# mri Documentation

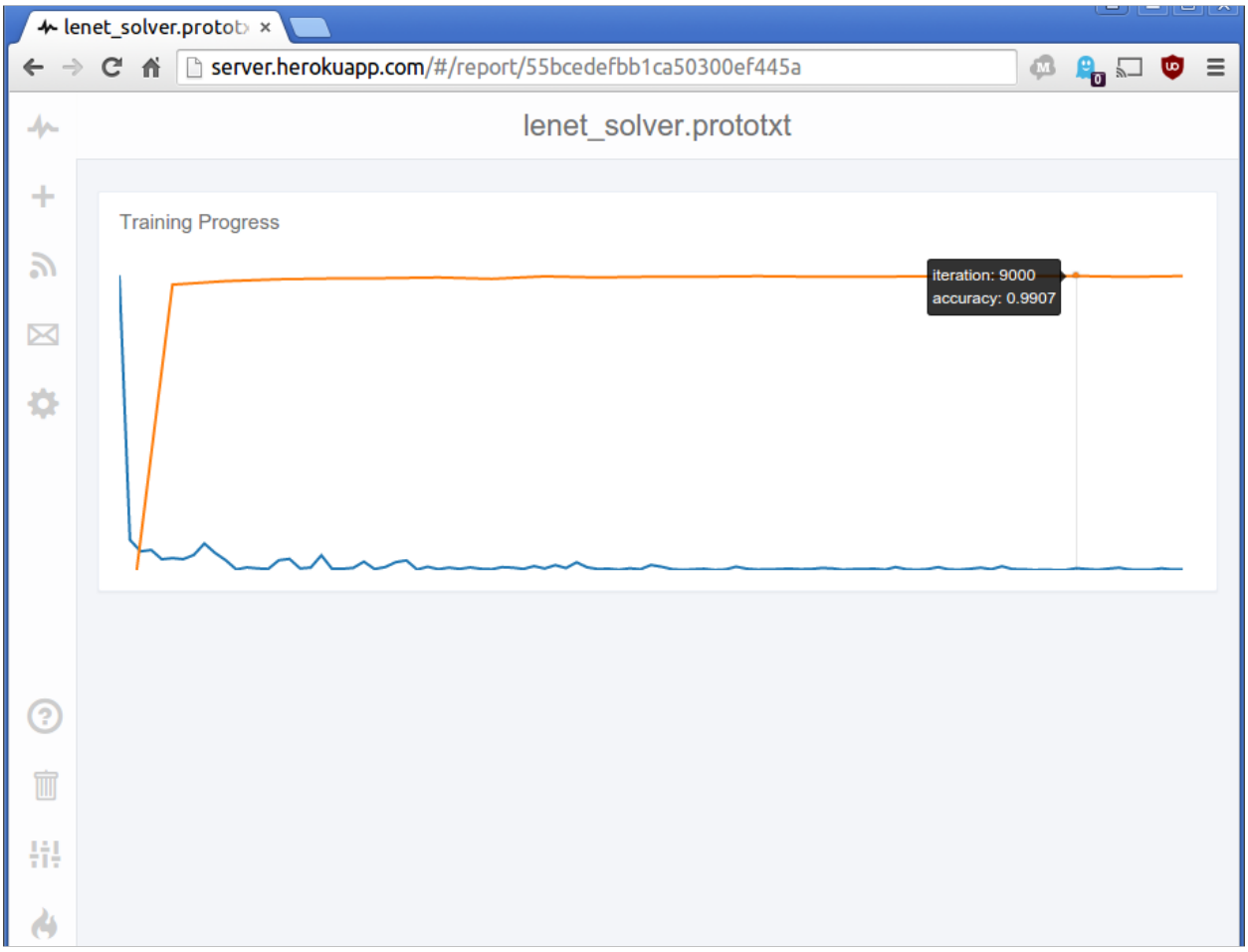
***Release 1.0.0***

**Nate Harada**

September 18, 2015

Contents

Mri is a set of tools and applications to allow you to easily monitor neural network training from anywhere.




