Intel® Deep Learning Studio

User Guide, Beta Release

Revision 1.0: December 2018

Terms and Conditions

Contents

Tables

Figures

Intel® Deep Learning Studio Introduction

Product Overview

The Intel® Deep Learning Studio 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 amongst teams easier. Intel DL Studio 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, Intel DL Studio also supports both batch and streaming inference, all in a single platform.

The Intel DL Studio client software runs on the following operating systems:

- Ubuntu* 16.04
- macOS*
- Windows* 10

Purpose of this Guide

This guide describes how to use the Intel DL Studio and discusses the following topics:

- Basic Concepts
- Client Installation and Configuration
- Getting Started
- Working with Datasets
- Working with Experiments
- Working with Template Packs
- Evaluating Experiments
- Evaluating Experiments with Inference Testing
- Managing Users and Resources
- CLI Commands

Basic Concepts

The following concepts and terms are relevant to using this software.

User

The user is a data scientist who wants to perform deep learning experiments to train models that will, after training and testing, be deployed in production. Using Intel DL Studio, the user can define and schedule containerized deep learning experiments using Kubernetes* on single or multiple worker nodes, and check the status and results of those experiments to further adjust and run additional experiments, or prepare the trained model for deployment.

Administrator

The administrator or admin creates and monitors users and resources. Admins cannot be users (data scientists), and are not permitted to perform any of the user experiments or related tasks. Also, users (data scientists) cannot be admins. 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 SDRAM 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

Data is the set of observations used to run experiments to train, test and validate your model. Unlabeled data is used in inference and prediction. Datasets are often downloaded from publicly available sources and include MNIST, CIFAR10, NORB, Caltech 256, and many others. Or you may have your own data. Intel processors are not limited to running experiments with highly regular, curated data sets like those listed above, but can process data from less curated sources.

Experiments

Performing deep learning experimentation is what the Intel DL Studio 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. 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 Intel DL Studio 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 processes singular data as it is received.

Client Installation and Configuration

The section provides instructions for installing and configuring Intel DL Studio to run on your client system.

For instructions to install and configure Intel DL Studio to run on the host server, refer to the *Intel® Deep Learning Studio Installation and Configuration Guide*.

Supported Operating Systems

The Intel DL Studio software runs on the following operating systems listed next. Please contact your Intel representative for information about how to acquire the installation package for your OS.

- Ubuntu 16.04
- macOS.
- Windows 10.

Required Software Packages

The following software *must* be installed on the client system *before* installing Intel DL Studio:

- kubectl version 1.10 or later: (https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl)
- docker version 18.03.0-ce or later: (https://docs.docker.com/install/)

Note: Helm is shipped together with Intel DL Studio.

Installation

To install the Intel DL Studio software package, do the following:

1. First, download and install the two *Required Software Packages* above, preferably in the order given.
2. Download and extract the Intel DL Studio client software package *for your operating system*. There is no installation utility. You can unpack this package and place the unpacked files in any location you prefer. Take note of the path.
3. Set KUBECONFIG environment variable to the Kubernetes configuration file provided by your Intel DL Studio Admin. Here, <PATH> is wherever your *config* file is located.
    - MacOS/Ubuntu:
      Execute: `export KUBECONFIG=<PATH>/<USERNAME>.config`
    - Windows:
      Execute: `set KUBECONFIG=<PATH>\<USERNAME>.config`
4. (OPTIONAL) Add the package `dlsctl` path to your terminal PATH. `DLSCTL_HOME` should be the path to the dlsctl application folder:
    - MacOS/Ubuntu:
      Execute: `export PATH=$PATH:DLSCTL_HOME`

- Windows:
  Execute: `set PATH=%PATH%;DLSCTL_HOME`

Note: If you want to set the variables permanently, you can add the variables to your .bashrc, .bash_profile, or Windows system PATH. Alternatively, you may want to set the PATH and KUBECONFIG variables in the “Environment Variables” window. This is accessed by opening the Control Panel > System and Security > System > Advanced system settings, and accessing Environment variables. This is an administrator function only.

Getting Started

This section of the guide provides brief examples for performing some of the most essential and valuable tasks supported by Intel DL Studio software.

Note: Several examples in this section require access to the internet, to download data, scripts, etc.

The section discusses the following topics:

- Verifying Installation
- Overview of dlsctl Commands
- Example Experiments
- Adding Metrics with the Metrics API (Optional)
- Launch TensorBoard*
- Viewing Experiment Output Folder
- Removing Experiments (Optional)

Verifying Installation

Check that the required software packages are available in terminal by PATH and verified that correct version is used.

Note 1: Docker* can contain kubectl* binary in a version that is not supported. Please verify that the PATH environment variable has the correct order of paths and that the kubectl from docker package is not used.

Note 2: If you are behind a proxy, remember to set the *HTTP_PROXY*, *HTTPS_PROXY* and *NO_PROXY* environment variables. For Ubuntu, these variables must be set or the software will not work correctly.

Enter the following command to verify your installation.

- **Execute**: `dlsctl verify`

If any installation issues are found, the command returns information about their cause (which application should be installed and in which version). This command also checks if the CLI can connect to Intel DL Studio and if port forwarding to Intel DL Studio is working properly.

If no issues are found, a message indicates checks were successful. The following is displayed:

```
This OS is supported.
draft verified successfully.
kubectl verified successfully.
kubectl server verified successfully.
helm client verified successfully.
helm server verified successfully.
docker client verified successfully.
docker server verified successfully.
```

Overview of dlsctl Commands

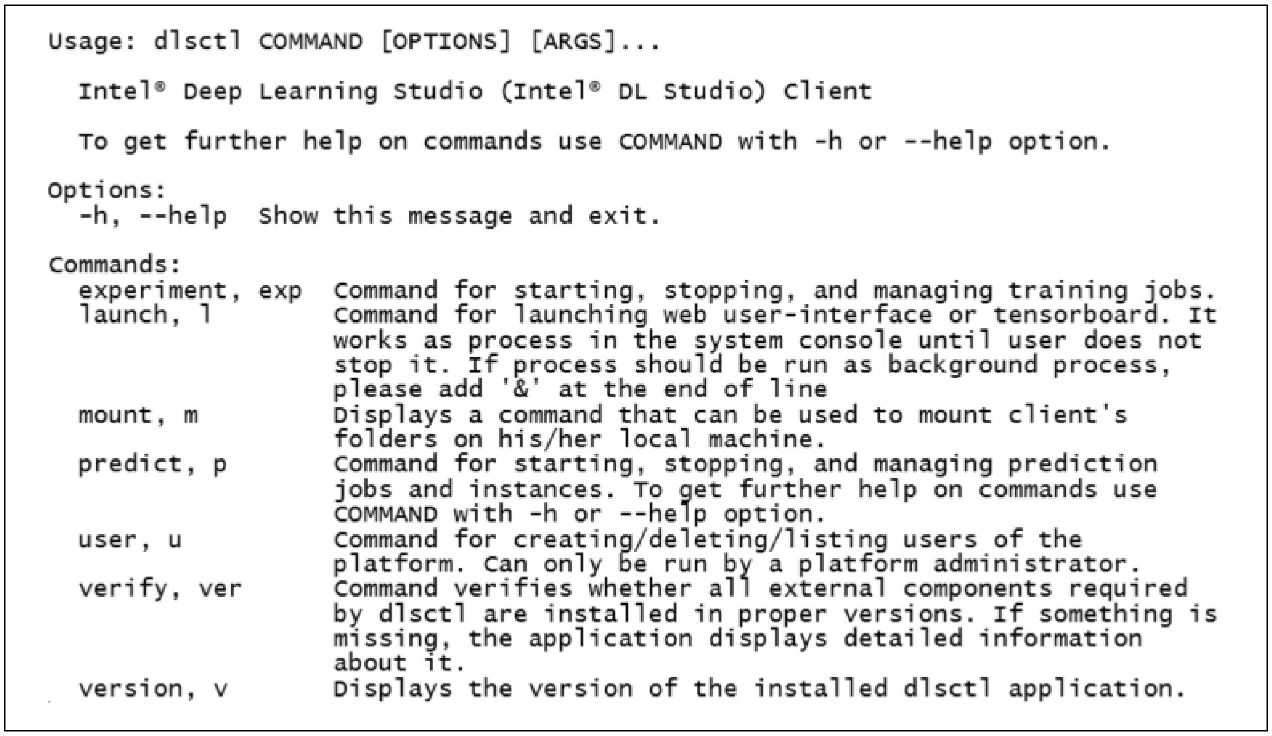
Each dlsctl command has at least two options:

- `-v, --verbose` Set verbosity level:
    - o `-v` for INFO - basic logs on INFO/EXCEPTION/ERROR levels are displayed.
    - o `-vv` for DEBUG - detailed logs on INFO/DEBUG/EXCEPTION/ERROR levels are displayed.
- `-h, --help` - the application displays the usage and options available for a specific command or subcommand.

Access help for any command with the `--help` or `-h` parameter. The following command provides a list and brief description of all dlsctl commands.

Execute: `dlsctl --help`

The results are shown below.



```
Usage: dlsctl COMMAND [OPTIONS] [ARGS]...

  Intel® Deep Learning Studio (Intel® DL Studio) Client

  To get further help on commands use COMMAND with -h or --help option.

Options:
  -h, --help  Show this message and exit.

Commands:
  experiment, exp  Command for starting, stopping, and managing training jobs.
  launch, l        Command for launching web user-interface or tensorboard. It
                   works as process in the system console until user does not
                   stop it. If process should be run as background process,
                   please add '&' at the end of line
  mount, m         Displays a command that can be used to mount client's
                   folders on his/her local machine.
  predict, p       Command for starting, stopping, and managing prediction
                   jobs and instances. To get further help on commands use
                   COMMAND with -h or --help option.
  user, u          Command for creating/deleting/listing users of the
                   platform. Can only be run by a platform administrator.
  verify, ver      Command verifies whether all external components required
                   by dlsctl are installed in proper versions. If something is
                   missing, the application displays detailed information
                   about it.
  version, v       Displays the version of the installed dlsctl application.
```

Example Experiments

The Intel DL Studio installation includes sample training scripts and utility scripts that can be run to demonstrate how to use Intel DL Studio. This section describes how to use these scripts.

Examples Folder Content

The `examples` folder in the dlsctl installation contains these following experiment scripts:

- `mnist_single_node.py` - training of digit classifier in single node setting
- `mnist_multinode.py` - training of digit classifier in distributed TensorFlow setting
- `mnist_horovod.py` - training of digit classifier in Horovod

There are also the following two utility scripts, `mnist_converter_pb.py` and `mnist_checker.py` which are related to inference process and model verification.

Note: Experiment scripts must be written in Python.

Launching Training Using the Scripts

Launching training in Intel DL Studio in these examples is performed with the following command:

```
   dlsctl experiment submit -t template SCRIPT_LOCATION
```

with:

- `template = tf-training-tfjob` and `SCRIPT_LOCATION = examples/mnist_single_node.py` for single node training. The template parameter in this case is optional.
- or `template = multinode-tf-training-tfjob` and `SCRIPT_LOCATION = examples/mnist_multinode.py` for multinode training
- or `template = multinode-tf-training-horovod` and `SCRIPT_LOCATION = examples/mnist_horovod.py` for Horovod training

Note: All examples assumes macOS. Any line-wraps below are *not* intended.

Submitting an Experiment

Note: The included example scripts do not require external data source. They download the MNIST dataset and save it locally.

Note: Templates referenced here have some set CPU and Memory requirements. Please refer to Working with Template Packs on page 28 for more info about changing those if you want to.

Note: For more info about experiment submit command please refer to submit Subcommand on page 52.

This example will show how to submit a mnist experiment and write the TensorBoard data to a folder in your Intel DL Studio output folder. Enter the following command.

Syntax:

Following is the basic syntax for experiment submit command:

```
dlsctl experiment submit [options] SCRIPT_LOCATION -- [script_parameters]
```

Enter the following command to run this example:

Execute:

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

Where:

- `-t,`   TEXT: Name of a template that will be used by Intel Dl Studio to create a description of a job to be submitted. By default, this is a single-node tensorflow training. Template is chosen. List of available templates might be obtained by Issuing dlsctl experiment template_list command.
- -- `SCRIPT_LOCATION`: The path and name of the Python script use to perform this experiment.

Result of this Command

The execution of the submit command may take a few minutes the first time. During this time, resources are allocated, the experiment package is created using parameters, scripts, and settings, a docker image is built then pushed to the Kubernetes cluster. When the experiment submission is complete, the following result is displayed:

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

Viewing Experiment Status

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

Syntax: `dlsctl experiment list [options]`

Execute: `dlsctl experiment list --brief`

Experiment status displays as below.

The `--brief` option returns a short version of results shown below. *This is an example only.*

```
| Experiment            | Submission date     | Owner   | State    |
|----------------------+---------------------+---------+----------|
| mnist-single-node-tb  | 2018-11-16 00:19:19 | dan     | QUEUED   |
| mnist-tb              | 2018-11-13 21:50:37 | dan     | COMPLETE |
| mnist-tb-2-1          | 2018-11-13 23:04:18 | dan     | COMPLETE |
| mnist-tb-2-2          | 2018-11-13 23:04:20 | dan     | COMPLETE |
| mnist-tb-2-3          | 2018-11-13 23:04:23 | dan     | COMPLETE |
| my-experiment         | 2018-11-13 22:43:16 | dan     | QUEUED   |
| one-experiment        | 2018-11-16 00:03:09 | dan     | QUEUED   |
| parameter-range-1     | 2018-11-15 22:44:10 | dan     | QUEUED   |
| parameter-range-2     | 2018-11-15 22:44:13 | dan     | QUEUED   |
| parameter-range-3     | 2018-11-15 22:44:15 | dan     | QUEUED   |
| single-experiment     | 2018-11-16 00:00:54 | dan     | QUEUED   |
```

Monitoring Training

There are three ways to monitor training in Intel Dl Studio, all which are discussed in the following sections.

- Viewing Experiment Logs
- Adding Experiment Metrics
- Using Tensorboard

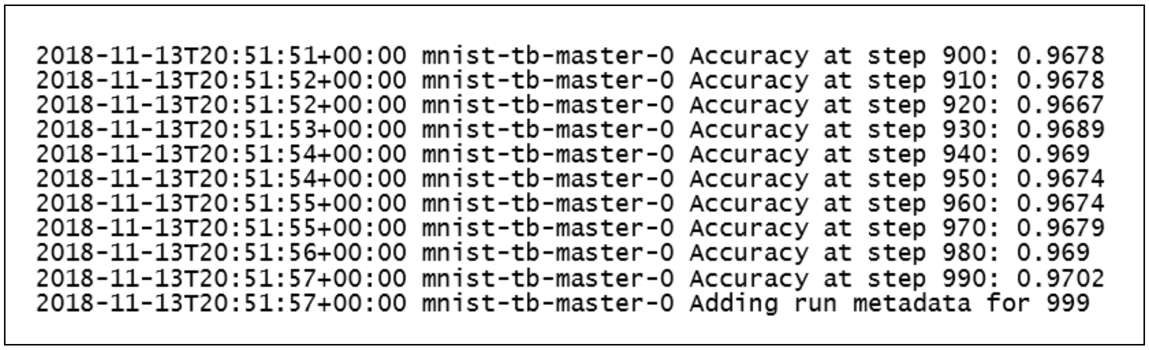
Viewing Experiment Logs

Use the following command to view the experiment log.

