes epoch-level statistics based on the batches accumulated since the prior epoch.

noggin.plotter.LivePlot.set_train_batch

`LivePlot.`**`set_train_batch`**(*metrics: Dict[str, numbers.Real], batch_size: numbers.Integral, plot: bool = True*)

Record batch-level measurements for train-metrics, and (optionally) plot them.

Parameters

metrics [Dict[str, Real]] Mapping of metric-name to value. Only those metrics that were registered when initializing LivePlot will be recorded.

batch_size [Integral] The number of samples in the batch used to produce the metrics. Used to weight the metrics to produce epoch-level statistics.

plot [bool] If True, plot the batch-metrics (adhering to the refresh rate)

noggin.plotter.LivePlot.set_train_epoch

`LivePlot.`**`set_train_epoch`**()

Record and plot an epoch for the train-metrics.

Computes epoch-level statistics based on the batches accumulated since the prior epoch.

noggin.plotter.LivePlot.show

LivePlot.**show**()
Calls `matplotlib.pyplot.show()`. For visualizing a static-plot

noggin.plotter.LivePlot.to_dict

LivePlot.**to_dict**()
Records the state of the plotter in a dictionary.

This is the inverse of *from_dict()*

Returns

Dict[str, Any]

Notes

To save your plotter, use this method to convert it to a dictionary and then pickle the dictionary.

noggin.plotter.LivePlot.to_xarray

LivePlot.**to_xarray**(*train_or_test: str*) → Tuple[xarray.core.dataset.Dataset, xarray.core.dataset.Dataset]
Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

Parameters

train_or_test [str] Either 'train' or 'test' - specifies which measurements to be returned

Returns

Tuple[xarray.Dataset, xarray.Dataset] The batch-level and epoch-level datasets. The metrics are reported as data variables in the dataset, and the coordinates corresponds to the batch-iteration count.

Notes

The layout of the resulting data sets are:

```
Dimensions:     (iterations: num_iterations)
Coordinates:
  * iterations  (iterations) int64 1 2 3 ...
Data variables:
    metric0      (iterations) float64 val_0 val_1 ...
    metric1      (iterations) float64 val_0 val_1 ...
    ...
```

Each metric can be accessed as an attribute of the resulting data-set, e.g. `dataset.metric0`, or via the 'get-item' syntax, e.g. `dataset['metric0']`. This returns a data-array for that metric.

Data sets collected from multiple trials of an experiment can be combined using *concat_experiments()*.

__init__(*metrics: Union[str, Sequence[str], Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]], Dict[str, Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]]]], max_fraction_spent_plotting: float = 0.05, last_n_batches: Optional[int] = None, nrows: Optional[int] = None, ncols: int = 1, figsize: Optional[Tuple[int, int]] = None*)

Parameters

metrics [Union[str, Sequence[str], Dict[str, valid-color], Dict[str, Dict['train'/'test', valid-color]]]] The name, or sequence of names, of the metric(s) that will be plotted.

`metrics` can also be a dictionary, specifying the colors used to plot the metrics. Two mappings are valid:

- '<metric-name>' -> color-value (specifies train-metric color only)
- '<metric-name>' -> {'train'/'test' : color-value}

max_fraction_spent_plotting [float, optional (default=0.05)] The maximum fraction of time spent plotting. The default value is `0.05`, meaning that no more than 5% of processing time will be spent plotting, on average.

last_n_batches [Optional[int]] The maximum number of batches to be plotted at any given time. If `None`, all data will be plotted.

nrows [Optional[int]] Number of rows of the subplot grid. Metrics are added in row-major order to fill the grid.

ncols [int, optional, default: 1] Number of columns of the subplot grid. Metrics are added in row-major order to fill the grid.

figsize [Optional[Sequence[float, float]]] Specifies the width and height, respectively, of the figure.