Included in Mri is:

- Mri-server: A webserver based on Reportr that allows you to easily monitor training from anywhere.
- Mri-client-python: A Python interface to the server API for use with Theano/Pylearn/Blocks/Keras/etc.
- Mri-app: An interface to Caffe to enable easy hyperparameter testing and monitoring of multiple network architectures.

**Warning:** Mri is a new project and is currently under development. All projects and APIs are subject to change, and while incompatability warnings will be released, you should not expect long term stability. Windows is not yet supported, although as long as you have a proper Python installation you should be okay.

**Tip:** If you are interested in Mri, please feel free to open issues on Github, and to contribute via pull requests.

Contents:

CHAPTER 1

# Getting Started

This is a minimal getting started guide for Mri. For more detailed instructions on each component see its respective page.

## 1.1 Deploying A Server

Getting started with Mri is a two step process. First you will need to get the server component running. While you can run the server on your own machine, the easiest way to run the server is to use a free Heroku account.

By deploying this app to Heroku you can easily start your own private instance of Mri-server with minimal configuration. Heroku is a scalable cloud platform that offers a free-tier sufficient to run your own Mri-server. Simply click the button below to get started. You will need to create an account and provide credit card information.

---

**Tip:** Don't worry – credit card information is only used if you try and scale up your Mri-server to multiple machines. As long as you stick to the default free dyno you will not be charged.

---

Heroku will ask you to set authentication variables and will automatically set the remaining configuration variables. You will also be given the option to choose an app name.

## 1.2 Using Caffe as a Client

Caffe users can use the Mri-app to run and monitor Caffe training, as well as automatically generate tasks for hyperparameter testing.

First you will need to install the Python client:

```
$ pip install -e git+https://github.com/Mri-monitoring/Mri-python-client.git#egg=mri-master
```

Then install the application:

```
$ git clone https://github.com/Mri-monitoring/Mri-app.git
$ cd Mri-app
$ cp config.Template config.txt
```

Make the appropriate edits to the config file (see template):

```
$ cd ..
$ pip install -r requirements.txt
$ python setup.py install
```

Then run the app:

```
$ cd mriapp
$ python MriApp.py --override_solver /path/to/solver.prototxt
```

## 1.3 Using the Python API as a Client

Mri offers a Python API to allow easy integration with Python based deep learning packages.:

```
$ pip install -e git+https://github.com/Mri-monitoring/Mri-python-client.git#egg=mri-master
```

CHAPTER 2

# Mri Server

The Mri-server is built on Reportr and so additional info can be found there, as well as on the server README.

## 2.1 Automatic Deployment

Deploying via Heroku is the easiest way to get started with Mri.

By deploying this app to Heroku you can easily start your own private instance of Mri-server with minimal configuration. Heroku is a scalable cloud platform that offers a free-tier sufficient to run your own Mri-server. Simply click the button below to get started. You will need to create an account and provide credit card information.

**Tip:** Don't worry – credit card information is only used if you try and scale up your Mri-server to multiple machines. As long as you stick to the default free dyno you will not be charged.

Heroku will ask you to set authentication variables and will automatically set the remaining configuration variables. You will also be given the option to choose an app name.

## 2.2 Manual Deployment

To run Mri-server locally you will need to install the Heroku toolbelt.

```
$ git clone https://github.com/Mri-monitoring/Mri-server.git
$ npm install .
```

To run it locally, you should use foreman (configuration can be stored in a env file):

```
$ foreman start
```

To deploy it on Heroku:

```
$ heroku config:set MONGODB_URL=mongodb://...
$ heroku config:set AUTH_USERNAME=...
$ heroku config:set AUTH_PASSWORD=...
$ git push heroku master
```

CHAPTER 3

# Mri Application (Caffe)

The Mri application allows you to monitor Caffe training, as well as easily test hyperparameters and different architectures all at once. See getting started for instructions on how to install the Mri-app

## 3.1 Task List or Standalone

Mri-app offers two modes: a simple prototxt based mode and a more complete task list mode.