Syntax: `dlsctl experiment logs [options] EXPERIMENT_NAME`

Execute: `dlsctl experiment logs mnist–tb`

A log displays as follows. *The result below is an example only*.

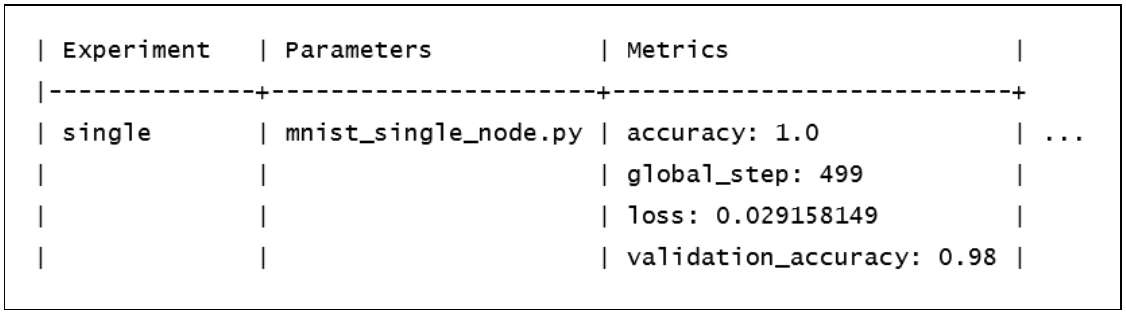


```
2018-11-13T20:51:51+00:00 mnist-tb-master-0 Accuracy at step 900: 0.9678
2018-11-13T20:51:52+00:00 mnist-tb-master-0 Accuracy at step 910: 0.9678
2018-11-13T20:51:52+00:00 mnist-tb-master-0 Accuracy at step 920: 0.9667
2018-11-13T20:51:53+00:00 mnist-tb-master-0 Accuracy at step 930: 0.9689
2018-11-13T20:51:54+00:00 mnist-tb-master-0 Accuracy at step 940: 0.969
2018-11-13T20:51:54+00:00 mnist-tb-master-0 Accuracy at step 950: 0.9674
2018-11-13T20:51:55+00:00 mnist-tb-master-0 Accuracy at step 960: 0.9674
2018-11-13T20:51:55+00:00 mnist-tb-master-0 Accuracy at step 970: 0.9679
2018-11-13T20:51:56+00:00 mnist-tb-master-0 Accuracy at step 980: 0.969
2018-11-13T20:51:57+00:00 mnist-tb-master-0 Accuracy at step 990: 0.9702
2018-11-13T20:51:57+00:00 mnist-tb-master-0 Adding run metadata for 999
```

Adding Experiment Metrics (Optional)

Experiments launched in Intel DL Studio can output additional kind of metrics using the `publish` function from the experiment metrics API. To see an example of metrics published with the `single` experiment executed in the above example, enter this command:
Execute: `dlsctl experiment list`

Following are example results (only a fragment is shown):



```
| Experiment    | Parameters            | Metrics                    |
|--------------+----------------------+---------------------------+
| single        | mnist_single_node.py | accuracy: 1.0              | ...
|               |                      | global_step: 499           |
|               |                      | loss: 0.029158149          |
|               |                      | validation_accuracy: 0.98 |
```

To add metrics *to an experiment file you have created*, you need to edit the experiment file to use the `experiment_metrics.api` and then publish the accuracy in your experiment file. Perform the following steps:

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

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

2. Add metrics for publishing of last step's accuracy by adding this code in the 'def feed_dict' definition after the for loops:

   ```
   metrics = {}
   metrics["accuracy_step_{}".format(i)] = str(acc)
   publish(metrics)
   ```

3. Save the changes.

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

   Execute: `dlsctl <new_name> experiment list`

Information About 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 loosing some logs) due to conflicting names. To avoid this, adding metrics for multinode experiments should be done using one of the two following methods:

1. The key of a certain metric should also contain a node identificator from which this metric comes. Creation of such identificator can be done in the following ways:
   - For horovod multinode training jobs, result of the `rank()` function provided by the `horovod` package can be used as a node's identificator.
   - For `tfjob` multinode training jobs, a user can take all necessary info from the TF_CONFIG environment variable. Here is an example piece of a code creating such identificator:

```
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 identificator
node_id = '-'.join(job_name,task_index)
```

2. Only one node should store metrics. Deciding which node should store metrics can be done in the following ways:
   a. 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.
   b. 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 identificator of a current worker can be obtained as described in method 1 above. A user should choose one of such identificators and store logs only from a node having this chosen id.

Launching TensorBoard

Generally, every file that the training script outputs to `/mnt/output/experiment` (accessed from the perspective of training script launched in Intel DL Studio) is accessible from the outside after mounting the output directory with command provided by `dlsctl mount`. (Refer to Mounting Storage to View Experiment Output for more information.)

When training scripts output Tensorflow summaries to `/mnt/output/experiment`, they can be automatically picked up by Tensorboard instance launched with command:

Syntax: `dlsctl launch tensorboard [options] EXPERIMENT_NAME`

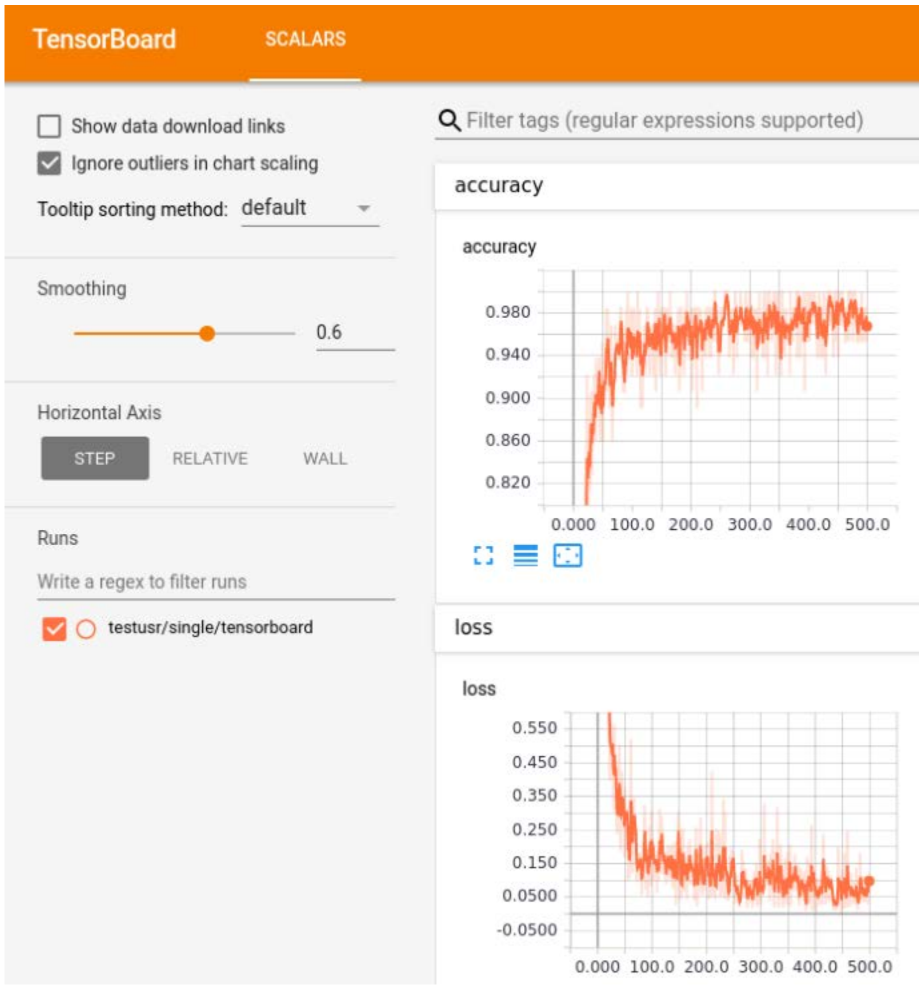
Execute: `dlsctl launch tensorboard single`

The following message displays. The port number is an example only.

```
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 1 shows the browser display of TensorBoard dashboard with the experiment's results.

Figure 1: TensorBoard Dashboard - Example Only



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
☑ testusr/single/tensorboard

Inference

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

1. Prepare data for model input.

2. Acquire a trained model.
3. Run 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 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`. Downloaded MNIST dataset will be stored there as well as 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. Max value is 10000

Running:

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

will create `/tmp/mnist_test/conversion_out` folder, fill it with 100 protobuf requests, and cache labels for these requests in `/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 `dlsctl mount` command and mount output storage locally. Example scripts all save servable models in their models subdirectory. To use models like this for inference, you will have to mount `input` storage too, because models have to be accessible from inside of the cluster.

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

```
mkdir /mnt/input
mkdir /mnt/input/single
mkdir /mnt/output
... mount command provided with dlsctl mount used to mount output storage
to /mnt/output
... mount command provided with dlsctl mount used to mount input storage
to /mnt/input
cp /mnt/output/single/models/* -Rf /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 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 see 42.

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

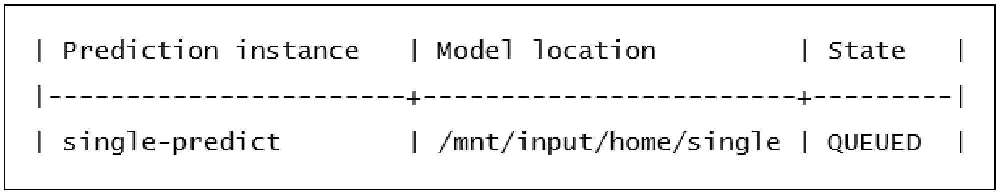
Execute these commands:

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

The next command will create a prediction instance. **Execute**:

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

Following are the example results of this command:



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

Note: 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.

Note: `mnist_converter_pb.py` creates requests to the `mnist` model. Note that `--model-name mnist` is where this `mnist` name is given to the prediction instance.

Note: For more info about predict command, please refer to predict Command on page 63.

After the prediction instance completes (can be checked using the `predict list` command), you can collect instance responses from `output` storage. In our example, it would contain 100 protobuf responses. These can be validated in our case using `mnist_checker.py`. Running the following command locally:

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

will display the error rate calculated for this model and this sample of the test set.

Viewing Experiment Output Folder

You can use the following steps to mount the output folder and view TensorBoard data files.

Mount a folder to your DLS4E namespace output directory:

1. macOS/Ubuntu: First, mount your DLS4E output directory to a local folder. Create a folder for mounting named `my_output_folder`.
   Execute: `mkdir my_output_folder`
2. To see the correct mount options and command:
   Execute: `dlsctl mount`

3. Use the mounting command that was displayed to mount Intel DL Studio storage to your local machine. Following are examples of mounting the local folder to the Intel DL Studio output folder for each OS:
   - MacOS:
     `mount_mbfs //'USERNAME:PASSWORD'@CLUSTER-URL/output my_output_folder`
   - Ubuntu:
     ```
     sudo mount.cifs -o username=USERNAME,password=PASSWORD,rw,uid=1000 \
     //CLUSTER-URL/output my_output_folder
     ```
   - Windows: Use Y: drive as mount point.
     `net case Y: \\CLUSTER-URL\output /user:USERNAME PASSWORD`
4. Navigate to the mounted location.
   - MacOS/Ubuntu only: Navigate to my_output_folder
   - Windows only: Open Explorer Window and navigate to Y: drive.
5. See the saved event file by navigating to mnist-single-node/tensorboard. Example file: events.out.tfevents.1542752172.mnist-single-node-master-0
6. Unmount DLS Storage using one of the below commands:
   - MacOS: `umount output my_output_folder`
   - Ubuntu: `sudo umount my_output_folder`
   - Windows: Eject or `net use Y: /delete`

For more information on mounting, refer to Working with Datasets on page 19.

Removing Experiments (Optional)

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. The experiment will only be removed from the experiment list. The experiment's artifacts will remain in the Intel DL Studio storage output folder. Logs will be removed.

Syntax: `dlsctl experiment cancel [options] EXPERIMENT_NAME`

Execute: `dlsctl experiment cancel –-purge <your_experiment>`

Working with Datasets

The section covers the following topics:

- Uploading Datasets
- dlsctl mount Command
- Mount and Access Folders
- Uploading and Using Shared Dataset Example

Uploading Datasets

Intel DL Studio offers two ways for users to upload and use datasets for experiments:

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, e.g., other .py files, datasets, etc. If the option is not included, the files will not be included in the experiment.

Syntax:

```
dlsctl experiment submit --script_folder_location DATASET-PATH
SCRIPT_LOCATION
```

This option is recommended for small datasets or datasets that are NOT reused often. In using this option, the dataset will be uploaded each time the submit command is executed which may add to the overall submission time.

Note: `script_folder_location` is also used for script files, not only for datasets.

Upload to DL Studio Storage and Reference Location

Depending on the Intel DL Studio configuration, the application 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 instructions in this chapter for information how to access and use Intel DL Studio storage.

This option is recommended for large datasets, data that will be reused often, and data that will be shared among users.

dlsctl mount Command

The 'mount' command displays another command that can be used to mount Intel DL Studio folders to a user's local machine. When a user executes the command, information similar the following is displayed (this example is for macOS). Use the following command to mount those folders (all of the following is displayed, although this is an example only).

Note: The *dlsctl mount* command also returns a command to unmount a folder.

```
Use the following command to mount those folders:
- replace <MOUNTPOINT> with a proper location on your local machine)
- replace <DLS4E_FOLDER> with one of the following:
        - input - User's private input folder (read/write)
          (can be accessed as /mnt/input/home from training script).
        - output - User's private output folder (read/write)
          (can be accessed as /mnt/output/home from training script).
        - input-shared - Shared input folder (read/write)
          (can be accessed as /mnt/input/root/public from training script).
        - output-shared - Shared output folder (read/write)
          (can be accessed as /mnt/output/root/public from training script).
        - input-output-ro - Full input and output directories, read only.
Additionally, 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>.
-------------------------------------------------------------------
mount_smbfs //'USERNAME:PASSWORD'@CLUSTER-URL/<DLS4E_FOLDER> <MOUNTPOINT>
Use following command to unmount previously mounted folder:
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.
```

IMPORTANT: Intel DL Studio uses the mount command that is native to each operating system, so the command printed out may not appear as in the example.

Note: All variables are shown in upper-case letters.

Mount and Access Folders

The following table displays the access permissions for each mounting folder.

Table 1: Access Permissions for Mounting Folders

DLS4E 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

output-shared	/mnt/output/root/public	read/write	read/write
input-output-ro		read	read

Uploading and Using Shared Dataset Example

The default configuration is to mount `my-input` and `my-output` folders to Intel DL Studio storage. Perform these steps below to mount an `input-shared` folder to Intel DL Studio storage so that you and other users can use shared input data when performing training.

