DEEP-Hybrid-DataCloud documentation Documentation

Release DEEP-1 (Genesis)

DEEP-Hybrid-DataCloud consortium

Jan 14, 2019

Contents

CHAPTER 1

User documentation

If you are a user (current or potential) you should start here.

1.1 User documentation

Todo: Documentation is being written at this moment.

If you want a quickstart guide, please check the following link.

1.1.1 Quickstart Guide

Todo: Provide information on (at least):

1. How to download and test a model (i.e. get docker, go to the marketplace, get the model, run it).
2. Use cookiecutter to develop a model
3. Integrate an existing model with DEEPaaS
4. Create a container from a model

Download module from the marketplace

1. go to DEEP Open Catalog
2. Browse available modules
3. Find the module and get it either from Docker Hub (easy) or Github (pro)

Run a module locally

Docker Hub way (easy)

Prerequisites

- docker
- If you want GPU support you can install nvidia-docker along with docker or install udocker instead of docker. udocker is entirely a user tool, i.e. it can be installed and used without any root priveledges, e.g. in a user environment at HPC cluster.

1. **Run the container**

To run the Docker container directly from Docker Hub and start using the API simply run the following:

Via docker command:

```
$ docker run -ti -p 5000:5000 -p 6006:6006 deephdc/deep-oc-module_of_interest
```

With GPU support:

```
$ nvidia-docker run -ti -p 5000:5000 -p 6006:6006 deephdc/deep-oc-module_of_
↪interest
```

Via udocker:

```
$ udocker run -p 5000:5000 -p 6006:6006 deephdc/deep-oc-module_of_interest
```

Via udocker with GPU support:

```
$ udocker pull deephdc/deep-oc-module_of_interest
$ udocker create --name=module_of_interest deephdc/deep-oc-module_of_interest
$ udocker setup --nvidia module_of_interest
$ udocker run -p 5000:5000 -p 6006:6006 module_of_interest
```

2. **Access the module via API**

To access the downloaded module via API, direct your web browser to http://127.0.0.1:5000. If you are training a model, you can go to http://127.0.0.1:6006 to monitor the training progress (if such monitoring is available for the model).

For more details on particular models, please, read *model* documentation.

Github way (pro)

Prerequisites

- docker

Using Github way allows to modify the Dockerfile for including additional packages, for example.

1. Clone the DEEP-OC-module_of_interest github repository:

```
$ git clone https://github.com/indigo-dc/DEEP-OC-module_of_interest
```

2. Build the container:

```
$ cd DEEP-OC-module_of_interest
$ docker build -t deephdc/deep-oc-module_of_interest .
```

3. Run the container and access the module via API as described *above*

Note: One can also clone the source code of the module, usually located in the 'module_of_interest' repository.

Run a module on DEEP Pilot Infrastructure

Prerequisites

- DEEP-IAM registration
- oidc-agent installed and configured for DEEP-IAM
- orchent tool

If your are going to use DEEP-Nextcloud for storing you data you also have to:

- Register at DEEP-Nextcloud
- Include rclone installation in your Dockerfile (see *rclone howto*)
- Include call to rclone in your code (see *rclone howto*)

In order to submit your job to DEEP Pilot Infrastructure one has to create TOSCA YAML file.

The submission is then done via

```
$ orchent depcreate ./topology-orchent.yml '{}'
```

If you also want to access DEEP-Nextcloud from your container via rclone, you can create a following bash script for job submission:

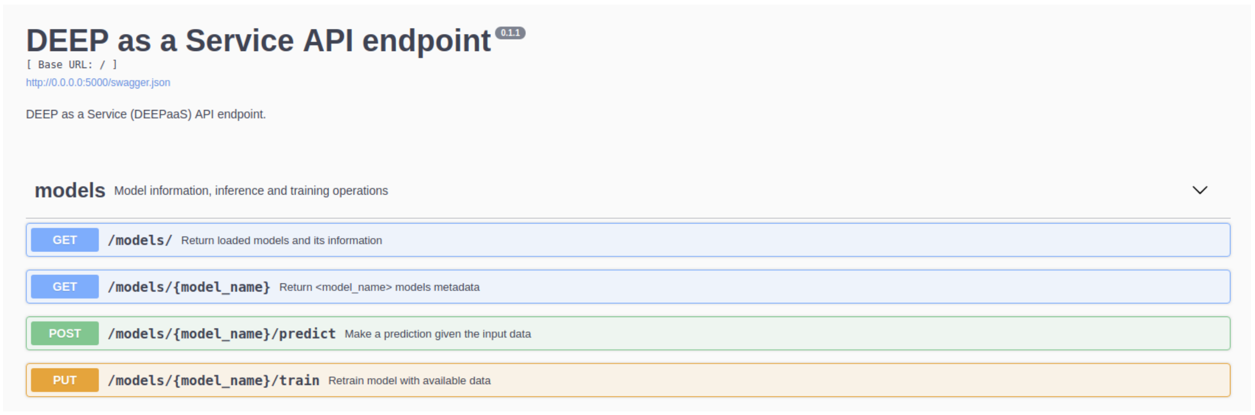
```
#!/bin/bash

orchent depcreate ./topology-orchent.yml '{ "rclone_url": "https://nc.deep-hybrid-
↪datacloud.eu/remote.php/webdav/",
                                            "rclone_vendor": "nextcloud",
                                            "rclone_user": <your_nextcloud_username>
                                            "rclone_pass": <your_nextcloud_password> }
↪'
```

To check status of your job

```
$ orchent depshow <Deployment ID>
```

Integrate your model with the API



DEEP as a Service API endpoint 0.1.1

[Base URL: /]

http://0.0.0.0:5000/swagger.json

DEEP as a Service (DEEPaaS) API endpoint.

models Model information, inference and training operations

GET /models/ Return loaded models and its information

GET /models/{model_name} Return <model_name> models metadata

POST /models/{model_name}/predict Make a prediction given the input data

PUT /models/{model_name}/train Retrain model with available data

The DEEPaaS API enables a user friendly interaction with the underlying Deep Learning modules and can be used both for training models and doing inference with the services. Check the full *API guide* for the detailed info.

An easy way to integrate your model with the API and create Dockerfiles for building the Docker image with the integrated *DEEPaaS API* is to use our *DEEP UC template* when developing your model.

A more in depth documentation, with detailed description on the archicture and components is provided in the following sections.

1.1.2 Overview

Architecture overview

There are several different components in the DEEP-HDC project that are relevant for the users. Later on you will see how each *different type of user* can take advantage of the different components.

The marketplace

This is one of the most important components. It is a catalogue of modules that every user can have access to. We can divide the modules in two different categories:

- **Models** are modules (eg. an image classifier) that an user can train on their own data to create a new service. They have not already been trained to perform any particular task.
- **Services** are models that have been trained for a specific task (eg. an plant classifier). They usually derive from model, although sometimes, for very specific tasks, a module can be both a model and a service.

For more information have a look at the marketplace.

Todo: If everyone agrees in this nomenclature, tags should be changed in the marketplace. Also models –> modules in the marketplace navigation bar

Todo: Add links to the marketplace url of the plant classifier in Tensorflow (which is still not uploaded to the marketplace)

The API

The DEEPaaS API is a key component for making the modules accessible to everybody. It provides a graphical interface that the user can take advantage of to easily interact with the module. It is available for both inference and training. Advanced users that want to create new modules can make them *compatible with the API* to make them available to the whole community.

For more information take a look on the full *API guide*.

The storage resources

Storage is essential for user that want to create new services by training models on their custom data. For the moment we support saving data into DEEP-Nextcloud.

Todo: Check rclone integration with Dropbox and Google Drive

Our different user roles

DEEPaaS is focused with three different types of users in mind, which vary in their machine learning knowledge and the usage they will make of DEEPaaS.

