Neronet User Guide

Contents

- Contents
- Introduction
- Installation
- Specifying experiments
- **Command Line Interface:** – Using Neronet CLI
  - Specifying computing nodes in Neronet CLI
  - Specifying nodes and node groups by manually updating nodes.yaml
  - Deleting computing nodes in Neronet CLI
  - Specifying and configuring experiments in Neronet CLI
  - Deleting defined experiments from Neronet
  - Submitting experiments to computing Nodes
  - Fetching data about submitted experiments
  - Terminating a currently running experiment
  - Status report
  - Cleaning Neronet's databases
- **GUI:** – Starting Neronet GUI
  - Specifying and configuring experiments in Neronet GUI
  - Deleting defined experiments from Neronet
  - Submitting experiments and batches of experiments to computing Nodes
  - Specifying nodes in Neronet GUI
  - Monitoring log output in Neronet GUI
  - Status report in Neronet GUI
- **Advanced Functionalities:** – Output processing
  - Auto termination
  - Plotting output
- Example use case:

Introduction

Neronet is a python-based, framework agnostic tool for computational researchers that is made to enable easy

- specification of computational experiments and inheritance of parameter values
- batch submission of experiment jobs to computing nodes
- monitoring of logs and parameter values for ongoing experiments
- access to experiment information during and after the run
- configurable notifications on experiment state and progress
- configurable criteria for experiment autotermination
- logging of experiment history

Neronet can be used either via command-line interface or via GUI.

Installation

All components of the neronet application, including both the parts run in nodes and the parts run in the researcher's local machine are implemented using python 2.7, so before installation please make sure that the correct version of python is installed on your local machine and on the computing nodes you intend to be use. Using other python versions may cause complications and is therefore not recommended. Then download neronet folder and proceed by setting up your initial node setup and setting up your preferences.

Pip installation

Start by running the command below on the command line of your local machine. Note: python 2.7 is required. :: sudo pip install neronet

The command will load all components related to neronet and install them to the system. It will also create the folder `~/.neronet` that contain your preferences and cluster setup. Proceed by opening the folder and then defining your initial cluster setup and preferences in the respective files.

If you want to use GUI, it is necessary to also install pyQT. See section GUI/installation for further instructions.

Using Neronet CLI

To start your Neronet CLI application, run nerocli on your local machine's command line.

Example: :: nerocli –status

Specifying computing nodes

You can specify nodes either via command line or by manually updating the `nodes.yaml` file. See the next section for more information on the format when updating the nodes.yaml file manually.

To add computing nodes via command line use the following format: :: Usage: nerocli –addnode ID SSH_ADDRESS Example: nerocli –addnode triton triton.cs.hut.fi

- `ID` is a user defined id of the node.
- `SSH_ADDRESS` is the ssh address of the node.

The information given via CLI is then automatically updated to `nodes.yaml`. If you want to save other information about a specific node besides the node's address and id, you must manually write them to the nodes.yaml file.

Specifying nodes and node groups by manually updating nodes.yaml

Although nodes can be easily specified via neronet CLI or GUI, manually updating the config files gives the user some additional options and is sometimes more versatile.

Open `~/.neronet/nodes.yaml` using your favorite text editor and fill in the following information.

The format of `nodes.yaml` is as follows. From here on out we will explain the formats of important files by first showing an example file and then explaining the important points.

Example:

```
nodes:
    triton:
        ssh_address: triton.aalto.fi
        hard_disk_space: 1000GB
    gpu1:
        ssh_address: gpu1
    gpu2:
        ssh_address: gpu2
groups:
    gpu: [gpu1, gpu2]
default_node: triton
```

The specification of a node must start with the user-specified node-id on a separate line. The following lines containing the node's information must be indented and contain at least the ssh_address: (f.ex `triton.aalto.fi`).

Additionally, it is possible to specify optional information on the node such as the hard disk space. However, these are purely for the user and are not used internally.

It is also possible to group some of your nodes under a single virtual cluster name using the following format: `GROUP_ID: [NODE_ID, NODE_ID, ...]` (f.ex `gpu: [gpu1, gpu2]` in the example above). Then later on you can submit your experiments to that virtual cluster name and let neronet automatically divide the work between the actual nodes.