1. Linux/macOS only: Create a folder for mounting named my-input.
   Execute: `mkdir my-input`
2. Use the mount command to display the command that should be used to mount your local folder/machine to your Intel DL Studio input folder.
   Execute: `dlsctl mount`
3. Use the mounting command that was displayed to mount your local machine to Intel DL Studio storage. Examples of mounting the local folder to the Intel DL Studio output folder for each OS:
   - MacOS: `mount_mbfs //'USERNAME:PASSWORD'@CLUSTER-URL/input my-input`
   - Ubuntu: `sudo mount.cifs -o username=USERNAME,password=PASSWORD,rw,uid=1000 //CLUSTER-URL/input my-input`
   - Windows: Use Y: drive as mount point
     `net case Y: \\CLUSTER-URL\input /user:USERNAME PASSWORD`
4. Copy input data or files to this folder for use when submitting experiments. After copying, the files will be located in Intel DL Studio storage and can be used by any user for their experiments.
   a. Using the MNIST example Submit an Experiment on page 11 in the *Getting Started* section, you can download the MNIST dataset from this link:
      Mnist Dataset: http://yann.lecun.com/exdb/mnist
   b. Create a mnist folder in the Intel DL Studio input-shared folder.
   c. Copy the downloaded files to the folder.
   d. Submit an experiment referencing the new shared dataset. From the examples folder,
      Execute: dlsctl experiment submit --name mnist-shared-input \ mnist_with_summaries.py -- --data_dir==/mnt/input/home/mnist

If you want to to copy your data to shared folder use `input-shared` instead of `input` in step 2. Doing so lets any Intel DL Studio user can use the same path to reference the mnist dataset on your shared DL Storage.

Working with Experiments

This section provides instructions about the following topics:

- Mounting Experiment Input to Intel DL Studio Storage
- Launching Jupyter Interactive Notebook
- Submitting a Single Experiment
- Submitting Multiple Experiments
- Running an Experiment on Multiple Nodes
- Mounting Storage to View Experiment Output
- Canceling Experiments

Note: Files located in the input storage are accessible through Jupyter Notebooks. Only files that are written to /output/home/ are persistently stored. Therefore, changes made to other files, including model scripts, during the session will not be saved after the session is closed. It is recommended to save session data to the output/<experiment> folder for future use.

Mounting Experiment Input to Intel DL Studio Storage

Perform these steps to mount a local folder/machine to Intel DL Studio storage and use the files when performing training.

Note: Names below are examples only.

1. Linux/macOS: Create a folder for mounting named `my_input`.
   Execute: `mkdir my_input`
2. Use the mount command to display the command that should be used to mount your local folder/machine to your Intel DL Studio input folder.
   Execute: `dlsctl mount`
3. **Execute**: `mount_smbfs` or `net use` or `mount.cfis` as appropriate. The command is dependent on the operating system. For Linux/macOS users, the MOUNTPOINT is the `my_input` folder. For Windows users, choose a drive letter as mount point.
4. Linux/macOS: Navigate to `my_input`
   Windows: Open Explorer Window and navigate to the input/home folder.
5. Copy input data or files to this folder for use when submitting experiments.

Launching Jupyter* Interactive Notebook

You can use Jupyter* Notebook to run and display the results of you experiments. This release of Intel® DL Studio supports Python 3 an 2.7 for scripts. Launch Jupyter Notebook using the following command:

Syntax: `dlsctl experiment interact [options]`

Execute: `dlsctl experiment interact`

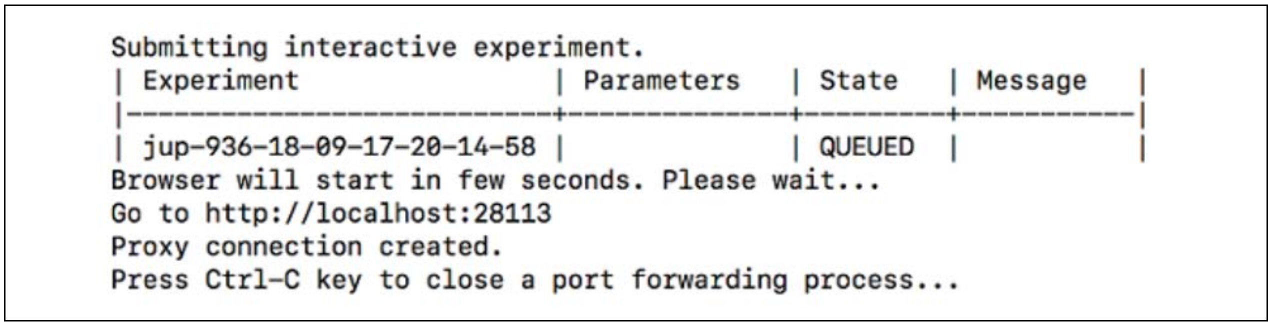
Other available options to this command are:

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

Note: Files located in the input storage are accessible through Jupyter Notebooks. Only files that are written to /output/home/ are persistently stored. Therefore, changes made to other files, including model scripts, during the session will not be saved after the session is closed. It is recommended to save session data to the output/<experiment> folder for future use.

Files that are accessible through the Jupyter Notebook are the same folders that is accessible to the user for experiments.

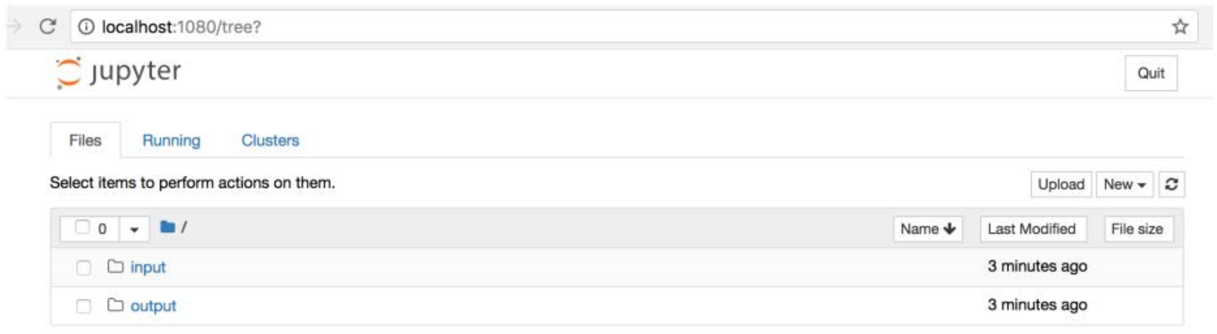
The following result displays.



```
Submitting interactive experiment.
| Experiment                  | Parameters    | State    | Message    |
|-----------------------------+---------------+----------+------------|
| jup-936-18-09-17-20-14-58 |               | QUEUED   |            |
Browser will start in few seconds. Please wait...
Go to http://localhost:28113
Proxy connection created.
Press Ctrl-C key to close a port forwarding process...
```

Jupyter Notebook will launch in your default web browser. The following displays.

Figure 2: Jupyter Notebook—Example Only

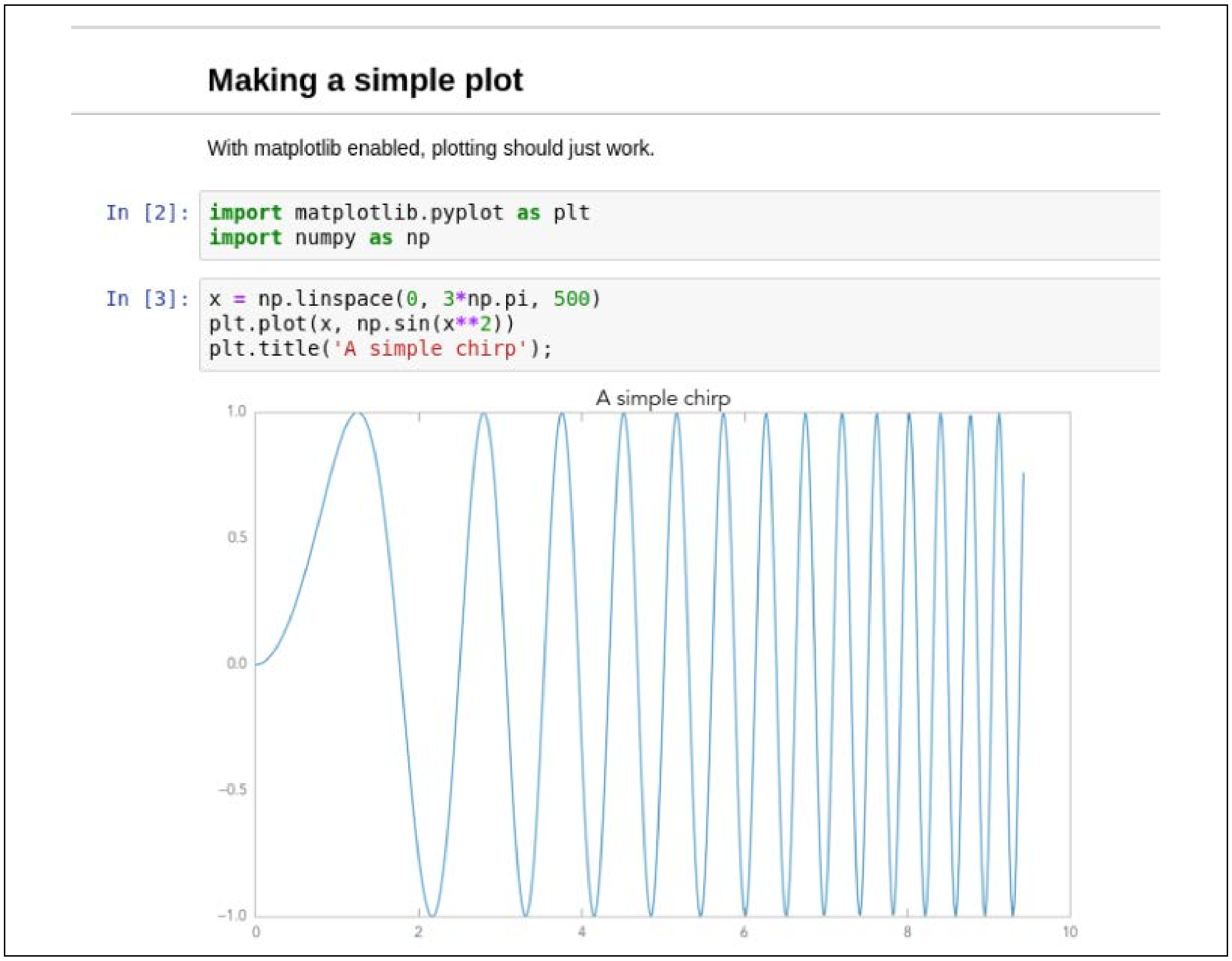


Select items to perform actions on them.

input 3 minutes ago
output 3 minutes ago

An example active Jupyter Notebook, showing a simple experiment plot.

Figure 3: Jupyter Notebook Simple Experiment Plot—Example Only



Making a simple plot

With matplotlib enabled, plotting should just work.

```
In [2]: import matplotlib.pyplot as plt
        import numpy as np
```

```
In [3]: x = np.linspace(0, 3*np.pi, 500)
        plt.plot(x, np.sin(x**2))
        plt.title('A simple chirp');
```

Submitting a Single Experiment

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 Intel DL Studio software.

Refer to Submit an Experiment on page 11.

Submitting Multiple Experiments

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

Your experiment script must be written in Python. Storage locations for your input and output folders are determined by the mount command. Refer to Working with Datasets on page 19 and Mounting Experiment Input to Intel DL Studio Storage on page 22.

To submit multiple individual experiments that use the same script, use the following syntax for this command (line wrap is not intended).

Syntax: `dlsctl exp submit --parameter_range SCRIPT_LOCATION [-- SCRIPT_PARAMETERS]`

Note: `SCRIPT_LOCATION` above refers to values (a set or of a range) of a single parameter.

Execute: `dlsctl experiment submit --parameter_range lr "{0.1, 0.2, 0.3}" < mnist_single_node> -- --data_dir=/mnt/input/root/public/mnist`

Refer to Working with Datasets for instructions on uploading the dataset to the "input_shared" folder.

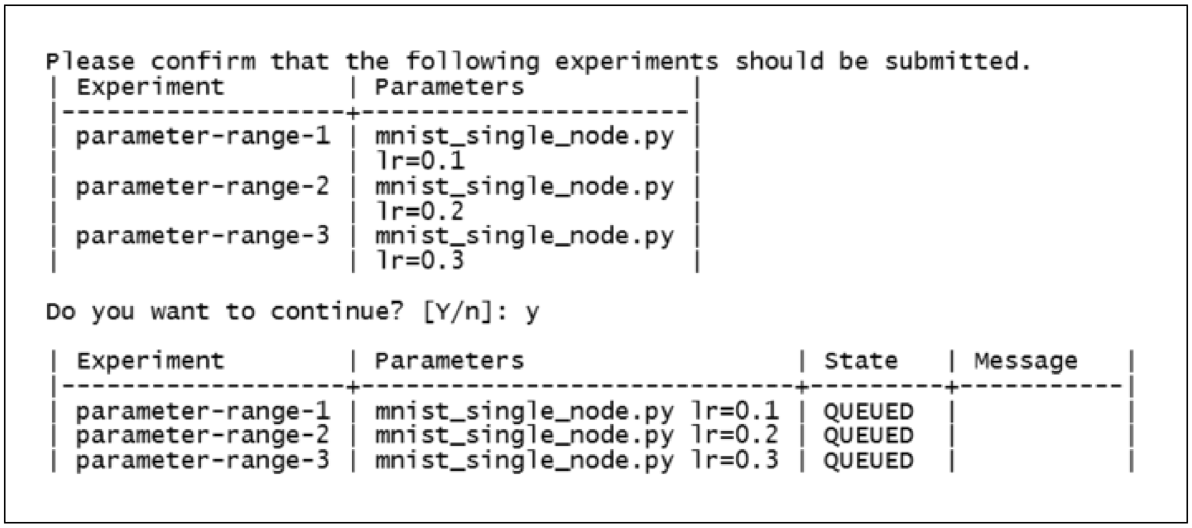
Parameters can include either:

- `parameter_range` argument that defines the name of a parameter together with its values expressed as either a range or an explicit set of values, or
- `parameter_sets` argument that specifies a number of distinct combinations of parameter values.

An *example* of this command using `parameter_range` is shown below (line wrap is not intended).

Execute: `dlsctl experiment submit --name parameter_range --parameter_range lr "{0.1, 0.2, 0.3}" mnist_single_node.py`

The following result displays.



```
Please confirm that the following experiments should be submitted.
| Experiment         | Parameters           |
|--------------------+----------------------|
| parameter-range-1  | mnist_single_node.py |
|                    | lr=0.1               |
| parameter-range-2  | mnist_single_node.py |
|                    | lr=0.2               |
| parameter-range-3  | mnist_single_node.py |
|                    | lr=0.3               |

Do you want to continue? [Y/n]: y

| Experiment         | Parameters                  | State    | Message    |
|--------------------+-----------------------------+----------+------------|
| parameter-range-1  | mnist_single_node.py lr=0.1 | QUEUED   |            |
| parameter-range-2  | mnist_single_node.py lr=0.2 | QUEUED   |            |
| parameter-range-3  | mnist_single_node.py lr=0.3 | QUEUED   |            |
```

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 Intel DL Studio 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 on page 19.

This experiment uses a template. For more information, refer to Working with Template Packs.

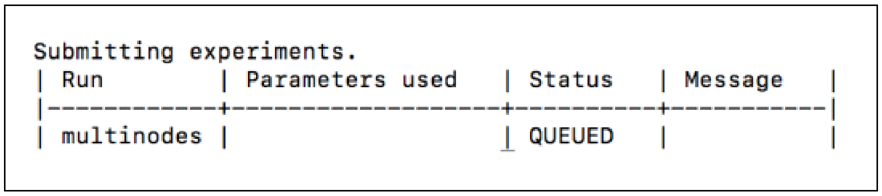
To run a multi-node experiment, the script must support it. Following is the generic syntax (line wrap is not intended).

