Nauta User Guide

Enterprise Edition 1.1

Document Revision 1.1: October 2019

Document Revision History

Document Revision Number	Date	Comments
1.1	October 2019	Initial Release of Enterprise Edition, 1.1 Release.

Terms and Conditions

Contents

Tables

Figures

Nauta Introduction

Product Overview

The Nauta software provides a multi-user, distributed computing environment for running deep learning model training experiments. Results of experiments, can be viewed and monitored using a command line interface, web UI and/or TensorBoard*.

You can use existing data sets, use your own data, or downloaded data from online sources, and create public or private folders to make collaboration among teams easier. Nauta runs using the industry leading Kubernetes* and Docker* platform for scalability and ease of management.

Templates are available (and customizable) on the platform to take the complexities out of creating and running single and multi-node deep learning training experiments without all the systems overhead and scripting needed with standard container environments. To test your model, Nauta also supports both batch and streaming inference, all in a single platform.

The Nauta client software has been validated on the following operating systems and versions:

- Ubuntu* (16.04, 18.04)
- RedHat* 7.6
- macOS* High Sierra (10.13)

Nauta User Guide Purpose

This guide describes how to use the Nauta and discusses the following topics main topics:

Nauta Basic Concepts

Within this user guide, the following concepts and terms are relevant to using this software: user, administrator, resources, data, experiments, and predictions, all of which are described below.

This section discusses the following main topics:

User

In this context, the User is a Data Scientist who performs deep learning experiments to train models that will, after training and testing, be deployed in production. Using Nauta, the user can define and schedule containerized deep learning experiments using Kubernetes* on single or multiple worker nodes. The user also checks the status and results of those experiments to further adjust and runs additional experiments, or prepares the trained model for deployment.

Administrator

In this context, the *Administrator* or *Admin* creates and monitors users and resources. An important key concept to remember is that Admins *cannot* be users (data scientists); and, users (data scientists) cannot be Admins. Admins *are not* permitted to perform any of the user experiments or related tasks. An admin who wants to run experiments must create a separate user account for that purpose.

Resources

In this context, *Resources* are the system compute and memory resources the user will assign to a model training experiment. The user can specify the number of processing nodes and the amount of memory in the system that will be reserved for a given experiment or job. The job will not be allowed to exceed the specified memory limit. In a multi-user environment, care should be taken to not dedicate too many resources to a given job, because other applications and services may be impacted.

Data

In this context, *Data* is the set of observations used to run experiments to train, test and validate your model.

Experiments

Performing deep learning experimentation is what the Nauta application was developed for, and each experiment is executed by a deep learning script. You can run a single experiment, or run multiple experiments in parallel using the same script, or run different multiple experiments with different scripts (see the experiment Command, page 84 for more details). The script needs to be tailored to process whatever data you are using to train your model.

Predictions

After experiments have been run and the model has been trained, you can pass in new (unlabeled) data exemplars, to obtain predicted labels and other details returned. This process is called inference. In general, generating predictions involves pre-processing the new input data, running it through the model, and then collecting the results from the last layer of the network.

The Nauta software supports both batch and streaming inference. Batch inference involves processing a set of prepared input data to a referenced trained model and writing the inference results to a folder. Streaming inference is where the user deploys the model on the system and streaming inference instance processes singular data as it is received.

Client Installation and Configuration

The section provides instructions for installing and configuring Nauta to run on your client system. For instructions to install and configure Nauta to run on the host server, refer to the *Nauta Installation, Configuration, and Administration Guide*.

This section discusses the following main topics:

Supported Operating Systems

This release of the Nauta client software has been validated on the following operating systems and versions.

- Ubuntu (16.04, 18.04)
- Red Hat 7.6
- macOS High Sierra (10.13)

Required Software Packages

The following software *must* be installed on the client system *before* installing Nauta client:

- kubectl version 1.15 or later: Install Kubectl
- git version 1.8.3.1 or later.

Installation

Complete the following steps to install the Nauta client software package:

1. Download and install the *Required Software Package* above, preferably in the order given.
2. There *is no* installation utility. Unpack this package and place the unpacked files in any preferred location. Take note of the path.
3. Set `KUBECONFIG` environment variable to the Kubernetes configuration file provided by your Nauta administrator. The `<PATH>` is located wherever your config file is stored.
   - For **MacOS/Ubuntu**, enter:

     ```
     export KUBECONFIG=<PATH>/<USERNAME>.config
     ```

4. **Optional**: Add the package `nctl` path to your terminal *PATH*. `NCTL_HOME` should be the path to the nctl application folder:

   For **MacOS/Ubuntu**, enter:

   ```
   export PATH=$PATH:NCTL_HOME
   ```

Setting Variables Permanently

Should you want to permanently set the variables, you can add the variables to your:

- .bashrc
- .bash_profile

Getting Started

This section of the guide provides brief examples for performing some of the most essential and valuable tasks supported by Nauta.

Note: Several commands and training scripts in this section require access to the internet to download data, scripts, and so on.

The section discusses the following topics:

Verifying Installation

Check that the required software packages are available in the terminal by `PATH` and verify that the correct version is used (see Confirm Installation below).

Proxy Environment Variables

If you are behind a proxy, remember to set your:

- `HTTP_PROXY`, `HTTPS_PROXY` and `NO_PROXY` environment variables
- `http_proxy`, `https_proxy` and `no_proxy` environment variables

Confirm Installation

Execute the following command to verify your installation has completed:

```
nctl verify
```

Confirmation Message

If any installation issues are found, the command returns information about the cause: which application should be installed and in which version. This command also checks if the CLI can connect to Nauta; and, if port forwarding to Nauta is working correctly. If no issues are found, a message indicates that the checks were successful. The following examples are the results of this command:

```
This OS is supported.
kubectl verified successfully.
helm client verified successfully.
git verified successfully.
helm server verified successfully.
kubectl server verified successfully.
packs resources' correctness verified successfully.
```

Overview of nctl Commands

Each nctl command has at least three options:

1. `-v, --verbose` - Set verbosity level:
   - `-v` for INFO - Basic logs on INFO/EXCEPTION/ERROR levels are displayed.
   - `-vv` for DEBUG - Detailed logs on INFO/DEBUG/EXCEPTION/ERROR levels are displayed.
2. `-h, --help` - The application displays the usage and options available for a specific command or subcommand.
3. `-f, --force` - Force command execution by ignoring (most) confirmation prompts.

Accessing Help

To access help for any command, use the `--help` or `-h` parameters. The following command provides a list and brief description of all nctl commands.

```
nctl --help
```

Help Command Output

The results are shown below.

```
Usage: nctl COMMAND [options] [args]...

  Nauta Client

Displays additional help information when the -h or --help COMMAND is
  used.

Options:
  -h, --help  Displays help messaging information.

Commands:
  config, cfg      Set limits and requested resources in templates.
  experiment, exp  Start, stop, or manage training jobs.
  launch, l        Launch the web user-interface or TensorBoard. Runs as a
                   process in the system console until the user stops the
                   process. To run in the background, add '&' at the end of
                   the line.
  model, mo        Manage the processing, conversion, and packaging of models.
  mount, m         Displays a command that can be used to mount a client's
                   folder on their local machine.
  predict, p       Start, stop, and manage prediction jobs and instances.
  template, tmp    Manage experiment templates used by the system.
  user, u          Create, delete, or list users of the platform. Can only be
                   run by a platform administrator.
  verify, ver      Verifies if all required external components contain the
                   proper installed versions.
  version, v       Displays the version of the installed nctl application.
```

Example Experiments

The Nauta installation includes sample training scripts and utility scripts, contained in the `examples` folder, that can be run to demonstrate how to use Nauta. This section describes how to use these scripts.

Examples Folder Content

The `examples` folder in the nctl installation contains the following experiment scripts and scripts folders:

- `mnist_checker.py` - This a utility script used for the inference process and model verification.
- `mnist_converter_pb.py` - This a utility script used for the inference process and model verification.
- `mnist_horovod.py` - Training of digit classifier in Horovod.
- `mnist_input_data.py` - Functions for downloading and reading mnist data.
- `mnist_multinode.py` - Training of digit classifier in distributed TensorFlow setting.
- `mnist_saved_model.py` - Training of digit classifier with saving the model at the end (requires `mnist_tensorboard.py`).
- `mnist_single_node.py` - Training of digit classifier in single node setting `mnist_input_data.py` file).

Additional example scripts for various neural networks are included and have been validated on the Nauta platform.

Utility Scripts

The following are the utility scripts used for the inference process and model verification:

- `mnist_converter_pb.py`
- `mnist_checker.py`

Note: Experiment scripts *must be* written in Python.

Submitting an Experiment

Launch the training experiments with Nauta using the following:

Syntax:

```
nctl experiment submit [options] SCRIPT-LOCATION [-- script-parameters]
```

Where: The path and name of the Python script used to perform this experiment.

```
-- SCRIPT-LOCATION
```

Note: For more info about experiment submit command, refer to submit Subcommand, page 85.

Example Experiments

To submit the example experiments, use the following:

Single Node Training

For *single node* training (template parameter in this case is optional), use the following:

```
nctl experiment submit -t  tf-training-single examples/mnist_single_node.py --name
single
```

Multinode Training

For *multinode* training, use the following:

```
nctl experiment submit -t multinode- tf-training-single examples/mnist_multinode.py --
name multinode
```

Horovod Training

For Horovod training, use the following:

```
nctl experiment submit -t tf-training-horovod examples/mnist_horovod.py --name horovod
```

The included example scripts *do not* require an external data source. The scripts automatically download the MNIST dataset. Templates referenced here have set CPU and Memory requirements. The list of available templates can be obtained by issuing `nctl template list` command.

To change template-related requirements (if desired), refer to the template packs documentation (Working with Template Packs, page 44).

Note: To run TensorBoard, TensorBoard data *must be* written to a folder in the directory `/mnt/output/experiment`. This example script satisfies this requirement; however, your scripts *must* meet the same requirement.

The following example shows how to submit a `MNIST` experiment and write the TensorBoard data to a folder in your Nauta output folder.

Execute the following command to run this example:

```
nctl experiment submit -t tf-training-single examples/mnist_single_node.py --name single
```

Result of this Command

The execution of the submit command may take a few minutes the first time. When the experiment submission is completed, the following result is displayed:

```
Submitting experiments.
| Experiment    | Parameters            | State    | Message    |
|---------------+-----------------------+----------+------------|
| single        | mnist_single_node.py | QUEUED   |            |
```

Running an Experiment using PyTorch Framework

Nauta provides a separate template with the PyTorch framework, which is named `pytorch-training`. If you want to run an experiment based on a PyTorch framework, pass the `pytorch-training` value as the `-t / --template` option when executing the `experiment submit` command.

```
nctl experiment submit --name pytorch --template pytorch-training
examples/pytorch_mnist.py
```

Result of this Command

The previous command runs an experiment using the `pytorch_mnist.py` example delivered together with the `nctl` application. The following result displays showing the queued job.

```
Submitting experiments.
| Name     | Parameters          | Status    | Message    |
|---------+--------------------+----------+----------|
| pytorch | pytorch_mnist.py    | QUEUED    |            |
```

Viewing Experiment Status

Use the following command to view the status of all your experiments:

Syntax:

```
nctl experiment list [options]
```

Example:

Execute this command:

```
nctl experiment list --brief
```

As shown below, an experiment's status displays. This is an example only. The `--brief` option returns a short version of results shown below.

```
| Name                              | Submission date        | Owner    | Status     |
|-----------------------------------+------------------------+---------+-----------|
| mnist-sing-209-19-08-26-18-03-43 | 2019-08-26 06:05:05 PM | user1    | CANCELLED |
| multinode                         | 2019-08-26 06:06:32 PM | user1    | QUEUED     |
| multinodes                        | 2019-09-19 01:38:33 AM | user1    | QUEUED     |
| para-range-1                      | 2019-09-19 01:25:21 AM | user1    | QUEUED     |
| para-range-2                      | 2019-09-19 01:25:23 AM | user1    | QUEUED     |
| para-range-3                      | 2019-09-19 01:25:23 AM | user1    | QUEUED     |
| pytorch                           | 2019-08-26 06:58:01 PM | user1    | QUEUED     |
| single                            | 2019-08-26 06:05:32 PM | user1    | QUEUED     |
| single2                           | 2019-09-20 05:31:06 PM | user1    | COMPLETE   |
```

Monitoring Training

There are four ways to monitor training in Nauta, all which are discussed in the following sections.

Viewing Experiment Logs

To view the experiment log, execute the following command.

Syntax:

```
nctl experiment logs [options] EXPERIMENT-NAME
```

Example:

Execute this command:

```
nctl experiment logs single
```

As shown below, a log displays the example results.

```
2019-03-20T16:11:38+00:00 single-master-0 Step 0, Loss: 2.3015756607055664, Accuracy: 0.078125
2019-03-20T16:11:44+00:00 single-master-0 Step 100, Loss: 0.13010963797569275, Accuracy: 0.921875
2019-03-20T16:11:49+00:00 single-master-0 Step 200, Loss: 0.07017017900943756, Accuracy: 0.984375
2019-03-20T16:11:55+00:00 single-master-0 Step 300, Loss: 0.08880224078893661, Accuracy: 0.984375
2019-03-20T16:12:00+00:00 single-master-0 Step 400, Loss: 0.15115690231323242, Accuracy: 0.953125
2019-03-20T16:12:07+00:00 single-master-0 Validation accuracy: 0.980400025844574
```

Adding Experiment Metrics

Experiments launched in Nauta can output additional kinds of metrics using the *publish function* from the experiment metrics API. Execute the following command to see an example of metrics published with the single experiment executed in the above example, and execute the following command:

```
nctl experiment list
```

A *partial* example result is shown below.

```
| Name                              | Parameters           | Metrics    | Submission date
|-----------------------------------+----------------------+------------+------------------------+
| mnist-sing-209-19-08-26-18-03-43 | mnist_single_node.py |            | 2019-08-26 06:05:05 PM |
| multinode                         | mnist multinode.py   |            | 2019-08-26 06:06:32 PM |
| pytorch                           | pytorch mnist.py   |              | 2019-08-26 06:58:01 PM |
| single                            | mnist_single_node.py |            | 2019-08-26 06:05:32 PM |
```

Adding Experiment Metrics: Instructions

To add metrics to an experiment, you need to edit the experiment script to use the `experiment_metrics.api` and then publish the metric that you wish to display. Complete the following steps in the script to publish a metric.

1. Add the metrics library API with the following entry in your experiment script:

   ```
   from experiment_metrics.api import publish
   ```

2. To add a metric, publish `dict key` and string value. Using the validation accuracy metric as an example, the metric is published in `the mnist_single_node.py` example.

   ```
   publish({"validation_accuracy": str(validation_accuracy_val)})
   ```

3. Save the changes.
4. Submit the experiment again, but with a different name.
5. The published metrics can now be viewed.

   ```
   nctl experiment list
   ```

Saving Metrics for Multinode Experiments

Storing at the same time two (or more) metrics with the same key from two different nodes may lead to errors (such as losing some logs) due to conflicting names. To avoid this, adding metrics for multinode experiments should be done using one of the two following methods: *Method 1* or *Method 2*.

Method 1

The key of a certain metric should also contain a node identifier from which this metric derives. To create an identifier, use one of the following:

- For `horovod` multinode training jobs, result of the `rank()` function provided by the Horovod package can be used as a node's Identifier.

- For `tfjob` multinode training jobs, a user can take all necessary info from the `TF_CONFIG` environment variable. An example piece of a code creating such Identifier, is:

  Node Identifier Example

```
tf_config = os.environ.get('TF_CONFIG')
if not tf_config:
    raise RuntimeError('TF_CONFIG not set!')

tf_config_json = json.loads(tf_config)

job_name = tf_config_json.get('task', {}).get('type')
task_index = tf_config_json.get('task', {}).get('index')
# final node identifier
node_id = '-'.join(job_name,task_index)
```

Method 2

Only one node should store metrics. Use the following to decide which node should store metrics:

- For `horovod` multinode training jobs, the Horovod python library provides the `rank()` function that returns a number of a current worker. *Master* is marked with the number 0, so only a pod with this number should store logs.

- For `tfjob` multinode training jobs, because there is *no* dedicated master node, a user should choose which worker should be responsible for storing metrics. The identifier of a current worker can be obtained as described in *Method 1* above. Furthermore, a user should choose an identifier and store the logs, but only from a node that has this chosen ID.

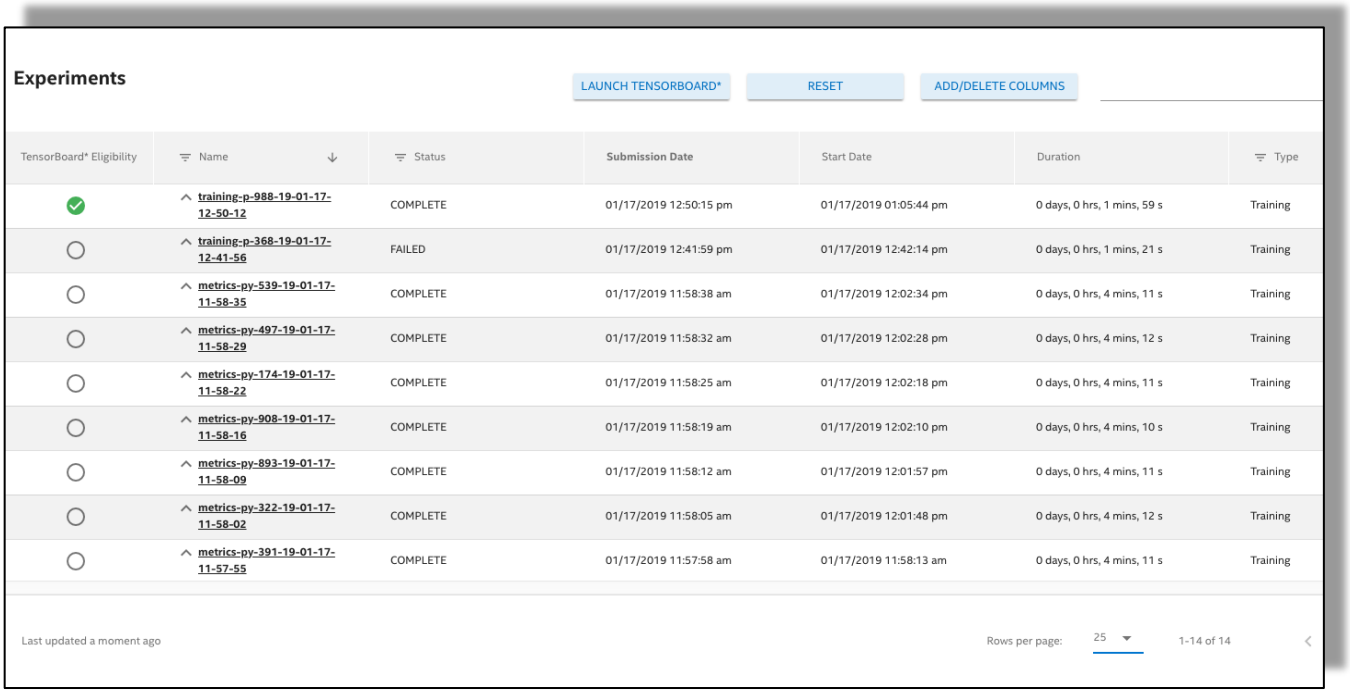
Viewing Experiment Results from the Web UI

The web UI lets you explore the experiments you have submitted. To view your experiments at the web UI, execute the following command at the command prompt:

```
nctl launch webui
```

The following screen displays (this is an example only).