Deleting computing nodes

If you want to remove all information regarding a specific computing node from neronet's database, type the following command:

Remove a computing node: :: Usage: nerocli –delnode ID Example: nerocli –delnode triton

Specifying and configuring experiments

Neronet supports experiments written using any programming language or framework as long as the experiments are runnable with a command of the format `RUN_COMMAND-PREFIX CODE_FILE PARAMETERS`, f.ex. `python2.7 main.py 1 2 3 4 file.txt`

Start by writing your experiment code and save all experiments you deem somehow related to a single folder. Then include a YAML configuration file in your folder and name it `config.yaml`. It is also possible to create the YAML configuration file template with the following command:

Create a config.yaml template: :: Usage: nerocli –template EXP_ID RUN_COMMAND-PREFIX CODE_FILE PARAMETERS Example: nerocli –template theanotest python theanotest.py N feats training_steps

In the configuration file you are to specify all the different experiments you want to run using the following format. Please read this section carefully for it contains plenty of important information.

```yaml
collection: lang_exp
run_command_prefix: python3
main_code_file: main.py
outputs: stdout
+lang_exp1:
    parameters:
        hyperparamx: [1,2,34,20]
        hyperparamy: 2
        data_file: data/1.txt
        hyperparamz: 2
    parameter_format: '{hyperparamx} {hyperparamy} {data_file} {hyperparamz}'
    conditions:
```

```
        error_rate_over_50:
            variablename: error_rate
            killvalue: 50
            comparator: gt
            when: time 6000
            action: kill
        error_rate_over_35:
            variablename: error_rate
            killvalue: 35
            comparator: geq
            when: time 6000
            action: warn

    +lang_exp3:
        parameters:
            hyperparamz: 2

+lang_exp2:
    run_command_prefix: python2
    main_code_file: main2.py
    parameters:
        hyperparamx: kh
        hyperparamy: nyt
        data_file: data/2.txt
        hyperparamz: 400
    parameter_format: '{hyperparamx} {hyperparamy} {data_file} {hyperparamz}'
```

- The information on the `config.yaml` file is divided to blocks that have the same indentation.
- Each experiment specification must begin with a row containing the experiment id (f.ex in the example above three experiments are specified: `lang_exp1`, `lang_exp2` and `lang_exp3`) and be followed by a block containing all the experiment's attributes. Do not use the reserved words, list of which can be found at the end of this section. The experiment ids must be unique within the same config file.
- Experiment ids must begin with `+` character, otherwise neronet won't recognize the new experiment.
- Each different experiment specification must have the following attributes.
  - `main_code_file`: The path to the code file that is to be run when executing the experiment.
  - `run_command_prefix`: The prefix of the run command f.ex `python2`.

  - Your experiments should be runnable with a command of the form `RUN_COMMAND_PREFIX MAIN_CODE_FILE PARAMETER_STRING` f.ex in the example above `lang_exp2` must be runnable with the command `python2 main2.py kh nyt data/2.txt 400`

- If multiple experiments have the same attribute values, it is not necessary to re-write every attribute for every experiment. The experiments defined in inner blocks automatically inherit all the attribute values specified in outer blocks. For example in the example above `lang_exp1` and `lang_exp2` inherit the `run_command_prefix`, `main_code_file` and `outputs` values from the outmost block and `lang_exp3` inherits all the parameter values from `lang_exp1`. If you don't want to inherit a specific value, just specify it again in the inner block and it is automatically overwritten. For example in the example above `lang_exp3` uses different `hyperparamz` and `parameter_format` values than its parent `lang_exp1`.

- If you place multiple parameter values within brackets and separated by a comma (like in the example above `lang_exp1` – `hyperparamx: [1,2,34,20]`)Neronet will automatically generate different experiments for each value specified within brackets. (f.ex `lang_exp1` would be run with the parameters `1 2 data/1.txt 2`, `2 2 data/1.txt 2`, `34 2 data/1.txt 2` and `20 2 data/1.txt 2`)

After your experiment folder contains the config file of the correct format and all the code and parameter files, you can then submit the folder to your Neronet application with the following command.

Submit the experiment folder to neronet locally: :: Usage: nerocli –addexp FOLDER Example: nerocli –addexp ~/experiments/lang_exp

Reserved Words: :: experiment_id run_command_prefix main_code_file parameters parameter_format outputs output_line_processor output_file_processor plot collection required_files conditions custom_msg path warning variablename killvalue comparator when action

Deleting defined experiments

To delete a specified experiment from your Neronet application's database you can use the following command.

Example: :: nerocli –delexp EXPERIMENT_ID

Using the command above doesn't delete the experiment folder or any files within it. It only removes the experiment's information from Neronet's database. It also doesn't affect the children of the experiment.

Submitting experiments to computing clusters

The following command will submit an experiment to a specified cluster.

Submit an experiment to a computing node: :: Usage: nerocli –submit EXPERIMENT_ID CLUSTER_ID Example: nerocli –submit lang_exp triton

`EXPERIMENT_ID` is the name of the experiment you are about to submit.

`CLUSTER_ID` can be any node id or node group id specified in the `nodes.yaml` file or via CLI or GUI. If you have specified a default node in preferences.yaml (see *Installation*), you can leave `CLUSTER_ID` blank to automatically submit your experiments to the specified default node.