In task list, you will specify a list of tasks in the configuration file and Mri-app will run all of those tasks in order. In this mode each task requires some setup – a task requires a model file, a solver file, and a task file. Use the hyperparameter scripts to automatically generate these tasks based on a set of hyperparameters to test:

```
$ python MriApp.py
```

In prototxt mode, simply pass a prototxt file to Mri-app and Mri will run on that single prototxt. While less flexible, this mode requires almost no setup:

```
$ python MriApp.py --override_solver /path/to/solver.prototxt
```

## 3.2 Task Files

A task file specifies additional information to Mri-app for each task to take place. At the moment, training is the only supported task. A task file looks like this:

```
{
    "directives": [
        {
            "type": "train",
            "parameters": {
                "solver": "/path/to/solver/solver.prototxt",
                "resume": "/path/to/snapshots/snapshots_iter_20000.solverstate"
            }
        }
    ],
    "id": "some_id_194183591835",
    "title": "This net title"
}
```

The *resume* field is optional and will pass *–snapshot* to Caffe to allow training to resume from a *.solverstate* file.

## 3.3 Configuration

The Mri configuration file is a plain-text file that contains all the required configuration settings. See the configuration template for an example. Note that not all modules are required, i.e. if you only plan to use the matplotlib-dispatch option, you do not need any other dispatch configuration settings.

## 3.4 Hyperparameter Testing

Included in the *script* directory is a python script that makes it easy to test many hyperparameters in Caffe.

To use, first copy the configuration example to *config* and add your own values. The model and solver templates should simply replace any fields to be replaced by *%{field-name}%*. For example, you may have the following excerpt in your *solver.protobuf* file:

```
# The base learning rate, momentum and the weight decay of the network.
base_lr: %{base_lr}%
momentum: 0.9
weight_decay: %{weight_decay}%
# The learning rate policy
lr_policy: "step"
```

Then, create the hyperparameter file:

```
base_lr: 1e-5, 1e-6, 1e-7
weight_decay: 0.2, 0.1, 0.01
```

The field names can be in either the model or solver, just use unique names

To run script:

```
python generate_tasks random -n 5       # Generates 5 random tasks sampled from the hyperparameter di
python generate_tasks grid              # Generate tasks for all possible hyperparameter values
```

CHAPTER 4

# Mri Python Client

The Python client for Mri offers a simple interface to the Mri-server. See Getting Started for installation instructions.

## 4.1 Tutorial

For a commented version, see *examples/python_bindings*.

Import

```
>>> from mri import MriServer
>>> from mri.event import TrainingEvent
>>> import time
```

Setup server (one time)

```
>>> server = MriServer(SERVER_ADDR, USER, PASS)
>>> task = {'title': 'Example Bindings', 'id': '001'}
>>> dispatch = server.new_dispatch(task)
>>> dispatch.setup_display('iteration', ['iteration', 'loss', 'accuracy'])
```

Train!

```
>>> for i in range(1, 10):
>>>     training_data = {'iteration': i, 'loss': -i, 'accuracy': i/10.0}
>>>     event = TrainingEvent(training_data, 'iteration')
>>>     print('Sending event {} to server'.format(event))
>>>     dispatch.train_event(event)
>>>     time.sleep(1)
```

CHAPTER 5

# Python API Reference

**Warning:** This API reference is built based on docstrings only.

The API reference contains details about the user-facing interfaces of Mri-client-python. We recommend that end users use the MriServer module for most use-cases.

## 5.1 mri package

### 5.1.1 Subpackages

#### mri.dispatch package

##### mri.dispatch.BaseDispatch module

**class** mri.dispatch.BaseDispatch.**BaseDispatch**
Bases: object

Base class to dispatch new actions to whatever backend you want

**setup_display**(*time_axis*, *attributes*)
Create whatever front end we're using

**train_event**(*event*)
Parse a line of output from a training caffe object

**train_finish**()
Call once training is finished

##### mri.dispatch.MatplotlibDispatch module

**Warning:** The Matplotlib dispatch currently only fully supports the Qt4Agg backend, as this is the only backend that we found to reliably work with automatic updating of the plot.