Syntax: `dlsctl exp submit [options] --template [MULTINODE_TEMPLATE_NAME] \ SCRIPT_LOCATION [-- SCRIPT_PARAMETERS]`

The template `multinode-tf-training-tfjob` is included with Intel DL Studio software. Following is an example command using this template (line wrap is not intended):

Execute:
```
dlsctl experiment submit --name multinodes \
--template multinode-tf-training-tfjob \
--name mnist-multi-node mnist_multi_nodes.py \
-- -- data_dir=/mnt/input/root/public/mnist
```

The following result displays showing the queued job.



```
Submitting experiments.
| Run         | Parameters used     | Status    | Message    |
|-------------+---------------------+-----------+------------|
| multinodes  |                     | QUEUED    |            |
```

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

- `-p workers_Count 2`
- `-p pServersCount 1`

Mounting Storage to View Experiment Output

Refer to the section Working with Datasets on page 19.

Canceling Experiments

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

Execute: `dlsctl experiment cancel[options] EXPERIMENT_NAME`

This command stops and cancels any experiment *queued* or *in progress*. Completed experiments cannot be cancelled. Cancels any experiment based on the name of an experiment/pod/status of a pod. If any such object is found the command queries if these objects (one or more) should be cancelled.

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: `dlsctl experiment cancel -match EXPERIMENT_NAME`

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

Syntax: `dlsctl experiment cancel –-pod-ids [pod_ID] EXPERIMENT_NAME`

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

Execute: `dlsctl experiment cancel --pod-status [PENDING, RUNNING, SUCCEEDED, FAILED, UNKNOWN] EXPERIMENT_NAME`

Any of the above criteria can be combined.

You can also purge all information concerning any experiment using the `-p` or `--purge` option.

Working with Template Packs

This section discusses the following topics:

- What is a Template Pack?
- The Pack Anatomy
- Provided Template Packs
- Customizing the Provided Packs
- Creating a New Template Pack
- Placeholders in values.yaml

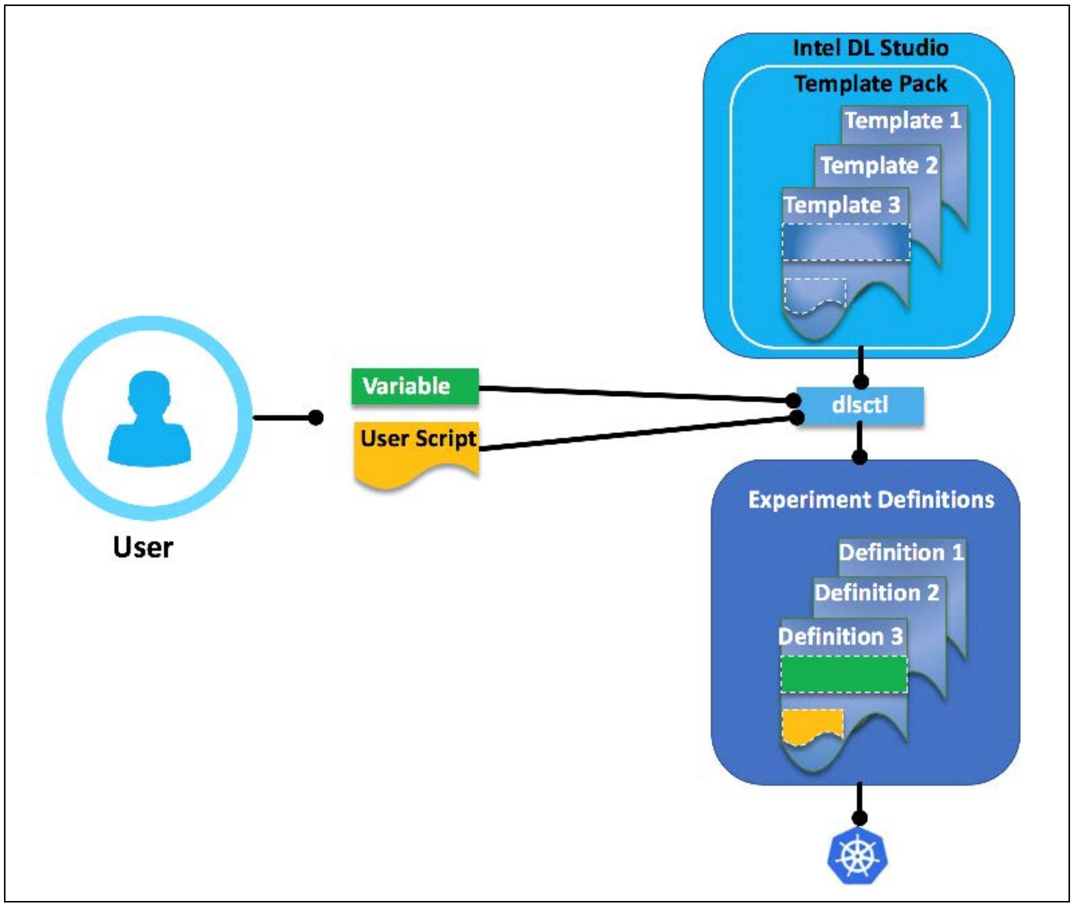
What is a Template Pack?

Every experiment run on the Intel® Deep Learning Studio application utilizes a template pack. 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 4: Template Pack



Individual elements within a pack are referred to as templates because they contain a number of placeholders (some are required, while others are optional), that are substituted with appropriate values by Intel DL Studio software during experiment submission. These placeholders define items such as: experiment name, user namespace, the address of the local Intel DL Studio 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 Intel DL Studio template packs.

Note 1: While Intel DL Studio 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 Intel DL Studio to track and manage the resulting experiments. Refer to Creating a New Template Pack on page 32 for details.

Note 2: All supported Intel DL Studio template packs are distributed with the dlsctl package.

The Anatomy of a Pack

Location

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

```
DLSCTL_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/
```

Where:

- `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 `dlsctl` to perform substitution on "Placeholders in values.yaml " on page 33.
- `templates` folder groups all the YAML files that provide definitions for various Kubernetes (K8s) entities, which define the packs deployment and runtime environment. These definitions are referred to as templates as they may include Helm template placeholders substituted for actual values in the process of deploying the chart on the cluster.

Provided Template Packs

The Intel DL Studio 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- prefix in the name support multi-node experiments, while those with a single- prefix 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 2.

Table 2: 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 Intel DL Studio, it is possible to adjust the compute resource requirements either on per-experiment basis or permanently (refer to Customizing the Provided Packs on page 31.

The template packs provided with Intel DL Studio are listed below:

- **multi-tf-training-horovod** - A TensorFlow multi-node training job based on Horovod using Python 3.
- **multi-tf-training-horovod-py2** - A TensorFlow multi-node training job based on Horovod using Python 2.
- **multi-tf-training-tfjob** - A TensorFlow multi-node training job based on TF-operator using Python 3.
- **multi-tf-training-tfjob-py2** - A TensorFlow multi-node training job based on TF-operator using Python 2.
- **single-tf-training-tfjob** - A TensorFlow single-node training job based on TF-operator using Python 3.
- **single-tf-training-tfjob-py2** - A TensorFlow single-node training job based on TF-operator 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 30, 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 values.yaml

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

1. Users 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. Users may alter some of the parameters listed in `values.yaml` file temporarily and *only* for a single experiment. To do so, the user 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 on page 51 for more details).

Advanced users who 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`, etc.

Where to Start

Creating new template packs for Intel DL Studio 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 exhaustively described in Helm documentation.

A Template Pack in Five Simple Steps

Once a working Helm chart is available, the process of adapting it for use as an Intel DL Studio template is as follows:

1. Pick the pack's name. 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 won't work.
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 Intel DL Studio 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 `DLS4E.CommandLine`. (Refer to DLS4E.CommandLine on page 33.) This will allow the commandline parameters specified in the `dlstctl 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 `DLS4E.ExperimentImage`. (Refer to DLS4E.ExperimentImage on page 34.)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 Intel DL Studio 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. Intel DL Studio 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,* etc) 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).

Placeholders in values.yaml

DLS4E.CommandLine

The DLS4E.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 `dlsctl 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 DLS4e.CommandLine %}nt
- {{ arg }}
```

```
{% endfor %}
```

DLS4E.ExperimentImage

The DLS4E.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 DLS4e 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: {{ DLS4e.ExperimentImage }}

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

Evaluating Experiments

This section discusses the following topics:

- Viewing Experiments Using the CLI
- Viewing Experiment Logs and Results Data
- Viewing Experiment Results at the Web UI
- Launching TensorBoard to View Experiments

Viewing Experiments Using the CLI

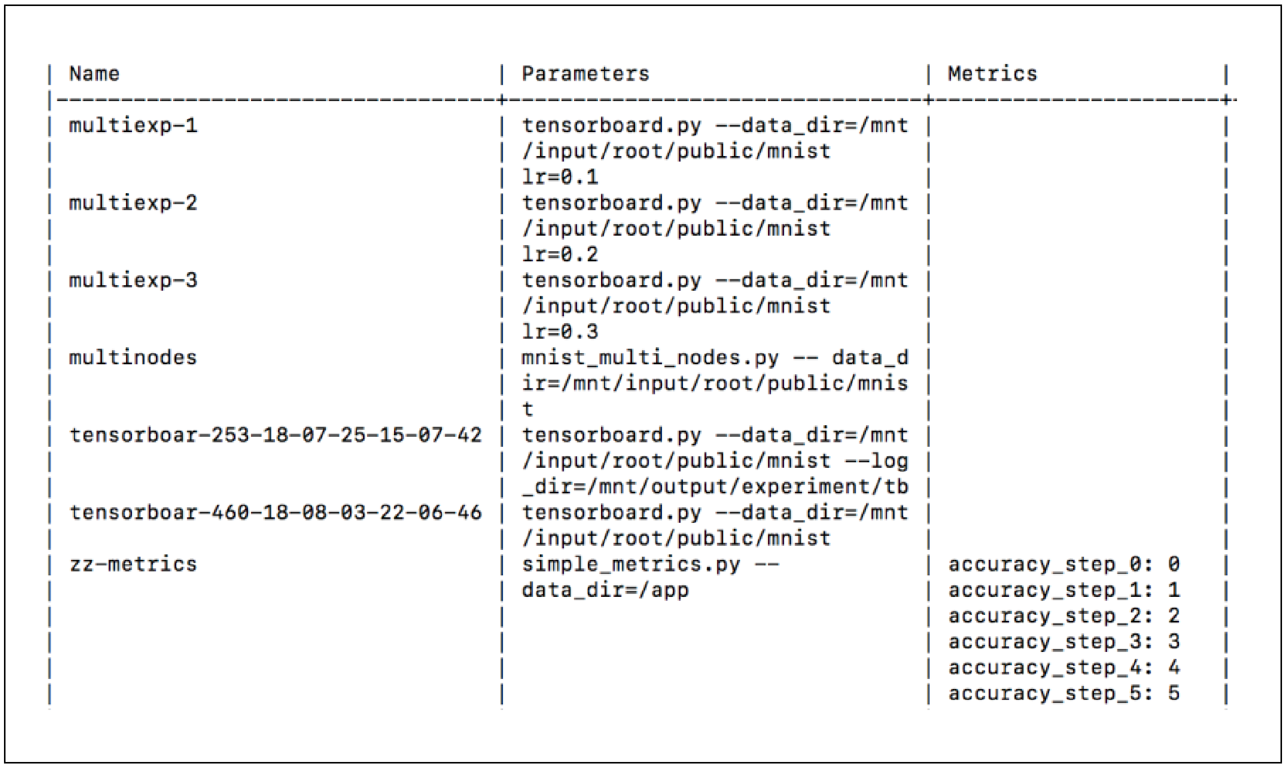
Viewing all Experiments

To list *all* experiments you have submitted, run the next command. The possible returned statuses are QUEUED, RUNNING, COMPLETE, CANCELED, FAILED, and CREATING.

Syntax: `dlsctl experiment list [options]`

Execute: `dlsctl experiment list`

Following are example results (not all columns are shown).



```
| Name                                | Parameters                      | Metrics                 |
|-------------------------------------+---------------------------------+-------------------------+
| multiexp-1                          | tensorboard.py --data_dir=/mnt |                         |
|                                     | /input/root/public/mnist        |                         |
|                                     | lr=0.1                          |                         |
| multiexp-2                          | tensorboard.py --data_dir=/mnt |                         |
|                                     | /input/root/public/mnist        |                         |
|                                     | lr=0.2                          |                         |
| multiexp-3                          | tensorboard.py --data_dir=/mnt |                         |
|                                     | /input/root/public/mnist        |                         |
|                                     | lr=0.3                          |                         |
| multinodes                          | mnist_multi_nodes.py -- data_d |                         |
|                                     | ir=/mnt/input/root/public/mnis |                         |
|                                     | t                               |                         |
| tensorboar-253-18-07-25-15-07-42    | tensorboard.py --data_dir=/mnt |                         |
|                                     | /input/root/public/mnist --log |                         |
|                                     | _dir=/mnt/output/experiment/tb |                         |
| tensorboar-460-18-08-03-22-06-46    | tensorboard.py --data_dir=/mnt |                         |
|                                     | /input/root/public/mnist        |                         |
| zz-metrics                          | simple_metrics.py --            | accuracy_step_0: 0      |
|                                     | data_dir=/app                   | accuracy_step_1: 1      |
|                                     |                                 | accuracy_step_2: 2      |
|                                     |                                 | accuracy_step_3: 3      |
|                                     |                                 | accuracy_step_4: 4      |
|                                     |                                 | accuracy_step_5: 5      |
```

Viewing a Single Experiment's Details

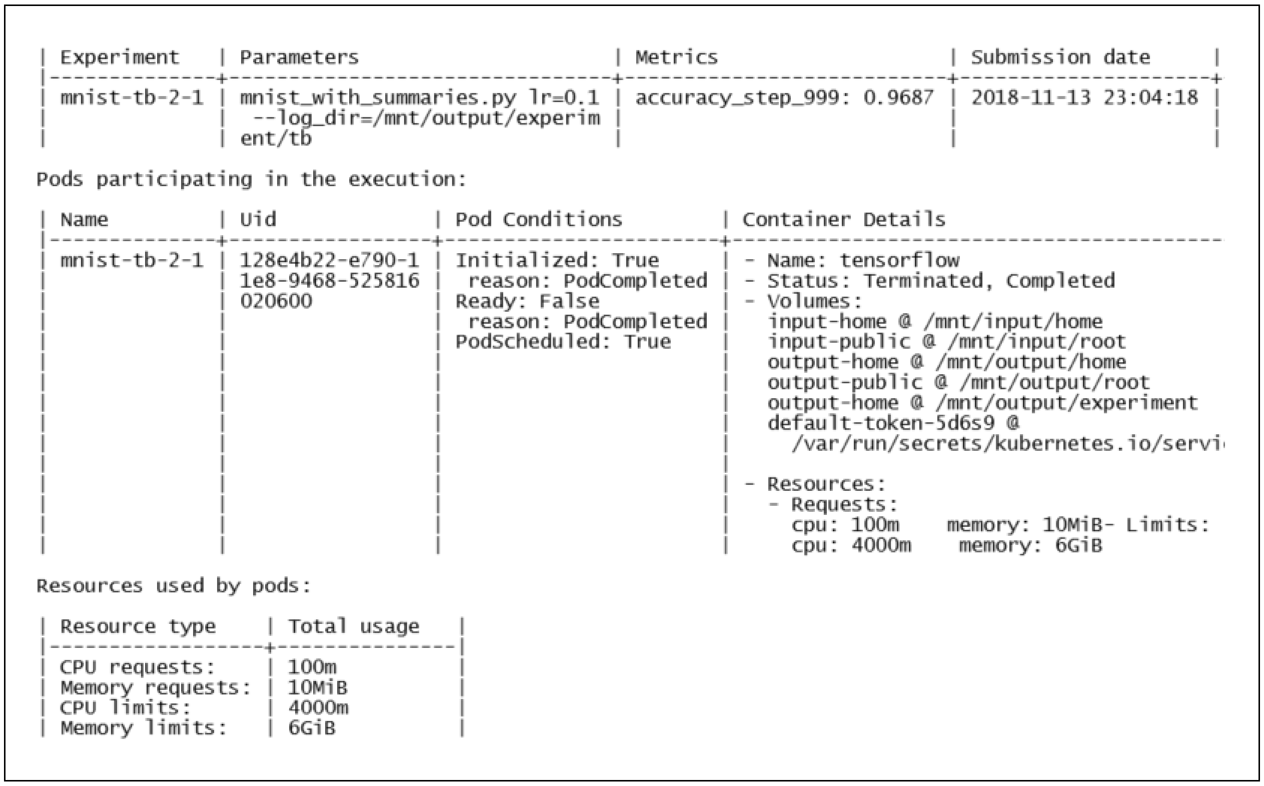
The primary purpose of the next command is to provide Kubernetes pod-level information and container information for this experiment. This includes the pod ID, the POD status, information about input and output volumes used in this experiment, and CPU and memory resources requested to perform this experiment.

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

Syntax: `dlsctl experiment view [options] EXPERIMENT_NAME`

Execute: `dlsctl experiment view mnist-tb-2-1`

Following are *example results* and not all information is shown.