Fetching data about submitted experiments

To see the current state of the submitted experiments it is necessary to first fetch the data from clusters. In Neronet CLI this is done by typing the following command:: nerocli –fetch

After that you can see the current state of your experiments by typing::
nerocli –status

Terminating a currently running experiment

If you need to manually terminate an experiment that is currently running in a node, type the following command

Terminate an experiment: :: nerocli –terminate EXPERIMENT_ID

Status report

The status command gives status information regarding configurations and any specified nodes and experiments.

Example: :: Usage: nerocli –status [ARGS]

`ARGS` can refer to experiment or node ids.

Overall status: :: nerocli –status

The command above will print the overall status information. That is, printing the number of experiments with each of the different experiment states, the list of defined clusters and their current states and finally the list of experiments and their current states.

Experiment status: :: nerocli –status lang_exp3

Node status: :: Usage: nerocli –status CLUSTER_ID Example nerocli –status triton

Cleaning Neronet's databases

If you want to remove all data currently existing in neronet's databases, that is all specified experiments, their results and information on computing nodes, type the following command:: nerocli –clean

GUI

Installation

As pyqt is not included with pip, it is required to be installed from package manager. You can download QT for python with `apt-get install python-qt4` Make sure you have configured path correctly. You can check you current path with `import sys print sys.path`

Gui is included in pip install. You can open gui with `nerogui`

Specify nodes

Specify nodes by writing nodes short name to node name field. Write nodes address and select its type from dropdown menu and hit add node to add it.

Specify experiments

Specify experiments by pressing "Add experiment" A dialog should open. Navigate to the folder where your experiment is configured (the one containing config.yaml) and hit open. If specifying the experiment was successful, the table of experiments will be updated.

You can also drag and drop multiple folders to the NeroGUI window to add them.

Submit experiments

You can submit experiments by selecting experiment and folder and hitting submit button.

Submit batches of experiments

You can select multiple experiments by holding ctrl and pressing every experiment you want to send. After selecting the experiments, choose cluster and hit submit.

Experiment status report

Selecting experiment will update log view with the information accosiated with experiment.

Node status report

Selecting experiment will update log view with the information accosiated with cluster.

Accessing status folder

You can get into the folder where experiment is defined by double clicking experiment.

Fetch data from nodes

Hit refresh to update status(es) of the experiment(s).

Deleting experiments

You can delete experiments by selecting experiment(s) and pressing delete key

Manipulating experiments table

Right clicking will open menu where you can select parameters which you want to view. By pressing headers you can sort your experiments.

Plotting experiments

You can plot some function of your experiment by pressing the experiment and selecting plots in plot-tab.

Creating a new experiment

Navigate to experiment tab and type command you wish to run your experiment f.ex "python test.py x y". Program will create you a template config.yaml.

Duplicate experiment

Select experiment and press duplicate buton in experiment-tab.

Advanced functionalities

Output processing

To allow Neronet to undestand your experiments output you must define output processing functions for neronet. These are defined in the `config.yaml` file for the experiment. There are two different types of output processing functions you can define according to your output file format.

The first one is output line processor which receives a line of output file as input and outputs the line interpeted as a dict.

It is defined in config.yaml as such:

config.yaml

```
output_line_processor:
    output_file_name: module_name function_name other_arguments
```

- For each output file you want to be understood this way you should define a seperate output line processor
- The other arguments are defined as a string separated with spaces
- `output_file_name` must be a output file defined in the experiment outputs
- `module_name` refers to the module where you specify the output reading functions
- `function_name` refers to the name of the output line processing function
- `other_arguments` are other arguments you would want Neronet to pass to your function. Other arguments can also contain strings of characters with spaces by using quotes: "I am a string passed as an argument".

The other one is output file processor which recieves the filename of the output file as input and outputs the file interpeted as a dict. Its definition in the `config.yaml` doesn't differ from the output line processors, but `output_line_processor` is replaced with `output_file_processor`.

These functions should follow the following format when defined in the module.

module

```
def some_output_line_reader(output_line, *optional_args):
    #Process optional arguments
    ...
    #Process the line data
    ...
    #Map the line data into dictionary so that neroman can understand it
    ...
    return output_dict

def some_output_file_reader(output_file, *optional_args):
    #Process optional arguments
    ...
    #Read and process the output file
    ...
    #Map the line data into dictionary so that neroman can understand it
    ...
    return output_dict
```

- The users are free to name the functions and parameters in any way they see fit.
- `output_line` is a string containing a line of the output.
- `output_file` is a file object returned by open.
- `optional_args` are other arguments the user wants the function to receive. You can also name give names to the other optional arguments, but then you must take special care that output processors receive the correct amount of arguments.

The user should take special care that the functions are valid python and can actually used to read the user input. If the user functions fail at any point Neronet cannot use the functions and will give warnings to the user.