Figure 1: Viewing Experiment Results from the Web UI—Example Only



Experiments

LAUNCH TENSORBOARD* RESET ADD/DELETE COLUMNS

TensorBoard* Eligibility	Name	Status	Submission Date	Start Date	Duration	Type
✓	training-p-988-19-01-17-12-50-12	COMPLETE	01/17/2019 12:50:15 pm	01/17/2019 01:05:44 pm	0 days, 0 hrs, 1 mins, 59 s	Training
○	training-p-368-19-01-17-12-41-56	FAILED	01/17/2019 12:41:59 pm	01/17/2019 12:42:14 pm	0 days, 0 hrs, 1 mins, 21 s	Training
○	metrics-py-539-19-01-17-11-58-35	COMPLETE	01/17/2019 11:58:38 am	01/17/2019 12:02:34 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-497-19-01-17-11-58-29	COMPLETE	01/17/2019 11:58:32 am	01/17/2019 12:02:28 pm	0 days, 0 hrs, 4 mins, 12 s	Training
○	metrics-py-174-19-01-17-11-58-22	COMPLETE	01/17/2019 11:58:25 am	01/17/2019 12:02:18 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-908-19-01-17-11-58-16	COMPLETE	01/17/2019 11:58:19 am	01/17/2019 12:02:10 pm	0 days, 0 hrs, 4 mins, 10 s	Training
○	metrics-py-893-19-01-17-11-58-09	COMPLETE	01/17/2019 11:58:12 am	01/17/2019 12:01:57 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-322-19-01-17-11-58-02	COMPLETE	01/17/2019 11:58:05 am	01/17/2019 12:01:48 pm	0 days, 0 hrs, 4 mins, 12 s	Training
○	metrics-py-391-19-01-17-11-57-55	COMPLETE	01/17/2019 11:57:58 am	01/17/2019 11:58:13 am	0 days, 0 hrs, 4 mins, 11 s	Training

Last updated a moment ago Rows per page: 25 1-14 of 14

Note: If you are using CLI through remote access, you will need to setup a X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, you can use the URL provided by `nctl` command, as shown in the example Web UI screen above.

Web UI Columns

- **Name:** The left-most column lists the experiments by name.
- **Status:** This column reveals experiment's current status, one of: QUEUED, RUNNING, COMPLETE, CANCELLED, FAILED, CREATING.
- **Submission Date:** This column gives the submission date in the format: MM/DD/YYYY, hour:min:second AM/PM.
- **Start Date:** This column shows the experiment start date in the format: MM/DD/YYYY, hour:min:second AM/PM.
- **Duration:** This column shows the duration of execution for this experiment in days, hours, minutes and seconds.
- **Type:** Experiment Type can be Training, Jupyter, or Inference. Training indicates that the experiment was launched from the CLI. Jupyter indicates that the experiment was launched using Jupyter Notebook.

Note: You can perform the tasks discussed below at the Nauta web UI.

Expand Experiment Details

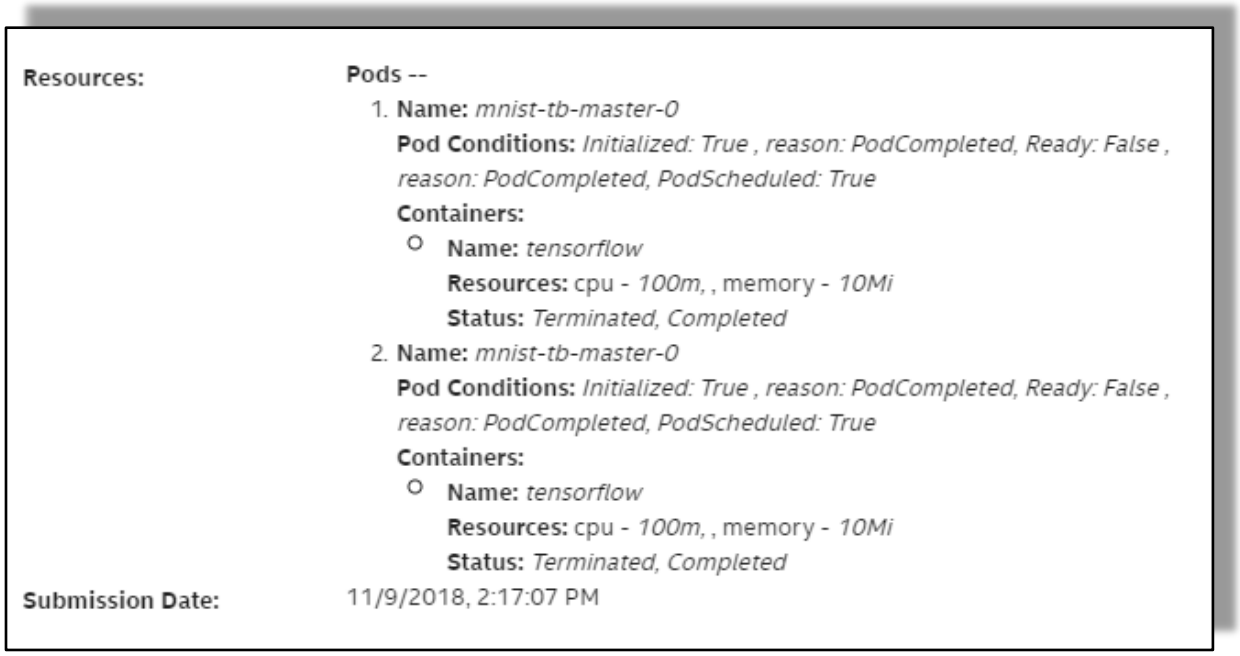
Click the *listed experiment name* to see additional details for that experiment. The following details are examples only. This screen is divided into left and right-side frames.

Left-most Frame

The left-side frame of the experiment details window shows Resources and Submission Date (as shown in the Figure 2).

- **Resources** are assigned to that experiment: specifically, the assigned pods, their status, container information including the CPU, and memory resources assigned.
- Displays the **Submission Date** and time.

Figure 2: Experiment Details—1

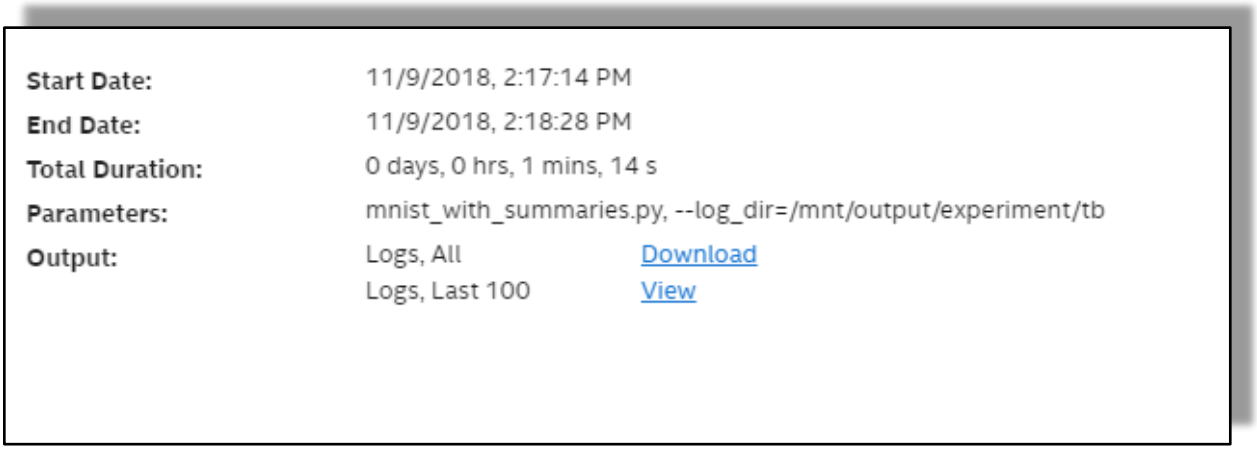


Right-side Frame

The right-side frame (see Figure 3) of the experiment details window shows Start Date, End Date, Total Duration, Parameters, and Output.

- **Start Date:** The day and time this experiment was launched.
- **End Date:** The day and time this experiment was launched.
- **Total Duration:** The actual duration this experiment was instantiated.
- **Parameters:** The experiment script file name and the log directory.
- **Output:** Clickable links to download all logs and view the last 100 log entries.

Figure 3: Experiment Details—2



Searching on Experiments

In the **Search** field at the far right of the UI, enter a string of alphanumeric characters to match the experiment name or other parameters (such as user), and list only those matching experiments. This Search function lets the user search fields in the entire list, *not* just the experiment name or parameters.

Adding and Deleting Columns

ADD/DELETE COLUMNS Button

Click **ADD/DELETE COLUMNS** button to open a scrollable dialogue. Here, the columns currently in use are listed first with their check box checked. Scroll down to see more, available columns listed next, unchecked.

Check/Uncheck Column Headings

Click to check and uncheck and select the column headings you prefer. Optional column headings include parameters such as **Pods**, **End Date**, **Owner**, **Template**, **Time in Queue**, and so on.

Column Heading Metrics

Column headings also include metrics that have been setup using the Metrics API, for a given experiment, and you can select to show those metrics in this display as well.

Column Additions and Deletions

Column additions and deletions you make are retained between logins.

Launching Kubernetes Dashboard

1. Click the **Hamburger Menu** at the far left of the UI to open a left frame.
2. Click **Resources Dashboard** to open the Kubernetes resources dashboard.

   Note: Refer to Accessing the Kubernetes Resource Dashboard, page 79.

Launching TensorBoard

Generally, every file that the training script outputs to `/mnt/output/experiment` (accessed from the perspective of training script launched in Nauta) is accessible from the outside after mounting the output directory with command provided by `nctl mount`.

Use the following command to launch TensorBoard and to view graphs of this model's results. Refer to Working with Datasets, page 31 for more information.

When training scripts output TensorFlow summaries to `/mnt/output/experiment`, they can be automatically picked up by a TensorBoard instance launched with this command:

Syntax:

```
nctl launch tensorboard [options] EXPERIMENT-NAME
```

Execute the following command:

```
nctl launch tensorboard single
```

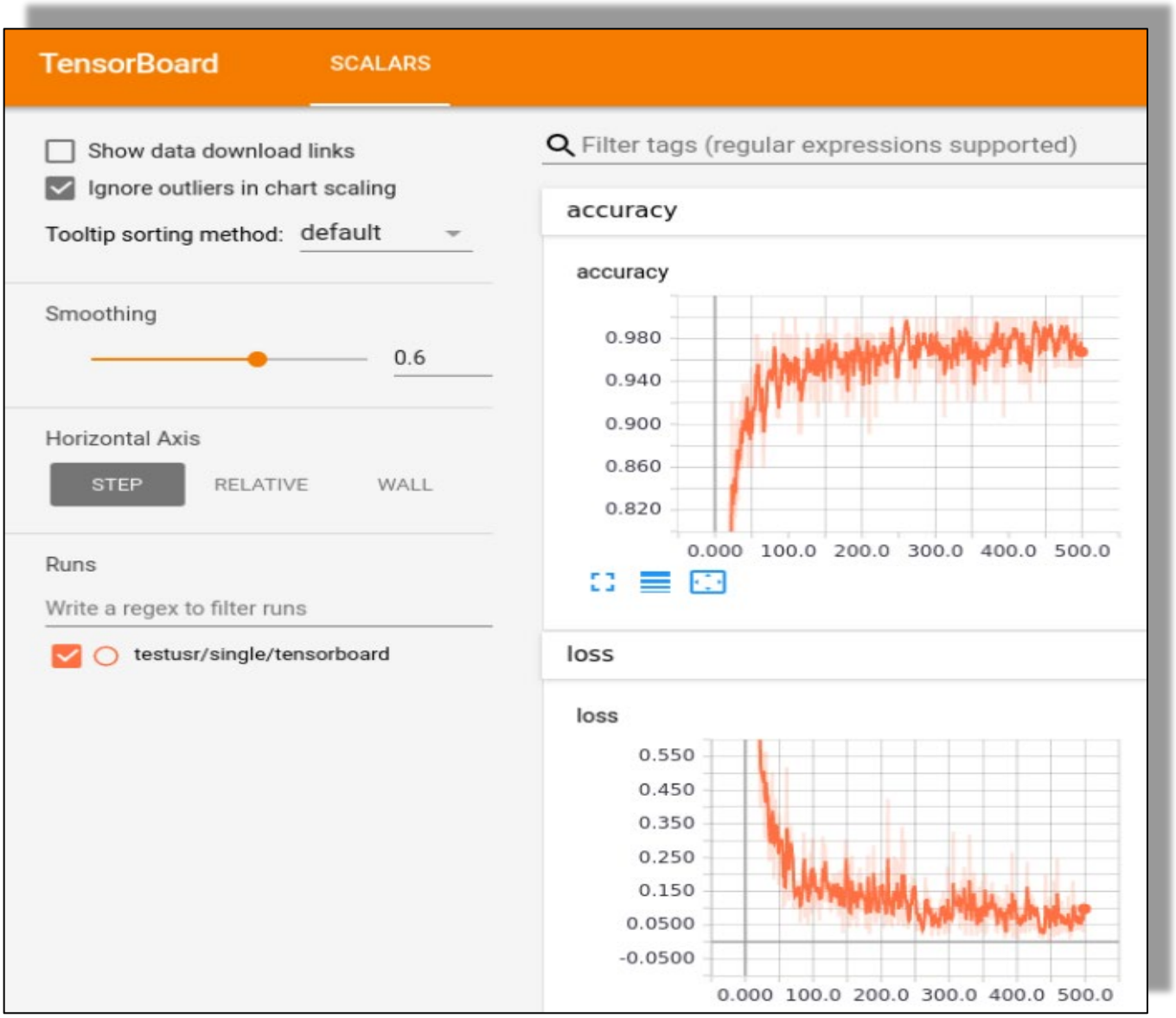
Note: If you are using CLI through remote access, you will need to setup a X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, you can use the URL provided by `nctl`.

The following message displays the example port number.

```
Please wait for Tensorboard to run...
Go to http://localhost: 58218
Proxy connection created.
Press Ctrl-C key to close a port forwarding process...
```

Figure 4, page 27 shows the browser display of TensorBoard dashboard with the experiment’s results.

Figure 4: TensorBoard Dashboard—Example Only



Inference

To perform inference testing (using `predict batch` command in this example), you need to:

1. Prepare the data for model input.
2. Acquire a trained model.
3. Run a prediction instance with trained model on this data.

Data Preparation

The example `mnist_converter_pb.py` script, located in the `examples` folder, can be used for data preparation. This script prepares the sample of the `MNIST` test set and converts it to `protobuf` requests acceptable by the served model. This script is run locally and requires `tensorflow`, `numpy`, and `tensorflow_serving` modules. The `mnist_converter_pb.py` script takes two input parameters:

- `--work_dir` which defaults to `/tmp/mnist_test`. It is a path to directory used as `workdir` by this script and `mnist_checker.py`. The downloaded `MNIST` dataset is stored there, as well as the converted test set sample and labels cached for them.
- `--num_tests` which defaults to 100. It is a number of examples from test set which will be converted. The maximum value is 10000.

Execute the following command:

```
python examples/mnist_converter_pb.py
```

Creates the `/tmp/mnist_test/conversion_out` folder. Fill it with 100 `protobuf` requests, and cache labels for these requests in the `/tmp/mnist_test/labels.npy` file.

Trained Model

Servable models (as with other training artifacts) can be saved by a training script. As previously mentioned, to access these you have to use the command provided by the `nctl mount` command and mount output storage locally. All example scripts save servable models in their model's subdirectory. To use models like this for inference mount `input` storage too, because models have to be accessible from inside of the cluster.

For the *single* experiment example, execute these commands:

```
mkdir -p /mnt/input/single
mkdir /mnt/output
... mount command provided with nctl mount used to mount output storage to /mnt/output
... mount command provided with nctl mount used to mount input storage to /mnt/input
cp -Rf /mnt/output/single/* /mnt/input/single/
```

After these steps, `/mnt/input/single` should contain:

```
/mnt/input/single/:
00001
/mnt/input/single/00001:
saved_model.pb  variables
/mnt/input/single/00001/variables:
variables.data-00000-of-00001  variables.index
```

Running a Prediction Instance

The following provides a brief example of running inference using the `batch` command. For more information, refer to Evaluating Experiments with Inference Testing on page 65.

Before running the `batch` command, copy `protobuf` requests to input storage, because they need to be accessed by the prediction instance too.

Execute these commands:

```
mkdir /mnt/input/data
cp /tmp/mnist_test/conversion_out/* /mnt/input/data
```

To create a prediction instance, execute these commands:

```
nctl predict batch -m /mnt/input/home/single -d /mnt/input/home/data --model-name mnist
--name single-predict
```

The following are the example results of this command:

```
| Prediction Instance   | Model location        | State     |
|-------------+----------------------+---------+-----------|
| single-predict        | mnt/input/home/single | QUEUED    |
```

Notice the additional **`home`** directory in path to both model and input data. This is how the path looks from the perspective of the prediction instance. The `mnist_converter_pb.py` creates requests to the `MNIST` model. The `--model-name mnist` is where this `MNIST` name is given to the prediction instance.

Note: Refer to predict Command, page 108 for additional predict command information.

Prediction Instance Complete

After the prediction instance completes (can be checked using the `predict list` command), collect instance responses from `output` storage.

In the example, it contains *100* *`protobuf`* responses. These can be validated using `mnist_checker.py`.

Running the following command locally will display the error rate calculated for this model and this sample of the test set.

```
python examples/mnist_checker.py --input_dir /mnt/output/single-predict
```

Removing Experiments

An experiment that has been completed and is no longer needed can be removed from the experiment list using the `cancel` command and its `--purge` option.

- If the `--purge` option *is not* set in the cancel command, the experiment will only change status to `CANCELLED`.
- If the `--purge` option is set in the `cancel` command, experiment objects and logs will be irreversibly removed (the experiment's artifacts will remain in the Nauta storage output folder).

Syntax:

```
nctl experiment cancel [options] EXPERIMENT-NAME
```

Execute this command, substituting your experiment name:

```
nctl experiment cancel --purge <your_experiment>
```

Cancelling Experiments

Refer to Cancelling Experiments, page 43.

Working with Datasets

The section covers the following main topics:

Uploading Datasets

Nauta uses NFS to connect to a storage location where each user has folders that have been setup to store experiment input and output data. This option allows the user to upload files and datasets for private use and for sharing. Once uploaded, the files are referenced by the path.

All data in the folders are retained until the user manually removes it from the NFS storage. Refer to the following sections in this chapter for information on how to access and use Nauta storage.

nctl mount Command

The `mount` command displays another command that can be used to mount Nauta folders to a user's local machine. When a user executes the command, information similar the following is displayed (the example shown is for macOS only).

Use the following command (`nctl mount`) to mount those folders as shown in the example in Figure 5 (next page, **zoom in as desired**).

Figure 5: nctl mount Command Output

```
|=============================================================================================|
|   <NAUTA_FOLDER>  |   Code Reference Path     |   User Access     |   Shared Access          |
|---------------------------------------------------------------------------------------------|
|   input           |   /mnt/input/home         |   read/write      |   -                      |
|   output          |   /mnt/output/home        |   read/write      |   -                      |
|   input-shared    |   /mnt/input/root/public  |   read/write      |   read/write             |
|   output-shared   |   /mnt/output/root/public |   read/write      |   read/write             |
|   input-output-ro |                           |   read            |   read                   |
|=============================================================================================|
|   Each experiment has a special folder that can be accessed                                 |
|   as /mnt/output/experiment from training script. This folder is shared by Samba            |
|   as output/<EXPERIMENT_NAME>.                                                              |
|=============================================================================================|
|   <MOUNTPOINT>    |   folder/drive location on your local machine                           |
|=============================================================================================|


Command to unmount previously mounted folder:

sudo umount <MOUNTPOINT> [-fl]

In case of problems with unmounting (disconnected disk etc.) try out -f (force) or -l (lazy) options. For more info
about these options refer to man umount.


Example usage:

    - Mounting a local folder (mylocalfolder) to the user's Nauta input folder:

        sudo mount.cifs -o username=JANEDOE,password=lqS9P5kQ0TFzMmscCY21ZklDDKZtdBeH,rw,uid=10001
//10.91.120.152/input mylocalfolder

    - Code reference path:

        data_dir=/mnt/input/home/<Uploaded-DataFolder>

    - Unmount the folder:

        umount mylocalfolder
```

Other nctl mount and mount Information

The `nctl mount` command also returns a command to unmount a folder. Nauta uses the mount command that is native to each operating system, so the command printed out *may not* appear as in this example. In addition, *all variables* are shown in upper-case.