**class** mri.dispatch.MatplotlibDispatch.**MatplotlibDispatch**(*task_params*, *img_folder*)
Bases: *mri.dispatch.BaseDispatch.BaseDispatch*

Display events via Matplotlib backend. This class requires some heavy dependencies, and so trying to run it without Matplotlib and Numpy installed will result in pass-thru behavior

**Parameters**

- **task_params** (*dict*) – Dictionary of the task json specification, including name and ID number
- **img_folder** (*string*) – Folder to save output images to

**setup_display**(*time_axis*, *attributes*, *show_windows=False*)

**train_event**(*event*)
Plot a basic training and testing curve via Matplotlib

**Parameters** **event** (TrainingEvent.TrainingEvent) – Event to add to Matplotlib plot

**train_finish**()
Save our output figure to PNG format, as defined by the save path *img_folder*

### mri.dispatch.MriServerDispatch module

**class** mri.dispatch.MriServerDispatch.**MriServerDispatch**(*task_params*, *address*, *username*, *password*)
Bases: *mri.dispatch.BaseDispatch.BaseDispatch*

Display events via the mri-Server front-end. For this dispatch, we will treat each task as a separate report. There may be multiple visualizations on the server for a report, and there may be multiple directives in a task. These two, however, aren't necessarily one-to-one. Each dispatch is isolated from the server itself to emphasize the stateless nature of the monitoring tool and avoid deadlikes and the like. That means that to perform actions like clearing the server or clearing a specific visualization you'll need to create an MriServer class instead.

**Parameters**

- **task_params** (*dict*) – Dictionary of the task json specification, including title and ID number
- **address** (*string*) – Server address, generally a hosted URL
- **username** (*string*) – Username for the mri-server
- **password** (*string*) – Password for the mri-server

**setup_display**(*time_axis*, *attributes*)
Create a report for this dispatch, usually done at init

**Parameters**

- **time_axis** (*string*) – Name of attribute representing time eg. iterations, epoch, etc
- **attributes** (*list*) – List of strings representing attributes to plot eg. loss, accuracy, learning rate, etc. If time_axis attribute is present it will be ignored.

**Returns** **result** – Result of the report creation request

**Return type** requests.Response

**train_event**(*event*)
Dispatch training events to the mri-server via REST interface

**Parameters**

- **event** (TrainingEvent.TrainingEvent) – Info for this training event
- **event_url** (*string*) – URI to send post events to in mri-server (shouldn't need to change)

**Returns** **result** – Result of the training event request

**Return type** requests.Response

**train_finish**()
Final call for training, can be used to issue alerts/etc. Currently unused.

## mri.event package

### mri.event.BaseEvent module

**class** mri.event.BaseEvent.**BaseEvent**
Bases: object

Base container for new events

### mri.event.TrainingEvent module

**class** mri.event.TrainingEvent.**TrainingEvent**(*attributes*, *time_axis*)
Bases: *mri.event.BaseEvent.BaseEvent*

Container for training events. Training events must have a time-axis variable, and must have at least one other attribute to be valid

**Parameters**

- **time_axis** (*string*) – Defines which attribute represents time (eg iteration number or epoch, etc)
- **attributes** (*dict*) – Dictonary of attributes for this training event. Must include the time axis attribute and at least one other

## mri.utilities package

### mri.utilities.cd module

**class** mri.utilities.cd.**cd**(*newPath*)
Bases: future.types.newobject.newobject

Context manager for changing the current working directory

### mri.utilities.send_request module

mri.utilities.send_request.**send_request**(*address*, *protocol*, *data*, *auth*)
Send an HTTP request

**Parameters**

- **address** (*string*) – Full address for this request
- **protocol** (*string*) – HTTP protocol request to make
- **data** (*dict*) – JSON object of data to pass in the request
- **auth** (*tuple*) – (username, pass) for server

**Returns** **result** – Response from the server, includes response code, encoding, and text

**Return type** requests.Response

### mri.utilities.server_consts module

Contains the constants for the mri-server. These shouldn't be user-changable but should be editable in source just in case

**class** mri.utilities.server_consts.**ServerConsts**
Bases: object

**class API_URL**
Bases: object

**EVENT = '/api/events'**

**REPORT = '/api/reports'**

**REPORT_ID = '/api/report/'**

CHAPTER 6

# Application Reference

**Warning:** This API reference is built based on docstrings only.

This is the API reference for the Mri-app. Mri-app is not designed to be a user-facing API, so use these docs for hacking and at your own risk