The basic user

This user wants to use the services that have been already developed and *test them with their data*, and therefore don't need to have any machine learning knowledge. For example, they can take an already trained network for plant classification that has been containerized, and use it to classify their own plant images.

What DEEPaaS can offer to you:

- a *catalogue* full of ready-to-use services to make inference with your data
- an *API* to easily interact with the services
- solutions to run the inference in the Cloud or in your local resources
- the ability to develop complex topologies by composing different modules

The intermediate user

The intermediate user wants to *retrain an available model* to perform the same task but fine tuning it to their own data. They still might not need high level knowledge on modelling of machine learning problems, but typically do need basic programming skills to prepare their own data into the appropriate format. Nevertheless, they can re-use the knowledge being captured in a trained network and adjust the network to their problem at hand by re-training the network on their own dataset. An example could be a user who takes the generic image classifier model and retrains it to perform seed classification.

What DEEPaaS can offer to you:

- the ability to train out-of-the-box a model of the *catalogue* on your personal dataset
- an *API* to easily interact with the model
- access to data *storage resources* via DEEP-Nextcloud to save your dataset

- the ability to deploy the developed service on Cloud resources
- the ability to share the service with other users in the user's catalogue

The advanced user

The advanced users are the ones that will *develop their own machine learning models* and therefore need to be competent in machine learning. This would be the case for example if we provided an image classification model but the users wanted to perform object localization, which is a fundamentally different task. Therefore they will design their own neural network architecture, potentially re-using parts of the code from other models.

What DEEPaaS can offer to you:

- a ready-to-use environment with the main DL frameworks running in a dockerized solution running on different types of hardware (CPUs, GPUs, etc)
- access to data *storage resources* via DEEP-Nextcloud to save your dataset
- the ability to deploy the developed module on Cloud resources
- the ability to share the module with other users in the user's *catalogue*
- the possibility to *integrate your module* with the *API* to enable easier user interaction

Todo: Add links to image classifier-tf, plant classifier-tf, seed classifier-tf once they will be uploaded to the marketplace

DEEP Use-Cases template

To simplify development and as an easy way integrate your model with the *DEEPaaS API*, a project template, cookiecutter-data-science*[0], is provided in our GitHub.

In order to create your project based on the template, one has to install and then run cookicutter tool as follows:

```
$ cookiecutter https://github.com/indigo-dc/cookiecutter-data-science
```

You will be asked several questions, e.g.:

- User account at github.com, e.g. 'indigo-dc'
- Project name
- Repository name for the new project
- (main) author name
- Email of the author (or contact person)
- Short description of the project
- Application version
- Choose open source license
- Version of python interpreter
- Docker Hub account name
- Base image for Dockerfile

[0] The more general cockiecutter-data-science template was adapted for the purpose of DEEP.

Note: These parameters are defined in `cookiecutter.json` in the cookiecutter-data-science source.

When these questions are answered, following two repositories will be created locally and immediately linked to your github.com account:

```
~/DEEP-OC-your_project
~/your_project
```

your_project repo

Main repository for your model with the following structure:

```
├─ data                    Placeholder for the data
│
├─ docs                    Documentation on the project; see sphinx-doc.org for details
│
├─ docker                  Directory for development Dockerfile(s)
│
├─ models                  Trained and serialized models, model predictions, or model␣
↪summaries
│
├─ notebooks               Jupyter notebooks. Naming convention is a number (for␣
↪ordering),
│                             the creator's initials (if many user development),
│                             and a short `_` delimited description,
│                             e.g. `1.0-jqp-initial_data_exploration.ipynb`.
│
├─ references              Data dictionaries, manuals, and all other explanatory␣
↪materials.
│
├─ reports                 Generated analysis as HTML, PDF, LaTeX, etc.
│
├─ your_project            Main source code of the project
│   │
│   ├── __init__.py           Makes your_project a Python module
│   │
│   ├── dataset               Scripts to download and manipulate raw data
│   │
│   ├── features              Scripts to prepare raw data into features for modeling
│   │
│   ├── models                Scripts to train models and then use trained models to␣
↪make predictions
│   │
│   └── visualization         Scripts to create exploratory and results oriented␣
↪visualizations
│
├─ .dockerignore           Describes what files and directories to exclude for␣
↪building a Docker image
│
├─ .gitignore              Specifies intentionally untracked files that Git should␣
↪ignore
│
├─ Jenkinsfile             Describes basic Jenkins CI/CD pipeline
│
```

(continues on next page)

(continued from previous page)

```
├─ LICENSE                 License file

├─ README.md               The top-level README for developers using this project.

├─ requirements.txt        The requirements file for reproducing the analysis␣
↪environment,
│                              e.g. generated with `pip freeze > requirements.txt`

├─ setup.cfg               makes project pip installable (pip install -e .)

├─ setup.py                makes project pip installable (pip install -e .)

├─ test-requirements.txt   The requirements file for the test environment

└─ tox.ini                 tox file with settings for running tox; see tox.testrun.org
```

Certain files, e.g. `README.md`, `Jenkinsfile`, `setup.cfg`, development Dockerfiles, `tox.ini`, etc are pre-populated based on the answers you provided during cookiecutter call (see above).

DEEP-OC-your_project

Repository for the integration of the *DEEPaaS API* and your_project in one Docker image.

```
├─ Dockerfile     Describes main steps on integrationg DEEPaaS API and
│                     your_project application in one Docker image
│
├─ Jenkinsfile    Describes basic Jenkins CI/CD pipeline

├─ LICENSE        License file

├─ README.md      README for developers and users.
```

All files get filled with the info provided during cookiecutter execution (see above).

DEEPaaS API

The DEEPaaS API enables a user friendly interaction with the underlying Deep Learning modules and can be used both for training models and doing inference with services.

Integrate your model with the API

To make your Deep Learning model compatible with the DEEPaaS API you have to:

1. **Define the API methods for your model**

Create a Python file (named for example `api.py`) inside your package. In this file you can define any of the *API methods*. You don't need to define all the methods, just the ones you need. Every other method will return a `NotImplementError` when queried from the API.

Here is example of the implementation of the methods

2. **Define the entrypoints to your model**

You must define the entrypoints to this file in the `setup.cfg` as following:

```
[entry_points]
deepaas.model =
    pkg_name = pkg_name.api
```

Here is an example of the entrypoint definition in the `setup.cfg` file.

Methods

get_metadata()
Return metadata from the exposed model.

Returns dictionary containing the model's metadata. Possible keys of this dict can include: 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-email', 'License', etc.

get_train_args(**args*)
Function to expose to the API what are the typical parameters needed to run the training.

Returns a dict of dicts with the following structure to feed the DEEPaaS API parser:

```
{ 'arg1' : {'default': '1',      #value must be a string (use json.dumps to␣
↪convert Python objects)
            'help': '',          #can be an empty string
            'required': False    #bool
            },
  'arg2' : {...
            },
...
}
```

train(*user_conf*)
Function to train your model.

Parameters **user_conf** (*dict*) – User configuration to train a model with custom parameters. It has the same keys as the dict returned by `get_train_args()`. The values of this dict can be loaded back as Python objects using `json.loads`. An example of a possible `user_conf` could be:

```
{'num_classes': 'null',
 'lr_step_decay': '0.1',
 'lr_step_schedule': '[0.7, 0.9]',
 'use_early_stopping': 'false'}
```

predict_file(*path*, ***kwargs*)
Perform a prediction from a file in the local filesystem.

Parameters **path** (*str*) – Path to the file

Returns dictionary of predictions

Return type dict

predict_data(*data*, ***kwargs*)
Perform a prediction from the data passed in the arguments. This method will use the raw data that is passed in the `data` argument to perfom the prediction.

Parameters **data** – raw data to be analized

predict_url (*url*, **args*)
Perform a prediction from a remote URL. This method will perform a prediction based on the data stored in the URL passed as argument.

Parameters **url** (*str*) – URL pointing to the data to be analized

A set of various HowTo examples.