Auto termination

- Additionally, if you want neronet to autoterminate an experiment or give you a warning under certain circumstances you can use the conditions-attribute. Neronet supports warnings and autotermination based on a variable exceeding, falling below or reaching a predetermined value. The conditions-attribute must be followed by a block containing the specifications of the conditions and actions to perform

    - Start by giving a unique ID to your condition. f.ex in the example above 'lang_exp1' has two conditions set: 'error_rate_over_50' and 'error_rate_over_35'. Do not use the reserved words, list of which can be found at the end of this section. Then specify the following attributes on the following block.
    - variablename: This is the name of the variable you want to monitor
    - killvalue: This is the value to which you want neronet to compare the monitored variable
    - comparator: Either 'gt' (greater that), 'lt' (less than), 'eq' (equal to), 'geq' (greater than or equal to) or 'leq' (less than or equal to). Use 'gt' if you want a warning when the value of the variable monitored exceeds killvalue, 'lt' if you want a warning when the variable falls below killvalue and 'eq' if you want a warning when the variable reaches killvalue.
    - when: The value of this attribute can be either 'immediately' or 'time MINUTES' where MINUTES is the time interval in minutes after which the warning condition is checked and action performed.
    - action: Specifies what you want neronet to do when the warning condition is fulfilled. The value of this attribute is either 'kill' (if you want the experiment to be terminated when the warning condition is fulfilled) or 'warn' (if you want to see a warning message when the condition is fullfilled)
    - The log output from the experiment code must contain rows of the format: 'VARIABLENAME VALUE'. So that neronet is able to follow the variable values. F.ex. in the example above the log output of lang_exp1 must contain rows of the form 'error_rate 24.3334', 'error_rate 49', 'error_rate 67.01', etc... The row must not contain anything else.

Plotting output

To allow Neronet to plot your output you must define plotting functions for your outputs. These are defined in the `config.yaml` for the experiment.

It is defined in config.yaml as such:

config.yaml

```
plot:
    plot_name: module_name function_name output_filenames other_arguments
```

- Each plot you want to be generated should be defined on their own lines under the `plot` line.
- `plot_name` should be the name of the plot generated.
- `module_name` refers to the module where you specify the plotting function
- `function_name` refers to the name of the plotting functions
- `output_filenames` specifies the output file used for plotting. The output file should have a output processing function defined so that Neronet can give them to the plotting function. Either line or file processer works.
- `other_arguments` are other arguments you would want Neronet to pass to your function. To allow your plotting function to use your experiment output you can use variables defined in the output dicts. Each argument that is contained in the output is replaced with a tuple containing the name of the variable as the first element and the data of the output pertaining to that variable as the second element. Other arguments can also contain strings of characters with spaces by using quotes: "I am a string passed as an argument".

These functions should follow the following format when defined in the module.

module

```
def plotting_function(plot_name, feedback, save_image, *optional_args):
    #Process optional arguments
    ...
    #Process the line data
    ...
    #Map the line data into dictionary so that neroman can understand it
    ...
    return feedback
```

- The users are free to name the functions and parameters in any way they see fit.
- `plot_name` is a string containing the name of the plot to be saved.

- `feedback` is a special variable that can be used for combining plots. It doesn't need to be used and is set to None by Neronet when normally plotting.
- `save_image` is also a special variable that can be used for combining plots. It doesn't need to be used and is set to None by Neronet when normally plotting.
- `optional_args` are other arguments the user wants the function to receive. You can also name give names to the other optional arguments, but then you must take special care that output processors receive the correct amount of arguments.

The user should take special care that the functions are valid python and can actually used to read the user input. If the user functions fail at any point Neronet cannot use the functions and will give warnings to the user.

Example use case

Assume we have folder `~/mytheanotest` which contains an experiment named `script.py` and we want to submit it to `kosh.aalto.fi` to be run there. We proceed as follows:

Define a node where the experiment is to be run:: nerocli –addnode kosh kosh.aalto.fi

Neronet requires some information about each experiment, which is why we create the file `~/mytheanotest/config.yaml` with the following content:

```
collection: None
run_command_prefix: 'python'
main_code_file: 'script.py'
outputs: 'results'
parameters_format: '{N} {feats} {training_steps}'
theanotest:
    parameters:
        N: 400
        feats: 784
        training_steps: 10000
```

Now we let Neronet know about the experiment by registering it:: nerocli –addexp ~/mytheanotest

Finally, we submit the experiment to be run in the node:: nerocli –submit theanotest kosh

Before submitting of course you need to make sure that all the dependencies of the experiment file are available in the node.

While the experiment is running we can check its status by first fetching information:: nerocli –fetch

And then checking the status with:: nerocli –status theanotest

Eventually the experiment state will show as `finished` and the results will be synced to the `~/.neronet/results/theanotest` folder.