## 6.1 mriapp package

### 6.1.1 Subpackages

#### mriapp.process package

##### mriapp.process.BaseProcess module

**class** mriapp.process.BaseProcess.**BaseProcess**(*directive_params*, *config*, *action_handler*)

Bases: object

Base class for processes like Caffe solvers

**Parameters**

- **directive_params** (*dict*) – Dictionary from the JSON directive parameters
- **config** (*dict*) – Dictionary of configuration options
- **action_handler** (*Queue*) – Thread-safe queue that transfers events across threads

**alive**

Returns true if the process is currently running

**Returns** **running** – True if process is still running

**Return type** boolean

**test**()

Currently unused

**train**()

#### mriapp.process.CaffeProcess module

**class** mriapp.process.CaffeProcess.**CaffeProcess**(*directive_params*, *config*, *action_handler*)
Bases: *mriapp.process.BaseProcess.BaseProcess*

Class for running Caffe

**Parameters**

- **directive_params** (*dict*) – Dictionary from the JSON directive parameters
- **config** (*dict*) – Dictionary of configuration options
- **action_handler** (*Queue*) – Thread-safe queue that transfers events across threads

**alive**
Returns true if the process is currently running

**Returns** **running** – True if process is still running

**Return type** boolean

**test**()
Currently unused

**train**()
Start solver, we'll context switch to the caffe_root directory because Caffe has issues not being the center of the universe.

#### mriapp.process.DummyProcess module

**class** mriapp.process.DummyProcess.**DummyProcess**(*directive_params*, *config*, *action_handler*)
Bases: object

Dummy process for unit testing

**Parameters**

- **directive_params** (*dict*) – Dictionary from the JSON directive parameters
- **config** (*dict*) – Dictionary of configuration options
- **action_handler** (*Queue*) – Thread-safe queue that transfers events across threads

**alive**
Live for a certain number of iterations

**test**()
Unused

**train**()
Unused

### mriapp.retrieve package

#### mriapp.retrieve.BaseRetrieve module

**class** mriapp.retrieve.BaseRetrieve.**BaseRetrieve**
Bases: object

Base class to retrieve new solver jobs

**retrieve_file**(*location*)
Fetch a file and return a json/dict representation

**retrieve_task**()
Retrieve the next task and return a json/dict representation. A task can contain multiple directives and files to fetch. This is a generator

#### mriapp.retrieve.LocalRetrieve module

**class** mriapp.retrieve.LocalRetrieve.**LocalRetrieve**(*task_record*)
Bases: object

Retrieve new solver jobs from local filesystem

**Parameters** **task_record** (*string*) – File on the local system containing a list of folders with tasks

**__del__**()
Make sure we close our file on object deletion

**retrieve_file**(*location*)
Basically the identity function - we don't need to get any network data for local files, so we'll just pass back the file handle

**Returns** **location** – Local location of a file.

**Return type** string

**retrieve_task**()
Retrieve the next task and return a json/dict representation.

A task can contain a number of directives, but will generally contain the training task for Caffe, including the locations of model/solvers. This function is a generator and will allow iteration through the entire set of possible tasks.

**Yields** **task** (*dict*) – A task located on the local drive

### mriapp.utilities package

#### mriapp.utilities.line_parser module

mriapp.utilities.line_parser.**parse_caffe_train_line**(*line*)
Parse a line from Caffe's training output

**Parameters** **line** (*string*) – Line to parse from training

**Returns** **training_event** – A (possibly incomplete) dict with the parsed information

**Return type** dictionary

#### mriapp.utilities.verify_config module

mriapp.utilities.verify_config.**verify_config**(*filename*)
Verify a configuration file prior to loading the full program. This will hopefully prevent unfortunate config errors where some events have already occurred, such as accessing the server

**Parameters** **filename** (*string*) – Config file to test

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

## m

## Symbols



## A



## B



## C



## D



## E



## L



## M



## P



## R



## S

## T



## V