Mount and Access Folders

Table 1 displays the access permissions for each mounting folder.

Table 1: Access Permissions for Mounting Folders

Nauta Folder	Reference Path	User Access	Shared Access
`input`	`/mnt/input/home`	read/write	-
`output`	`/mnt/output/home`	read/write	-
`input-shared`	`/mnt/input/root/public`	read/write	read/write

Nauta Folder	Reference Path	User Access	Shared Access
`output-shared`	`/mnt/output/root/public`	read/write	read/write
`input-output-ro`		read	read

Uploading and Using Dataset Example

The default configuration is to mount local folders to a Nauta user's private input and output storage folders. Execute the following steps below to mount a local folder, `my_input`, to Nauta storage so that input data can be referenced from the storage when performing training.

1. **Linux/macOS only:** Create a folder for mounting named `my_input` folder:

```
mkdir my_input
```

2. Use `nctl mount` to display the mounting command for your operating system.

```
nctl mount
```

3. Enter the `mount` command that is provided by `nctl mount` using the input as the NAUTA-FOLDER and my-input folder as the MOUNTPOINT. Examples of mounting the command:
   - **MacOS only:**

```
mount_mbfs //'USERNAME:PASSWORD'@CLUSTER-URL/input my_input
```

   - **Ubuntu only:**

```
sudo mount.cifs -o username=USERNAME,password=PASSWORD,rw,uid=1000 //CLUSTER-
URL/input my_input
```

4. Navigate to the mounted location:
   - **MacOS/Ubuntu only:** Navigate to `my-input` folder.
5. Copy a dataset or files to the folder for use in experiments. The files will be located in the Nauta storage until deleted.
6. Using the MNIST example from Submitting an Experiment on page 16, you can download the `MNIST` dataset from this link: MNIST Dataset.
7. Create a `MNIST` folder in the Nauta input folder.
8. Copy the downloaded files to the folder.
9. Submit an experiment referencing the new shared dataset. From the `nctl home` directory, run this command:

```
nctl experiment submit --name mnist-input examples/mnist_single_node.py -- --
data_dir=/mnt/input/home/MNIST
```

Uploading and Using a Shared Dataset

If you want to copy your data to a shared folder, use input-shared instead of input in step 3. Using the shared Nauta storage will allow all Nauta users to use the same `MNIST` dataset by referencing the shared path:

```
/mnt/input/root/public/MNIST
```

Uploading During Experiment Submission

Uploading additional datasets or files is an option available for the *submit* command, using the following option:

```
-sfl, --script-folder-location
```

Where `--script-folder-location` is the name of a folder with additional files used by a script. For example, other `.py` files, datasets, and so on. If the option *is not* included, the files *will not* be included in the experiment.

Syntax:

```
nctl experiment submit --script-folder-location DATASET-PATH SCRIPT-LOCATION
```

This option may be used only for small datasets for development purposes (datasets larger than several MB should be uploaded using standard mechanism described above).

WARNING: Submitting large amount of data using this option will prolong an experiments' submission time.

Working with Experiments

This section provides instructions about the following topics:

Launching Jupyter Interactive Notebook

Use the Jupyter Notebook to run and display the results of your experiments.

Launching Jupyter Interactive Notebook Instructions

This release of Nauta supports Python 3 and 2.7 for scripts.

Syntax:

```
nctl experiment interact [options]
```

Options, include:

- `name` - The name of this Jupyter Notebook session.
- `filename` - File with a notebook or a Python script that should be opened in Jupyter Notebook.

For detailed command syntax information, refer to: experiment interact Subcommand, page 94.

Execute this command to launch Jupyter:

```
nctl experiment interact
```

Tunneling

If you are using CLI through remote access, you will need to setup an X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, use the URL provided by `nctl`. The following result displays.

```
Submitting experiments.
| Experiment                  | Parameters | State   | Message    |
|-----------------------------+------------+--------+-----------|
| jup-936-18-09-17-20-14-58   |            | QUEUED |            |

Browser will start in a few seconds. Please wait...
Go to http://locahost:28113
Proxy connection created.
Press Ctrl-C key to close a port forwarding process...
```

Figure 6 shows the Jupyter Notebook launched in your default web browser.

Figure 6: Jupyter Notebook—Example Only

Cancelling a Jupyter Notebook

In Nauta, running a Jupyter notebook is performed through an interact session. The session remains open and continues to run in your browser until closed Therefore, you must manually cancel the interact session, or it will continue to use/allocate resources.

Steps to Manage and Cancel Interacts

1. To see all running jobs, execute:

```
nctl experiment list --status RUNNING
```

2. To cancel a running interact job, execute:

```
nctl experiment cancel [options] [EXPERIMENT-NAME]
```

   - `EXPERIMENT-NAME` is the interact session name.
   - Use the `--purge` option if you need to remove session from experiment list. For purge information. Refer to Removing Experiments, page 30.

3. To verify that cancellation has completed, enter this command:

```
nctl experiment list --status RUNNING
```

Submitting a Single Experiment

Your script *must be* written to process your input data as it is presented, or conversely, your data must be formatted to be processed by your script. No specific data requirements are made by the Nauta software.

Note: Refer to Submitting an Experiment, page 16 for more information.

Submitting Multiple Individual Experiments

This section describes how to launch multiple experiments using the same script.

Storage locations for your input and output folders are determined by the mount command. Refer to Working with Datasets, page 31 and Mounting Experiment Input to Nauta Storage, page 42.

To submit multiple individual experiments that use the same script, use the following syntax for this command.

Syntax:

```
nctl experiment submit --parameter-range TEXT SCRIPT_NAME [-- script-parameters]
```

Example:

An example command is shown below:

```
nctl experiment submit --parameter-range lr "{0.1, 0.2, 0.3}"
examples/mnist_single_node.py -- --data_dir=/mnt/input/root/public/MNIST
```

Refer to Working with Datasets, page 31 for instructions on uploading the dataset to the `input_shared` folder.

Parameter Ranges and Parameter Sets

Parameters can include either:

- The `parameter-range` is an option of the submit subcommand together with its values expressed as either a range or an explicit set of values

-Or-

- The `parameter-set` is an option of the submit subcommand that specifies a number of distinct combinations of parameter values.

Example:

An *example* of this command using `parameter-range` is shown below.

```
nctl experiment submit --name para-range --parameter range lr "{0.1, 0.2, 0.3}"
examples/mnist_single_node.py -- --data_dir=/mnt/input/root/public/MNIST
```

The following result displays.

```
Please confirm that the following experiments should be submitted.
| Name          | Parameters                                |
|---------------+-------------------------------------------|
| para-range-1 | mnist_single_node.py                      |
|               | lr=0.1                                    |
|               | --data_dir=/mnt/input/root/public/MNIST |
| para-range-2 | mnist_single_node.py                      |
|               | lr=0.2                                    |
|               | --data_dir=/mnt/input/root/public/MNIST |
| para-range-3 | mnist_single_node.py                      |
|               | lr=0.3                                    |
|               | --data_dir=/mnt/input/root/public/MNIST |
Do you want to continue? [Y/n]: y
| Name          | Parameters                       | Status    | Message    |
|---------------+----------------------------------+-----------+------------|
| para-range-1 | mnist_single_node.py lr=0.1 -- | QUEUED    |            |
|               | data_dir=/mnt/input/root/publi |           |            |
|               | c/MNIST                          |           |            |
| para-range-2 | mnist_single_node.py lr=0.2 -- | QUEUED    |            |
|               | data_dir=/mnt/input/root/publi |           |            |
|               | c/MNIST                          |           |            |
| para-range-3 | mnist_single_node.py lr=0.3 -- | QUEUED    |            |
|               | data_dir=/mnt/input/root/publi |           |            |
|               | c/MNIST                          |           |            |
```

Note: Your script *must be* written to process your input data as it is presented, or conversely, your data *must be* formatted to be processed by your script. No specific data requirements are made by the Nauta software.

Run an Experiment on Multiple Nodes

This section describes how to submit an experiment to run on multiple processing nodes, to accelerate the job. Storage locations for your input and output folders are determined by the mount command. Refer to the section Working with Datasets, page 31.

This experiment uses multinode template. For more information, refer to Working with Template Packs, page 44.

To run a multi-node experiment, the script must support it. The following is the generic syntax (the line wrap *is not* intended).

Syntax:

```
nctl experiment submit [options] --template [MULTINODE-TEMPLATE_NAME] SCRIPT-LOCATION
[-- script-parameters]
```

Example:

The template `tf-training-multi` is included with Nauta software. The following is an example command using this template (line wrap *is not* intended):

```
nctl experiment submit --name multinodes --template tf-training-multi
examples/mnist_multinode.py -- --data_dir=/mnt/input/root/public/MNIST
```

Result:

The following result displays (a partial) queued job.

```
Submitting experiments.
| Name        | Parameters                      | Status    | Message    |
|-------------+---------------------------------+-----------+------------|
| multinodes  | mnist_multinode.py -- data_dir  | QUEUED    |            |
|             | =/mnt/input/root/public/MNIST   |           |            |
```

In the previous command, to optionally set the number of workers and servers, set these as parameters below. The default values are 3 worker nodes and 1 parameter server. The following parameters are set to 2 worker nodes and 1 parameter server.

```
-p workersCount 2
-p pServersCount 1
```

Mounting Experiment Input to Nauta Storage

To attach Nauta's storage to a local `folder` and use the files when performing training, refer to the section Working with Datasets, page 31 to mount additional experiment data for use when submitting experiments.

Note: The names used below are for example purposes only.

Mounting Experiment Output to Nauta Storage

To mount a local `folder` to the Nauta storage and use the files when performing training, execute the following steps.

Note: The names used below are for example purposes only.

Linux/macOS

1. Create a folder for mounting and name it `my_output`, by executing the following command:

   ```
   mkdir my_output
   ```

2. Execute the `mount` command to display the command you should use to mount your local `folder` to your Nauta input folder.

   ```
   nctl mount
   ```

3. Enter `mount_smbfs` or `mount.cfis` as appropriate. Be aware, these commands are dependent on the operating system you are using.

   Note: The `MOUNTPOINT` is your `my_output` folder.

4. Navigate to `my_output` folder.
5. Use the output folder (`my_output`) to review the results of the training. For example, trained models.

Unmounting Local Folder from Storage

Perform these steps to unmount previously mounted Nauta storage from a local folder.

1. Use the `mount` command to display the command that should be used to unmount your local folder from your Nauta input folder.

   ```
   nctl mount ls
   ```

2. Execute the unmount using `umount` as appropriate to display the command to unmount your local folder from your Nauta input folder.

   Note: This command is dependent on the operating system.

Cancelling Experiments

The `nctl experiment cancel` command stops and cancels any experiment queued or in progress. Furthermore, the command also cancels any experiment based on the name of an *experiment/pod/status* of a pod. If any such an object is found, the command queries if these objects (one or more) and should be cancelled.

Cancelling One or More Experiments

To cancel one or more experiments, execute the following command:

```
nctl experiment cancel[options] EXPERIMENT-NAME
```

The value of this argument should be created using rules described here. Use this command to cancel one or more experiments with matching or partially-matching names, a matching pod ID, matching pod status, or combinations of these criteria. For example, the following command will cancel all experiments with a matching or partially matching name:

Syntax:

```
nctl experiment cancel --match EXPERIMENT-NAME
```

Cancelling All Experiments with a Matching Pod-ID

The following command will cancel all experiments with a matching pod-ID, using one or more comma-separated IDs:

Syntax:

```
nctl experiment cancel --pod-ids [pod_ID] EXPERIMENT-NAME
```

Cancelling All Experiments with a Matching Pod-Status

The following command will cancel all experiments with a matching pod-status, using one of the following statuses: `[PENDING, RUNNING, SUCCEEDED, FAILED, UNKNOWN]`:

```
nctl experiment cancel --pod-status [PENDING, RUNNING, SUCCEEDED, FAILED, UNKNOWN]
EXPERIMENT-NAME
```

Note: Any of the above criteria can be combined.

Purging an Experiment

You can also purge all experiment-related information using the `-p` or `--purge` option. Refer to the Purging Process, page 78 for more information.

Cancelling one or More Experiments Using the force Command

To cancel one or more experiments with the `force` command, execute the following command:

```
nctl experiment cancel -f [options] EXPERIMENT-NAME
```

Working with Template Packs

This section discusses the following topics:

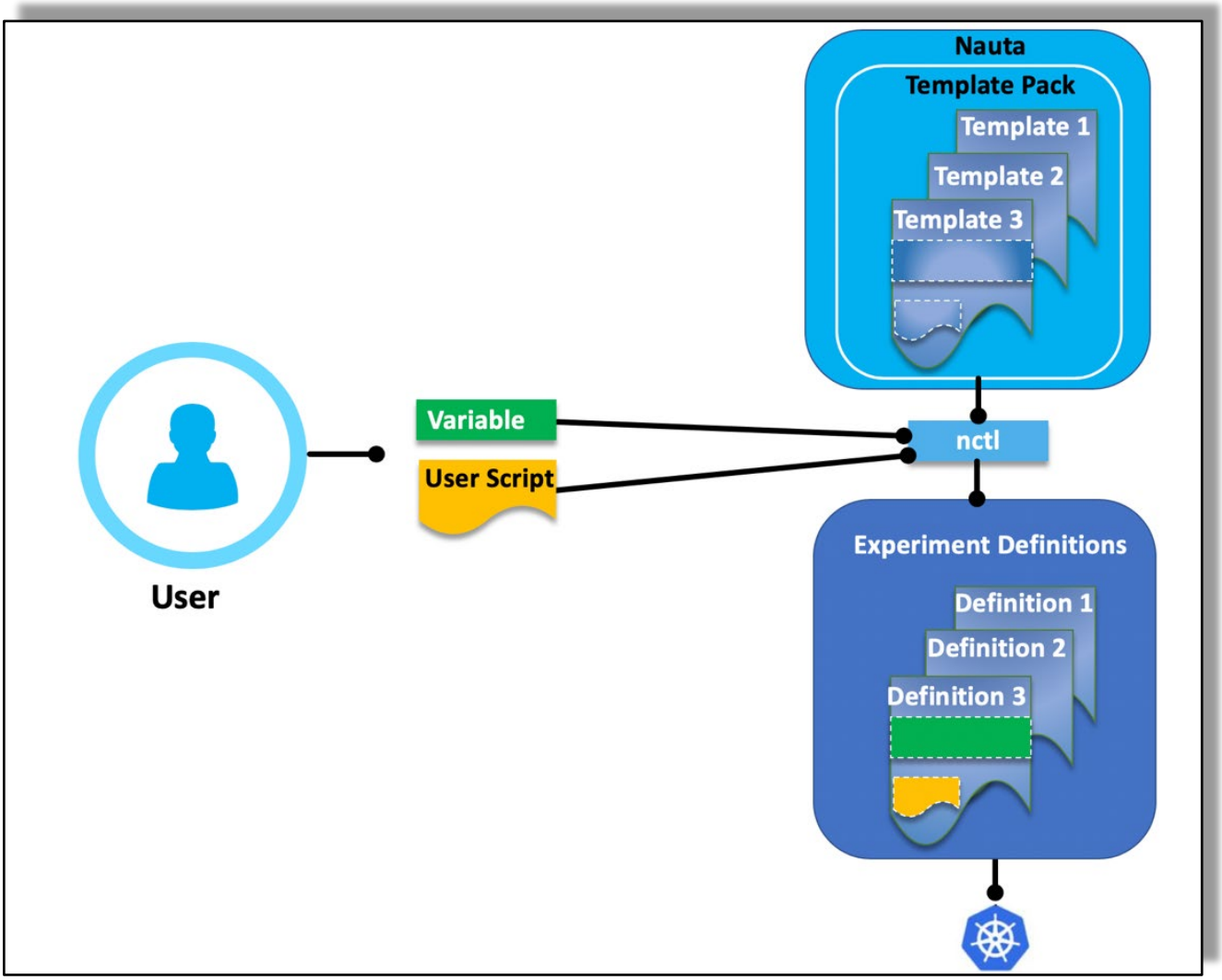
What is a Template Pack?

Every experiment run on the Nauta application utilizes a template pack (as shown in Figure 7). For each experiment, a template pack defines the experiment's complete runtime environment and any supporting infrastructure required to run that experiment.

Each template pack includes a number of elements or templates that together define the Kubernetes (K8s) application, which executes a user-provided experiment script based on specific supporting technology.

Each template pack includes templates that define a Dockerfile, the Kubernetes service definition, deployments, jobs, a configuration map and any other standard Kubernetes elements needed to create a runtime environment for an experiment instance.

Figure 7: Template Pack



Individual elements within a pack are referred to as templates because they contain a number of placeholders that are substituted with appropriate values by Nauta software during experiment submission. Some placeholders are required, while others are optional. These placeholders define items such as: experiment name, user namespace, the address of the local Nauta Docker registry, and other variables that may change between different experiment runs.

The core Kubernetes definitions within each pack are grouped into Helm packages referred to as *Charts*. Helm is the de-facto standard for Kubernetes application packaging and reusing this package format allows leveraging of the large resource of community-developed Helm charts when creating new Nauta template packs.

Note 1: While Nauta is able to re-use Helm charts mostly verbatim, there are a number of required placeholders that need to be added to these charts for Nauta to track and manage the resulting experiments. Refer to Creating a New Template Pack, page 49 for details.

Note 2: All officially supported Nauta template packs are distributed together with the `nctl` package.

Pack Anatomy

Location

When the `nctl` package is installed on the client machine, the template packs that come with the official package are deposited in the folder:

```
NAUTA_HOME/config/packs
```

Each pack resides in a dedicated sub-folder, named after the pack.

The Pack Folder Structure

The individual items that form a single pack are laid out in its folder as follows:

```
<PACK_NAME>/
   Dockerfile
   charts/
      Chart.yaml
      values.yaml
      templates/
```

Table 2: Template Pack Structure Additional Information

Dockerfile	Charts	Chart.yaml	values.yaml	templates
`Dockerfile` is the Docker file that defines the Docker image which serves as the runtime for the experiment's script supplied by the user. Any dependencies needed to build the Docker image must be placed in this directory, next to the *`Dockerfile`*.	*`charts`* is a directory that hosts the Helm chart that specifies the definitions of all Kubernetes entities used to deploy and support the experiment's Docker image in the cluster.	*`Chart.yaml`* provides the key metadata for the chart, such as name and version, and about the chart.	*`values.yaml`* serves a key role as it provides definitions for various Helm template placeholders (refer to Helm's Chart Template Guide for details) used throughout the chart (mostly in the individual Kubernetes definitions contained within the templates sub-folder). This file is also parsed and analyzed by `nctl` to perform substitution on *placeholders in* *`values.yaml`* on page 50	*`templates`* folder groups all the YAML files that provide definitions for various Kubernetes (K8s) entities, which define the packs deployment and runtime environment.

Provided Template Packs

The Nauta software is shipped with a number of built-in template packs that represent the types of experiments officially supported and validated.

For each of the packs there are two versions provided: one that supports Python 2.7.x user scripts (packs with `-py2` suffix in the name) and one that supports Python 3.5.x user scripts.