1.1.3 HowTo's

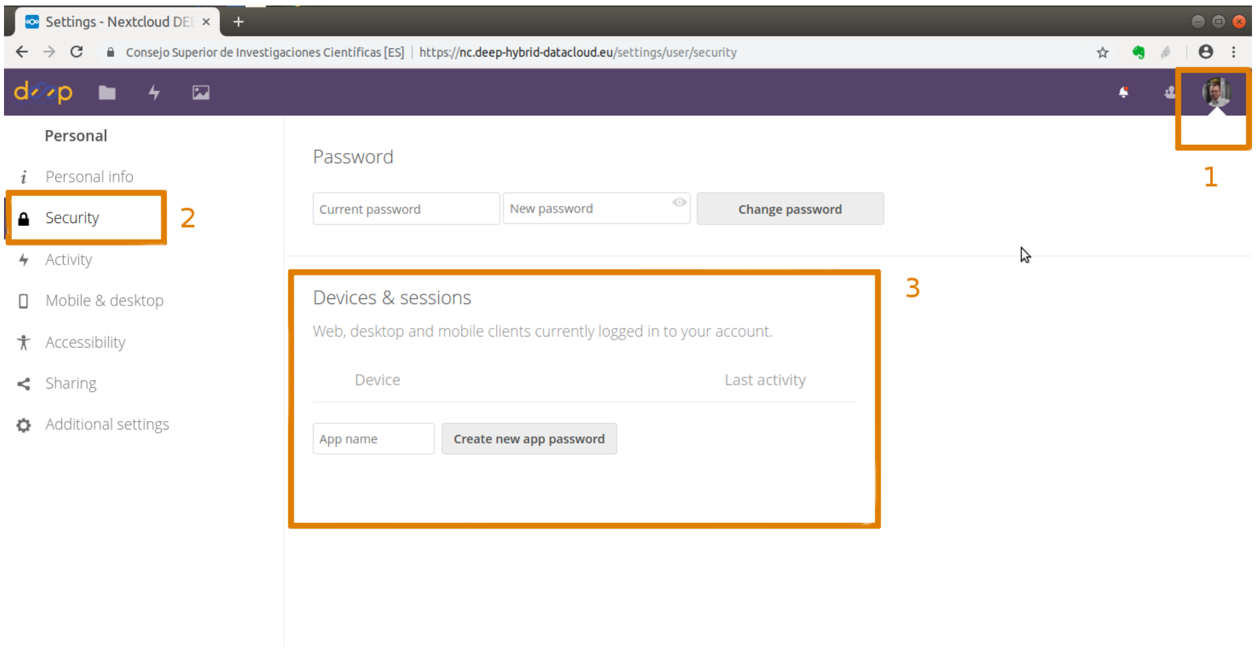
How to use rclone

Installation of rclone in Docker image (pro)

All the applications in the DEEP Open Catalog are packed within a Docker containing rclone installed by default. If you want to create a Docker containing your own application, you should install rclone in the container to be able to access the data stored remotely. The following lines are an example of what has to be added in the Dockerfile when installation is based on Ubuntu. For other Linux flavors, please, refer to the rclone official site

```
# install rclone
      RUN wget https://downloads.rclone.org/rclone-current-linux-amd64.deb && \
      dpkg -i rclone-current-linux-amd64.deb && \
      apt install -f && \
      rm rclone-current-linux-amd64.deb && \
      apt-get clean && \
      rm -rf /var/lib/apt/lists/* && \
      rm -rf /root/.cache/pip/* && \
      rm -rf /tmp/*
```

Nextcloud configuration for rclone



After login into DEEP-Nextcloud with your DEEP-IAM credentials, go to (1) **Settings (top right corner)** → (2) **Security** → (3) **Devices & sessions**. Set a name for you application and clik on **Create new app password**. That user and password is what one needs to include in the rclone config file (`rclone.conf`) to run locally or in the orchent script to generate the deployment when running remotely (see *here* and *here*).

Creating rclone.conf

You can install rclone at your host or run Docker image with rclone installed (see installation steps of rclone above). In order to create the configuration file (`rclone.conf`) for rclone:

```
$ rclone config
 choose "n"  for "New remote"
 choose name for DEEP-Nextcloud, e.g. deep-nextcloud
 choose "Type of Storage" \u2192 "Webdav" (24)
 provide DEEP-Nextcloud URL for webdav access: https://nc.deep-hybrid-datacloud.eu/
↪remote.php/webdav/
 choose Vendor, Nextcloud (1)
 specify "user" (see "Nextcloud configuration for rclone" above). Your username␣
↪starts with "DEEP-IAM-..."
 specify password (see "Nextcloud configuration for rclone" above).
 by default rclone.conf is created in your $HOME/.config/rclone/rclone.conf
```

Important: The rclone.conf file should be in your host, i.e. outside of container. **DO NOT STORE IT IN THE CONTAINER**

Then one has two options:

If your know under what user your run your application in the container (e.g. if docker or nvidia-docker is used, most probably this is 'root') you can mount your host `rclone.conf` into the container as:

```
$ docker run -ti -v $HOSTDIR_WITH_RCLONE_CONF/rclone.conf:/root/.config/rclone/rclone.
↪conf <your-docker-image>
```

i.e. you mount `rclone.conf` file itself directly as a volume.

One can also mount rclone directory with the `rclone.conf` file:

```
$ docker run -ti -v $HOSTDIR_WITH_RCLONE_CONF:/root/.config/rclone <your-docker-image>
```

A more reliable way can be to mount either rclone directory or directly `rclone.conf` file into a pre-defined location and not (container) user-dependent place:

```
$ docker run -ti -v $HOSTDIR_WITH_RCLONE_CONF:/rclone <your-docker-image>
```

One has, however, to call rclone with `--config` option to point to the `rclone.conf` file, e.g:

```
$ rclone --config /rclone/rclone.conf ls deep-nextcloud:/Datasets/dogs_breed/models
```

Example code on usage of rclone from python

Simple example

A simple call of rclone from python is via `subprocess.Popen()`

```
import subprocess

# from deep-nextcloud into the container
command = (['rclone', 'copy', 'deep-nextcloud:/Datasets/dogs_breed/data', '/srv/dogs_
↪breed_det/data'])

result = subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output, error = result.communicate()
```

Advanced examples

More advanced usage includes calling rclone with various options (ls, copy, check) in order to check file existence at Source, check if after copying two versions match exactly.

- rclone_call

```
def rclone_call(src_path, dest_dir, cmd = 'copy', get_output=False):
    """ Function
       rclone calls
    """
     if cmd == 'copy':
            command = (['rclone', 'copy', '--progress', src_path, dest_dir])
     elif cmd == 'ls':
              command = (['rclone', 'ls', src_path])
         elif cmd == 'check':
              command = (['rclone', 'check', src_path, dest_dir])

     if get_output:
             result = subprocess.Popen(command, stdout=subprocess.PIPE,␣
↪stderr=subprocess.PIPE)
     else:
             result = subprocess.Popen(command, stderr=subprocess.PIPE)
     output, error = result.communicate()
     return output, error
```

Todo: This Python code should be corretly indented

- rclone_copy

```
def rclone_copy(src_path, dest_dir, src_type='file'):
    """ Function for rclone call to copy data (sync?)
    :param src_path: full path to source (file or directory)
            :param dest_dir: full path to destination directory (not file!)
            :param src_type: if source is file (default) or directory
            :return: if destination was downloaded, and possible error
            """
    error_out = None

    if src_type == 'file':
        src_dir = os.path.dirname(src_path)
     dest_file = src_path.split('/')[-1]
      dest_path = os.path.join(dest_dir, dest_file)
    else:
        src_dir = src_path
      dest_path =  dest_dir
```

(continues on next page)