```
| Experiment    | Parameters                       | Metrics                    | Submission date     |
|---------------+----------------------------------+----------------------------+---------------------+
| mnist-tb-2-1  | mnist_with_summaries.py lr=0.1   | accuracy_step_999: 0.9687  | 2018-11-13 23:04:18 |
|               |   --log_dir=/mnt/output/experim  |                            |                     |
|               | ent/tb                           |                            |                     |

Pods participating in the execution:

| Name          | Uid               | Pod Conditions          | Container Details
|---------------+-------------------+-------------------------+-------------------------------------
| mnist-tb-2-1  | 128e4b22-e790-1   | Initialized: True       | - Name: tensorflow
|               | 1e8-9468-525816   |   reason: PodCompleted  | - Status: Terminated, Completed
|               | 020600            | Ready: False            | - Volumes:
|               |                   |   reason: PodCompleted  |   input-home @ /mnt/input/home
|               |                   | PodScheduled: True      |   input-public @ /mnt/input/root
|               |                   |                         |   output-home @ /mnt/output/home
|               |                   |                         |   output-public @ /mnt/output/root
|               |                   |                         |   output-home @ /mnt/output/experiment
|               |                   |                         |   default-token-5d6s9 @
|               |                   |                         |     /var/run/secrets/kubernetes.io/servi
|               |                   |                         |
|               |                   |                         | - Resources:
|               |                   |                         |   - Requests:
|               |                   |                         |     cpu: 100m    memory: 10MiB- Limits:
|               |                   |                         |     cpu: 4000m    memory: 6GiB

Resources used by pods:

| Resource type     | Total usage   |
|-------------------+---------------|
| CPU requests:     | 100m          |
| Memory requests:  | 10MiB         |
| CPU limits:       | 4000m         |
| Memory limits:    | 6GiB          |
```

Viewing Experiment Logs and Results Data

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

Separate from logs, the results or output of an experiment can be found by mounting the user's output folder or output-shared folder. A training script should write to the Intel DL Studio output folder to save any output files.

Use the following command to view logs from a given experiment (this is an example only).

Syntax: `dlsctl experiment logs [options] EXPERIMENT_NAME`

Execute: `dlsctl experiment logs mnist-tb`

The following result displays an example log.

```
2018-11-13T20:51:51+00:00 mnist-tb-master-0 Accuracy at step 900: 0.9678
2018-11-13T20:51:52+00:00 mnist-tb-master-0 Accuracy at step 910: 0.9678
2018-11-13T20:51:52+00:00 mnist-tb-master-0 Accuracy at step 920: 0.9667
2018-11-13T20:51:53+00:00 mnist-tb-master-0 Accuracy at step 930: 0.9689
2018-11-13T20:51:54+00:00 mnist-tb-master-0 Accuracy at step 940: 0.969
2018-11-13T20:51:54+00:00 mnist-tb-master-0 Accuracy at step 950: 0.9674
2018-11-13T20:51:55+00:00 mnist-tb-master-0 Accuracy at step 960: 0.9674
2018-11-13T20:51:55+00:00 mnist-tb-master-0 Accuracy at step 970: 0.9679
2018-11-13T20:51:56+00:00 mnist-tb-master-0 Accuracy at step 980: 0.969
2018-11-13T20:51:57+00:00 mnist-tb-master-0 Accuracy at step 990: 0.9702
2018-11-13T20:51:57+00:00 mnist-tb-master-0 Adding run metadata for 999
```

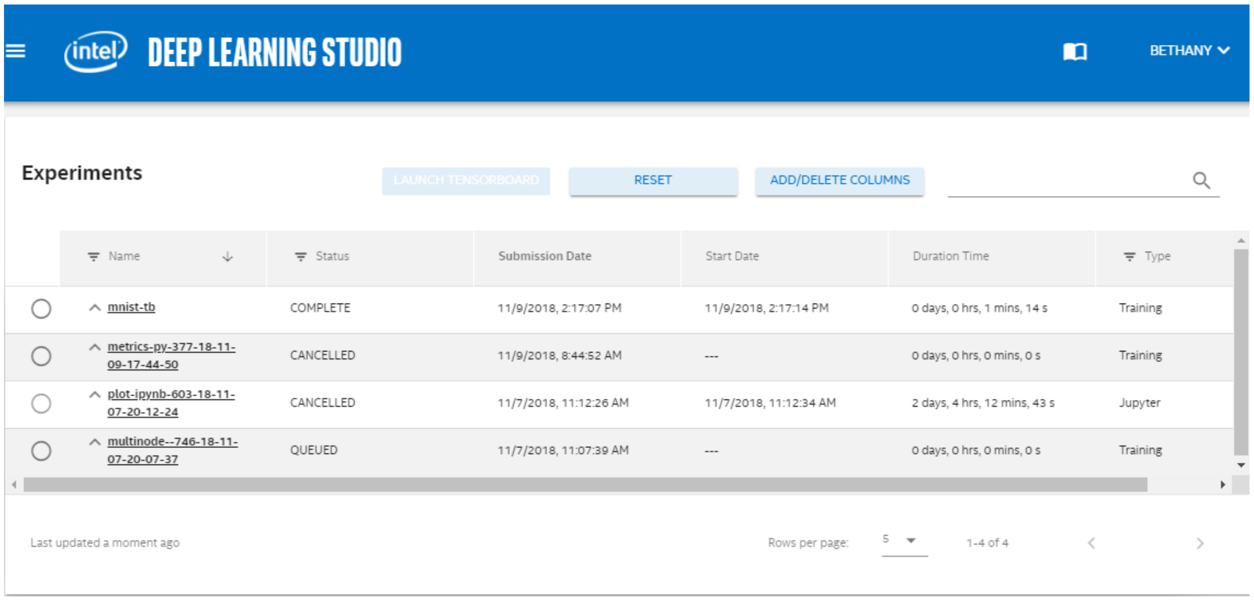
Viewing Experiment Results at the Web UI

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

Execute: `dlsctl launch webui`

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

Figure 5: Viewing Experiment Results from the Web UI—Example Only



Experiments

Name	Status	Submission Date	Start Date	Duration Time	Type
mnist-tb	COMPLETE	11/9/2018, 2:17:07 PM	11/9/2018, 2:17:14 PM	0 days, 0 hrs, 1 mins, 14 s	Training
metrics-py-377-18-11-09-17-44-50	CANCELLED	11/9/2018, 8:44:52 AM	---	0 days, 0 hrs, 0 mins, 0 s	Training
plot-ipynb-603-18-11-07-20-12-24	CANCELLED	11/7/2018, 11:12:26 AM	11/7/2018, 11:12:34 AM	2 days, 4 hrs, 12 mins, 43 s	Jupyter
multinode--746-18-11-07-20-07-37	QUEUED	11/7/2018, 11:07:39 AM	---	0 days, 0 hrs, 0 mins, 0 s	Training

Last updated a moment ago   Rows per page: 5   1-4 of 4

The web UI shows the six columns listed below. Each of these column headings are clickable, so to re-sort the listing of experiments based on that column heading, ascending or descending order, click on that column heading.   Here are the six columns:

- **Name**: The left-most column lists the experiments by name.
- **Status**: This column reveals experiment's current status, one of: QUEUED, RUNNING, COMPLETE, CANCELED, 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. The Start Date (or time) will always be after the Submission Date (or time).
- **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. Inference means that training is largely complete and you have begun running predictions (Inference) with this model. (If the model used for inference was already present, there was no training performed on DL Studio.)

You can perform the tasks discussed below at the Intel DL Studio web UI.

Expand Experiment Details

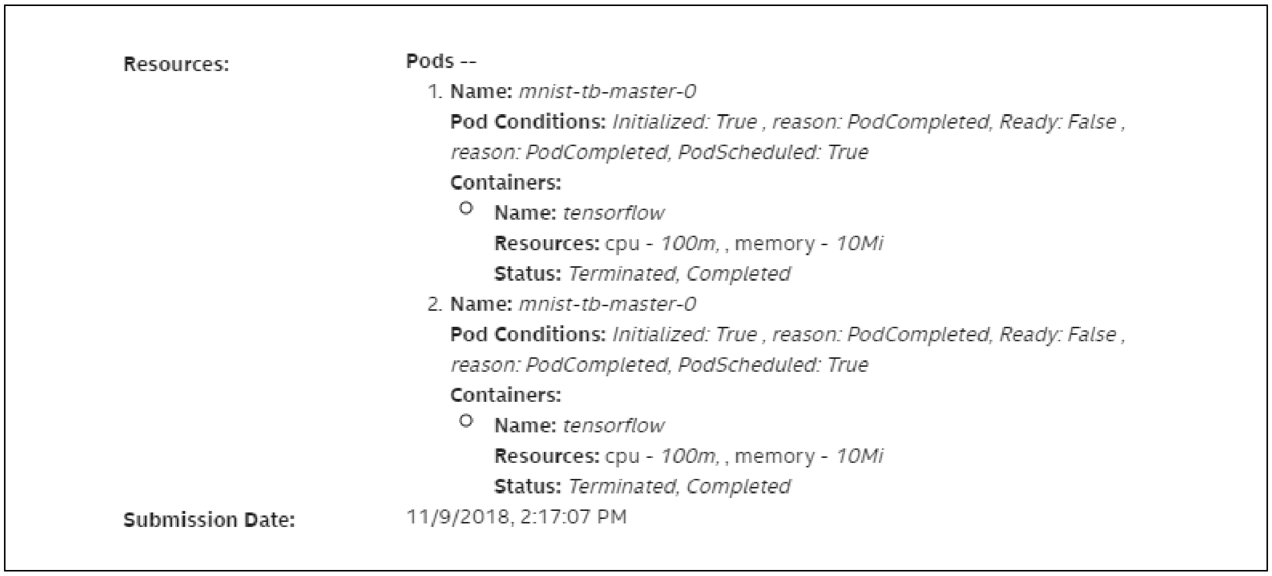
Click on a **listed experiment name** to see additional details for that experiment. The following details are examples only.

This screen is divided into two frames. The left-side frame shows:

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

See Figure 6.

Figure 6: Experiment Details - 1



Resources: Pods --

1. **Name:** *mnist-tb-master-0*
   Pod Conditions: *Initialized: True , reason: PodCompleted, Ready: False , reason: PodCompleted, PodScheduled: True*
   Containers:
   - **Name:** *tensorflow*
     Resources: cpu - *100m*, , memory - *10Mi*
     Status: *Terminated, Completed*
2. **Name:** *mnist-tb-master-0*
   Pod Conditions: *Initialized: True , reason: PodCompleted, Ready: False , reason: PodCompleted, PodScheduled: True*
   Containers:
   - **Name:** *tensorflow*
     Resources: cpu - *100m*, , memory - *10Mi*
     Status: *Terminated, Completed*

Submission Date: 11/9/2018, 2:17:07 PM

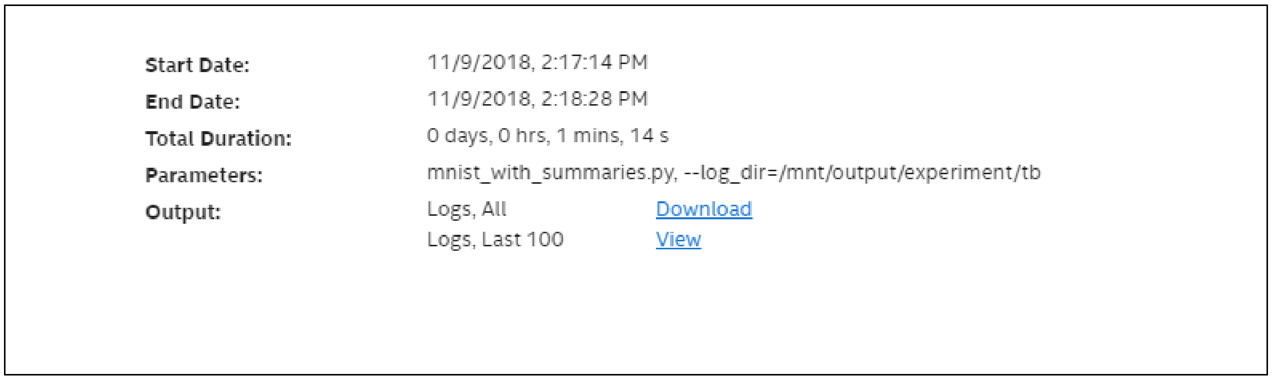
The right-side frame of the experiment details windows shows:

- o **Start Date**: The day and time this experiment was launched.
- o **End date**: The day and time this experiment was launched.
- o **Total Duration**: The actual duration this experiment was instantiated.
- o **Parameters**: The experiment script file name and the log directory.

- **Output**: Clickable links to download all logs and view the last 100 log entries.

See Figure 7

Figure 7: Experiment Details - 2



Start Date:	11/9/2018, 2:17:14 PM	
End Date:	11/9/2018, 2:18:28 PM	
Total Duration:	0 days, 0 hrs, 1 mins, 14 s	
Parameters:	mnist_with_summaries.py, --log_dir=/mnt/output/experiment/tb	
Output:	Logs, All	Download
	Logs, Last 100	View

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/Deleting Columns

Click **ADD/DELETE COLUMNS** 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. 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, etc.

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 on page 39.

Opening Kubernetes Dashboard

Click the “hamburger menu” ☰ at the far left of the UI to open a left frame. Click **Resources Dashboard** to open the Kubernetes resources dashboard. Refer to Accessing the Kubernetes Resource Dashboard on page 49.

Launching TensorBoard to View Experiments

You can launch TensorBoard from the the Intel DL Studio Web UI or the CLI. These methods are described below.

Launching TensorBoard from the Web UI

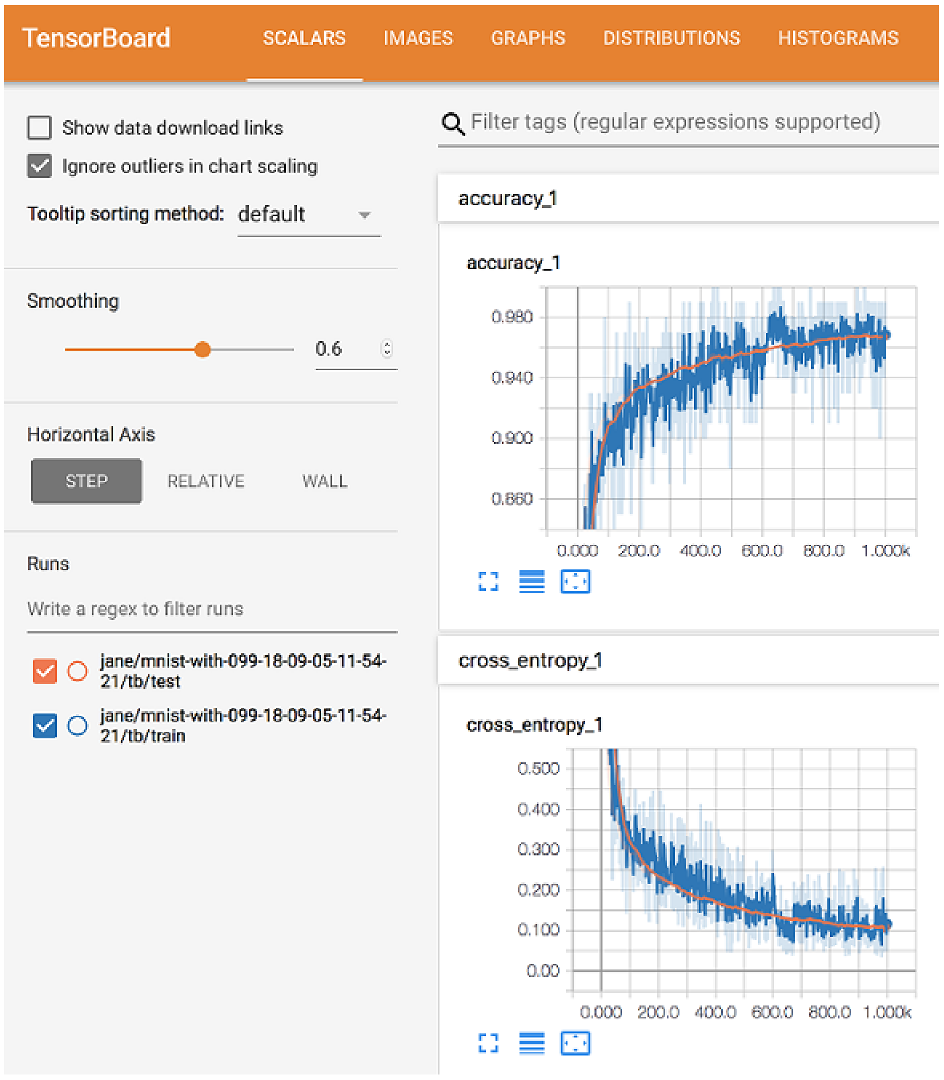
Note: To view the experiment’s results in TensorBoard, TensorBoard data must be written to a folder in the directory `/mnt/output/experiment`.

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

1. Open the web ui. **Execute**: `dlsctl launch webui`
2. At the web UI, identify the experiment that you want to see displayed in TensorBoard. Click on 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.

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

Figure 8: Launch TensorBoard from the Web UI – Example Only



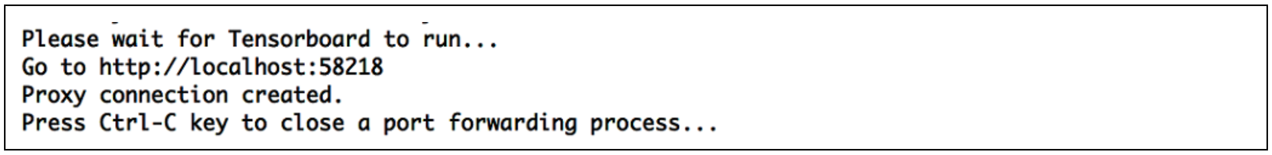
☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: STEP
Runs:
☑ jane/mnist-with-099-18-09-05-11-54-21/tb/test
☑ jane/mnist-with-099-18-09-05-11-54-21/tb/train

Launching TensorBoard from the CLI

To launch TensorBoard from the CLI, do the following.

Execute: `dlsctl launch tb <experiment_name>`

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 `dlsctl experiment view` command:

```
dlsctl experiment view -tensorboard EXPERIMENT_NAME
```

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

Evaluating Experiments with Inference Testing

This section discusses the following topics:

- Using the predict Command
- Batch Inference Example
- Streaming Inference Example

Using the predict Command

Use the predict command to start, stop, and manage prediction jobs. Please refer to predict Command on page 63 for a detailed description of this command.

Batch Inference

Example Flow

Following is the general flow of this example:

1. The user has acquired the dataset and the trained model.
2. The user converts dataset into serialized protocol buffers (PBs). (Refer to https://developers.google.com/protocol-buffers/ as one information source.)
3. The user invokes `dlsctl mount`.
4. The user mounts the Samba input folder by invoking the command displayed in the previous step.
5. The user copies the serialized PBs and the trained model to the just-mounted share.
6. The user runs `dlsctl predict batch` command.

MNIST Example

You need to have preprocessed MNIST data for feeding the batch inference. You can use already-preprocessed data located in 'parsed' directory, or generate data on your own (described in MNIST Data Preprocessing below). Perform the following steps:

1. Mount Samba input share to your directory, assumed /mnt/input . Use the command printed by `dlsctl mount`.
2. Copy 'parsed' and 'trained_mnist_model' to `/mnt/input`.
3. **Execute**:
```
dlsctl predict batch --model-location \
/mnt/input/home/trained_mnist_model --data /mnt/input/home/parsed \
--model-name mnist
```

Important notes:

- Paths provided in locations such as `--model-location` and `--data` need to point for files/directory from the container's context, not from user's filesystem or mounts. These paths can be mapped using instructions from `dlsctl mount`. For example, if you've mounted Samba `/input` and copied files there, you should pass `/mnt/input/home/<file>`.
- `--model-name` is optional, but it must match the model name provided during data preprocessing, since generated requests must provide which servable they target. In the

`mnist_converter_pb.py` script you can find `request.model_spec.name = 'mnist'`, which saves model name in requests, and that name must match value passed as `--model-name`. If not provided it assumes that model name is equal to last directory in model location:
`/mnt/input/home/trained_mnist_model -> trained_mnist_model`

MNIST Data Preprocessing

Perform the following steps:

1. Create venv. **Execute**: `make venv`
2. Run the mnist_converter_pb.py script using just generated venv:

```
source .venv/bin/activate
python mnist_converter_pb.py
```

Description of Files

- output - example PBs responses from batch inference downloaded after dlsctl predict batch process.
- parsed - example PBs parsed MNIST dataset generated by mnist_converter_pb.py script ready as an input for dlsctl predict batch.
- trained_mnist_model - example MNIST trained model used for inference.
- batch_client.py - prototype of batch inference wrapper client.
- checker.py - script checking MNIST inference error rate from PBs responses.
- mnist_converter_pb.py - example preprocessing script for preparing predict batch input. handles downloading MNIST dataset and saving PBs to 'parsed' folder.
- mnist_input_data.py - dependency for mnist_converter_pb.py script for managing original MNIST dataset.
- requirements.txt - required Python dependencies for invoking all Python scripts here.
- requirements-dev.txt - optional Python dependency for e.g. auto formatting Python code.

References

- https://www.tensorflow.org/serving/serving_basic
- https://developers.google.com/protocol-buffers/docs/pythontutorial
- https://github.com/tensorflow/serving/blob/master/tensorflow_serving/example/mnist_client.py
- https://www.tensorflow.org/serving/docker

Streaming Inference

Example Flow

Following is the basic task flow for this example.

1. The user has saved a trained Tensorflow Serving compatible model.
2. The user will be sending data for inference in JSON format, or in binary format using gRPC API.
3. The user runs `dlsctl predict` launch command.

4. The user sends inference data using the `dlsctl 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 https://github.com/tensorflow/serving repository. We will use the *saved_model_half_plus_two_cpu* model for showing streaming prediction capabilities.

In order to use that model for streaming inference, perform following steps:

1. Clone https://github.com/tensorflow/serving repository:
   Execute: `git clone https://github.com/tensorflow/serving`
2. Perform step 3 or step 4 below, based on preference.
3. Run the following command. **Execute**:
   ```
   dlsctl predict launch --local_model_location <directory where you
   have cloned Tensorflow Serving>/serving/tensorflow_serving/ \
   servables/tensorflow/testdata \ /saved_model_half_plus_two_cpu
   ```
4. Alternatively to step 3, you may want to save a trained model on input share, so it can be reused by other experiments/prediction instances. In order to to this, run these commands:
   a. Use the `mount` command to mount DLS4E input folder to local machine.
      Execute: `dlsctl mount`
      Use the resulting command printed by `dlsctl mount`. After executing command printed by `dlsctl mount` command, you will be able to access input share on your local file system.
   b. Now copy the *saved_model_half_plus_two_cpu* model to input share:
      Execute:
      ```
      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>
      ```
   c. Run the following command. **Execute**:
      ```
      dlsctl predict launch --model-location \
      /mnt/input/saved_model_half_plus_two_cpu
      ```

Note: `--model-name` can be passed optionally to `dlsctl predict launch` command. If not provided, it assumes that 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, dlsctl will create a streaming inference instance that can be used in multiple ways, as described below.

Streaming Inference with dlsctl predict stream Command

The `dlsctl predict stream` command allows 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 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` operation. We've 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`. In order 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 `dlsctl predict launch` command executes; you can also use dlsctl predict list command for listing running prediction instances). Then run following command:

```
dlsctl 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 used model and the expected results. Please refer to https://www.tensorflow.org/serving/api_rest for more detailed information. These method verbs are:

- CLASSIFY
- REGRESS
- PREDICT

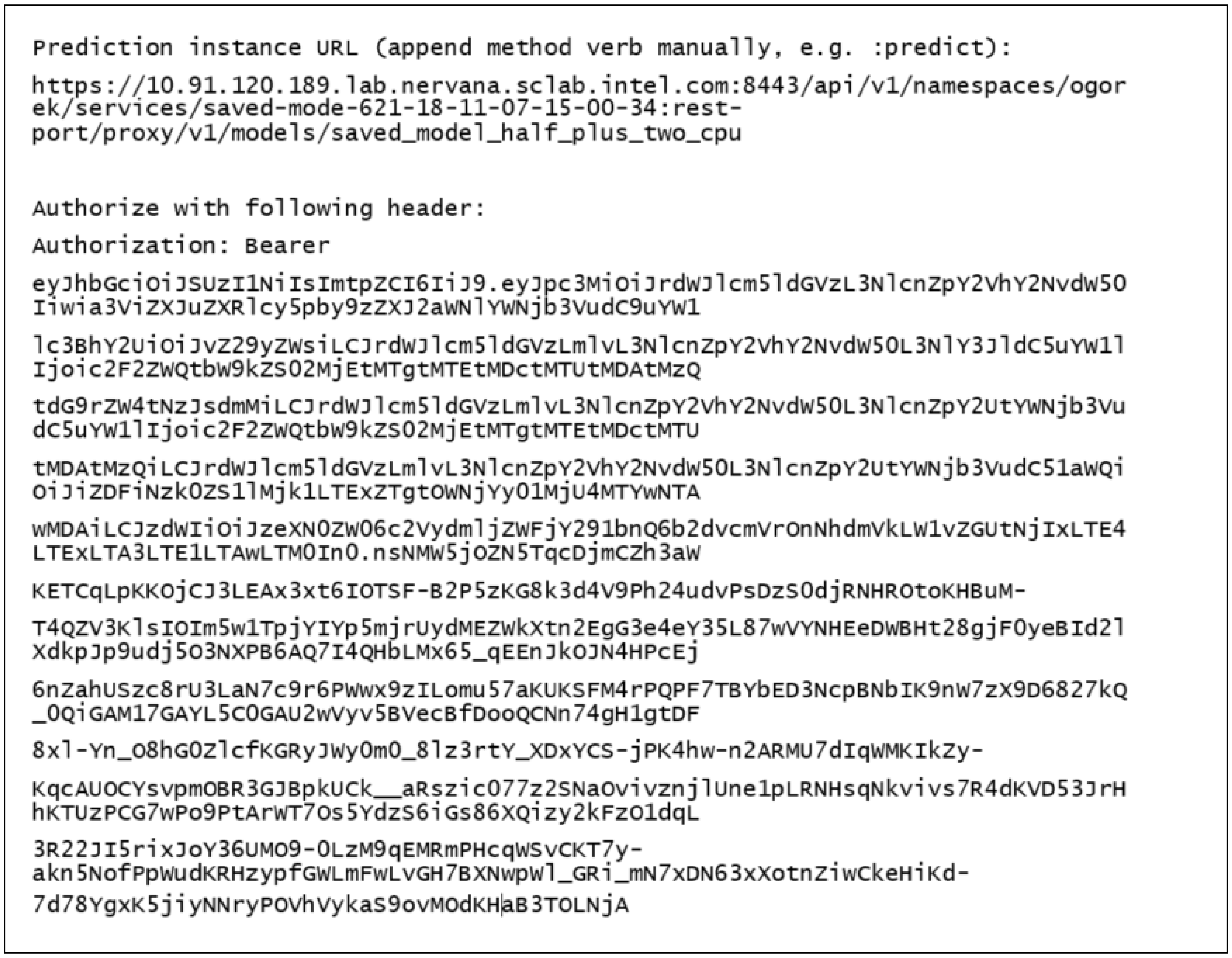
By default, `dlsctl predict stream` will use the `PREDICT` method verb. You can change it by passing the `--method-verb` parameter to the `dlsctl predict stream` command, for example:

```
dlsctl 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-collecting scripts and 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://10.91.120.189.lab.nervana.sclab.intel.com:8443/api/v1/namespaces/ogor
ek/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
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50
Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1
lc3BhY2UiOiJvZ29yZWsiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlY3JldC5uYW1l
Ijoic2F2ZWQtbW9kZS02MjEtMTgtMTEtMDctMTUtMDAtMzQ
tdG9rZW4tNzJsdmMiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3Vu
dC5uYW1lIjoic2F2ZWQtbW9kZS02MjEtMTgtMTEtMDctMTU
tMDAtMzQiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQi
OiJiZDFiNzk0ZS1lMjk1LTExZTgtOWNjYy01MjU4MTYwNTA
wMDAiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6b2dvcmVrOnNhdmVkLW1vZGUtNjIxLTE4
LTExLTA3LTE1LTAwLTM0In0.nsNMW5jOZN5TqcDjmCZh3aW
KETCqLpKKOjCJ3LEAx3xt6IOTSF-B2P5zKG8k3d4V9Ph24udvPsDzS0djRNHROtoKHBuM-
T4QZV3KlsIOIm5w1TpjYIYp5mjrUydMEZWkXtn2EgG3e4eY35L87wVYNHEeDWBHt28gjF0yeBId2l
XdkpJp9udj5O3NXPB6AQ7I4QHbLMx65_qEEnJkOJN4HPcEj
6nZahUSzc8rU3LaN7c9r6PWwx9zILomu57aKUKSFM4rPQPF7TBYbED3NcpBNbIK9nW7zX9D6827kQ
_0QiGAM17GAYL5C0GAU2wVyv5BVecBfDooQCNn74gH1gtDF
8xl-Yn_O8hG0ZlcfKGRyJWy0m0_8lz3rtY_XDxYCS-jPK4hw-n2ARMU7dIqWMKIkZy-
KqcAUOCYsvpmOBR3GJBpkUCk__aRszic077z2SNaOvivznjlUne1pLRNHsqNkvivs7R4dKVD53JrH
hKTUzPCG7wPo9PtArWT7Os5YdzS6iGs86XQizy2kFzO1dqL
3R22JI5rixJoY36UMO9-0LzM9qEMRmPHcqWSvCKT7y-
akn5NofPpWudKRHzypfGWLmFwLvGH7BXNwpWl_GRi_mN7xDN63xXotnZiwCkeHiKd-
7d78YgxK5jiyNNryPOVhVykaS9ovMOdKHaB3TOLNjA
```

Accessing the REST API with curl

Here is an example of 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. You can 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 background, use this command:

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

Please note local port number of tunnel you entered above ; it will be produced by kubectl port-forward if you do not explicitly specify it.