Packs with multi-suffix in the name support multinode experiments, while those with a single-suffix are designed for single node experiments only.

All packs are optimized for non-trivial deep learning tasks executed on Intel's two socket Xeon systems, and therefore the default compute configuration is the following in Table 3.

Table 3: Compute Configurations for Template Packs

Type	CPU	Memory	Total Experiment per Node
Single-node packs	1 CPU per node	~0.4 available memory	2
Multi-node packs	2 CPUs per node	~0.9 available memory	1

In general, the single node packs are configured to take roughly half of the available resources on a single node (so that the user can *fit* two experiments on a single node), while multi-node packs utilize the entire resources on each node that participates in the multi-node configuration.

While these defaults are intended to guarantee the best possible experience when training on Nauta, it is possible to adjust the compute resource requirements either on per-experiment basis or permanently (refer to Customizing the Provided Packs, page 48 for more information).

The Nauta software should contain at least the following template packs (list of template packs delivered together with the Nauta software depends on the content of the template zoo repository at the moment of building a certain version of the Nauta nctl client). Table 4 describes the template packs provided with Nauta.

Table 4: Template Pack Structure Additional Information

Template Name	Template Description
`jupyter`	An interactive session based on Jupyter Notebook using Python 3.
`jupyter-py2`	An interactive session based on Jupyter Notebook using Python 2.
`ovms-inference-batch`	An OpenVINO model server inference job for batch predictions.
`ovms-inference-stream`	OpenVINO model server inference job for streaming predictions on a deployed instance.
`pytorch-training`	A PyTorch single or multi-node training job using Python 3.
`pytorch- training-py2`	A PyTorch single or multi-node training job using Python 2.
`tf-inference-batch`	A TensorFlow Serving inference job for batch predictions.
`tf-inference-stream`	A TensorFlow Serving inference job for streaming predictions on a deployed instance.
`tf-training-horovod`	A TensorFlow multi-node training job based on Horovod using Python 3.
`tf-training-horovod-py2`	A TensorFlow multi-node training job based on Horovod using Python 2.
`tf-training-multi`	A TensorFlow multi-node training job based on TfJob using Python 3.
`tf-training-multi-py2`	A TensorFlow multi-node training job based on TfJob using Python 2.
`tf-training-single`	A TensorFlow single-node training job based on TfJob using Python 3.
`tf-training-single-py2`	A TensorFlow single-node training job based on TfJob using Python 2.

Customizing the Provided Packs

Any customizations to template packs revolve mostly around the `values.yaml` file included in the pack's underlying Helm chart. As mentioned in The Pack Folder Structure on page 46, this file provides key definitions that are referenced throughout the rest of the Helm chart, and therefore it plays a crucial role in the process of converting the chart's templates into actual Kubernetes definitions deployed on the cluster.

By convention, the definitions contained in the `values.yaml` file typically reference parameters that are intended to be customized by end-users, so in most cases it is safe to manipulate those without corrupting the pack.

Note: This is in contrast to parameters *not* intended for customization. In addition, these parameters typically live within the templates themselves.

Altering Parameters Listed in YAML File

When altering parameters listed in the `values.yaml` file, there are two approaches:

1. You *may* manually modify the pack's `values.yaml` file using a text editor. Any modifications done using this approach will be permanent and apply to all subsequent experiments based on this pack.

2. You *may* alter some of the parameters listed in the `values.yaml` file temporarily, and *only* for a single experiment. To do so, you *may* specify alternative values for any of the parameters listed in `values.yaml` using the `--pack_param` option when submitting an experiment (refer to CLI Commands, page 81 for more details).

If you are an advanced user and want full control over how their experiments are deployed and executed on the Kubernetes cluster may also directly modify the templates residing in the `<PACK-NAME>/charts/templates/` folder. Doing this, however, requires a good grasp of Kubernetes concepts, notation, and debugging techniques, and is therefore ***not recommended***.

Creating a New Template Pack

Prerequisites

Creating a new pack, while not overly complex, requires some familiarity with the technologies that packs are built on. Therefore, it is recommended to have at least some working experience in the following areas do this:

- Creating/modifying Helm charts and specifically using the Helm Templates.
- Defining and managing Kubernetes entities such as `pods`, `jobs`, `deployments`, `services`, and so on.

Where to Start

Creating new template packs for Nauta is greatly simplified by leveraging the relatively ubiquitous *Helm* chart format as the foundation.

Thus, the starting point for a new template pack is typically an existing Helm chart that packages the technology of choice for execution on a *K8s* cluster. Consider creating a chart from scratch only if an existing chart *is not* available. The process of creating a new *Helm* chart from scratch is described in the Official Helm Documentation.

A Template Pack in Five Simple Steps

Once a working Helm chart is available, the process of adapting it for use as a Nauta template is as follows:

1. Name the Pack.

   The name should be unique and not conflict with any other packs available in the local packs folder. After naming the pack, create a corresponding directory in the packs folder and populate its charts subfolder with the contents of the chart. Do not forget to set this pack name also in the `chart.yaml` file. Otherwise the new template *will not* work.

   Instead of creating a completely new folder from scratch, you can also copy an existing one using the `nctl template copy` command and modify its content according to your needs.

2. Create a Dockerfile.

   This Dockerfile will be used to build the image that will host the experiment's scripts. As such, it should include all libraries and other dependencies that experiments based on this pack will use at runtime.

3. Update `values.yaml` (or create it if it *does not* exist).

   The following items that must be placed in the chart's `values.yaml` file in order to enable proper experiment tracking:

- The `podCount` element must be defined and initialized with the expected number of experiment pods that must enter the Running state in order for Nauta to consider the experiment as started.

- If the experiment script to be used with the pack accepts any command-line arguments, then a commandline parameter must be specified and assigned the value of `NAUTA.CommandLine`. (Refer to NAUTA.CommandLine, page 50). This will allow the command line parameters specified in the `nctl experiment submit` command to be propagated to the relevant Helm chart elements (by referencing the `commandline` parameter specified in `values.yaml`)

- An image parameter must be specified and assigned the value of `NAUTA.ExperimentImage`. (Refer to NAUTA.ExperimentImage, page 51).

- The actual name of this parameter *does not* matter as long as it is properly referenced wherever a container image for the experiment is specified within the chart templates.

4. Add tracking labels.
   - The `podCount` element specified above indicates how many pods to expect within a normally functioning experiment based on this pack. The way Nauta identifies the pods that belong to particular experiment is based on specific labels that need to be assigned to each pod that *should* be included in the `podCount` number. The label in question is `runName` and it needs to be assigned the value corresponding to the name of the current Helm release (by assigning the Helm `{{ .Release.name }}` template placeholder).

     Note: Not all pods within an experiment need to be accounted for in `podCount` and assigned the aforementioned label. Nauta only needs to track the pods in which the runtime state is representative of the overall experiment status. If, for instance, an experiment is composed of a *master* pod which in turn manages its fleet of worker pods, then its sufficient to set `podCount` to 1 and only track the *master* as long as it's state (*PENDING, RUNNING, FAILED,* and so on) is representative for the entire group.
5. Update container image references.
   - All container image definitions with the chart's templates that need to point to the image running the experiment script (as defined in the `Dockerfile` in step #1) need to refer to the corresponding `image` Helm template placeholder as previously defined in `values.yaml` (step #3 above).

Nauta values.yaml Placeholders

NAUTA.CommandLine

The `NAUTA.CommandLine` placeholder, when placed within the `values.yaml` file, will be substituted for the list of command line parameters specified when submitting an experiment via `nctl experiment submit` command.

To pass this list as the command line into one of the containers defined in the pack's templates, it needs to be first assigned to a parameter within `values.yaml` This parameter then needs to be referenced within the chart's templates just like any other Helm template parameter.

The following example snippet shows the placeholder being used to initialize a parameter named `commandline`:

```
    commandline:
      args:
        {% for arg in NAUTA.CommandLine %}
        - {{ arg }}
        {% endfor %}
```

NAUTA.ExperimentImage

The `NAUTA.ExperimentImage` placeholder carries the full reference to the Docker image resulting from building the Dockerfile specified within the pack.

During experiment submission the image will be built by Docker and deposited in the Nauta Docker Registry under the locator represented by this placeholder.

Hence, the placeholder shall be used to initialize a template parameter within the `values.yaml` file, that will later be referenced within the chart's templates to specify the experiment image.

Below is a sample definition of a parameter within `values.yaml`, followed by a sample reference to the image in pod template.

```
    <values.yaml>
    image: {{ NAUTA.ExperimentImage }}

    <pod.yaml>
    containers:
    - name: tensorflow
      image: "{{ .Values.image }}"
```

Template Pack Management

The Nauta `nctl` client is shipped with an initial set of template packs. This initial set is created from thee template packs that are stored in the *template zoo* folder when the build of a certain version of a `nctl` application is completed.

Template zoo GitHub

Template zoo is a GitHub repository with template packs created by Intel and a community members for different purposes. Refer to the publicly available template zoo GitHub Repository for more information.

The Nauta `nctl` command provides a special command: template Command, page 116 that makes interaction with this *template zoo* much easier.

This command provides the following options:

- `list` - Lists templates available locally and remotely.
- `install` - Installs locally a template that is remotely available. If a template pack with the same name exists locally, but in a version older than the one available remotely, the local version will be upgraded to the remote one.
- `copy` - Makes a copy of a locally existing template pack. Such a copy might be then freely extended by a user.

Evaluating Experiments

This section discusses the following topics:

Viewing Experiments Using the CLI

Viewing all Experiments

To list *all* experiments you have submitted, run the `nctl experiment` command. As shown in Table 5, the possible returned statuses are:

Table 5: Returned Experiment Status

QUEUED	RUNNING	COMPLETE	CANCELLED	FAILED	CREATING
Experiment has been scheduled but *is not* yet running	Experiment is running	Experiment completed successfully	Experiment has been cancelled by a user	Experiment has failed	Experiment is being created

Syntax:

```
nctl experiment list [options]
```

Example:

An example experiment list is shown below using the list option.

```
nctl experiment list
```

Result:

The following (partial output) example results are shown below.

```
| Experiment   | Parameters           | Metrics                      | Submission date        |
|------------+----------------------+------------------------+-------------------------+
| single       | mnist single node.py | accuracy: 0.96875            | 2019-03-20 05:03:12 PM |
|              |                      | global_step: 499             |                        |
|              |                      | loss: 0.08342029             |                        |
|              |                      | validation_accuracy: 0.9818 |                        |
| single2      | mnist single node.py | accuracy: 0.953125           | 2019-03-20 05:06:19 PM |
|              |                      | global_step: 499             |                        |
|              |                      | loss: 0.078533165            |                        |
|              |                      | validation_accuracy: 0.9838 |                        |
```

Viewing a Single Experiment's Details

The primary purpose of the next command is to provide Kubernetes pod-level information and container information for this experiment. This includes the pod ID, the POD status, information about input and output volumes used in this experiment, and CPU and memory resources requested to perform this experiment.

Use the following command to view a single experiment’s details (this is an example only):

Syntax:

```
nctl experiment view [options] EXPERIMENT-NAME
```

Example:

An example experiment view (the example name used is `single`) is shown below.

```
nctl experiment view single
```

Result:

The following (*partial output*) example results are shown below.

```
| Name   | Parameters           | Metrics   | Submission date        |
|--------+----------------------+-----------+------------------------+
| single | mnist_single_node.py |           | 2019-08-26 06:05:32 PM |


Pods participating in the execution:


| Name            | Uid             | Pod Conditions | Container Details
|
|-----------------+-----------------+----------------------+-------------------|
| single-master-0 | 55101ad9-c81b-1 | reason: Unschedulable,| - Name: tensorflow
|                 | 1e9-a56e-525816 | message: 1/1 tasks
|                                     in gang               | - Status: Not created


Resources used by pods:


| Resource type    | Total usage         |
|------------------+---------------------|
| CPU requests:    | 19000m              |
| Memory requests: | 87GiB 180MiB 135KiB |
| CPU limits:      | 19000m              |
| Memory limits:   | 87GiB 180MiB 135KiB |
Experiment is in QUEUED state due to insufficient number of cpus.
```

The volumes list includes mount mode for each volume (in <> brackets), which can be either `ro` (read-only) or `rw` (read-write).

Viewing Experiment Logs and Results Data

Each experiment generates logs and this information is generated during the run of the experiment and is saved. If an experiment did not print out data during execution, the logs are blank.

Separate from the logs, the results or output of an experiment can be found by mounting the user's output folder or output-shared folder. A model file should write to the Nauta output folder to save any output files.

Execute following command to view logs from a given experiment where `single` is shown in the experiment logs, this indicates the name of the experiment).

Syntax:

```
nctl experiment logs [options] EXPERIMENT-NAME
```

Example:

```
nctl experiment logs single
```

Result:

The following result displays a partial *example log* (where `single` is shown in the experiment logs this indicates the name of the experiment).

```
2019-09-20T15:31:29+00:00 single-master-0 Step 0, Loss: 2.3285014629364014, Accuracy: 0.078125
2019-09-20T15:31:38+00:00 single-master-0 Step 100, Loss: 0.20747436583042145, Accuracy: 0.9375
2019-09-20T15:31:48+00:00 single-master-0 Step 200, Loss: 0.1862923502922058, Accuracy: 0.953125
2019-09-20T15:31:58+00:00 single-master-0 Step 300, Loss: 0.1068115308880806, Accuracy: 0.984375
2019-09-20T15:32:07+00:00 single-master-0 Step 400, Loss: 0.0432920977473259, Accuracy: 0.984375
2019-09-20T15:32:17+00:00 single-master-0 Validation accuracy: 0.9832000136375427
```

Note: Logs generated with sub-millisecond frequency may appear out of order when displayed. This is caused by the 1ms resolution of the underlying logging solution.

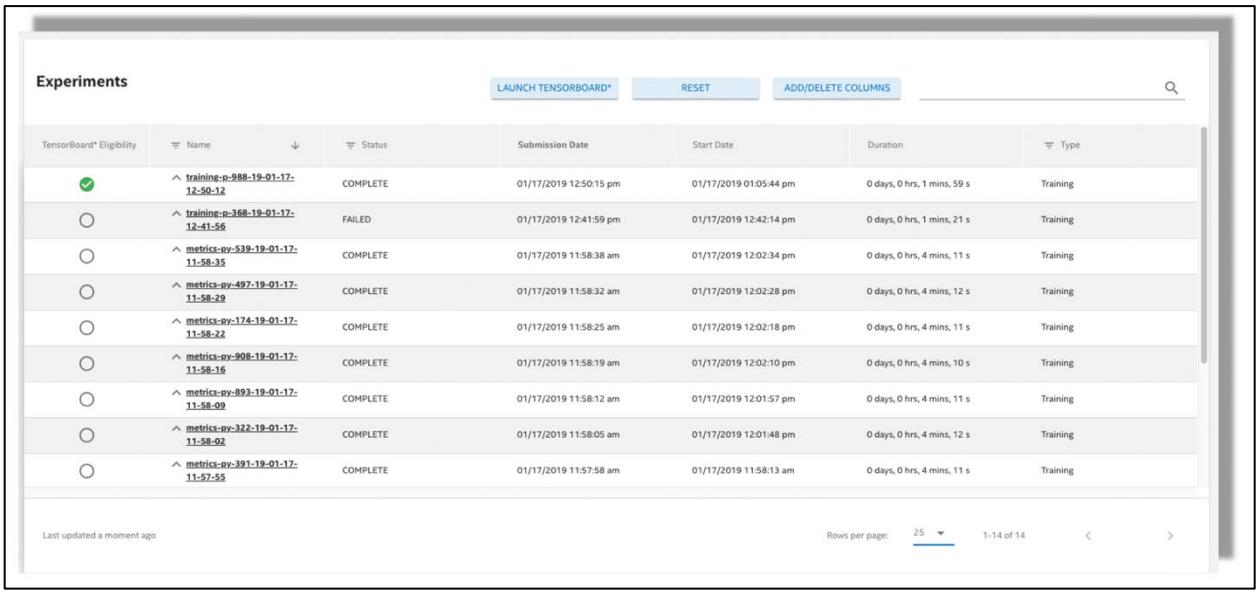
Viewing Experiment Results at the Web UI

The web UI lets you explore the experiments you have submitted. To view your experiments at the web UI, execute the following command at the command prompt:

```
nctl launch webui
```

Figure 8 shows an example of the experiment results in the Web UI screen.

Figure 8: Viewing Experiment Results from the Web UI—Example Only



TensorBoard* Eligibility	Name	Status	Submission Date	Start Date	Duration	Type
✓	training-p-988-19-01-17-12-50-12	COMPLETE	01/17/2019 12:50:15 pm	01/17/2019 01:05:44 pm	0 days, 0 hrs, 1 mins, 59 s	Training
○	training-p-368-19-01-17-12-41-56	FAILED	01/17/2019 12:41:59 pm	01/17/2019 12:42:14 pm	0 days, 0 hrs, 1 mins, 21 s	Training
○	metrics-py-539-19-01-17-11-58-35	COMPLETE	01/17/2019 11:58:38 am	01/17/2019 12:02:34 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-497-19-01-17-11-58-29	COMPLETE	01/17/2019 11:58:32 am	01/17/2019 12:02:28 pm	0 days, 0 hrs, 4 mins, 12 s	Training
○	metrics-py-174-19-01-17-11-58-22	COMPLETE	01/17/2019 11:58:25 am	01/17/2019 12:02:18 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-908-19-01-17-11-58-16	COMPLETE	01/17/2019 11:58:19 am	01/17/2019 12:02:10 pm	0 days, 0 hrs, 4 mins, 10 s	Training
○	metrics-py-893-19-01-17-11-58-09	COMPLETE	01/17/2019 11:58:12 am	01/17/2019 12:01:57 pm	0 days, 0 hrs, 4 mins, 11 s	Training
○	metrics-py-322-19-01-17-11-58-02	COMPLETE	01/17/2019 11:58:05 am	01/17/2019 12:01:48 pm	0 days, 0 hrs, 4 mins, 12 s	Training
○	metrics-py-391-19-01-17-11-57-55	COMPLETE	01/17/2019 11:57:58 am	01/17/2019 11:58:13 am	0 days, 0 hrs, 4 mins, 11 s	Training

Note: If you are using CLI through remote access, you will need to setup a X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, you can use the URL provided by `nctl` command.

Web UI Columns

- **Name:** The left-most column lists the experiments by name.
- **Status:** This column reveals experiment's current status, one of: QUEUED, RUNNING, COMPLETE, CANCELLED, FAILED, CREATING.
- **Submission Date:** This column gives the submission date in the format: MM/DD/YYYY, hour:min:second AM/PM.
- **Start Date:** This column shows the experiment start date in the format: MM/DD/YYYY, hour:min:second AM/PM.
- **Duration:** This column shows the duration of execution for this experiment in days, hours, minutes and seconds.
- **Type:** Experiment Type can be Training, Jupyter, or Inference. Training indicates that the experiment was launched from the CLI. Jupyter indicates that the experiment was launched using Jupyter Notebook.

Note: You can perform the tasks discussed below at the Nauta web UI.