(continued from previous page)

```
    # check first if we find src_path
    output, error = rclone_call(src_path, dest_dir, cmd='ls')
    if error:
     print('[ERROR] %s (src):\n%s' % (src_path, error))
     error_out = error
        dest_exist = False
    else:
      # if src_path exists, copy it
      output, error = rclone_call(src_path, dest_dir, cmd='copy')
     if not error:
         # compare two directories, if copied file appears in output
            # as not found or not matching -> Error
          print('[INFO] File %s copied. Check if (src) and (dest) really match..' %␣
↪(dest_file))
          output, error = rclone_call(src_dir, dest_dir, cmd='check')
         if 'ERROR : ' + dest_file in error:
            print('[ERROR] %s (src) and %s (dest) do not match!' % (src_path, dest_
↪path))
                 error_out = 'Copy failed: ' + src_path + ' (src) and ' + \
                            dest_path + ' (dest) do not match'
                dest_exist = False
         else:
            output, error = rclone_call(dest_path, dest_dir,
                                        cmd='ls', get_output = True)
            file_size = [ elem for elem in output.split(' ') if elem.isdigit() ][0]
            print('[INFO] Checked: Successfully copied to %s %s bytes' % (dest_path,␣
↪file_size))
              dest_exist = True
        else:
          print('[ERROR] %s (src):\n%s' % (dest_path, error))
          error_out = error
     dest_exist = False

    return dest_exist, error_out
```

Todo: This Python code should be corretly indented

Develop a model using DEEP UC template

1. Prepare DEEP UC environment

Install cookiecutter (if not yet done)

```
$ pip install cookiecutter
```

Run the DEEP UC cookiecutter template

```
$ cookiecutter https://github.com/indigo-dc/cookiecutter-data-science
```

Answer all questions from DEEP UC cookiecutter template with attentions to `repo_name` i.e. the name of your github repositories, etc. This creates two project directories:

```
~/DEEP-OC-your_project
~/your_project
```

Go to `github.com/your_account` and create corresponding repositories: `DEEP-OC-your_project` and `your_project` Do `git push origin master` in both created directories. This puts your initial code to `github`.

2. Improve the initial code of the model

The structure of `your_project` created using DEEP UC template contains the following core items needed to develop a DEEP UC model:

```
requirements.txt
data/
models/
{{repo_name}}/dataset/make_dataset.py
{{repo_name}}/features/build_features.py
{{repo_name}}/models/model.py
```

2.1 Installing development requirements

Modify `requirements.txt` according to your needs (e.g. add more libraries) then run

```
$ pip install -r requirements.txt
```

You can modify and add more `source files` and put them accordingly into the directory structure.

2.2 Make datasets

Source files in this directory aim to manipulate raw datasets. The output of this step is also raw data, but cleaned and/or pre-formatted.

```
{{cookiecutter.repo_name}}/dataset/make_dataset.py
{{cookiecutter.repo_name}}/dataset/
```

2.3 Build features

This step takes the output from the previous step *Make datasets* and creates train, test as well as validation ML data from raw but cleaned and pre-formatted data. The realisation of this step depends on the concrete Use Case, the aim of the application as well as available technological backgrounds (e.g. high-performance supports for data processing).

```
{{cookiecutter.repo_name}}/features/build_features.py
{{cookiecutter.repo_name}}/features/
```

2.4 Develop models

This step deals with the most interesting phase in ML i.e. modelling. The most important thing of DEEP UC models is located in `model.py` containing DEEP entry point implementations. DEEP entry points are defined using *API methods*. You don't need to implement all of them, just the ones you need.

```
{{cookiecutter.repo_name}}/models/model.py
{{cookiecutter.repo_name}}/models/
```

3. Create a docker containe for your model

Once your model is well in place, you can encapsulate it by creating a docker container. For this you need to create a Dockerfile. This file will contain the information about the Docker, including the type of operating system you want to run on and the packages you need installed to make your package run.

The simplest Dockerfile could look like this:

```
FROM ubuntu:18.04

WORKDIR /srv

#Download and install your model package
RUN git clone https://github.com/your_git/your_model_package && \
cd image-classification-tf && \
python -m pip install -e . && \
cd ..

#Install DEEPaaS API
pip install deepaas

# Install rclone
RUN wget https://downloads.rclone.org/rclone-current-linux-amd64.deb && \
dpkg -i rclone-current-linux-amd64.deb && \
apt install -f && \
rm rclone-current-linux-amd64.deb && \
apt-get clean && \
rm -rf /var/lib/apt/lists/* && \
rm -rf /root/.cache/pip/* && \
rm -rf /tmp/*

# Expose API on port 5000 and tensorboard on port 6006
EXPOSE 5000 6006

CMD deepaas-run --listen-ip 0.0.0.0
```

For more details on rclone or on DEEPaas API you can check *here* and here respectively.

If you want to see an example of a more complex Dockerfile, you can check it here.

In order to compile the Dockerfile, you should choose a name for the container and use the docker build command:

```
docker build -t your_container_name -f Dockerfile
```

You can then upload it to Docker hub so that you can download the already compiled image directly. To do so, follow the instructions here.

Install and configure oidc-agent

1. Installing oidc-agent

oidc-agent is a tool to manage OpenID Connect tokens and make them easily usable from the command line. Installation instructions and full documentation can be found here.

2. Configuring oidc-agent with DEEP-IAM

Prerequisites

- DEEP-IAM registration

- Start oidc-agent:

```
$ eval $(oidc-agent)
```

- Run:

```
$ oidc-gen
```

You will be asked for the name of the account to configure. Let's call it **deep-iam**. After that you will be asked for the additional client-name-identifier, you should choose the option:

```
[2] https://iam.deep-hybrid-datacloud.eu/
```

Then just click Enter to accept the default values for Space delimited list of scopes [openid profile offline_access].

- After that, if everything has worked properly, you should see the following messages:

```
Registering Client ...
Generating account configuration ...
accepted
```

- At this point you will be given a URL. You should visit it in the browser of your choice in order to continue and approve the registered client.
- For this you will have to login into your DEEP-IAM account and accept the permissions you are asked for.
- Once you have done this you will see the following message:

```
The generated account config was successfully added to oidc-agent. You don't have␣
↪to run oidc-add
```

Next time you want to start oidc-agent from scratch, you will only have to do:

```
$ eval $(oidc-agent)
oidc-add deep-iam
Enter encryption password for account config deep-iam: ********
success
```

- You can print the token:

```
$ oidc-token deep-iam
```

2.1 Usage with orchent

- You should set OIDC_SOCK (this is not needed, if you did it before):

```
$ eval (oidc-agent)
oidc-add deep-iam
```

- Set the agent account to be used with orchent:

```
$ export ORCHENT_AGENT_ACCOUNT=deep-iam
```

- You also need to set ORCHENT_URL, e.g:

```
$ export ORCHENT_URL="https://deep-paas.cloud.cnaf.infn.it/orchestrator"
```

Train a model locally

This is a step by step guide on how to train a model from the Marketplace with your own dataset.

1. Get Docker

The first step is having Docker installed. To have an up-to-date installation please follow the official Docker installation guide.

1. Search for a model in the marketplace

The first step is to choose a model from the DEEP Open Catalog marketplace. For educational purposes we are going to use a general model to identify images. This will allow us to see the general workflow.

3. Get the model

Todo: Check that names of the docker containers are correct for the image classifier example.

Once we have chosen the model at the DEEP Open Catalog marketplace we will find that it has an associated docker container in DockerHub. For example, in the example we are running here, the container would be deephdc/deep-oc-image-classification-tf. This means that to pull the docker image and run it you should

```
.. code-block:: console
```

$ docker pull deephdc/deep-oc-image-classification-tf

4. Export rclone.conf file

When running the container you should export the rclone.conf file so that it can be reached from within the docker. You can see an example on how to do this here:

```
$ docker run -ti -v  -p 5000:5000 -p 6006:6006 -v  host_path_to_rclone.conf:/root/.
↪config/rclone/rclone.conf <your-docker-image>
```

You can see this last step explained more in detail *here*.