Now you can access 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:

```
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, like integrating data collecting scripts/applications with prediction instances. It should provide better performance than REST API.

In order 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 &
```

Please note 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 using a dedicated client gRPC client (such as: https://github.com/tensorflow/serving/blob/master/tensorflow_serving/example/mnist_client.py).

Alternatively, use gRPC CLI client of your choice (such as: grpcc or polyglot) and connect to:

`localhost:<local tunnel port number>`.

References

- https://www.tensorflow.org/serving/serving_basic
- https://www.tensorflow.org/serving/docker
- https://www.tensorflow.org/serving/api_rest

Managing Users and Resources

This section discusses the following topics:

- Creating a User Account
- Deleting a User Account
- Viewing all User Activity
- Accessing the Kubernetes Dashboard

Creating a User Account

The user is the data scientist who wants to perform deep learning experiments to train models that will, after training and testing, be deployed in the field. The user has full control (list/read/create/terminate) over his/her own experiments and has read access (list/read) of experiments belonging to other users on this cluster. Creating a new user account creates a user account configuration file compliant in format with kubectl configuration files.

Note 1: A user with the same name cannot be created immediately after its' removal. The reason is that the user's related Kubernetes objects are deleted asynchronously by Kubernetes and this can take some time. Consider waiting perhaps thirty minutes before creating a user with the same name as the user just deleted.

Note 2: User names are limited to 32 characters maximum, must be lowercase, no underscores, periods, or special characters, and must start with a letter not a number. You can use a hyphen to join user names, for example: john-doe.

To create a user, perform these steps:

1. The `dlsctl user create <username>` command sets up a namespace and associated roles for the named user on the cluster. It sets up "home" directories, named after the username, on the "input" and "output" network shares with file-system level access privileges.
   - **Execute** `dlsctl user create <username>`
2. The above command 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.
   - **Execute**: `cp <username>.config ~/<local_user_folder>/.`
3. Use the export command to set this variable for user:
   - **Execute**: `export KUBECONFIG=~/<local_user_folder>/<username>.config`
4. To verify that the new user has been created:
   - **Execute**: `dlsctl user list`

   The above command lists all users, including the new user just added.

Deleting a User Account

Only an administrator can delete a user.

Deleting a user removes that user's account from the Intel® DL Studio software; that user will not be able to log in to the system. This will halt and remove all experiments and pods, however all artifacts related to that user's account, such as the users input and output folders and all data related to past experiments he/she submitted will remain.

- **Execute**: `dlsctl user delete <user_name>`

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):

- **Execute**: `dlsctl user delete <user_name> -p`

Both commands above will ask for confirmation. Enter Y or Yes to delete the user.

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

Viewing All User Activity

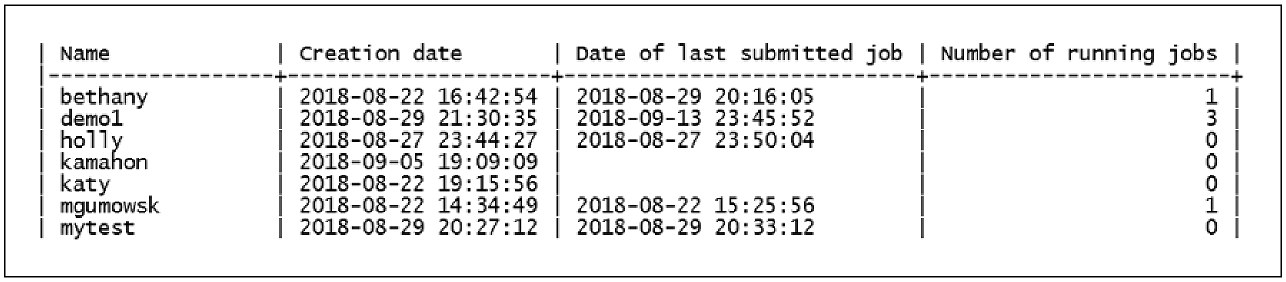
The command `dlsctl user list` displays all current users and all of their experiments (with status). The command shows the following information:

- 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. Previously deleted users are not shown.

- **Execute**: `dlsctl user list`

Following are example results (not all columns are shown).



```
| Name             | Creation date       | Date of last submitted job | Number of running jobs |
|------------------+---------------------+----------------------------+------------------------+
| bethany          | 2018-08-22 16:42:54 | 2018-08-29 20:16:05        |                      1 |
| demo1            | 2018-08-29 21:30:35 | 2018-09-13 23:45:52        |                      3 |
| holly            | 2018-08-27 23:44:27 | 2018-08-27 23:50:04        |                      0 |
| kamahon          | 2018-09-05 19:09:09 |                            |                      0 |
| katy             | 2018-08-22 19:15:56 |                            |                      0 |
| mgumowsk         | 2018-08-22 14:34:49 | 2018-08-22 15:25:56        |                      1 |
| mytest           | 2018-08-29 20:27:12 | 2018-08-29 20:33:12        |                      0 |
```

Accessing the Kubernetes Resource Dashboard

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. Here is an overview: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

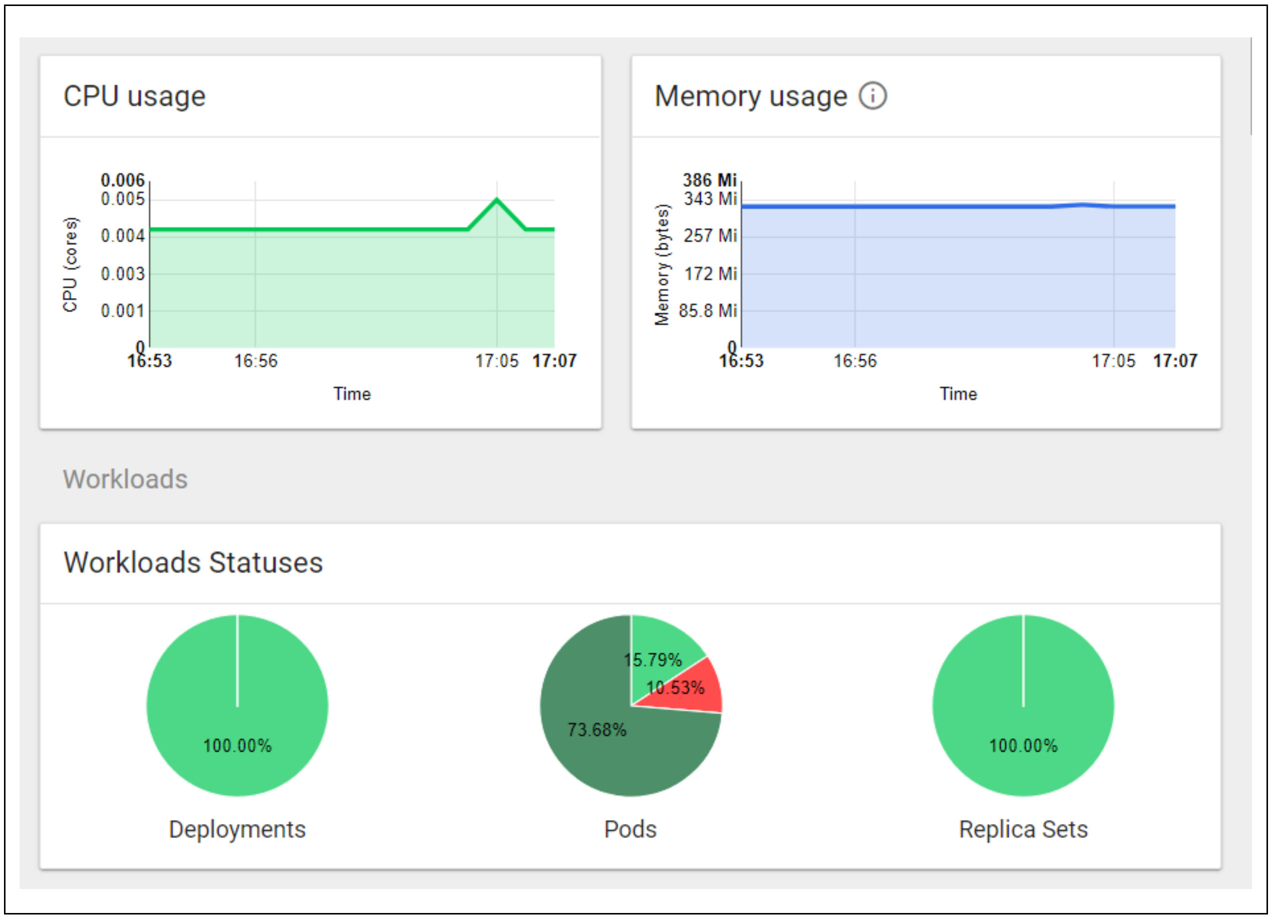
To access Kubernetes:

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.

Following is an example display of the Kubernetes dashboard.

Figure 9: Kubernetes Dashboard—Example Only



CLI Commands

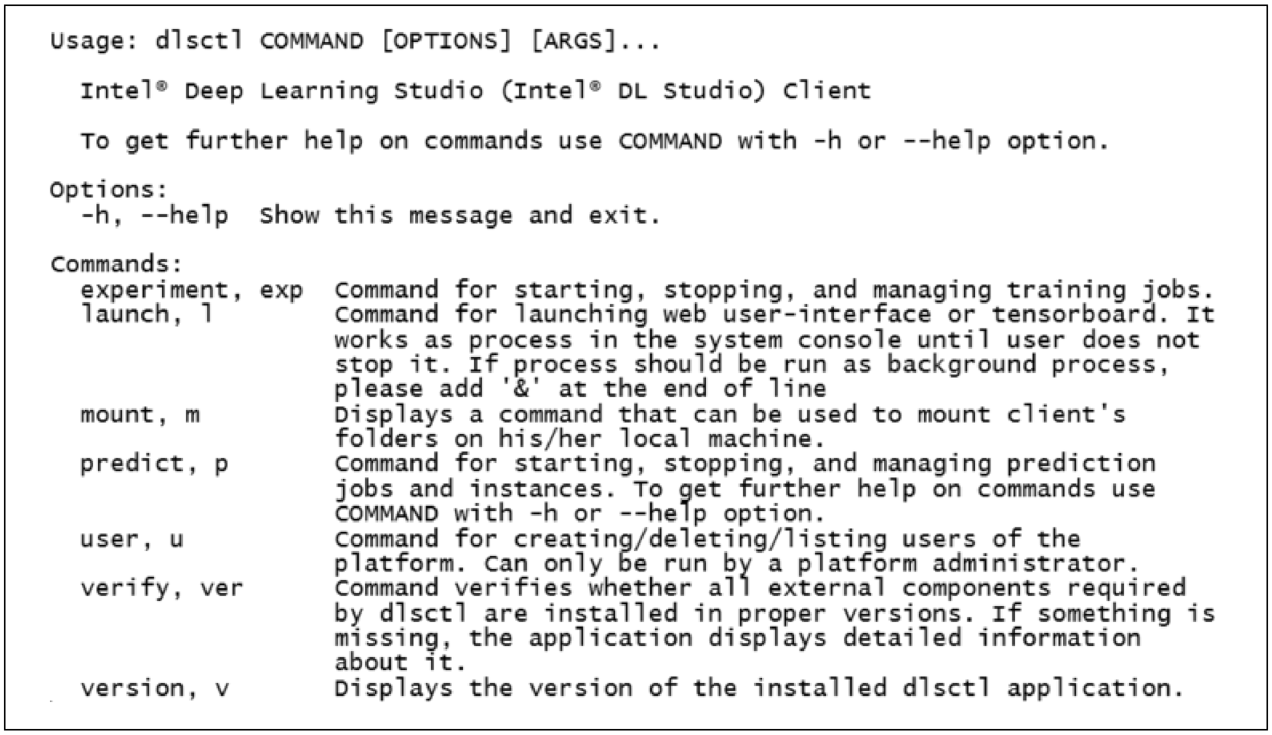
This section discusses the following topics:

- Viewing CLI Command Help at the Command Line
- experiment Command
- launch Command
- mount Command
- user Command
- verify Command
- version Command

Viewing CLI Command Help at the Command Line

The --`help` option provides man-page style help for each `dlsctl` command. You can view help for any command and subcommand, and all related parameters.

Entering `dlsctl --help` provides a listing of all dlsctl commands (without subcommands), as shown next.



```
Usage: dlsctl COMMAND [OPTIONS] [ARGS]...

  Intel® Deep Learning Studio (Intel® DL Studio) Client

  To get further help on commands use COMMAND with -h or --help option.

Options:
  -h, --help  Show this message and exit.

Commands:
  experiment, exp   Command for starting, stopping, and managing training jobs.
  launch, l         Command for launching web user-interface or tensorboard. It
                    works as process in the system console until user does not
                    stop it. If process should be run as background process,
                    please add '&' at the end of line
  mount, m          Displays a command that can be used to mount client's
                    folders on his/her local machine.
  predict, p        Command for starting, stopping, and managing prediction
                    jobs and instances. To get further help on commands use
                    COMMAND with -h or --help option.
  user, u           Command for creating/deleting/listing users of the
                    platform. Can only be run by a platform administrator.
  verify, ver       Command verifies whether all external components required
                    by dlsctl are installed in proper versions. If something is
                    missing, the application displays detailed information
                    about it.
  version, v        Displays the version of the installed dlsctl application.
```

You can view help for any command and available subcommand(s). The following example shows generic syntax; brackets are optional.