Expand Experiment Details

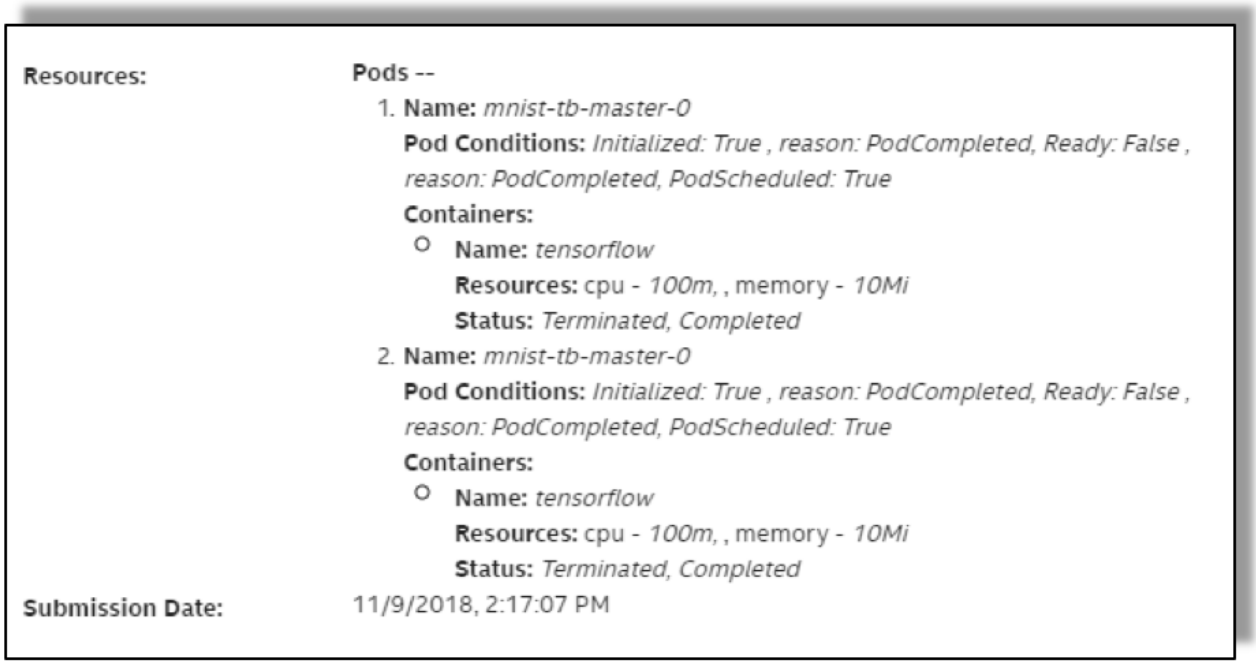
Click the *listed experiment name* to see additional details for that experiment. The following details are examples only. This screen is divided into left and right-side frames.

Left-side Frame

The left-side frame (see Figure 9, page 57) of the experiment details window shows Resources and Submission Date.

- **Resources** are assigned to that experiment: specifically, the assigned pods, their status, container information including the CPU, and memory resources assigned.
- Displays the **Submission Date** and time.

Figure 9: Experiment Details—1

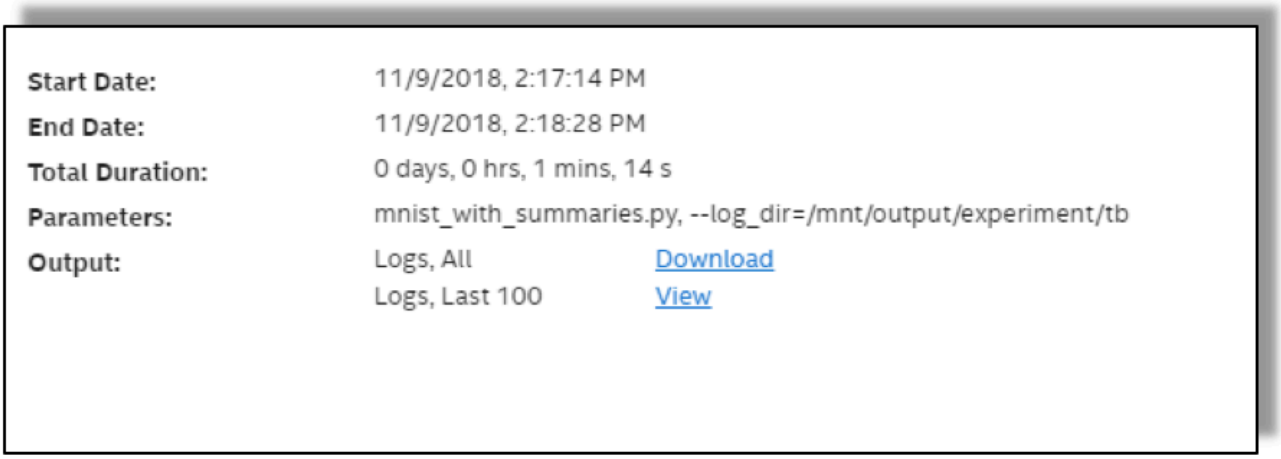


Right-side Frame

The right-side frame (see Figure 10) of the experiment details window shows Start Date, End Date, Total Duration, Parameters, and Output.

- **Start Date:** The day and time this experiment was launched.
- **End Date:** The day and time this experiment was launched.
- **Total Duration:** The actual duration this experiment was instantiated.
- **Parameters:** The experiment script file name and the log directory.
- **Output:** Clickable links to download all logs and view the last 100 log entries.

Figure 10: Experiment Details—2



Searching on Experiments

In the **Search** field at the far right of the UI, enter a string of alphanumeric characters to match the experiment name or other parameters (such as user), and list only those matching experiments. This Search function lets the user search fields in the entire list, *not* just the experiment name or parameters.

Adding and Deleting Columns

ADD/DELETE COLUMNS Button

Click **ADD/DELETE COLUMNS** to open a scrollable dialogue. The columns currently in use are listed first with their check box checked. Scroll down to see more, available columns listed next, unchecked.

Check/Uncheck Column Headings

Click to check and uncheck and select the column headings you prefer. Optional column headings include parameters such as Pods, End Date, Owner, Template, Time in Queue, and so on.

Column Heading Metrics

Column headings also include metrics that have been setup using the Metrics API, for a given experiment, and you can select to show those metrics in this display as well.

Refer to Launching TensorBoard to View Experiments, page 60 for more information.

Column Additions and Deletions

Column additions and deletions you make are retained between logins.

Launching Kubernetes Dashboard

1. Click the **Hamburger Menu** at the far left of the UI to open a left frame.
2. Click **Resources Dashboard** to open the Kubernetes resources dashboard.

   Note: Refer to Accessing the Kubernetes Resource Dashboard, page 79.

Launching TensorBoard to View Experiments

You can launch TensorBoard from the Nauta web UI or the CLI; both methods are described.

Launching TensorBoard from the Web UI

To view the experiment's results in TensorBoard, TensorBoard data must be written to a folder in the directory `/mnt/output/experiment`.

To launch TensorBoard from the web UI and view results for individual experiments, execute these steps:

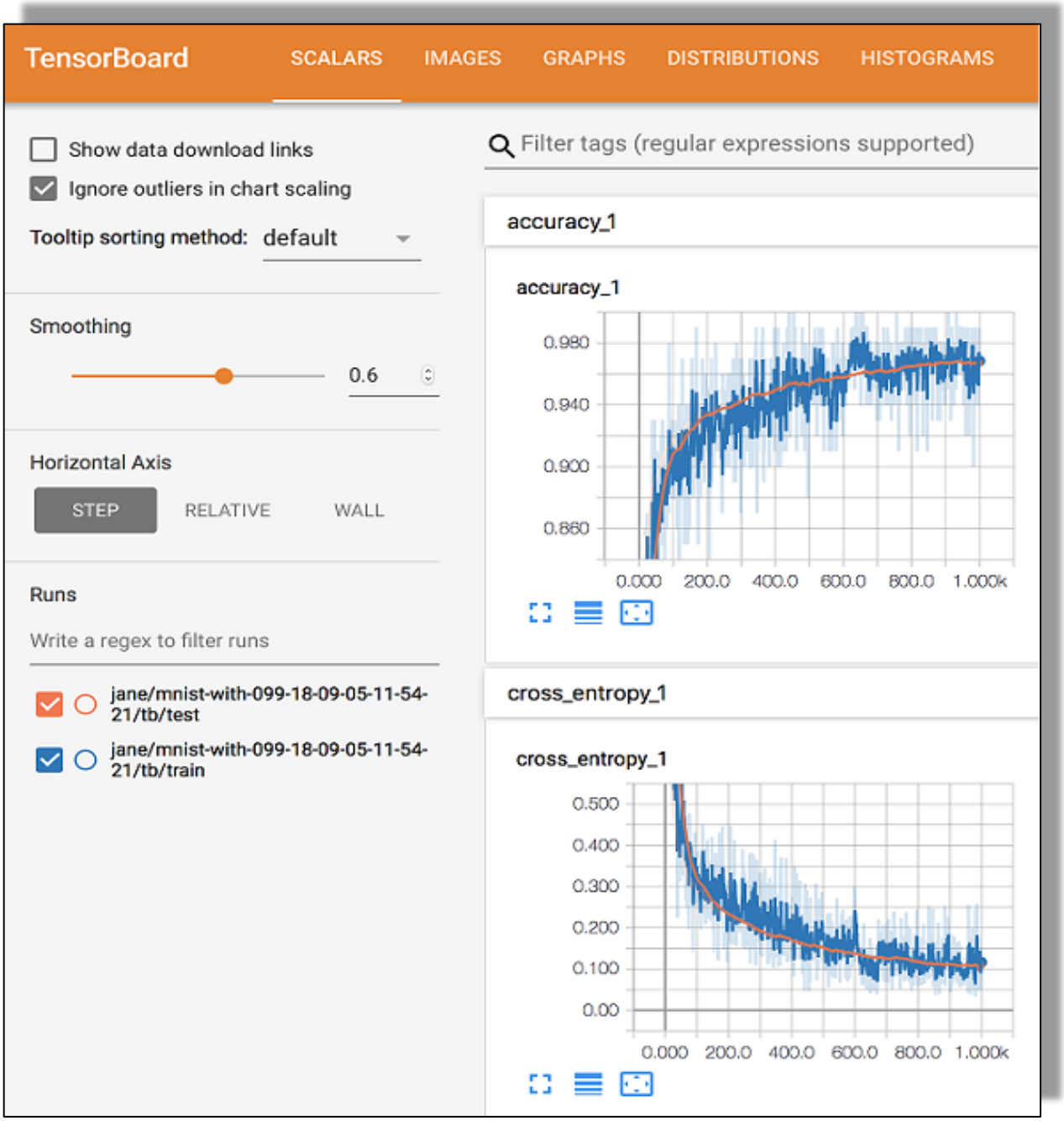
1. Open the web UI by executing this command:

   ```
   nctl launch webui
   ```

2. At the web UI, identify the experiment that you want to see displayed in TensorBoard. Click the check box to the left of the experiment name.
3. With an experiment selected (checked), the **LAUNCH TENSORBOARD** button becomes active. Click **LAUNCH TENSORBOARD**. TensorBoard is launched on a separate browser tab/window with graphs showing the experiment's results.

   Figure 11, page 61 shows an example TensorBoard UI.

Figure 11: Launch TensorBoard from the Web UI – Example Only



Launching TensorBoard from the CLI

1. To launch TensorBoard from the CLI, execute this command:

```
nctl launch tb <experiment-name>
```

2. The following result displays.

```
Please wait for Tensorboard to run...
Go to http://localhost: 58218
Proxy connection created.
Press Ctrl-C key to close a port forwarding process...
```

This command will launch a local browser. If the command was run with the `--no-launch` option, then you need to copy the returned URL into a web browser. TensorBoard is launched with graphs showing the experiment's results (as shown above).

You can also launch TensorBoard and with the `nctl experiment view` command:

```
nctl experiment view -tensorboard <experiment-name>
```

Note: This command exposes a TensorBoard instance with data from the named experiment.

MNIST Example

You can try out TensorBoard with MNIST classifier `mnist_tensorboard.py` from the Python examples in the `examples` folder. Run:

```
nctl experiment submit -p cpu 1 -p memory 8Gi examples/mnist_tensorboard.py -n tb-example
nctl exp view -tb tb-example
```

Note: `mnist_tensorboard.py` requires at least 8Gi of memory.

Exporting Models

The section discusses how to transform a model from one format to another using the model export functionality.

Obtaining a Model For Exporting

To use the flow for exporting models, select a model that will export to another format. To successfully do this, create a model using the `mnist_saved_model.py` example script, which is delivered together with `nctl` application. This script trains the model and then stores it in a shared folder.

To generate the model, use the following command:

```
nctl exp submit examples/mnist_saved_model.py -sfl examples/ -n generate-model --
/mnt/output/experiment
```

This command trains a model using TensorFlow framework and stores it in the `output/generate-model` shared folder. Passing this command to the `-sfl` (`--script-folder-location`) option is required, as the `mnist_saved_model.py` script requires the presence of the `mnist_input_data.py` script. This script is located in a folder with examples in the `nctl` distribution.

To check whether the script has been created, mount the locally shared folder (referenced above) and check if it contains the *One* subfolder (the script generates only *One* model, which is stored in a folder named as an ordinary number of this model).

Checking a List of Available Exports' Formats

To check what are the available exports' formats, execute the following command:

```
nctl model export formats
```

This command displays a list of formats, for example:

```
| Name     | Parameters description                                                          |
|----------+---------------------------------------------------------------------------------|
| openvino | --input_shape [x,y,....] - shape of an input                                    |
|          | --input [name] - names of input layers                                          |
|          | --output [name] - names of output layers                                        |
|          | Rest of parameters can be found in a description of OpenVino model optimizer |
```

Exporting the Model to OpenVINO Format

The model export formats command shows that you can export the model to `openvino` format. Use the `model export` command in the following format to export the model.

```
nctl model export <model_location> <format> -- <format_specific_parameters>
```

Where:

- `<model_location>` - This is the location of a model that is going to be exported.
- `<format>` - This is the format of the exported model.
- `<format_specific_paramaters>` - These are the parameters required during the export process. Their number and format is dependent on the model and the chosen format.

To export the model created in the previous step, use the following command:

```
nctl model export /mnt/output/home/generate-model/1 openvino -- --
input_shape [1,784] --input x --output y
```

The parameters `input_shape`, input and output are required to perform a successful export to `openvino` format.

- `input_shape` - Describes the shape of the input vector of the exported model.
- `input, output` - Describes the names of input and output vectors.

Successful execution of this command produces the following output:

```
| Operation      | Start date            | End date    | Owner      | State    |
|---------------+----------------------+------------+-----------+---------|
| openvino_1     | 2019-07-17T16:13:40Z |            | jdoe       | Queued   |


Successfully created export workflow
```

Note: The name of the operation is just an example, your naming may differ from the example.

The duration of the export operation depends on a chosen format. To check the status of the operation, use the following command:

```
nctl model status
```

This command returns a list of export operations with their statuses. If an export operation is finished, its status is `Succeeded`. When this occurs, an exported model can be found in the `output/openvino_1` shared folder. This folder contains the following files: `saved_model.bin`, `saved_model.mapping` and `saved_model.xml`.

If you export the operation and issues occur, details of those issues can be found in the logs from an export. To review the logs and issues, use the following command:

```
nctl model logs openvino_1
```

Evaluating Experiments with Inference Testing

Nauta provides you with the ability to test your trained model using TensorFlow Serving and OpenVINO Model Server (OVMS). OVMS is an OpenVINO serving component intended to provide hosting for the OpenVINO inference runtime.

For guidance on using Inference Testing to evaluate an experiment, refer to the topics shown below.

- For `nctl predict` command, its subcommands, and parameter information, refer to predict Command, page 108.

For How-to instructions for TensorFlow Serving:

- Refer to TensorFlow Serving Batch Inference Example, page 66 for running batch inference.
- Refer to TensorFlow Serving Streaming Inference Example, page 69 for running streaming inference.

To run prediction on OpenVINO Model Server, refer to Inference Example on OpenVINO Model Server

- For running prediction on OpenVINO Model Server, refer to Inference Example section: Inference on Models Served by the OpenVINO Model Server, page 73.

This section covers the following topics:

Using the predict Command

Use the `predict` command to start, stop, and manage prediction jobs. Refer to the predict Command, page 108 for a detailed description of this command.

TensorFlow Serving Batch Inference Example

Flow Example

Below are the general steps required to run batch inference on Nauta.

1. Acquire the dataset and the trained model.
2. Convert the dataset into *Serialized Protocol Buffers* (PBs). Refer to Protocol Buffers for additional PB information.
3. Mount the Samba shared folder by invoking the `nctl mount` command displayed (in the step 3).
4. Copy the serialized PBs and the trained model to the *just-mounted* shared folder.
5. Run `nctl predict batch` command.

Note: Be aware, if the general flow requirements *are not* met you *will not* be able to complete the example.

MNIST Example

You must preprocess `MNIST` data for feeding the batch inference. You can generate example data by executing the following steps:

MNIST Data Preprocessing

1. Execute the following command to create venv:

```
python3 -m venv .venv
```

2. Install required dependency in venv:

```
source .venv/bin/activate
pip install tensorflow-serving-api
```

3. Create a directory with two subdirectories named input and output.
4. Run the `mnist_converter_pb.py` script using just generated venv (from `nauta/applications/cli/example-python/package_examples`) using just-generated venv:

```
python mnist_converter_pb.py
```

The results of the conversion are stored in the `conversion_out` directory under `work_dir` parameter. The default is: `/tmp/mnist_test/conversion_out`. Copy them to your input directory.

Parameters of mnist_converter_pb.py

- `work_dir` – Location where files related with conversion will be stored. The default is: `/tmp/mnist_tests`.
- `num_tests` – Number of examples to convert. The default is: `100`.

Note: Results of conversion are stored in `conversion_out` directory under `work_dir` parameter. The default is: `/tmp/mnist_tests/conversion_out`.

Start Prediction

1. Run `nctl mount`.
2. Use the command printed by `nctl mount`. Replace the **<NAUTA_FOLDER>** with the *input* and with your input directory.
3. Use the same command, but this time replace the **<NAUTA_FOLDER>** with the *output* and with your output directory.
4. If you mounted the wrong directories, use the `sudo umount` [`name of the mounted directory`] command. You can run `mount` to check which directories have been mounted.
5. Deactivate and delete python virtual environment.
6. Execute the following command:

```
deactivate
rm -rf .venv/
```

7. Create model with a script `mnist_saved_model.py` to your input directory.
8. Execute the following command:

```
nctl experiment submit mnist_saved_model.py -sfl /nauta/applications/cli/examples --
name mn-model -- --training iteration=5 --model version=1 --export dir
/mnt/output/home/mn-model
```

   - **Notes:**
     - Where you see --name, this indicates the experiment name of your choice. This example uses mn-model.
     - The `-sfl/--script-folder-location` *must* include the name of the directory where Nauta examples are stored, including `mnist_saved_model.py`.
   - You can try out different a number of iterations.
9. Copy the directory with the name of your experiment from output folder to the input.
10. Execute the following command:

```
nctl predict batch --model-location /mnt/input/home/predict-model --data
/mnt/input/home/conversion_out --model-name mnist --name batch-predict
```

11. Use the script below to see predictions in a human-readable form.

```
import os
from tensorflow_serving.apis import predict_pb2
for i in range(100):
    with open(os.path.join(".", "{}.pb".format(i)), mode="rb") as pb_file:
            result_pb = pb_file.read()
            resp = predict_pb2.PredictResponse()
            resp.ParseFromString(result_pb)
            print(resp.outputs["scores"].float_val)
```

As a result of each sample, you will get an array of 10 elements that present the possibility of each sample being a given digit. The first element represents how likely it is that the picture represents "0", the last element represents "9". The higher the number, the more likely it is that this sample is that digit. An example is shown below.

```
[0.0156935453414917, 0.06918075680732727, 0.023996423929929733,
0.00025786852347664535, 0.07656218856573105, 0.05128718540072441,
0.1812051236629486, 0.02422264777123928, 0.0640382319688797,
0.49355611205101013]
```

In the example above, the highest value has the element of index 9, so the sample indicates that "9" is represented.

Other Important Information

Paths

Paths provided in locations such as, `--model-location` and `--data` need to point (for `files/directory`) from the container's context, *not* from a user's filesystem or mounts. These paths can be mapped using instructions from `nctl mount`.