We are using the port `5000` to deploy the API and the port `6006` to monitor the training (for example using Tensorboard).

4. Upload your data to storage resources

To run locally you have two options:

1. Have your data stored locally

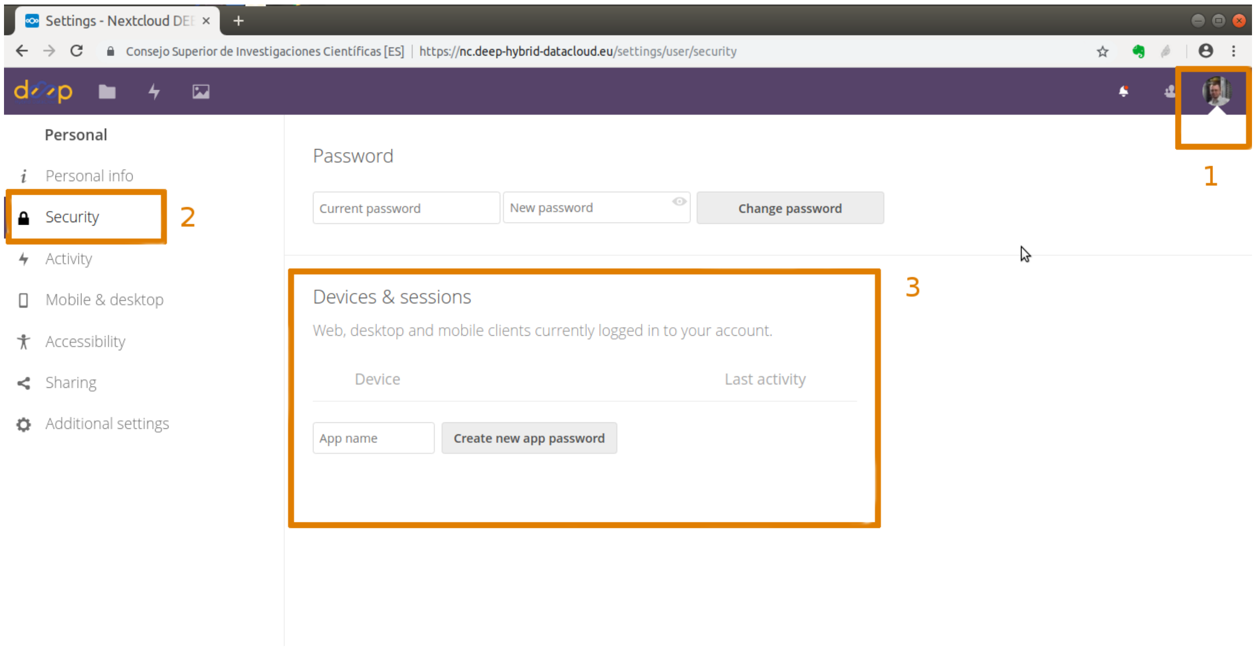
You should make sure that you export inside of the container all the folders you need for the training:

```
$ docker run -ti -v  -p 5000:5000 -p 6006:6006 -v path_to_local_folder:path_to_docker_
↪folder -v host_path_to_rclone.conf:/root/.config/rclone/rclone.conf <your-docker-
↪image>
```

2. Have your data at a remote storage resource

- Nextcloud

If you have the files you need for the training stored in Nextcloud you need first to login into DEEP-Nextcloud with your DEEP-IAM credentials. Then you have to go to: **(1) Settings (top right corner)** → **(2) Security** → **(3) Devices & sessions**



Set a name for your application (for this example it will be **deepnc**) and clik on **Create new app password**. This will generate <your_nextcloud_username> and <your_nextcloud_password> that you should to include in your rclone.conf file (see *more details*.).

Now you can create the folders that you need in order to store the inputs needed for the training and to retrieve the output. In order to be able to see these folders locally you should run either on your local host or on the docker container:

```
import subprocess

# from deep-nextcloud into the container
command = (['rclone', 'copy', 'deepnc:/path_to_remote_nextcloud_folder', 'path_to_
↪local_folder'])

results= subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output, error = result.communicate()
```

- Google Drive

TBC

- Dropbox

TBC

5. Train the model

Now comes the fun! Go to http://0.0.0.0:5000 and look for the `train` method. Modify the training parameters you wish to change and execute. If some kind of monitorization tool is available for this model you will be able to follow the training progress from http://0.0.0.0:6006.

6. Testing the training

Once the training has finished, you can directly test it by clicking on the `predict` method. There you can either upload the image your want to classify or give a URL to it.

Train a model remotely

This is a step by step guide on how to train a general model from the DEEP Open Catalog marketplace with your own dataset.

1. Choose a model

The first step is to choose a model from the DEEP Open Catalog marketplace. For educational purposes We are going to use a general model to identify images. Some of the model dependent details can change if using another model, but this tutorial will provide a general overview of the workflow to follow when using any of the models in the Marketplace. A demo showing the different steps in this HowTo has also be recorded and you can find it here *here*.

2. Prerequisites

Before being able to run your model at the Pilot Infraestructure you should first fulfill the following prerequisites:

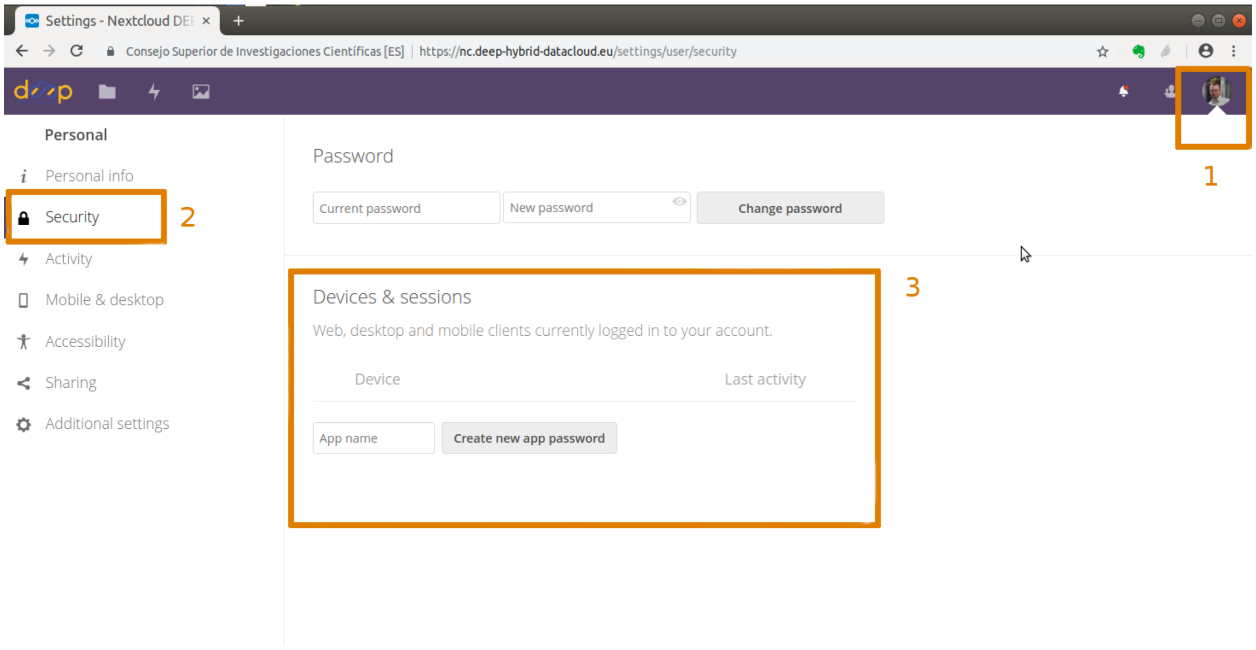
- DEEP-IAM registration
- Install rclone and configure it for DEEP-IAM. Instructions for this can be found *here*.
- Install oidc-agent and configure it for DEEP-IAM. Instructions for this can be found *here*. Make sure you follow the instructions in the *Usage with orchent* section.
- Install orchent tool

For this example we are going to use DEEP-Nextcloud for storing you data. This means you also have to:

- Register at DEEP-Nextcloud
- Follow the **Nextcloud configuration for rclone** *here*. This will give you <your_nextcloud_username> and <your_nextcloud_password>.