Methods

__init__(metrics, Sequence[str], Dict[str, ...)	**Parameters**
from_dict(plotter_dict)	Records the state of the plotter in a dictionary.
plot(plot_batches)	Plot the logged data.
set_test_batch(metrics, numbers.Real], ...)	Record batch-level measurements for test-metrics.
set_test_epoch()	Record and plot an epoch for the test-metrics.
set_train_batch(metrics, numbers.Real], ...)	Record batch-level measurements for train-metrics, and (optionally) plot them.
set_train_epoch()	Record and plot an epoch for the train-metrics.
show()	Calls `matplotlib.pyplot.show()`.
to_dict()	Records the state of the plotter in a dictionary.
to_xarray(train_or_test)	Returns xarray datasets for the batch-level and epoch-level metrics, respectively, for either the train-metrics or test-metrics.

Attributes

`figsize`	Returns the current size of the figure in inches.

Continued on next page

Table 12 – continued from previous page

last_n_batches	The maximum number of batches to be plotted at any given time.
max_fraction_spent_plotting	The maximum fraction of time spent plotting.
metric_colors	The color associated with each of the train/test and batch/epoch-level metrics.
metrics	A tuple of all the metric names
plot_objects	The figure-instance of the plot, and the axis-instance for each metric.
test_metrics	The batch and epoch data for each test-metric.
train_metrics	The batch and epoch data for each train-metric.

1.5.3 Documentation for noggin.xarray

metrics_to_xarrays(metrics, Dict[str, ...)	Given noggin metrics, returns xarray datasets for the batch-level and epoch-level metrics, respectively.
concat_experiments(*exps)	Concatenates xarray data sets from a sequence of experiments.

noggin.xarray.metrics_to_xarrays

noggin.xarray.**metrics_to_xarrays**(*metrics: Dict[str, Dict[str, numpy.ndarray]]*) → Tuple[xarray.core.dataset.Dataset, xarray.core.dataset.Dataset]

Given noggin metrics, returns xarray datasets for the batch-level and epoch-level metrics, respectively.

Parameters

metrics [Dict[str, Dict[str, ndarray]]] Live metrics reported as a dictionary, (e.g. via *LivePlot.train_metrics* or *LivePlot.test_metrics*)

Returns

MetricArrays[xarray.Dataset, xarray.Dataset] The batch-level and epoch-level datasets. The metrics are reported as data variables in the dataset, and the coordinates corresponds to the batch-iteration count.

Notes

The layout of the resulting data sets are:

```
Dimensions:     (iterations: num_iterations)
Coordinates:
  * iterations  (iterations) int64 1 2 3 ...
Data variables:
    metric0      (iterations) float64 val_0 val_1 ...
    metric1      (iterations) float64 val_0 val_1 ...
    ...
```

noggin.xarray.concat_experiments

noggin.xarray.**concat_experiments**(**exps*) → xarray.core.dataset.Dataset

Concatenates xarray data sets from a sequence of experiments.

Specifically, data sets that record identical metrics measured across several independent experiments will be concatenated along a new dimension, 'experiment', which tracks the experiment-index associated with the corresponding array of metrics.

Parameters

***exps: Dataset** One or more data sets recording metrics across independent runs of an experiment.

Returns

Dataset The recorded metrics joined into a single data set, along an experiment-index dimension.

Notes

The form of the resulting Dataset is:

```
Dimensions:     (experiment: num_exps, iterations: max_num_its)
Coordinates:
  * experiment  (experiment) int32 0 1 2 ...
  * iterations  (iterations) int64 1 2 3 ...
Data variables:
    metric0      (experiment, iterations) float64 val_0 val_1 ...
    metric1      (experiment, iterations) float64 val_0 val_1 ...
    ...
```

1.5.4 Documentation for noggin.utils

create_plot(metrics, Sequence[str], ...)	Create matplotlib figure/axes, and a live-plotter, which publishes "live" training/testing metric data, at a batch and epoch level, to the figure.
plot_logger(logger, plot_batches, ...)	Plots the data recorded by a *LiveLogger* instance.
save_metrics(path, pathlib.Path], liveplot, ...)	Save live-plot metrics to a numpy zipped-archive (.npz).
load_metrics(path, pathlib.Path])	Load noggin metrics from a numpy archive.