Execute: `dlsctl [command_name] [subcommand] --help`

experiment Command

This overall purpose of this command and subcommands is to submit and manage experiments. Following are the subcommands for the dlsctl experiment command.

- submit Subcommand
- list Subcommand
- cancel Subcommand
- view Subcommand
- logs Subcommand
- interact Subcommand
- template_list Subcommand

submit Subcommand

Synopsis

Submits training jobs. We can 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

```
dlsctl 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.

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, e.g., other .py files, data, etc. If not given, then its content won't be copied into the docker image created by the `dlsctl submit` command. `dlsctl` copies all content, preserving its structure, including subfolder(s).
`-t, --template <template_name> TEXT`	No	Name of a template that will be used by `dlsctl` 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 `dlsctl experiment template_list` command.
`-n, --name TEXT`	No	Name for this experiment.

`-p, --pack_param <TEXT TEXT>...`	No	Additional pack param in format: 'key value' or 'key.subkey.subkey2 value'. For lists use: 'key "['val1', 'val2']"' For maps use: 'key "{'a': 'b'}"'
`-pr, --parameter_range TEXT...` `[definition]` `<TEXT TEXT>...`	No	If the parameter is given, dlsctl will start 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. 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 `dlsctl` will launch a number of experiments equal to a number of values between x and y (including both values) with step step. - set of values - `{x, y, z}` - this form says that `dlsctl` 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, `dlsctl` will launch an experiment with a set of parameters defined in `[definition]` argument. Optional. Format of the `[definition]` argument is as follows : `{[param1_name]: [param1_value], [param2_name]: [param2_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 `ps` parameter is given more than once, then `dlsctl` will start as many experiments as there is occurrences of this parameter in a call.
`-e, --env TEXT`	No	Set of values of one or several parameters. Environment variables passed to training. User can pass as many environmental variables as it is needed - each variable should be in such case passed as a separate -e parameter.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Additional Remarks

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

Examples

```
dlsctl experiment submit mnist_single_node.py -sfl /data -- --
data_dir=/app/data --num_gpus=0
```

Starts a single node training job using `mnist_single_node.py` script located in a folder from which dlsctl command was issued. Content of the /data folder is copied into docker image (into /app folder - which is a work directory of docker images created using tf-training pack). Arguments `--data-dir` and `--num_gpus` are passed to a script.

list Subcommand

Synopsis

Displays 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

```
dlsctl 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.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

List of experiments matching criteria given in 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 table returned by this command.

```
| Experiment           | Parameters used     | Metrics        | Time submitted | Username | Status   |
+----------------------+---------------------+----------------+----------------+----------+----------|
| exp1-20181122:0830-1 | learningrate: 0.1   | loss: 0.05     |  20181122:0830 | jdoe     | Complete |
|                      | padding: 2          | accuracy: 0.9  |                |          |          |
|                      | layers: 10          |                |                |          |          |
| exp1-20181122:0830-2 | learningrate: 0.01  |                |  20181122:0830 | jdoe     | Running` |
| exp1-20181122:0830-3 | learningrate: 0.001 |                |  20181122:0830 | jdoe     | Queued   |
```

Examples

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

```
dlsctl experiment list
```

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

```
dlsctl experiment list -n train
```

cancel Subcommand

Synopsis

Cancels training chosen based on provided parameters.

Syntax

```
dlsctl experiment cancel [options] NAME
```

Arguments

Name	Required	Description
`NAME`	Yes	Name of an experiment/pod/status of a pod to be cancelled. If any such an object has been found - the command displays question whether this object should be cancelled.

Options

Name	Required	Description
`-m, --match TEXT`	No	If given, command searches for experiments matching the value of this option. This option cannot be used along with the NAME argument.
`-p, --purge`	No	If given - 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, command searches pods by their status and deletes them.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Description of a problem - if any occurs. Otherwise information that training job/jobs was/were cancelled successfully.

Examples

```
dlsctl experiment cancel t20180423121021851
```

Cancels experiment with `t20180423121021851` name.

view Subcommand

Synopsis

Displays basic details of an experiment, such as the name of an experiment, parameters, submission date, etc.

Syntax

```
dlsctl 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	Exposes TensorBoard instance with data from an experiment to a user.
`-u, --username TEXT`	No	Name of the user who submitted this experiment. If not given - only experiments of a current user are taken into account.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

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

Examples

```
dlsctl experiment view experiment_name_2 -t
```

Displays details of an `experiment_name_2` experiment and exposes TensorBoard instance with experiment's data to a user.

logs Subcommand

Synopsis

Displays logs from experiments - logs to be displayed are chosen based on parameters given in command's call.

Syntax

```
dlsctl experiment logs [options] EXPERIMENT_NAME
```

Arguments

Name	Required	Description
`EXPERIMENT_NAME`	Yes	Name of an experiment logs from which will be displayed.

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' - command returns logs with matching status from an experiment and matching EXPERIMENT_NAME.
`-m, --match TEXT`	No	If given, 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, 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.
`-f, --follow`	No	Specify if logs should be streamed. Only logs from a single experiment can be streamed.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Errors generate messages with description of their cause. Otherwise, this command returns experiment logs that can be filtered using command options.

Examples

```
dlsctl 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

Launches a local browser with Jupyter notebook. If script's name is given as a parameter of a command, then this script is displayed in a notebook.

Syntax

```
dlsctl experiment interact [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	No	Name of a Jupyter notebook session. If session with a given name already exists, then a user is connected to this session.
`-f, --filename TEXT`	No	Name of a script used in the Jupyter notebook's session.
`-p, --pack_param <TEXT TEXT>...`	No	Additional pack param 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 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. User can pass as many environmental variables as it is needed. Each variable should be in such case 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).
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

In case of any problems, it displays a message with a description of causes of problems. Otherwise it launches a default web browser with Jupyter notebook and displays the address under which this session is provided.

template_list Subcommand

Synopsis

The command returns a list of templates installed on a client machine. Template contains all details needed to properly deploy training job on a cluster.

Syntax

```
dlsctl experiment template_list [options]
```

Options

Name	Required	Description
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Command returns a List of existing templates, or the message "Lack of installed packs." message if there are no templates installed.

Examples

```
dlsctl experiment template_list
```

launch Command

Synopsis

This command launches a browser for the web UI or TensorBoard.

- webui Subcommand
- tensorboard Subcommand

webui Subcommand

Synopsis

Launches the Intel DL Studio web user interface with credentials.

Syntax

```
dlsctl launch webui [options]
```

Arguments

None.

Options

Name	Required	Description
`--no-launch`	No	Create tunnel without launching the web browser.
`-p, --port [port] INTEGER RANGE`	No	If given, application will be exposed on a local machine under [port] port.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Link to an exposed application.

Examples

```
dlsctl 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

Launches the TensorBoard* web user interface front-end with credentials, with the indicated experiment loaded.

Syntax

```
dlsctl launch tb [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 experiment that should be displayed in TensorBoard belongs to a current user - user has to give only its name. If this experiment is owned by another user - 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	INTEGER RANGE - Local port on which tensorboard service client will be started.
`-p, --port [port]`	No	If given, application will be exposed on a local machine under [port] port.
`-v, --verbose`	No	Set verbosity level: -v for INFO -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Link to an exposed application.

Example

```
dlsctl launch tensorboard experiment75
```

An example might look like this:

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

mount Command

Synopsis

The `mount` command displays another command that can be used to mount/unmount a client's folders on/from his/her local machine. See also, list Subcommand on page 63

Syntax

```
dlsctl mount [options]
```

Options

Name	Required	Description
`-v, --verbose`	No	Displays in real time information about commands being executed to run this command.
`-h, --help`	No	Show help message and exit.

Returns

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

list Subcommand

Synopsis

Displays a list of Intel DL Studio related folders mounted on a user's machine. If run using admin credentials, displays mounts of all users.

Syntax

```
dlsctl mount list
```

Returns

List of mounted folders. Each row contains additional information (i.e. 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 exposing Intel DL Studio shares. Other mounted folders are not taken into account.

predict Command

Synopsis

Use this command to start, stop, and manage prediction jobs.

- batch Subcommand
- cancel Subcommand
- launch Subcommand
- list Subcommand
- stream Subcommand

batch Subcommand

Synopsis

Starts a new batch instance that will perform prediction on provided data. Uses a specified dataset to perform inference. Results are stored in an output file.

Syntax

```
dlsctl predict batch [options]
```

Options

Name	Required	Description
`-n, --name`	No	Name of predict session.
`-m, --model-location TEXT`	Yes	Path to saved model that will be used for inference. Model must be located on one of the input or output system shares (e.g. /mnt/input/saved_model).
`-d, --data TEXT`	Yes	Location of a folder with data that will be used to perform the batch inference. Value should points out the location from one of the system's shares.
`-o, --output TEXT`	No	Location of a folder where outputs from inferences will be stored. Value should points out the location from one of the system's shares.
`-mn, --model-name`	No	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.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

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

cancel Subcommand

Synopsis

This command cancels prediction instance/s chosen based on criteria given as a parameter.

Syntax

```
dlsctl 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, 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, is removed from the system.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

Description of a problem - if any occurs. Otherwise information that training job/jobs was/were cancelled successfully.

launch Subcommand

Synopsis

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

```
dlsctl predict launch [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	No	The name of this prediction instance.
`-m, --model-location TEXT`	Yes	Path to saved model that will be used for inference. Model must be located on one of the input or output system shares (e.g. /mnt/input/home/saved_model).
`-l, --local_model_location PATH`	No	Local path to saved model that will be used for inference. Model content will be copied into an image.

`-mn, --model-name TEXT`	No	Name of a model passed as a servable name. By default it is the name of directory in model's location.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

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

Example

```
dlsctl 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://10.91.120.94.lab.nervana.sclab.intel.com:8443/api/v1/namespaces/mgumo
wsk/services/test/proxy/v1/models/home

Authorize with following header:
Authorization: Bearer
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50
Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJtZ3Vtb3dzayIsImt1YmVybmV0ZXMuaW8vc
2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJ
kZWZhdWx0LXRva2VuLWNrNXpkIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNl
LWFjY291bnQubmFtZSI6ImRlZmF1bHQiLCJrdW
Jlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiIzNDBlODEzNC1
hZDE1LTExZTgtYTcwMS01MjU4MTYwNDA2MDAiL
CJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6bWd1bW93c2s6ZGVmYXVsdCJ9.kTLiLKpi1MzNM
zXTmaSEbaCLYt4paZ_xFVT19aT6Fsf9ce-DTVu
RfAqd7Pf_CMktU9SJlZ_aN9WP35nysn-op8bFH5jLCJLSHBGBPkk7daU-
7WQF4FfAD5gurLYZ2arNMd-FROEG9DjPOomaPeE5GLMhHwzKBRFzprr_jI
QG72QcCOA7lb4lRnVusiYtlsgqnDPbhRH8XBGWk429HakSrrjiSBfKivpNXYzlZ9YSMQrZn-
ZbIpr_QSVfZlL_4IWmmvCvGOXTnqBKfOVbzf6ndsia5
hqTpgFV8mRkoblre2KiOvXPXy-
xvysbAC3CMKAVoxCbzl7TmD6xow9vbWaMvKEnJk4FRM4g5tjqz4khJnPlAMvYWYH6qLgJx0yyVWOY
NB5Iuf2m8UwF
mz7fEBvxDYAVAt1j-NOC-yorfnr-yQaxRbSKsvBRX-x_UqbpPL5yw_0R-XpTov4yj9t6ck4HC8h1-
mgbkRdrIzNIKpv4EnC2WSHp2U5RpiKWXCmZpRk
y27xXkYZ2vzHdRLinWqDyaq42w0tlkOoAKonKfcHLNCWnlgcjoYT1kZ7pUKRI5ZNjQiX4Tbd_kqmb
yNC6DWmMDBiufgpRLxyhxRJZ5BUCC_sTDX10PE
X5HcXfYIHtPqKK43ahStzpWKAspfqAVWX_gwBfhOeKd-kdbGTXP2QRQsQ
```

list Subcommand

Synopsis

Displays a list of inference instances with some basic information regarding each of them. Results are sorted using a date of creation of an inference instance - starting from the latest one.

Syntax

```
dlsctl 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 narrow down list to prediction instances that match status.
`-u, --uninitialized`	No	List uninitialized prediction instances, i.e., 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.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

List of inference instances matching criteria given in command's options.

stream Subcommand

Synopsis

Perform stream inference task on launched prediction instance.

Syntax

```
dlsctl predict stream [options]
```

Options

Name	Required	Description
`-n, --name TEXT`	Yes	Name of prediction session.
`-d, --data PATH`	Yes	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.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

user Command

Use this command to create, delete, and manage users.

- create Subcommand
- delete Subcommand
- list Subcommand

create Subcommand

Synopsis

Creates and initializes a new DLS4E user. This command must be executed when kubectl is used by a dlsctl 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

```
dlsctl user create [options] USERNAME
```

Arguments

Name	Required	Description
USERNAME	Yes	Name of a user that will be created. This value must a valid OS level user.

Options

Name	Required	Description
`-l, --list_only`	No	If given - content of the generated user's config file is displayed on the screen only. If not given - file with configuration is saved on disk.
`-f, --filename`	No	Name of file where user's configuration will be stored. If not given configuration is stored in the `config.<username>` file.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Additional remarks

In case of any errors during saving of a file with a configuration - the command displays a content of the configuration file on the screen - even if -l option wasn't chosen.

If an admin tries to create 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 the operation of a creation of a new user should be postponed for a while - until all user's objects are removed.

Returns

In case of any problems - message describing their cause/causes. Otherwise message is returned indicating success. If -list_only option was given - the command displays also a content of a configuration file.

Notes

User name must meet the following rules:

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

Examples

```
dlsctl user create jdoe
```

Creates user jdoe.

delete Subcommand

Synopsis

This command deletes a user with a given name. If option -p, --purge was added - it removes also all artifacts related to a removed user, like content of user's folders and data of experiments and runs.

```
dlsctl user delete USERNAME
```

Arguments

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

Additional remarks

Before removing a user, the commands asks for a final confirmation. If user chooses Yes - chosen user is deleted.

Deletion of a user may take a while to be fully completed. Command waits for up to 30 seconds for a complete removal of user. If after this time user hasn't been deleted completely - the command displays information that a user is still being deleted. In this case the user won't 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.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

Returns

A message regarding the command's completion. In case of any problems - short description of their causes.

Examples

```
dlsctl user delete jdoe -p
```

Removes jdoe user with all his/her artifacts.

list Subcommand

Synopsis

Lists all currently configured users.

Syntax

```
dlsctl user list [options]
```

Options

Name	Required	Description
`-c, --count INTEGER RANGE`	No	If given - command displays c last rows.
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

verify Command

Synopsis

Checks whether all prerequisites required by dlsctl are installed and have proper versions.

Syntax

```
dlsctl verify
```

Options

Name	Required	Description
`-v, --verbose`	No	Set verbosity level: -v for INFO, -vv for DEBUG
`-h, --help`	No	Show help message and exit.

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.
draft verified successfully.
kubectl verified successfully.
kubectl server verified successfully.
helm client verified successfully.
helm server verified successfully.
docker client verified successfully.
docker server verified successfully.
```

version Command

Synopsis

Returns the version of Intel Deep Learning Studio.

Syntax

```
dlsctl version
```

Options

Name	Required	Description
`-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 `dlsctl` application version of both client platform and server.