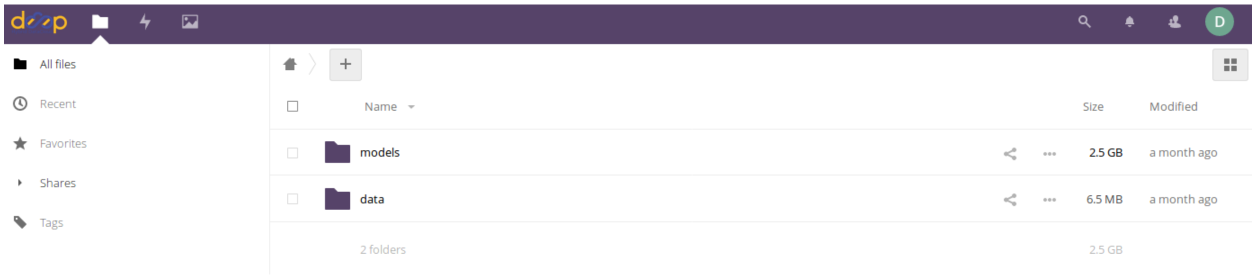
3. Upload your files to Nextcloud

Upload the files you need for the training to DEEP-Nextcloud. For this, after login into DEEP-Nextcloud with your DEEP-IAM credentials, go to: **(1) Settings (top right corner)** → **(2) Security** → **(3) Devices & sessions**



Set a name for your application (for this example it will be **deepnc**) and clik on **Create new app password**. This will generate <your_nextcloud_username> and <your_nextcloud_password> that you will need in the next step.

Now you can create the folders that you need in order to store the inputs needed for the training and to retrieve the output. These folders will be visible from within the container. In this example we just need two folders:



- A folder called **models** where the training weights will be stored after the training
- **A folder called data that contains two different folders:**
  - The folder **images** containing the input images needed for the training
  - The folder **dataset_files** containing a couple of scripts: *train.txt* indicating the relative path to the training images and *classes.txt* indicating which are the categories for the training

The folder structure and their content will of course depend on the model to be used. This structure is just an example in order to complete the workflow for this tutorial.

4. Orchent submission script

With <your_nextcloud_username> and <your_nextcloud_password> from the previous step, you can generate a bash script (i.e deploy.sh) to create the orchent deployment:

```
#!/bin/bash

orchent depcreate ./TOSCA.yml '{ "rclone_url": "https://nc.deep-hybrid-datacloud.eu/
↪remote.php/webdav/",
                                            "rclone_vendor": "nextcloud",
                                            "rclone_user": <your_nextcloud_username>
                                            "rclone_pass": <your_nextcloud_password> }
↪'
```

This script will be the *only place* where you will have to indicate your username and password. This file should be stored locally.

Important: DO NOT save the rclone credentials in the **CONTAINER** nor in the **TOSCA** file

5. The rclone configuration file

For running the model remotely you need to include in your git repository either an empty or 'minimal' rclone.conf file, ensure that rclone_confg in the TOSCA file (section) properly points to this file.The minimal rclone.conf content would correspond to something like:

```
[deepnc]
type = webdav
config_automatic = yes
```

Remember that the first line should include the name of the remote Nextcloud application (**deepnc** for this example).

6. Prepare your TOSCA file

In the orchent submission script there is a call to a TOSCA file (TOSCA.yml). A generic template can be found here. The sections that should be modified are the following (TOSCA experts may modify the rest of the template to their will.)

- Docker image to deploy. In this case we will be using deephdc/deep-oc-image-classification-tf:

```
dockerhub_img:
   type: string
   description: docker image from Docker Hub to deploy
   required: yes
   default: deephdc/deep-oc-image-classification-tf
```

- Location of the .rclone.conf (this file can be empty, but should be at the indicated location):

```
rclone_conf:
   type: string
   description: nextcloud link to access via webdav
   required: yes
   default: "/srv/image-classification-tf/rclone.conf"
```

For further TOSCA templates examples you can go here.

7. Create the orchent deployment

The submission is then done running the orchent submission script you generated in one of the previous steps:

```
./deploy.sh
```

This will give you a bunch of information including your deployment ID. To check status of your job

> $ orchent depshow <Deployment ID>

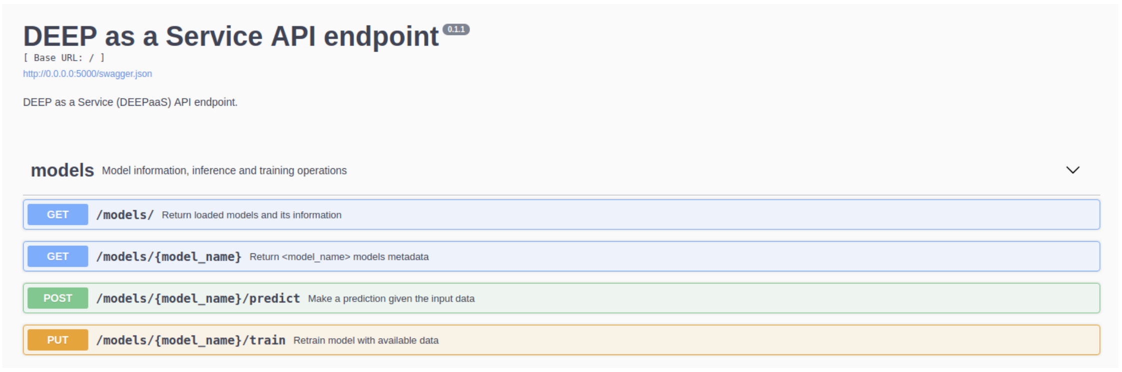
Once your deployment is in status **CREATED**, you will be given a web URL:

```
http://your_orchent_deployment:5000/
```

8. Go to the API, train the model

Now comes the fun!

Go to *http://your_orchent_deployment:5000/* and look for the `train` method. Modify the training parameters you wish to change and execute. If some kind of monitorization tool is available for this model you will be able to follow the training progress from *http://your_orchent_deployment:6006/*



DEEP as a Service API endpoint 0.1.1

[Base URL: /]

http://0.0.0.0:5000/swagger.json

DEEP as a Service (DEEPaaS) API endpoint.

models Model information, inference and training operations

GET /models/ Return loaded models and its information

GET /models/{model_name} Return <model_name> models metadata

POST /models/{model_name}/predict Make a prediction given the input data

PUT /models/{model_name}/train Retrain model with available data

9. Testing the training

Once the training has finished, you can directly test it by clicking on the `predict` method. There you can either upload the image your want to classify or give a URL to it.

Try a service locally

1. Get Docker

The first step is having Docker installed. To have an up-to-date installation please follow the official Docker installation guide.

2. Search for a model in the marketplace

The next step is to look for a service in the DEEP Open Catalog marketplace you want to try locally.

3. Get the model

You will find that each model has an associate Docker container in DockerHub. For eaxample DEEP OC Image Classification is associated with deephdc/deep-oc-image-classification-tf, DEEP OC Massive Online Data Streams is associated with deephdc/deep-oc-mods, etc.