noggin.utils.create_plot

noggin.utils.**create_plot**(*metrics: Union[str, Sequence[str], Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]], Dict[str, Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]]]], max_fraction_spent_plotting: float = 0.05, last_n_batches: Optional[int] = None, nrows: Optional[int] = None, ncols: int = 1, figsize: Optional[Tuple[int, int]] = None*) → Tuple[noggin.plotter.LivePlot, matplotlib.figure.Figure, numpy.ndarray]

Create matplotlib figure/axes, and a live-plotter, which publishes "live" training/testing metric data, at a batch and epoch level, to the figure.

Parameters

metrics [Union[str, Sequence[str], Dict[str, valid-color], Dict[str, Dict['train'/'test', valid-color]]]] The name, or sequence of names, of the metric(s) that will be plotted.

`metrics` can also be a dictionary, specifying the colors used to plot the metrics. Two mappings are valid:

- '<metric-name>' -> color-value (specifies train-metric color only)
- '<metric-name>' -> {'train'/'test' : color-value}

max_fraction_spent_plotting [float, optional (default=0.05)] The maximum fraction of time spent plotting. The default value is `0.05`, meaning that no more than 5% of processing time will be spent plotting, on average.

last_n_batches [Optional[int]] The maximum number of batches to be plotted at any given time. If `None`, all data will be plotted.

nrows [Optional[int]] Number of rows of the subplot grid. Metrics are added in row-major order to fill the grid.

ncols [int, optional, default: 1] Number of columns of the subplot grid. Metrics are added in row-major order to fill the grid.

figsize [Optional[Sequence[float, float]]] Specifies the width and height, respectively, of the figure.

Returns

Tuple[liveplot.LivePlot, matplotlib.figure.Figure, numpy.ndarray(matplotlib.axes.Axes)] (LivePlot-instance, figure, array-of-axes)

Examples

Creating a live plot in a Jupyter notebook

```
>>> %matplotlib notebook
>>> import numpy as np
>>> from noggin import create_plot, save_metrics
>>> metrics = ["accuracy", "loss"]
>>> plotter, fig, ax = create_plot(metrics)
>>> for i, x in enumerate(np.linspace(0, 10, 100)):
...     # training
...     x += np.random.rand(1)*5
...     batch_metrics = {"accuracy": x**2, "loss": 1/x**.5}
...     plotter.set_train_batch(batch_metrics, batch_size=1, plot=True)
...
...     # cue training epoch
...     if i%10 == 0 and i > 0:
...         plotter.plot_train_epoch()
...
...         # cue test-time computations
...         for x in np.linspace(0, 10, 5):
...             x += (np.random.rand(1) - 0.5)*5
...             test_metrics = {"accuracy": x**2}
...             plotter.set_test_batch(test_metrics, batch_size=1)
...         plotter.plot_test_epoch()
...
... plotter.plot()  # ensures final data gets plotted
```

Saving the logged metrics

```
>>> save_metrics("./metrics.npz", plotter) # save metrics to numpy-archive
```

noggin.utils.plot_logger

noggin.utils.**plot_logger**(*logger: noggin.logger.LiveLogger, plot_batches: bool = True, last_n_batches: Optional[int] = None, colors: Optional[Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None, Dict[str, Union[str, numbers.Real, Sequence[numbers.Real], None]]]]] = None, nrows: Optional[int] = None, ncols: int = 1, figsize: Optional[Tuple[int, int]] = None*) → Tuple[noggin.plotter.LivePlot, matplotlib.figure.Figure, Union[matplotlib.axes._axes.Axes, numpy.ndarray]]

Plots the data recorded by a `LiveLogger` instance.

Converts the logger to an instance of `LivePlot`.

