Neronet User Guide

Contents

Introduction

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

- batch submission of experiment jobs to computing clusters
- management of experiment queues
- 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 clusters 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 clusters 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 cluster setup and setting up your preferences.

1. Pip Installation

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

Example: :: 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.

2. Setting up the Initial cluster setup:

Open neronet/clusters.yaml file using your favorite text editor and fill in the following information.

The format of clusters.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: :: clusters: triton: ssh_address: triton.aalto.fi type: slurm hard_disk_space: 1000GB gpu1: ssh_address: gpu1 type: unmanaged gpu2: ssh_address: gpu2 type: unmanaged groups: gpu: [gpu1, gpu2]

The specification of a cluster must start with the user-specified cluster-id on a separate line. The following lines containing the cluster's information must be indented and contain at least the following attributes: ssh_address: (f.ex triton.aalto.fi) and type: (either 'unmanaged' or 'slurm'). If the cluster uses Simple Linux Utility for Resource Management (SLURM), its type is 'slurm', otherwise use 'unmanaged'.

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

It is also possible to group some of your clusters or unmanaged 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.

3. Setting up personal Information and preferences:

Open the file neronet/preferences.yaml and fill in your name, email and default cluster using the following format.

Example: :: name: John Doe email: john.doe@gmail.com default_cluster: triton

If you followed the instructions, your neronet application should be ready to run now. Proceed by starting neronet. The program will notify you if the installation failed for one reason or another.

Command Line Interface

Using Neronet CLI

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

Example: :: nerocli –status

Specifying and Configuring Experiments in Neronet CLI

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'. 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 a ton of important information.

Example: :: collection: lang_exp run_command_prefix: python3 main_code_file: main.py logoutput: 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.
- **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'
  - logoutput: The location to which the log output of the experiment is to be written. Can be either stdout or a file path.
  - parameters: This attribute is followed by a block containing all the unique parameters of this specific experiment. Parameter names can be arbitrary.
  - parameter_format: Specifies the order in which the parameters are given to the experiment code file in the form of a string. Write the attribute value within single quotes. Parameter names written within braces will be replaced by their values defined in the *parameters* section. F.ex in the example above lang_exp2 –parameter_format defines a parameter string 'kh nyt data/2.txt 400'. You can escape braces and special characters with backslashes in case your parameter names contain braces.
  - 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'**
- **Additionally, if you want neronet to autoterminate an experiment or give you a warning u**
  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), 'warn' (if you only want to see a warning message the next time you check the experiment status) or email (if you want to receive a warning email when the warning condition is fulfilled)
  - 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.

- 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 logoutput 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.

Example: :: Usage: nerocli –experiment FOLDER Example: nerocli –experiment ~/experiments/lang_exp

Reserved Words: :: ID run_command_prefix main_code_file logoutput parameters parameter_format warning: variablename killvalue comparator when action

Deleting Defined Experiments from Neronet

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

Example: :: nerocli –delete EXPERIMENT_ID

EXPERIMENT_ID is the 'ID' attribute defined on the topmost row of the experiment folder's config.yaml. Alternatively, if you only want to delete a certain experiment within a folder, you can use the format 'ID/experiment_Id' (see *specifying experiments* to find out what these attributes are). Commands of the format 'delete ID/experiment_Id' don't affect the experiment's children or parents.

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 and Batches of Experiments to Computing Clusters

The following command will submit a batch of experiments to a specified cluster.

Example: :: Usage: nerocli –submit CLUSTER_ID EXPERIMENT_ID Example: nerocli –submit triton lang_exp

EXPERIMENT_ID is the 'ID' attribute defined on the topmost row of the experiment folder's config.yaml. Alternatively, if you only want to submit a certain experiment within a folder, you can use the format 'ID/experiment_Id' (see *specifying experiments* to find out what these attributes are) Using 'all' as EXPERIMENT_ID will submit all specified but not submitted experiments.

CLUSTER_ID can be any cluster id or cluster group id specified in the clusters.yaml file or via CLI. Using 'any' as CLUSTER_ID will divide the work (if it can be divided) and submit it to all free clusters. If you have specified a default cluster in preferences.yaml (see *Installation*), you can leave CLUSTER_ID blank to automatically submit your experiments to the specified default cluster. F.ex 'submit lang_exp'.

Tasks can be submitted also by logical arguments: :: Usage: nerocli –submit CLUSTER_ID ARGUMENT

> #Specify an experiment and submit it instantly Example: nerocli –submit triton ~/experiments/lang_exp x

#Submit all experiments that were modified since 2015-11-23 Example: nerocli –submit triton tmod>2015-11-23

#Submit all that have a specified parameter Example: nerocli –submit triton params=*data/1.txt*

#Submit all defined but not submitted experiments Example: nerocli –submit any all

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

Specifying Clusters in Neronet CLI

You can specify clusters either via command line or by manually updating the clusters.yaml file. See the section *Installation* for more information on the format when updating the clusters.yaml file manually.

To add clusters via command line use the following format: :: Usage: nerocli –cluster ID SSH_ADDRESS TYPE Example: nerocli –cluster triton triton.cs.hut.fi slurm

ID is a user defined id of the cluster, SSH_ADDRESS is the ssh address of the cluster, TYPE is either slurm or unmanaged

The information given via CLI is then automatically updated to clusters.yaml. If you want to save other information on a specific cluster besides the cluster's address, name and type, you must manually write them to the clusters.yaml file.

Status report

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

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

ARGS can refer to experiment or cluster IDs, or be collection specifiers.

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_exp/lang_exp3

The experiment status report contains:

- The experiment's parameters
- The experiment's last modification date
- The experiment's current state and the times when the state has changed
- The final output, if the experiment is finished

The experiment state is either 'defined' (specified but not submitted to any cluster), 'submitted CLUSTER_ID' (submitted to a cluster but not yet running), 'running CLUSTER_ID', 'finished CLUSTER_ID' or 'terminated CLUSTER_ID'. CLUSTER_ID will be replaced with the correct cluster's id.

Collection status: :: #All experiments that were modified since 2015-11-23 Example: nerocli –status tmod>2015-11-23

> #All experiments that have a specified parameter Example: nerocli –status params=*data/1.txt*
>
> #All experiments that have the current state of 'defined' Example: nerocli –status defined

The collection status will contain a list of experiments in that collection and their current states.

All cluster statuses: :: nerocli –status clusters

Prints a list of all clusters and their current states. A cluster's current state is the number of experiments running in that cluster.

Single cluster status: :: Usage: nerocli –status CLUSTER_ID Example nerocli –status triton

Prints:

- The number of experiments submitted to and running in the given cluster
- The list of experiments submitted to and running in the given cluster
- The times when the experiments were submitted and started running

Example experiment

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 cluster where the experiment is to be run: `nerocli --cluster kosh kosh.aalto.fi unmanaged`

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 --experiment ~/mytheanotest`

Finally, we submit the experiment to be run in the cluster: `nerocli --submit kosh theanotest`

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

While the experiment is running, we can check its status with: `nerocli --status`

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

GUI

Installation

Specify clusters

Specify experiments

Submit experiments

Submit batches of experiments

Monitoring log output

Experiment status report

Collection status report

Neronet status report