Let call the service you selected `deep-oc-service_of_interest`. Please, download the container with:

```
$ docker pull deephdc/deep-oc-service_of_interest
```

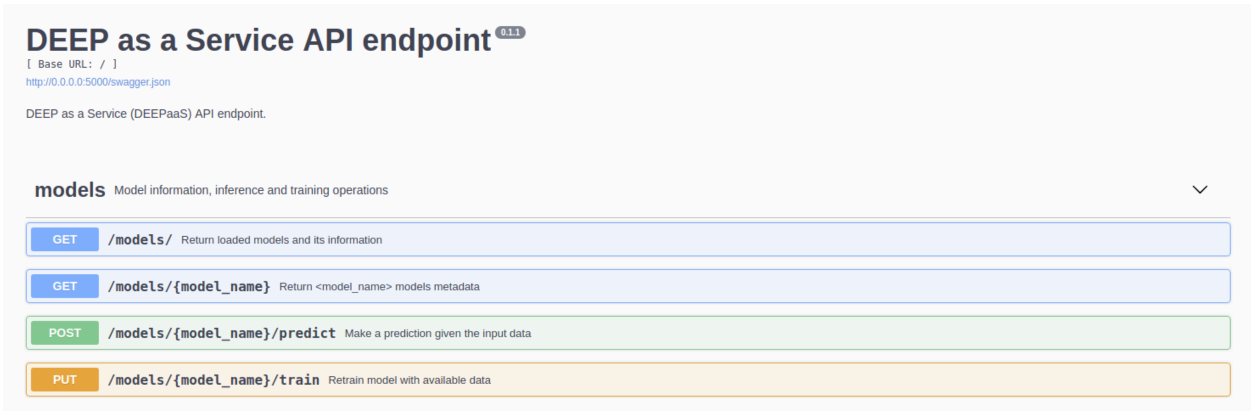
4. Run the model

Run the container with:

```
$ docker run -ti -p 5000:5000 deephdc/deep-oc-service_of_interest
```

5. Go to the API, get the results

Once running, point your browser to http://127.0.0.1:5000/ and you will see the API documentation, where you can test the service functionality, as well as perform other actions.



DEEP as a Service API endpoint 0.1.1

[Base URL: /]

http://0.0.0.0:5000/swagger.json

DEEP as a Service (DEEPaaS) API endpoint.

models Model information, inference and training operations

GET `/models/` Return loaded models and its information

GET `/models/{model_name}` Return <model_name> models metadata

POST `/models/{model_name}/predict` Make a prediction given the input data

PUT `/models/{model_name}/train` Retrain model with available data

The concrete results can vary from service to service. For example the plant classifier return a list of plant species along with the predicted probabilities.

Using Openstack API with OIDC tokens

Note: this guide is made for GNU/Linux distributions

Create file for OIDC

After downloading Openstack RC file (Identity API v3) from Horizon web panel, the following changes are necessary for the authentication with OIDC (review the missing fields):

```
#!/usr/bin/env bash
export OS_AUTH_URL="https://<keystone API url ending in /v3>"
export OS_AUTH_TYPE="v3oidcaccesstoken"
export OS_PROJECT_ID="<project id comes with downloaded RC v3 file>"
export OS_USERNAME="<username comes with downloaded RC v3 file>"
export OS_REGION_NAME="<comes with downloaded RC v3 file>"
# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE="<comes with downloaded RC v3 file>"
export OS_IDENTITY_API_VERSION=3
export OS_IDENTITY_PROVIDER="deep-hdc"
export OS_PROTOCOL="oidc"
export OS_ACCESS_TOKEN="<get access token from MitreID dashboard after login to␣
↪openstack horizon>"
```

The keystone API url can be obtained from Horizon web panel in Project->Access & Security->API Access, associated with the service name `Identity`.

The access token is obtained from the login in Openstack Horizon. It'll create an access token that can be reviewed in MitreID dashboard.

All the other attributes for authentication are in the RC v3 file downloaded from Horizon web panel.

Using Openstack CLI

After loading all necessary environment variables to the shell, just run openstack command. To create a RC file for OIDC you can do the following:

```
cat << EOF
#!/usr/bin/env bash
export OS_AUTH_URL="https://<keystone API url ending in /v3>"
export OS_AUTH_TYPE="v3oidcaccesstoken"
export OS_PROJECT_ID="<project id comes with downloaded RC v3 file>"
export OS_USERNAME="<username comes with downloaded RC v3 file>"
export OS_REGION_NAME="<comes with downloaded RC v3 file>"
# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE="<comes with downloaded RC v3 file>"
export OS_IDENTITY_API_VERSION=3
export OS_IDENTITY_PROVIDER="deep-hdc"
export OS_PROTOCOL="oidc"
export OS_ACCESS_TOKEN="<get access token from MitreID dashboard after login to␣
↪openstack horizon>"
EOF > oidc_RCfile.sh
```

Then execute the file to load variables definitions into shell environment (no need to give execution previleges):

```
. ./oidc_RCfile.sh
```

For example, to list all images available run the following command:

```
openstack image list
```

Since the list of images cloud be large, this should be better:

```
openstack image list | less
```

For more details about all available openstack cli commands review the following page:

Openstack CLI

Video demos

This page gathers different video demos available:

- Training a model remotely

  [Short description of the demo]

The following sections provide information on how several deep learning models have been developed and integrated with our platform.

1.1.4 Models

Toy example: dog breed recognition

A toy example to identify Dog's breed, "Dogs breed detector".

DEEP Open Catalogue: Image classification on TensorFlow

This is a plug-and-play tool to train and evaluate an image classifier on a custom dataset using deep neural networks running on TensorFlow. Further information on the package structure and the requirements can be found in the documentation in the git repository.

Workflow

1. Data preprocessing

The first step to train your image classifier if to have the data correctly set up.

1.1 Prepare the images

Put your images in the `./data/images` folder. If you have your data somewhere else you can use that location by setting the `image_dir` parameter in the `./etc/config.yaml` file.

Please use a standard image format (like `.png` or `.jpg`).

1.2 Prepare the data splits

First you need add to the `./data/dataset_files` directory the following files:

Mandatory files	Optional files
`classes.txt`, `train.txt`	`val.txt`, `test.txt`, `info.txt`

- `train.txt`, `val.txt` and `test.txt` files associate an image name (or relative path) to a label number (that has to *start at zero*).
- `classes.txt` file translates those label numbers to label names.

- `info.txt` allows you to provide information (like number of images in the database) about each class. This information will be shown when launching a webpage of the classifier.

You can find examples of these files here.

2. Train the classifier

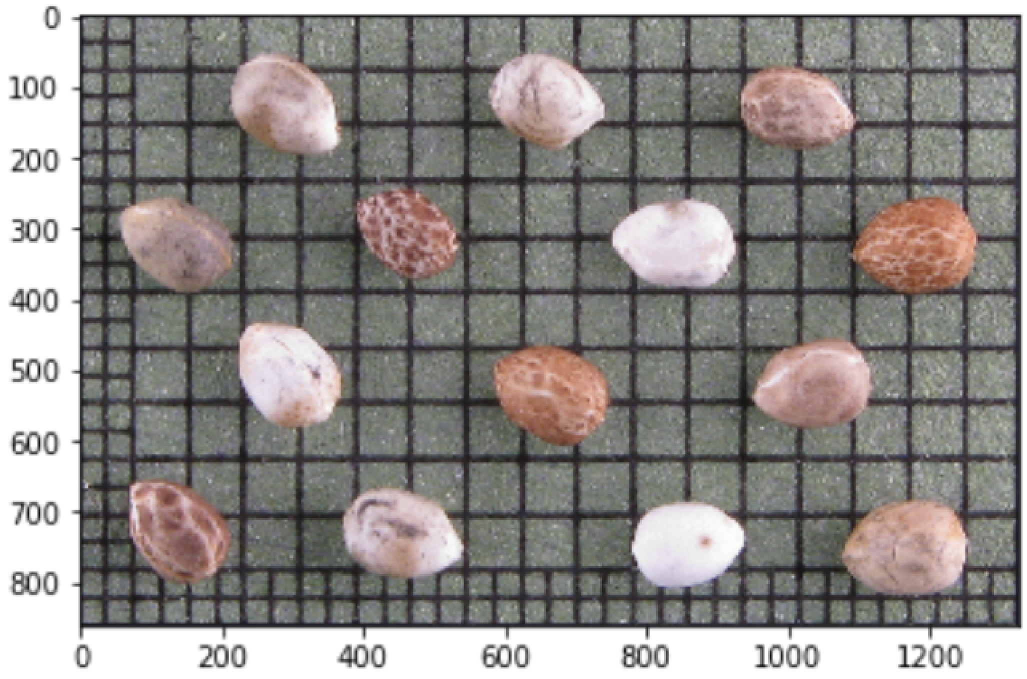
Before training the classifier you can customize the default parameters of the configuration file. To have an idea of what parameters you can change, you can explore them using the dataset exploration notebook. This step is optional and training can be launched with the default configurarion parameters and still offer reasonably good results.