Parameters

logger [LiveLogger] The logger whose train/test-split batch/epoch-level data will be plotted.

plot_batches [bool, optional (default=True)] If `True` include batch-level data in plot.

last_n_batches [Optional[int]] The maximum number of batches to be plotted at any given time. If `None`, all of the data will be plotted.

colors [Optional[Dict[str, Union[ValidColor, Dict[str, ValidColor]]]]] `colors` can be a dictionary, specifying the colors used to plot the metrics. Two mappings are valid:

- '<metric-name>' -> color-value (specifies train-metric color only)
- '<metric-name>' -> {'train'/'test' : color-value}

If `None`, default colors are used in the plot.

nrows [Optional[int]] Number of rows of the subplot grid. Metrics are added in row-major order to fill the grid.

ncols [int, optional, default: 1] Number of columns of the subplot grid. Metrics are added in row-major order to fill the grid.

figsize [Optional[Sequence[float, float]]] Specifies the width and height, respectively, of the figure.

Returns

Tuple[LivePlot, Figure, Union[Axes, np.ndarray]] The resulting plotter, matplotlib-figure, and axis (or array of axes)

noggin.utils.save_metrics

noggin.utils.**save_metrics**(*path: Union[str, pathlib.Path], liveplot: Union[noggin.plotter.LivePlot, noggin.logger.LiveLogger, None] = None, *, train_metrics: Dict[str, Dict[str, numpy.ndarray]] = None, test_metrics: Dict[str, Dict[str, numpy.ndarray]] = None*)

Save live-plot metrics to a numpy zipped-archive (.npz). A LivePlot-instance can be supplied, or train/test metrics can be passed explicitly to the function.

Parameters

path: PathLike The file-path used to save the archive. E.g. 'path/to/saved_metrics.npz'

liveplot [Optional[noggin.LivePlot]] The LivePlot instance whose metrics will be saves.

train_metrics [Optional[OrderedDict[str, Dict[str, numpy.ndarray]]]]

'<metric-name>' -> {'batch_data' -> array, 'epoch_domain' -> array, 'epoch_data' -> array}

test_metrics [Optional[OrderedDict[str, Dict[str, numpy.ndarray]]]]]

'<metric-name>' -> {'batch_data' -> array, 'epoch_domain' -> array, 'epoch_data' -> array}

noggin.utils.load_metrics

noggin.utils.**load_metrics**(*path: Union[str, pathlib.Path]*) → Tuple[Dict[str, Dict[str, numpy.ndarray]], Dict[str, Dict[str, numpy.ndarray]]]

Load noggin metrics from a numpy archive.

Parameters

path [PathLike] Path to numpy archive.

Returns

Tuple[OrderedDict[str, Dict[str, numpy.ndarray]], OrderedDict[str, Dict[str, numpy.ndarray]]] (train-metrics, test-metrics)

1.6 Changelog

This is a record of all past noggin releases and what went into them, in reverse chronological order. All previous releases should still be available on pip.

1.6.1 0.10.1 - 2019-07-21

Fixes bug which *last_n_batches* was specified for a *LivePlot* instance, and a metric's test-epochs were being plotted, but its train-epochs were not. In this scenario, *noggin* was not properly tracking the batch-iterations associated with the plotted epochs, and *all* of the test-epochs were being plotted.

1.6.2 0.10.0 - 2019-06-15

Normalizes the interfaces of *LiveLogger* and *LivePlot* so that they can be used as drop-in replacements for each other more seamlessly.

This is an API-breaking update for *LivePlot*, as it renames the methods `plot_train_epoch` and `plot_test_epoch` to `set_train_epoch` and `set_test_epoch`, respectively. As stated above, this is to match the interface of *LiveLogger*.

1.6.3 0.9.1 - 2019-06-06

Adds *plot_logger()*, which provides a convenient means for plotting the data stored by a *LiveLogger*, and to convert it into an instance of *LivePlot*

LivePlot not longer warns about a bad matplotlib backend if `max_fraction_spent_plotting` is set to `0.`

1.6.4 0.9.0 - 2019-05-27

This is the first public release of noggin on pypi.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

F

L

M

P

S

T