For example, if you have mounted Samba `/input` and copied the files there, you should pass:

```
/mnt/input/home/<file>
```

Model Name

The `--model-name` is optional, but it *must* match the model name provided during data preprocessing, since generated requests *must* define which servable they target.

In the `mnist_converter_pb.py` script, you can find the `request.model_spec.name = 'mnist'`. This saves the model name in requests, and that name must match a value passed as: `--model-name`.

If *not* provided, it assumes that the model name is equal to last directory in model location:

```
/mnt/input/home/trained_mnist_model --> trained_mnist_model
```

Useful External References

- Serving a TensorFlow Model
- Protocol Buffer Basics: Python
- mnist_client.py Script
- TensorFlow Serving with Docker

TensorFlow Serving Streaming Inference Example

Example Flow

1. Save a trained TensorFlow Serving compatible model.
2. Send the data for inference in JSON format, or in binary format using gRPC API.
3. Run the `nctl predict launch` command.
4. Send the inference data using the `nctl predict stream` command, TensorFlow Serving REST API, or TensorFlow Serving gRPC API.

TensorFlow Serving Basic Example

Launching a Streaming Inference Instance

Basic models for testing TensorFlow Serving are included in the following GitHub TensorFlow Serving Repository. This example will use the *`saved_model_half_plus_two_cpu`* model for showing streaming prediction capabilities.

Perform the following steps to use this model for streaming inference:

1. Clone the TensorFlow Serving Repository executing the following command:

```
git clone https://github.com/tensorflow/serving
```

2. Perform step 3 or step 4 below, based on preference.
3. Run the following command:

```
nctl predict launch --local-model-location <directory where you have cloned
Tensorflow
Serving>/serving/tensorflow_serving/servables/tensorflow/testdata/saved_model_hal
f_plus_two_cpu
```

4. Alternatively, for step 3, you *may* want to save a trained model on input shared folder, so it can be reused by other experiments/prediction instances. To do this, run these commands:
   a. Use the `mount` command to mount the Nauta input folder to a local machine:

```
nctl mount
```

   b. Run the resulting command printed by `nctl mount` (in this example, assuming that you will mount `/mnt/inputshare` described in nctl command Mounting Experiment Output to Nauta Storage, page 42). After executing the command printed by the `nctl mount` command, you will be able to access input share on your local file system.
   c. Use the resulting command printed by. After executing command printed by `nctl mount` command, you will be able to access input shared folder on your local file system.
   d. Copy the *`saved_model_half_plus_two_cpu`* model to input shared folder:

```
cp -r <directory where you have cloned Tensorflow
Serving>/serving/tensorflow_serving/servables/tensorflow/testdata/saved_model
_half_plus_two_cpu <directory where you have  mounted /mnt/input share>
```

   e. Run the following command:

```
nctl predict launch --model-location /mnt/input/saved_model_half_plus_two_cpu
```

Note: The `--model-name` can be passed optionally to `nctl predict launch` command. If *not* provided, it assumes that the model name is equal to the last directory in model location:

```
/mnt/input/home/trained_mnist_model -> trained_mnist_model
```

Using a Streaming Inference Instance

After running the `predict launch` command, `nctl` creates a streaming inference instance that can be used in multiple ways, as described below.

Streaming Inference with nctl predict stream Command

The `nctl predict stream` command allows for performing inference on input data stored in JSON format. This method is convenient for manually testing a trained model and provides a simple way to get inference results. For `saved_model_half_plus_two_cpu`, write the following input data and save it in the `inference-data.json` file:

```
{"instances": [1.0, 2.0, 5.0]}
```

The model `saved_model_half_plus_two_cpu` is a quite simple model: for given `x` input value it predicts result of `x/2 +2` operations. Having passed following inputs to the model: `1.0`, `2.0`, and `5.0`, and so expected predictions results are: `2.5`, `3.0`, and `4.5`.

To use that data for prediction, check the name of the running prediction instance with `saved_model_half_plus_two_cpu` model (the name will be displayed after `nctl predict launch` command executes; you can also use nctl predict list command for listing running prediction instances). Then, run following command:

```
nctl predict stream --name <prediction instance name> --data inference-data.json
```

The following results will be produced:

```
{ "predictions": [2.5, 3.0, 4.5] }
```

TensorFlow Serving exposes three different method verbs for getting inference results. Selecting the proper method verb depends on the model used and the expected results. Refer to RESTful API for more detailed information. These method verbs are:

- classify
- regress
- predict

By default, `nctl predict stream stream` will use the `predict` method verb. You can change it by passing the `--method-verb` parameter to the `nctl predict stream` command, for example:

```
nctl predict stream --name <prediction instance name> --data inference-data.json --
method-verb classify
```

Streaming Inference with TensorFlow Serving REST API

Another way to interact with a running prediction instance is to use TensorFlow Serving REST API. This approach could be useful for more sophisticated use cases, like integrating data-collection scripts/applications with prediction instances.

The URL and authorization header for accessing TensorFlow Serving REST API will be shown after prediction instance is submitted, as in the example below.

```
Prediction instance URL (append method verb manually, e.g. :predict):
https://192.168.0.1:8443/api/v1/namespaces/jdoe/services/saved-mode-621-18-11-07-
15-00-34:rest-port/proxy/v1/models/saved_model_half_plus_two_cpu

Authorize with following header:
Authorization: Bearer
1234567890abcdefghijklmnopqrstuvxyz
```

Accessing the REST API with curl

The example shows Accessing REST API using curl, with the following command:

```
curl -k -X POST -d @inference-data.json -H 'Authorization: Bearer <authorization
token data>' localhost:8501/v1/models/<model_name, e.g.
saved_model_half_plus_two_cpu>:predict
```

Using Port Forwarding

Alternatively, the Kubernetes port forwarding mechanism may be used. Create a port forwarding tunnel to the prediction instance with the following command:

```
kubectl port-forward service/<prediction instance name> :8501
```

Or, if you want to start a port forwarding tunnel in the background:, do the following:

```
kubectl port-forward service/<prediction instance name> <some local port
number>:8501 &
```

Note: The local port number of the tunnel you entered above; it will be produced by kubectl port-forward if you *do not* explicitly specify it.

You should be able to access the REST API on the following URL:

```
localhost:<local tunnel port number>/v1/models/<model_name, e.g.
saved_model_half_plus_two_cpu>:<method verb>
```

Example of Accessing REST API Using curl

To access REST API using curl, execute the following command:

```
curl -X POST -d @inference-data.json localhost:8501/v1/models/<model_name, e.g.
saved_model_half_plus_two_cpu>:predict
```

Streaming Inference with TensorFlow Serving gRPC API

Another way to interact with running prediction instance is to use TensorFlow Serving gRPC. This approach could be useful for more sophisticated use cases, such as integrating data collecting *scripts/applications* with prediction instances. Furthermore, it should provide better performance than REST API.

To access TensorFlow Serving gRPC API of running prediction instance, the Kubernetes port forwarding mechanism must be used. Create a port forwarding tunnel to a prediction instance with following command:

```
kubectl port-forward service/<prediction instance name> :8500
```

Or, if you want to start port forwarding tunnel in background:

```
kubectl port-forward service/<prediction instance name> <some local port
number>:8500 &
```

Note: The local port number of the tunnel you entered above; it will be produced by `kubectl port-forward` if you do not explicitly specify it.

You can access the gRPC API by using a dedicated client gRPC client (such as the following GitHub Python script: mnist_client.py). Alternatively, use gRPC CLI client of your choice (such as: Polyglot and/or gRPC) and connect to: `localhost:<local tunnel port number>`.

Useful External References

- Serving a TensorFlow Model
- TensorFlow Serving with Docker
- RESTful API

OpenVINO Model Server Overview

The OpenVino Model Server (OVMS) is an OpenVINO serving component intended to provide hosting for the OpenVINO inference runtime.

OVMS is an external API that is fully compatible with TF Serving providing an alternative prediction solution for Nauta users. OpenVINO provides one of the most performant inference solutions available on Intel platforms. In many cases (especially in small batch scenarios) it outperforms other inference engines including TF Serving.

OVMS does, however have a limited number of supported topologies and therefore cannot be used as the sole inference runtime on Nauta. It will be employed as an option for models that meet OpenVINO requirements.

Refer to the OpenVINO White Paper for more information.

Inference on Models Served by the OpenVINO Model Server

To perform a batch or stream inference with OVMS, a model in OpenVINO format is required. To obtain an MNIST model converted to OVMS format (shown in this example), refer to ***Exporting Models***, page 63 for complete instructions.

Mount the Input Directory and Copy OVMS Compatible Model

1. Mount the Nauta input directory via NFS using the following Mounting Experiment Input to Nauta Storage, page 42 instructions.
2. Copy the OVMS compatible model to the input directory.

Models Structure in the Input Directory

Place and mount the Models in a directory structure, as depicted in the example below.

```
models/
├── model1
│   ├── 1
│   │   ├── ir_model.bin
│   │   └── ir_model.xml
│   └── 2
│       ├── ir_model.bin
│       └── ir_model.xml
└── model2
    └── 1
        ├── ir_model.bin
        ├── ir_model.xml
        └── mapping_config.json
```

In case of MNIST model conversion with model export command, a directory storing one version of the model is created. Due to prediction prerequisite, the model directory structure must meet the following structure requirements:

```
models/
└── <directory from model export output>
    └── 1
        ├── saved_model.bin
        ├── saved_model.mapping
        └── saved_model.xml
```

Stream Inference

When the correct model structure is prepared, run model server instance with:

```
nctl predict launch -n ovmsexample --runtime ovms --model-location
/mnt/input/home/models/mnist
```

When `nctl predict list` reports the prediction as running (a partial example is shown below):

```
| Prediction instance   | Parameters              | Submission date        |
| Ovmsexample           | 2019-07-29 03:01:08 PM  | 2019-07-29 03:01:12 PM |
```

Perform stream inference by executing the following command:

```
nctl predict stream --name ovmsexample --data input.json
```

An example content of an `input.json` file can be found in examples of nctl located in:

```
<nctl_directory>/examples/ovms_inference
```

For `input.json` file delivered in the example result of the stream inference, it will be:

```
{"predictions": [[0.0006329981843009591, 1.111995175051561e-06,
0.00018445802561473101, 0.08759918063879013, 1.9286260055650928e-07,
0.9085237383842468, 2.53505368164042e-05, 0.0012352498015388846,
0.0017150170169770718, 8.265616634162143e-05]]}
```

The output of the prediction, in case of MNIST digit recognition model is a vector of 10 elements. This is the *Index* the vector that has highest value, and represents predicted class. In this case, the highest value was reported at *Index 5*, which corresponds to class of 'five' digits.

Note: Similar JSON files can be generated with python script in: `<nctl_directory>/examples/ovms_inference`, as shown in the example below.

```
cd <nctl_directory>/examples/ovms_inference
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

When venv is prepared and invoked, it appears as:

```
python generate_json.py --image_id <IMAGE ID>
```

The `IMAGE ID` argument determines with a picture from MNIST what will be used.

Batch Prediction

To perform batch prediction, generate the correct protobuffers for inference. This step is similar to MNIST Data Preprocessing, but with one difference. During the model conversion to an OV format, some of the model signatures information is missing. To perform prediction on the converted MNIST model, one additional parameter has to be used with `mnist_converter_pb.py`:

```
mnist_converter_pb.py --model_input_name="x/placeholder_port_0"
```

After file generation, move the directory that contains the `.pb` files to the `/input` mount point.

When all files are prepared, schedule a prediction with the following command:

```
nctl predict batch -n ovmsbatch -rt ovms --model-location
/mnt/input/home/models/mnist --data /mnt/input/home/ovms_inference
```

Note: The above command assumes that `.pb` files are stored in the `ovms_inference` directory in the `/input` shared folder.

When a batch prediction reaches the `FINISHED` state (as shown in the example), it displays the results (a partial example is shown below).

```
| Prediction instance         | Parameters | Submission date        |
| mnist-526-19-07-30-00-34-07 |            | 2019-07-30 12:34:35 AM |
```

To understand these results of the `/output` mount point, refer to Working with Datasets, page 31.

OVMS Prediction with Local Model

Nauta platform models can also be forwarded without the `/input` mount. This can be performed with the `--local-model-location` option.

```
nctl predict launch -n localovms --runtime ovms --local-model-location
/tmp/models/mnist/
```

Note: The above command assumes the MNIST in the OV format is stored in the `/tmp/models/mnist` folder.

Managing Users and Resources

This section discusses the following topics:

Creating a User Account

The user is the *Data Scientist* who performs deep learning experiments to train models that will, after training and testing, be deployed in the field. Creating a new user account creates a user account configuration file compliant in format with `kubectl` configuration files.

Experiments and User Access

The user has full control (`list/read/create/terminate`) over their own experiments, as well as read access (`list/read`) to experiments belonging to other users on this cluster.

Note: Only an Administrator *can create* a user account.

User Name Limitations

Users with the same name *cannot* be created directly after being removed. This is due to a user's related Kubernetes objects that are deleted asynchronously by Kubernetes and this can take some time. Consider waiting 10 minutes before creating a user with the same name.

In addition, user names are limited to a 32-character maximum and there are no special characters except for hyphens. However, all names *must start* with a letter. You can use a hyphen to join user names, for example: john-doe.

Create the User

Execute the following steps to create a user:

1. The `nctl user create <username>` command sets up a namespace and associated roles for the named user on the cluster. Furthermore, this command sets up home directories, named after the username, on the `input` and output network shares with the file-system level access privileges. Create the user:

   ```
   nctl user create <username>
   ```

2. The command above also creates a configuration file named `<username>.config` that the Admin provides to the user. The user then copies that file into a local folder.

3. Use the export command to set this variable for the user:

   ```
   export KUBECONFIG=/<local_user_folder>/<username>.config
   ```

4. Verify that the new user has been created with the following command:

   ```
   nctl user list
   ```

   The command above lists all users, including the new user just added. A partial example is shown below.

   ```
   | Name    | Creation date          | Date of last submitted job   |
   |---------+------------------------+------------------------------+
   | user1   | 2019-03-12 08:30:45 PM | 2019-02-27 07:55:13 PM       |
   | user2   | 2019-03-12 09:50:50 PM |                              |
   | user3   | 2019-03-12 09:51:31 PM |                              |
   ```

Deleting a User Account

Only an Administrator can delete user accounts. Deleting a user removes that user's account from the Nauta software and removes log in access to the system. The command halts and removes all experiments and pods; however, all artifacts related to that user's account, such as, the user's input and output folders and all data related to past experiments remains.

Removing a User

Execute the following command to remove a user:

```
nctl user delete <username>
```

Respond to this question to confirm the previous step.

```
Do you want to continue? [y/N]: Press y to confirm deletion.
```

Limitations

The command may take up to 30 seconds to delete the user. You may receive the message: `User is still being deleted`. Check the status of the user after a few minutes. Recheck as desired.

Using the purge Command

Use this command to permanently remove (*Purge*) all artifacts associated with the user, including all data related to past experiments submitted by that user (but excluding the contents of the user's input and output folders):

Purging Process

Execute the following command to purge a user:

```
nctl user delete <username> --purge
```

Respond to this question to confirm the previous step.

```
Do you want to continue? [y/N]: Press y to confirm deletion.
```

Limitations

The Nauta `user delete` command may take up to 30 seconds to delete the user. A new user with the same username *cannot* be created until after the delete command confirms that the first user with the same name has been deleted.

Viewing All User Activity

Use the `nctl user list` command to display all current users, as well as all of their experiments (with status). Furthermore, the command displays the following information:

- **Name:** user name
- **Creation date**: the date this user account was created
- **Date of last submitted job**: experiment
- **Number of running jobs**: experiments
- **Number of queued jobs:** experiments submitted, but not yet running

Administrators *are not* listed and previously deleted users *are not* shown. To *create* a user account, refer to Creating a User Account, page 77 and to *delete* a user account, refer to Deleting a User Account, page 78.

Execute the following command:

```
nctl user list
```

A partial example of the results is shown below.

```
| Name     | Creation date          | Date of last submitted job   |
|---------+------------------------+------------------------------+
| user1    | 2019-03-12 08:30:45 PM |  2019-03-02 05:25:14 PM      |
| user2    | 2019-03-12 09:50:50 PM |                              |
| user3    | 2019-03-12 09:51:31 PM |                              |
```

Kubernetes Resource Dashboard Overview

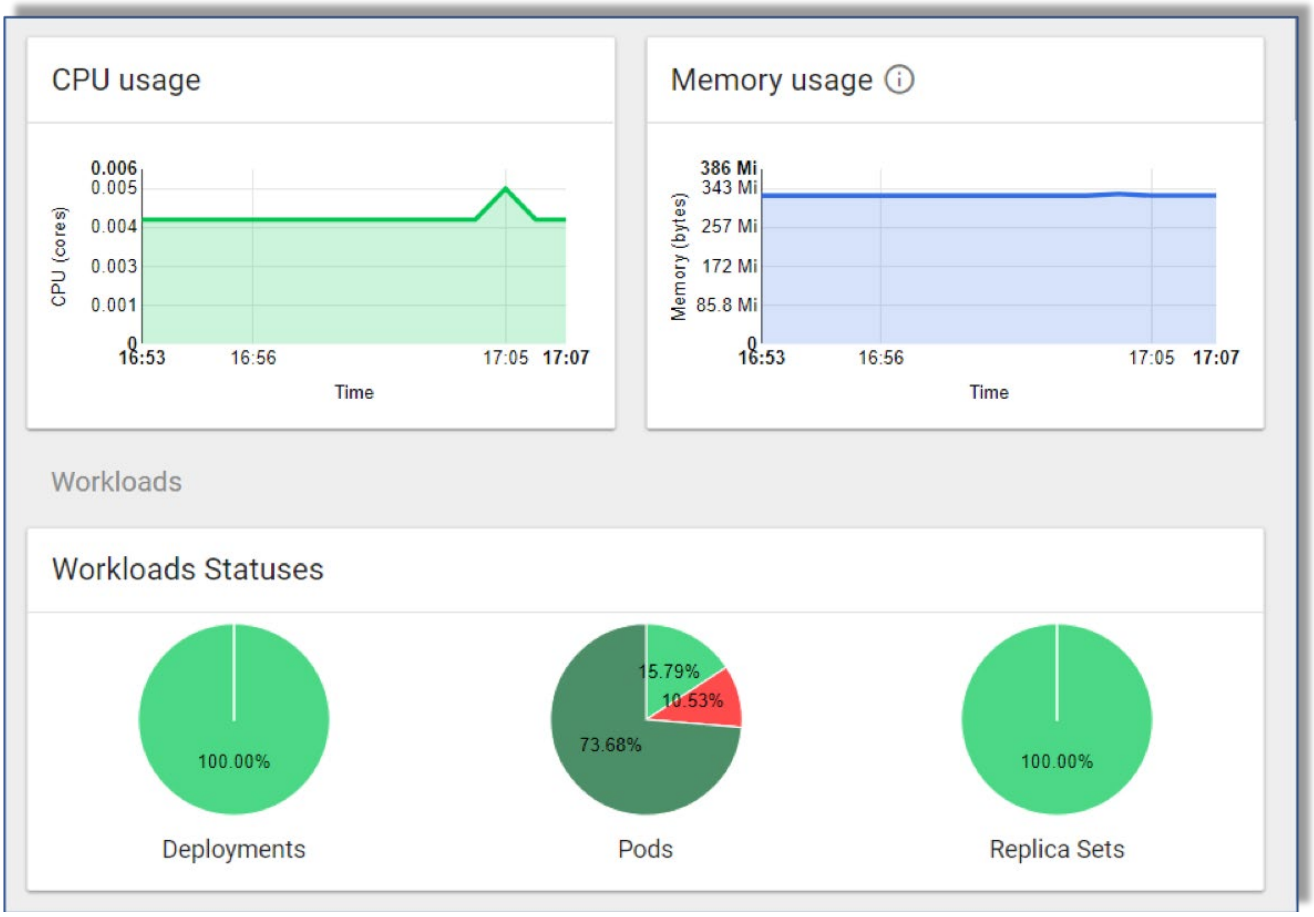
Kubernetes provides a way to manage containerized workloads and services, to manage resources given to a particular experiment and monitor workload statuses and resource consumption. Refer to Kubernetes Web UI (Dashboard) for detailed Kubernetes information.

To access Kubernetes:

1. Click the **Hamburger Menu** at the far left of the UI to open a left frame.
2. Click **Resources Dashboard** to open the Kubernetes resources dashboard in a new browser window/tab.

Figure 12 shows an example Kubernetes Dashboard.

Figure 12: Kubernetes Dashboard—Example Only



CLI Commands

The `--help` command provides man-page style help for each nctl command. You can view help for any command and subcommand, and all related parameters, on the next page: Viewing CLI Command Help, page 82.

This section discusses the following CLI Commands and Subcommand topics:

Viewing the CLI Commands Help

View man-page style help for any command and subcommand, and all related parameters by using the `--help` option.

nctl Help Commands Overview

Entering `nctl --help` or `nctl -h` provides a listing of all nctl commands (without subcommands), as shown below.

```
nctl -h
```

```
Usage: nctl COMMAND [options] [args]...

  Nauta Client

  Displays additional help information when the -h or --help COMMAND is
  used.

Options:
  -h, --help  Displays help messaging information.

Commands:
  config, cfg      Set limits and requested resources in templates.
  experiment, exp  Start, stop, or manage training jobs.
  launch, l        Launch the web user-interface or TensorBoard. Runs as a
                   process in the system console until the user stops the
                   process. To run in the background, add '&' at the end of
                   the line.
  model, mo        Manage the processing, conversion, and packaging of models.
  mount, m         Displays a command that can be used to mount a client's
                   folder on their local machine.
  predict, p       Start, stop, and manage prediction jobs and instances.
  template, tmp    Manage experiment templates used by the system.
  user, u          Create, delete, or list users of the platform. Can only be
                   run by a platform administrator.
  verify, ver      Verifies if all required external components contain the
                   proper installed versions.
  version, v       Displays the version of the installed nctl application.
```

You can view command-help for any command and available subcommand(s). The following example shows generic syntax; brackets are optional parameters, but a `[subcommand]` requires the `[command]`.

```
nctl [command_name] [subcommand] --help
```

config Command

Use the `config` command to adjust a packs' settings to the resources available on a cluster.

Synopsis

This command allows you to change the current system's settings concerning maximum and requested resources used by training jobs initiated by Nauta. The command takes the CPU number and the memory amount provided (by you) and calculates new values.

This calculation preserves the same coefficient between available resources and resources defined in every template, as it was before the execution of this command.

Syntax

```
nctl config [options]
```

Options

Name	Required	Description
`-c, --cpu`	Yes	This is the number of CPUs available on a cluster's node with the lowest number of CPU. Value should be given in format accepted by k8s. This can be a plain number or a number followed by 'm' suffix.
`-m, --memory`	Yes	This is the amount of a memory available on a cluster's node with the lowest amount of memory. Value should be given in format accepted by k8s. This can be a plain number or a number followed by a one of the following suffixes: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Note: The number of CPUs shown should be interpreted according to the following article: Meaning of CPU and the amount of memory given here should be interpreted according to the following article: Meaning of Memory.

Returns

In case of any problems, a message describing the cause/causes of the issues displays. Otherwise, message is returned indicating success.

Example

```
nctl config --cpu 10 --memory 8Gi
```

Outcome

This Calculates resources' settings for all packs installed together with `nctl` application. It assumes, that the maximal available number of CPU on a node is 10 and that this node provides 8Gb of RAM. Furthermore, limited and requested resources are calculated using those maximal values.

experiment Command

Use the `experiment` command to submit and manage experiments. This main command also includes the following subcommands:

- submit Subcommand, page 85
- list Subcommand, page 88
- cancel Subcommand, page 90
- experiment Command, page 84
- logs Subcommand, page 92
- interact Subcommand, page 94

submit Subcommand

Synopsis

Use the `submit` subcommand to submit training jobs. Use this command to submit single and multi-node training jobs (by passing `-t` parameter with a name of a multi-node pack), and many jobs at once (by passing `-pr`/`-ps` parameters).

Syntax

```
nctl experiment submit [options] SCRIPT-LOCATION [-- script-parameters]
```

Arguments

Name	Required	Description
`SCRIPT-LOCATION`	Yes	Location and name of a Python script with a description of training.
`script-parameters`	No	String with a list of parameters that are passed to a training script. All such parameters should be added at the end of command after "--" string.

Options

Name	Required	Description
`-sfl, --script_folder_location <folder_name> PATH`	No	Location and name of a folder with additional files used by a script, for example: other .py files, data, and so on. If not given, then its content *will not* be copied into the Docker image created by the `nctl submit` command. `nctl` copies all content, preserving its structure, including subfolder(s).
`-t, --template <template_name> TEXT`	No	Name of a template that will be used by `nctl` to create a description of a job to be submitted. If not given, a default template for single node TensorFlow training is used (`tf-training`). List of available templates can be obtained by issuing `nctl template list` command.
`-n, --name TEXT`	No	Name assigned to nctl template list.
`-p, --pack-param <TEXT TEXT>…`	No	Additional pack parameter in format: `'key value'` or `'key.subkey.subkey2 value'`. For maps use: `'key "['val1', 'val2']"' For maps use: 'key "{'a': 'b'}"'`

Name	Required	Description

`-pr, --parameter-range TEXT…`
`[definition]`
`<TEXT TEXT>…`
	No	If the parameter is given, `nctl` starts as many experiments as there is a combination of parameters passed in `-pr` options. Optional `[param-name]` is a name of a parameter that is passed to a training script. `[definition]`
		
		Contains values of this parameter that are passed to different instance of experiments. `[definition]` can have two forms:
		• **range:** `{x...y:step}` This form says that `nctl` will launch a number of experiments equal to a number of values between x and y (including both values) with `step`.
		• **set of values:** `{x, y, z}` This form says that `nctl` will launch number of experiments equal to a number of values given in this definition.

`-ps, --parameter-set [definition] TEXT`	No	If this parameter is given, `nctl` launches an experiment with a set of parameters defined in `[definition]` argument. An optional format of the [definition] argument is: `{[param1_name]: [parameter1_value], [parameter2_name]: [parameter2_value], ..., [paramn_name]:[paramn_value]}`. All parameters given in the `[definition]` argument will be passed to a training script under their names stated in this argument. If the `-ps` parameter is given more than once, then `nctl` will start as many experiments as there is occurrences of this parameter in a call.
`-e, --env TEXT`	No	This is the environment variable passed to training. You can pass as many environmental variables, as desired. Each variable should be passed as a separate `-e parameter`.
`-r, --requirements PATH`	No	This is the path to the file with experiment's pip requirements. Dependencies listed in this file will be automatically installed using pip.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Additional Remarks

For both types of parameters: `-ps` and `-pr`; if, the parameter stated in their definitions is also given in a `[script_parameters]` argument of the `nctl` command, then values taken from `-ps` and `-pr` are passed to a script.

If a combination of both parameters is given, then `nctl` launches a number of experiments equal to combination of values passed in those parameters. For example, if the following combination of parameters is passed to `nctl` command:

```
-pr param1 "{0.1, 0.2, 0.3}" -ps "{param2: 3, param4: 5}" -ps "{param6: 7}"
```

Then the following experiments will be launched:

```
param1 = 0.1, param2 = 3, param4 = 5, param6 - not set
param1 = 0.2, param2 = 3, param4 = 5, param6 - not set
param1 = 0.3, param2 = 3, param4 = 5, param6 - not set
param1 = 0.1, param2 = not set, param4 = not set, param6 - 7
param1 = 0.2, param2 = not set, param4 = not set, param6 - 7
param1 = 0.3, param2 = not set, param4 = not set, param6 - 7
```

Returns

This command returns a list of submitted experiments with their names and statuses. In case of problems during submission, the command displays message/messages describing the causes. Errors may cause some experiments *to not be* created and will be empty. If any error appears, then messages describing it are displayed with experiment's names/statuses.

If one or more of experiment *has not* been submitted successfully, then the command returns an exit code: `> 0`. The exact value of the code depends on the cause of error(s) that prevented submitting the experiment(s).

Example

```
nctl experiment submit --name para-range --parameter-range lr "{0.1, 0.2, 0.3}"
examples/mnist_single_node.py -- --data_dir=/mnt/input/root/public/MNIST
```

Starts multiple single node training jobs using `mnist_single_node.py` script located in the examples folder. Each training job uses a different learning rate value.

list Subcommand

Synopsis

Use the `list` subcommand to display a list of all experiments with some basic information for each, regardless of the owner. Results are sorted using the *date-of-creation* of the experiment, starting with the most recent experiment.

Syntax

```
nctl experiment list [options]
```

Options

Name	Required	Description
`-a, --all_users`	No	List contains experiments submitted by of all users.
`-n, --name TEXT`	No	A regular expression to filter list to experiments that match this expression.
`-s, --status`	No	QUEUED, RUNNING, COMPLETE, CANCELLED, FAILED, CREATING - Lists experiments based on indicated status.
`-u, --uninitialized`	No	List uninitialized experiments, that is, experiments without resources submitted for creation.
`-c, --count INTEGER RANGE`	No	An integer, command displays c last rows.
`-b, --brief`	No	Print short version of the result table. Only `'name'`, `'submission date'`, `'owner'` and `'state'` columns will be printed.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Displays a list of experiments matching a criteria given in the command's options. Each row contains the experiment name and additional data of each experiment, such parameters used for this certain training, time and date when it was submitted, name of a user which submitted this training and current status of an experiment. Below is an example returned by this command (the brief option is shown).

```
| Experiment                          | Submission date        | Owner | State     |
|-------------+----------------------+---------+-----------------------------|
| mnist-single-node-tb                | 2019-03-13 04:57:58 PM | user1 |  QUEUED   |
| mnist-tb                            | 2019-03-13 05:00:39 PM | user1 |  COMPLETE |
| mnist-tb 2-1                        | 2019-03-13 05:49:59 PM | user1 |  COMPLETE |
| test-experiment                     | 2019-03-13 06:00:39 PM | user1 |  QUEUED   |
| single-experiment                   | 2019-03-13 01:49:59 PM | user1 |  QUEUED   |
```

Examples

The following command displays all experiments submitted by a current user.

```
nctl experiment list
```

The following command displays all experiments submitted by a current user and with name starting with `train` word.

```
nctl experiment list -n train
```

cancel Subcommand

Synopsis

Use the `cancel` subcommand to cancel any training chosen based on provided parameters.

Syntax

```
nctl experiment cancel [options] NAME
```

Arguments

Name	Required	Description
`NAME`	Yes	The name of an *experiment/pod/status* of a pod to be cancelled. If any such an object is found, the command displays question whether this object should be cancelled.

Options

Name	Required	Description
`-m, --match TEXT`	No	If given, the command searches for experiments matching the value of this option. This option *cannot* be used along with the `NAME` argument.
`-p, --purge`	No	When used, all information concerning experiments is removed from the system.
`-i, --pod-ids TEXT`	No	Comma-separated pods IDs. If given, command matches pods by their IDs and deletes them.
`-s, --pod-status TEXT`	No	One of: `PENDING`, `RUNNING`, `SUCCEEDED`, `FAILED`, or `UNKNOWN`. If given, the command searches pods by their status and deletes them.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

The description of a problem; if, any problem occurs. Otherwise, displays the information that training job/jobs was/were cancelled successfully.

Example

```
nctl experiment cancel t20180423121021851
```

Outcome

This cancels the experiment with `t20180423121021851` name, as shown in the example.

view Subcommand

Synopsis

Use the `view` subcommand to display basic details of an experiment, such as the name of an experiment, parameters, submission date, and so on.

Syntax

```
nctl experiment view [options] EXPERIMENT-NAME
```

Arguments

Name	Required	Description
`EXPERIMENT-NAME`	Yes	Name of an experiment to be displayed.

Options

Name	Required	Description
`-tb, --tensorboard`	No	If given, the command exposes a TensorBoard instance with an experiment's data.
`-u, --username TEXT`	No	Name of the user who submitted this experiment. If not given, then only experiments of a current user are shown.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Displays details of an experiment. If `-tb, --tensorboard` option is given, then the command also returns a link to a TensorBoard's instance with data from the experiment.

Example

```
nctl experiment view experiment-name-2 -tb
```

Displays details of an `experiment-name-2` experiment and exposes TensorBoard instance with experiment's data to a user.

logs Subcommand

Synopsis

Use the `logs` subcommand to display the logs from experiments. Logs to be displayed are chosen based on parameters given in command's call.

Syntax

```
nctl experiment logs [options] EXPERIMENT-NAME
```

Arguments

Name	Required	Description
`EXPERIMENT-NAME`	Yes	Displays the name of experiment logs.

Options

Name	Required	Description
`-s, --min-severity`	No	Minimal severity of logs. Available choices are: • `CRITICAL` - Displays only CRITICAL logs • `ERROR` - Displays ERROR and CRITICAL logs • `WARNING` - Displays ERROR, CRITICAL and WARNING logs • `INFO` - Displays ERROR, CRITICAL, WARNING and INFO • `DEBUG` - Displays ERROR, CRITICAL, WARNING, INFO and DEBUG
`-sd, --start-date`	No	Retrieve logs produced from this date (format ISO-8061 - yyyy-mm-ddThh:mm:ss)
`-ed, --end-date`	No	Retrieve logs produced until this date (format ISO-8061 - yyyy-mm-ddThh:mm:ss)
`-i, --pod-ids TEXT`	No	Comma-separated pods IDs. If given, then matches pods by their IDs and only logs from these pods from an experiment with `EXPERIMENT_NAME` name will be returned.
`-p, --pod-status TEXT`	No	One of: `'PENDING'`, `'RUNNING'`, `'SUCCEEDED'`, `'FAILED'`, or `'UNKNOWN'` commands returns logs with matching status from an experiment and matching `EXPERIMENT-NAME`.
`-m, --match TEXT`	No	If given, this command searches for logs from experiments matching the value of this option. This option *cannot* be used along with the `NAME` argument.
`-o, --output`	No	If given, the logs are stored in a file with a name derived from a name of an experiment.
`-pa, --pager`	No	Display logs in interactive pager. Press *q* to exit the pager.
`-fl, --follow`	No	Specify if logs should be streamed. **Note:** Only logs from a single experiment can be streamed.

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Should issues arise, a message (or messages) with a description of their cause (or causes) displays. Otherwise, the logs are filtered based on command's parameters.

Example

```
nctl experiment logs experiment-name-2 --min-severity DEBUG
```

Displays logs from `experiment-name-2` experiment with severity `DEBUG` and higher (`INFO`, `WARNING`, and so on).

interact Subcommand

Synopsis

Use the `interact` subcommand to launch a local browser with a Jupyter notebook. If a script's name is given as a parameter of the command, then this script is displayed in a notebook.

Syntax

```
nctl experiment interact [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	No	The name of a Jupyter notebook's session. If session with a given name already exists, then you are connected to this session.
`-fl, --filename TEXT`	No	The file with a notebook shat should be opened in Jupyter notebook.
`-p, --pack-param <TEXT TEXT>...`	No	Additional pack parameter in format: `'key value'` or `'key.subkey.subkey2 value'`. • For lists use: `'key "['val1', 'val2']"'` • For maps use: `'key "{'a': 'b'}"'`
`--no-launch`	No	Run this command without a web browser starting, only proxy tunnel is created.
`-pn, --port-number INTEGER RANGE`	No	Port on which service will be exposed locally.
`-e, --env TEXT`	No	Environment variables passed to Jupyter instance and you can pass as many environmental variables as is desired. Each variable should be passed as a separate -e parameter.
`t, --template [jupyter,jupyter-py2]`	No	Name of a Jupyter notebook template used to create a deployment. Supported templates for interact command are: jupyter (python3) and jupyter-py2 (python2).
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Should issues arise, a message (or messages) with a description of their cause (or causes) displays. Otherwise, the command launches a default web browser with a Jupyter notebook, and displays the address under which this session is provided.

Example

Launches in a default browser a Jupyter notebook with `training_script.py` script.

```
nctl experiment interact --filename training_script.py
```

launch Command

Synopsis

Use the `launch` command launches a browser for Web UI or TensorBoard. This main command also includes the following subcommands:

- webui Subcommand, page 96
- tensorboard Subcommand, page 98

webui Subcommand

Synopsis

The `webui` subcommand launches the Nauta web user interface with credentials.

Note: If you are using CLI through remote access, you will need to setup an X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, you can use the URL provided by this command.

Syntax

```
nctl launch webui [options]
```

Arguments

None.

Options

Name	Required	Description
`--no-launch`	No	Run this command without a web browser starting; only proxy tunnel is created.
`-pn, --port-number INTEGER RANGE`	No	If given, the application will be exposed on a local machine under [port] port.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Link to an exposed application.

Example

```
nctl launch webui
```

This command returns a Go to URL. The following is an example only:

```
Launching...Go to http://localhost:14000?token=eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrd
WJlcm5ldVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uY
W1lc3BhY2UiOiJiZXRoYW55Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQu
NlcnZpY2Ut…
Proxy connection created.
Press Ctrl-C key to close a port forwarding process...
```

tensorboard Subcommand

Synopsis

Use the `tensorboard` subcommand to launch the TensorBoard web user interface front-end with credentials, with the indicated experiment loaded.

Note: If you are using CLI through remote access, you will need to setup an X server for tunneling over SSH with port forwarding or use SSH Proxy command tunneling. After establishing a tunnel from the gateway to your local machine, you can use the URL provided by this command.

Syntax

```
nctl launch tensorboard [options] EXPERIMENT-NAME
```

Arguments

Name	Required	Description
`EXPERIMENT-NAME`	Yes	Experiment name.

A user can pass one or more names of experiments separated with spaces. If an experiment that should be displayed in TensorBoard belongs to a current user, the user has to give only the name. If this experiment is owned by another user, the name of an experiment should be preceded with a name of this second user in the following format: `username/experiment-name`.

Options

Name	Required	Description
`--no-launch`	No	To create tunnel without launching web browser.
`-tscp, --tensorboard-service-client-port`	No	Local port on which TensorBoard service client will be started.
`-pn, --port INTEGER RANGE`	No	Port on which service will be exposed locally.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Link to an exposed application.

Examples

```
nctl launch tensorboard experiment75
```

An example might appear as:

```
http://127.0.0.1/tensorboard/token=AB123CA27F
```

model Command

Use the `model` command to manage model export related tasks. The Following are the subcommands for the `nctl model` command. This main command also includes the following subcommand. This main command also includes the following subcommands:

- status Subcommand, page 101
- export Subcommand, page 102
- logs Subcommand, page 104

status Subcommand

Synopsis

Displays a list of model export operations with their statuses, dates of start and finish, and the users who submitted those operations.

Syntax

```
nctl model status [options]
```

Options

Name	Required	Description
`-u, --username`	No	Name of a user to whom viewed operations belongs. If not given, only models of a current user are taken into account.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

This command displays a list of operations with their statuses, dates of start and finish, and displays the users who submitted those operations.

Example

```
nctl model status
```

Output

```
| Operation              | Start date           | End date             | Owner      | State     |
|------------------------------------------------+----------------------+------------------|
| openvino_1             | 2019-07-15T11:59:56Z | 2019-07-15T12:00:02Z | jdoe       | Succeeded |
| openvino_2             | 2019-07-02T14:39:38Z | 2019-07-02T14:39:47Z | jdoe       | Failed    |
| openvino_3             | 2019-07-16T14:29:54Z | 2019-07-16T14:30:00Z | jdoe       | Succeeded |
```

Displays details of 3 model export operations: two of them finished with success, one failed.

export Subcommand

Synopsis

Exports an existing model located in the `PATH` folder to a given `FORMAT` with given options. If the `formats` option is given, it displays a list of available export formats.

Syntax

```
nctl model export PATH/formats FORMAT [-- operation options]
```

Arguments

Name	Required	Description
`PATH/formats`	Yes	`PATH` - The location of a model that is going to be exported. Models can be stored only in shared folders. Furthermore, this command *does not* handle models located in local folders. The formats command (if given) displays a list of available formats.
`FORMAT`	No	This is the format of an exported model. A list of available formats can be obtained by executing the `export-list` command. **Note:** This is required if `PATH` has been given.
`operation options`	No	The string with a list of parameters that are passed to a workflow; this string is responsible for exporting a model. Add these parameters at the end of the command after: `--` string.

Options

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

This command displays a list of operations with their statuses, dates of start and finish, and displays the

Returns

Should issues occur, a message (or messages) with a description of their cause (or causes) displays. If an export's operation starts, then the related operation information and its details displays. If the `formats` option is given, a list of available export formats with a short description of parameters accepted by those formats displays.

Example 1

Wait for Tensorboard to run.

```
nctl model export /mnt/input/home/pretrained_model openvino -- --output ArgMax
```

Output 1

```
| Operation      | Start date            | End date    | Owner      | State    |
|---------------+----------------------+------------+-----------+---------|
| openvino_1     | 2019-07-17T16:13:40Z |             | jdoe       | Queued   |

Successfully created export workflow
```

Example 2

Exports an existing model located in the `pretrained_model` folder in `input` shared folder.

```
nctl model export formats
```

Output 2

Displays a list of available export formats with a short description of parameters accepted by them.

```
| Name     | Parameters description                                                          |
|----------+---------------------------------------------------------------------------------|
| openvino | --input_shape [x,y,....] - shape of an input                                    |
|          | --input [name] - names of input layers                                          |
|          | --output [name] - names of output layers                                        |
|          | Rest of parameters can be found in a description of OpenVino model optimizer |
```

logs Subcommand

Synopsis

The `logs` subcommand displays logs from a model export operation. Logs to be displayed are chosen based on parameters given in the command's call.

Syntax

```
nctl model logs [options] OPERATION-NAME
```

Arguments

Name	Required	Description
`OPERATION-NAME`	Yes	Name of a user to whom viewed operations belong.

Options

Name	Required	Description
`-sd, --start-date`	No	Retrieves the logs produced from this date (format ISO-8061 - yyyy-mm-ddThh:mm:ss).
`-ed, --end-date`	No	Retrieves the logs produced until this date (format ISO-8061 - yyyy-mm-ddThh:mm:ss).
`-m, --match TEXT`	No	If given, this command searches for logs from operations matching the value of this option. This option cannot be used along with the OPERATION-NAME argument.
`-o, --output`	No	If given, the logs are stored in a file with a name derived from a name of an experiment.
`-pa, --pager`	No	Displays the logs in interactive pager. Press q to exit the pager.
`-fl, --follow`	No	Specifies if the logs should be streamed. Only logs from a single experiment can be streamed.
`-f --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Should issues occur, a message (or messages) containing a description of their cause (or causes) displays. Otherwise, the logs are filtered based on command's parameters.

Example

```
nctl model logs openvino_2
```

Output

Displays logs from `openvino_2` model export operation.

mount Command

Use the `mount` command to display the operating system commands for mounting and unmounting Nauta folders. This main command also includes the following subcommand:

- list Subcommand, page 107

Note: *mount* is an operating system command so it might be better to continue using `nctl mount` here. The command displays both the mount and unmount commands.

Synopsis

The `mount` command by itself displays another command that can be used to mount/unmount a client's folders on or from a user's local machine.

Syntax

```
nctl mount [options]
```

Options

Name	Required	Description
`-f --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

This command returns another command that can be used to mount a client's folders on a user's local machine. It also shows what command should be used to unmount client's folder after it is no longer needed.

list Subcommand

Synopsis

Use the `list` subcommand to display a list of Nauta related folders mounted on a user's machine. If run using administrator credentials, it displays mounts of all users.

Syntax

```
nctl mount list
```

Options

Name	Required	Description
`-f --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

List of mounted folders. Each row contains additional information (for example: remote and local location) concerning those mounts. Set of data displayed by this command depends on operating system.

Additional Remarks

This command displays only those mounts that expose Nauta shares. Other mounted folders *are not* taken into account.

predict Command

Use the `predict` command to start, stop, and manage prediction jobs. This main command also includes the following subcommands:

batch Subcommand

Synopsis

Use the `batch` subcommand to start a new batch instance that performs prediction on provided data. This command uses a specified dataset to perform inference. The results are stored in an output file.

Syntax

```
nctl predict batch [options]
```

Options

Name	Required	Description
`-n, --name`	No	Name of predict session.
`-m, --model-location TEXT`	Yes	The path to saved model that will be used for inference. The model must be located on one of the input or output system shares (for example: `/mnt/input/saved_model`). The model content will be copied into an image.
`-l, --local_model_location PATH`	Yes	The local path to saved model that will be used for inference. The model content will be copied into an image.
`-d, --data TEXT`	Yes	Location of a folder with data that will be used to perform the batch inference. The value should point out the location from one of the system's shared folder.
`-o, --output TEXT`	No	The location of a folder where outputs from inferences will be stored. Value should point out the location from one of the system's shared folder.
`-p, --pack-param <TEXT TEXT>…`	No	Additional pack parameter in format: `'key value'` or `'key.subkey.subkey2 value'`. For maps use: `'key "['val1', 'val2']"' For maps use: 'key "{'a': 'b'}"'`
`-mn, --model-name`	No	The name of a model passed as a servable name. By default, it is the name of directory in model's location.
`-tr, --tf-record`	No	If given, the batch prediction accepts files in `TFRecord` formats. Otherwise, files should be delivered in *`protobuf`* format.
`-r, --requirements FILE`	No	Path to file with experiment's pip requirements. Dependencies listed in this file will be automatically installed using pip.
`rt, --runtime [tfserving|ovms]`	No	Determine runtime for prediction. Supported runtimes are 'Tensorflow serving' (tfserving) and 'OpenVINO Model Server (ovms). Default runtime is 'tfserving'.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG

`-h, --help`	No	Displays help messaging information.

Returns

Description of a problem, if any occurs. Otherwise, displays the information that the predict job was submitted.

cancel Subcommand

Synopsis

Use the `cancel` subcommand for prediction instance(s) chosen based on criteria given as a parameter.

Syntax

```
nctl predict cancel [options] [name]
```

Arguments

Name	Required	Description
`NAME`	No	Name of predict instance to be cancelled. [name] argument value can be empty when `match` option is used.

Options

Name	Required	Description
`-m, --match`	No	If given, the command searches for prediction instances matching the value of this option.
`-p, --purge`	No	If given, , then all information concerning all prediction instances, completed and currently running, are removed from the system.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

The description of a problem; if, any problem occurs. Otherwise, information that training job/jobs was/were cancelled successfully.

launch Subcommand

Synopsis

Use the `launch` subcommand starts a new prediction instance that can be used for performing prediction, classification and regression tasks on trained model. The created prediction instance is for streaming prediction only.

Syntax

```
nctl predict launch [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	No	The name of this prediction instance.
`-m, --model-location TEXT`	Yes	The path to saved model that will be used for inference. Model must be located on one of the input or output system shared folder (`e.g. /mnt/input/home/saved_model`).
`-l, --local_model_location PATH`	No	The local path to saved model that will be used for inference. Model content will be copied into an image.
`-mn, --model-name TEXT`	No	The name of a model passed as a servable name. By default, it is the name of directory in model's location.
`-p, --pack-param <TEXT TEXT>…`	No	Additional pack parameter in format: `'key value'` or `'key.subkey.subkey2 value'`. For maps use: `'key "['val1', 'val2']"' For maps use: 'key "{'a': 'b'}"'`
`-r, --requirements FILE`	No	Path to file with experiment's pip requirements. Dependencies listed in this file will be automatically installed using pip.
`-rt, --runtime [tfserving|ovms]`	No	Determine runtime for prediction. Supported runtimes are 'Tensorflow serving' (tfserving) and 'OpenVINO Model Server (ovms). Default runtime is 'tfserving'.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

Prediction instance URL and authorization token, as well as information about the experiment (name, model location, state).

```
nctl predict l -n test -m /mnt/input/home/experiment1

    | Prediction instance   | Model Location               | Status   |
    |-----------------------+------------------------------+----------|
    | test                  | /mnt/input/home/experiment1  | QUEUED   |

    Prediction instance URL (append method verb manually, e.g. :predict):
    https://192.168.0.1:8443/api/v1/namespaces/jdoe/services/test/proxy/v1/models/home

    Authorize with following header:
    Authorization: Bearer abcdefghijklmnopqrst0123456789
```

Note: This for example purposes only.

list Subcommand

Synopsis

Use the `list` subcommand to display a list of inference instances with some basic information regarding each of them. The results are sorted using a date of creation starting with the most recent, and filtered by optional criteria.

Syntax

```
nctl predict list [options]
```

Options

Name	Required	Description
`-a, --all_users`	No	Show all prediction instances, regardless of the owner.
`-n, --name TEXT`	No	A regular expression to narrow down list to prediction instances that match this expression.
`-s, --status [QUEUED, RUNNING, COMPLETE, CANCELLED, FAILED, CREATING]`	No	A regular expression to filter list to prediction instances with matching status.
`-u, --uninitialized`	No	List uninitialized prediction instances: for example, prediction instances without resources submitted for creation.
`-c, --count INTEGER RANGE`	No	If given, command displays c most-recent rows.
`-b, --brief`	No	Print short version of the result table. Only `'name'`, `'submission date'`, `'owner'` and `'state'` columns will be printed.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

List of inference instances.

stream Subcommand

Synopsis

Use the `stream` subcommand to perform stream inference tasks on a launched prediction instance.

Syntax

```
nctl predict stream [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	Yes	The name of prediction session.
`-d, --data PATH`	Yes	The path to JSON data file that will be streamed to prediction instance. Data must be formatted such that it is compatible with the `SignatureDef` specified within the model deployed in selected prediction instance.
`-m, --method-verb [classify, regress, predict]`	No	Method verb that will be used when performing inference. Predict verb is used by default.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

template Command

Use the `template` command to manage the template packs used by nctl application. This main command also includes the following subcommands:

- copy Subcommand, page 117
- install Subcommand, page 119
- list Subcommand, page 120

copy Subcommand

Synopsis

Use the `copy` subcommand to copy a locally existing template pack to a new template pack. Once copied, you can change the description and the version of a newly created template pack, if desired.

Syntax

```
nctl template copy [options] SRC_TEMPLATE_NAME DEST_TEMPLATE_NAME
```

Arguments

Name	Required	Description
`SRC_TEMPLATE_NAME`	Yes	This is the name of a template pack that will be copied. This pack *must* be available locally. Therefore, if a you want to make a copy of a remote template pack, *you must* first install it locally using the `template install` command.
`DEST_TEMPLATE_NAME`	Yes	This is the name of the copied template pack. If a template pack with a given name exists, the Nauta application displays the information about it and completes its action.

Options

Name	Required	Description
`-d, --description TEXT`	No	A description of a newly created template pack. If not given, nctl asks for a description during copying of the pack. The maximum length of a description is 255 characters.
`-ve, --version TEXT`	No	The version of a newly created template pack. If not given, the default `0.1.0` value is used as a version.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

When a template pack is copied successfully, a confirmation message displays. If an error occurs during execution of this command, the cause of the issue displays.

Example

```
nctl template copy --version 0.2.0 existing-pack new-pack
```

Additional Remarks

This subcommand creates a new template pack named new-pack based on a locally available template pack *existing-pack*. The version of a newly created pack is set to 0.2.0. You will be asked for a description during making a copy of a template pack.

install Subcommand

Synopsis

Use the `install` subcommand to install a template pack locally with a given name. If the template pack has been already installed, use this subcommand to update the template to the version residing on a remote repository.

Syntax

```
nctl template install TEMPLATE_NAME
```

Arguments

Name	Required	Description
`TEMPLATE_NAME`	Yes	The name of a template pack that should be installed/updated, as required.

Options

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

When an installation/update is successfully completed, a confirmation message displays. If an error occurs during execution of this command, the cause of the issue displays.

Example

```
nctl template install template-name
```

Additional Remarks

The following command installs/upgrade template with template-name name.

list Subcommand

Synopsis

Use the `list` subcommand to list the template packs and displays information about the available local packs on a remote repository.

Syntax

```
nctl template list
```

Options

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Additional Remarks

The configuration of the template zoo is stored in the `NAUTA_HOME/config/zoo-repository.config` file. This file contains location of a template zoo repository (under the `model-zoo-address` key).

Additionally, it can contain also a git access token (under the `access-token` key). The access token is needed in case when a template zoo repository is private, and credentials are needed to get access to it. A user can modify both values if needed to use a different repository with template packs.

Returns

The tables (shown below, show the full version) lists the available template packs. Each row contains the name and the description of a template (as well as the versions) of remote and local template packs. If one of these versions is empty, this indicates that this template pack does not have this certain version. A partial and full version is shown (next page, **zoom in as desired**).

See Table 4 for complete a list and descriptions of the template packs provided with Nauta.

Template name	Template description	Local version
jupyter	An interactive session based on Jupyter Notebook	0.1.0

Template name	Template description	Local version	Remote version
jupyter	An interactive session based on Jupyter Notebook using Python 3.	0.1.0	0.1.0
jupyter-py2	An interactive session based on Jupyter Notebook using Python 2.	0.1.0	0.1.0
openvino-inference-batch	An OpenVINO model server inference job for batch predictions.	0.1.0	0.1.0
openvino-inference-stream	An OpenVINO model server inference job for streaming predictions on a deployed instance.	0.1.0	0.1.0
pytorch-training	A PyTorch multi-node training job using Python 3.	0.0.1	0.0.1
pytorch-training-py2	A PyTorch multi-node training job using Python 2.	0.0.1	0.0.1
tf-inference-batch	A TensorFlow Serving inference job for batch predictions.	0.1.0	0.1.0
tf-inference-stream	A TensorFlow Serving inference job for streaming predictions on a deployed instance.	0.1.0	0.1.0
tf-training-horovod	A TensorFlow multi-node training job based on Horovod using Python 3.	0.2.2	0.2.2
tf-training-horovod-py2	A TensorFlow multi-node training job based on Horovod using Python 2.	0.2.2	0.2.2
tf-training-multi	A TensorFlow multi-node training job based on TfJob using Python 3.	0.1.0	0.1.0
tf-training-multi-py2	A TensorFlow multi-node training job based on TfJob using Python 2.	0.1.0	0.1.0
tf-training-single	A TensorFlow single-node training job based on TfJob using Python 3.	0.1.0	0.1.0
tf-training-single-py2	A TensorFlow single-node training job based on TfJob using Python 2.	0.1.0	0.1.0

Note: If an error occurs during execution of this command, the cause of the issue displays.

user Command

Use the `user` command to create, delete, and manage users. This main command also includes the following subcommands:

- create Subcommand, page 123
- delete Subcommand, page 125
- list Subcommand, page 126

create Subcommand

Synopsis

Use the `create` subcommand to create and initialize a new Nauta user. This command *must be* executed when `kubectl` is used by a nctl command entered by a k8s administrator. If this command is executed by someone other than a k8s administrator, it fails. By default, this command saves a configuration of a newly created user to a file. The format of this file is compliant with a format of kubectl configuration files.

Syntax

```
nctl user create [options] USERNAME
```

Arguments

Name	Required	Description
`USERNAME`	Yes	Name of a user that will be created.

Options

Name	Required	Description
`-lo, --list-only`	No	If given, the content of the generated user's config file is displayed on the screen only. If not given, the file with configuration is saved on disk.
`-fl, --filename TEXT`	No	The name of file where user's configuration will be stored. If not given, the configuration is stored in the `config.<username>` file.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Additional Remarks

In case of any errors during saving of a file with a configuration, the command displays the content of the configuration file on the screen, even if `-lo` option *was not* used.

If an administrator creates a user with a name that was used previously by a deleted user, it may happen that the create command displays information that the previous user is still being deleted, even if the previous user *is not* listed on a list of existing users. In this case, before creating a new user, postpone the operation for 10 minutes, until all the user's objects are removed.

Returns

If any issues occur, a message is displayed describing their cause/causes. Otherwise, a message is returned indicating success. If the `--list-only` option was given, the command also displays the content of a configuration file.

User Name Requirements

The *User Name must* meet the following requirements:

1. Cannot be longer than 32-characters.
2. Cannot be an empty string.
3. Must conform to Kubernetes naming convention, and can only contain lower-case alphanumeric characters and "-" and "."

User Name Limitations

If an administrator creates a user with a name that was used previously by a deleted user, the `create` command displays the previous user is still being deleted, even if the previous user *is not* listed on a list of existing users.

In this case, before creating a new user, postpone the operation for a short period (at least 3 minutes) until all the user's objects are removed.

Example

```
nctl user create jdoe
```

Outcome

This creates the user `jdoe`, as shown in the example.

delete Subcommand

Synopsis

The `delete` subcommand deletes a user with a given name. If the option `-p, --purge` was used, it also removes all artifacts related to a that removed user, such as the content of user's folders, experiment's data, and runs.

Syntax

```
nctl user delete USERNAME
```

Arguments

Name	Required	Description
`USERNAME`	Yes	The name of a user who should be removed from the system.

Additional Remarks

Before removing a user, the command asks for a final confirmation. If a you choose `Yes`, the chosen user is deleted. Deletion of a user may take a while (a few minutes) to be fully completed.

If after this time a user *has not* been deleted completely, the command displays information that a user is still being deleted. In this case the user *will not* be listed on a list of existing users, but there *is no possibility* to create a user with the same name until the command completes and the user is deleted.

Options

Name	Required	Description
`-p, --purge`	No	If set, the system also removes all logs generated by the user's experiments.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

A message regarding the command's completion. If issues occur, a short description of the cause(s) displays.

Example

```
nctl user delete jdoe -p
```

Outcome

This removes the created jdoe user along with all their artifacts.

list Subcommand

Synopsis

Use the `list` subcommand to list all currently configured users.

Syntax

```
nctl user list [options]
```

Options

Name	Required	Description
`-c, --count INTEGER RANGE`	No	If given, the command displays c last rows.
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

verify Command

Use the `verify` command to check whether all prerequisites required by `nctl` are installed and have proper versions.

Synopsis

Checks whether all prerequisites required by `nctl` are installed and have proper versions. Also refer to version Command, page 128 for more information.

Syntax

```
nctl verify
```

Options

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Displays help messaging information.

Returns

In the case of any installation issues, the command returns information about their cause (which application should be installed and in which version). If no issues are found, a message indicates checks were successful.

Example

```
This OS is supported.
kubectl verified successfully.
helm client verified successfully.
git verified successfully.
helm server verified successfully.
kubectl server verified successfully.
packs resources' correctness verified successfully.
```

version Command

Use the `version` command to return the version of Nauta, as desired.

Synopsis

Returns the version of Nauta software. Also refer verify Command, page 127 for more information.

Syntax

```
nctl version
```

Options

Name	Required	Description
`-f, --force`	No	Force command execution by ignoring (most) confirmation prompts.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

The version command returns the currently installed `nctl` application version of both client platform and server.

Example

```
| Component        | Version                   |
|------------------+---------------------------|
| nctl application | 1.1.0-ent-20191010050128 |
| nauta platform   | 1.1.0-ent-20191010050128 |
```

Note: The output shown is an example only.