Once you have customized the configuration parameters in the `./etc/config.yaml` file you can launch the training running `./imgclas/train_runfile.py`. You can monitor the training status using Tensorboard.

After training check the prediction statistics notebook to see how to visualize the training statistics.

3. Test the classifier

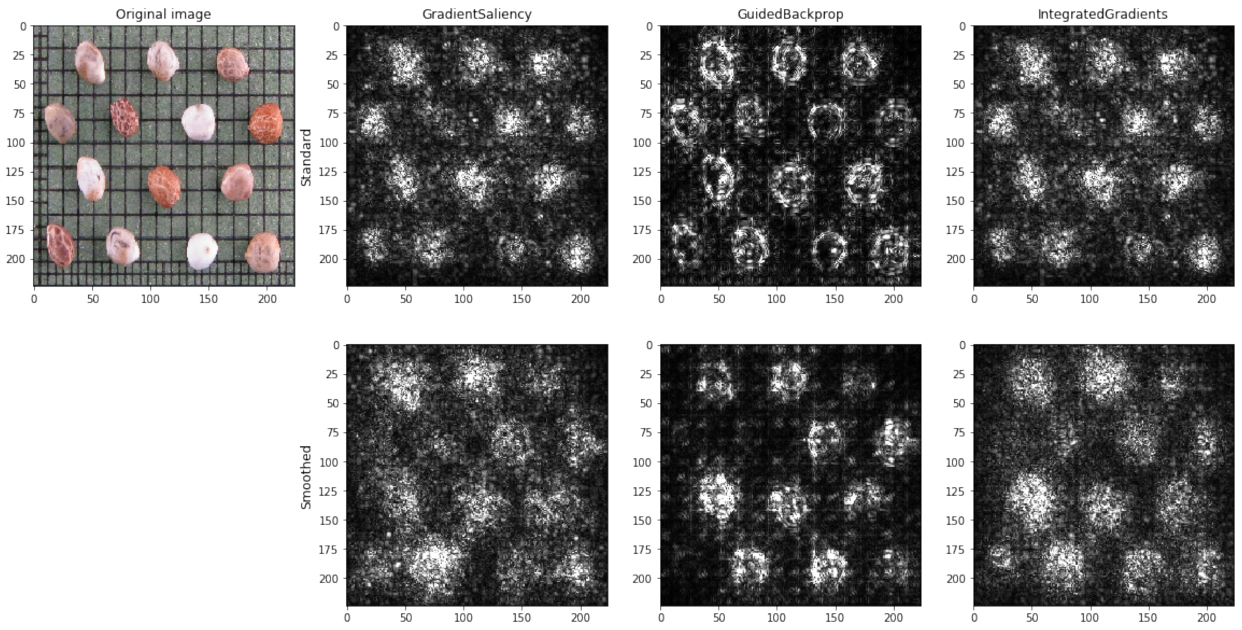
You can test the classifier on a number of tasks: predict a single local image (or url), predict multiple images (or urls), merge the predictions of a multi-image single observation, etc. All these tasks are explained in the computing prediction notebooks.



```
[59.8%] Genus Sambucus
[12.8%] Genus Rubus
[7.3%] Genus Papaver
[6.1%] Genus Polycarpon
[4.4%] Genus Astragalus
```

You can also make and store the predictions of the `test.txt` file (if you provided one). Once you have done that you can visualize the statistics of the predictions like popular metrics (accuracy, recall, precision, f1-score), the confusion matrix, etc by running the predictions statistics notebook.

By running the saliency maps notebook you can also visualize the saliency maps of the predicted images, which show what were the most relevant pixels in order to make the prediction.



Finally you can launch a simple webpage to use the trained classifier to predict images (both local and urls) on your favorite brownser.

Launching the full DEEPaas API

Preliminaries for prediction

If you want to use the API for prediction, you have to do some preliminary steps to select the model you want to predict with:

- Copy your desired `.models/[timestamp]` to `.models/api`. If there is no `.models/api` folder, the default is to use the last available timestamp.
- In the `.models/api/ckpts` leave only the desired checkpoint to use for prediction. If there are more than one chekpoints, the default is to use the last available checkpoint.

Running the DEEPaaS API

To access this package's complete functionality (both for training and predicting) through the DEEPaaS API you have to follow the instructions here: *Run a module on DEEP Pilot Infrastructure*

CHAPTER 2

Technical documentation

If you are searching for technical notes on various notes.

2.1 Technical documentation

These pages contain technical notes software documentations, guides, tutorials, logbooks and similar documents produced with DEEP Hybrid DataCloud project

2.1.1 Mesos

Introduction

Mesos 1.0.0 added first-class support for Nvidia GPUs. The minimum required Nvidia driver version is `340.29`

Enabling GPU support in a Mesos cluster is really straightforward (as stated in the official project documentation and as documented in this page). It consists in the following steps:

1. configuring the agent nodes in order to expose the available gpus as resources to be advertised to the master nodes;
2. enabling the framework GPU_RESOURCES capability so that the master includes the GPUs in the resource offers sent to the frameworks.

Mesos exposes GPUs as a simple `SCALAR` resource in the same way it always has for CPUs, memory, and disk.

An important remark is that currently the GPU support is available for the Mesos containerizer and not for the Docker containerizer. Anyway the Mesos containerizer is now able to run docker images natively through the Universal Container Runtime (UCR).

The following limitations can, on the other hand, have impacts on the deployment of Long-Running services (Marathon) requiring GPUs:

- The UCR does not support the following: runtime privileges, Docker options, force pull, named ports, numbered ports, bridge networking, port mapping.

It is important to remember that the task definition must be properly written in order to specify the right containerizer (type=MESOS).

For Marathon:

```
{
  "id": "test",
  "cpus": 2,
  "mem": 2048,
  [...]
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "tensorflow/tensorflow"
    }
  }
}
```

See also https://mesosphere.github.io/marathon/docs/native-docker.html#provisioning-containers-with-the-ucr

For Chronos:

```
{
  "name": "test-gpu",
  "command": "",
  "cpus": 1,
  "mem": 4096,
  [...]
  "container": {
    "type": "MESOS",
    "image": "tensorflow/tensorflow"
  },
  "schedule": "..."
}
```

The GPU support is fully implemented and officially documented in Mesos and Marathon Framework whereas Chronos Framework does not support GPU resources yet. Anyway there is a pull request (still open) that seems in good shape and we have decided to give it a try.

Testbed Setup

Nodes characteristics

Tested Components Versions

Prepare the agent (slave) node

Download the driver repo from http://www.nvidia.com/Download/index.aspx?lang=en-us choosing the proper version.

Install the downloaded .deb file (repo), install the driver and reboot:

```
dpkg -i nvidia-diag-driver-local-repo-ubuntu1604-390.30_1.0-1_amd64.deb
apt-key add /var/nvidia-diag-driver-local-repo-390.30/7fa2af80.pub
apt-get update
apt-get install nvidia-390
reboot
```

Alternatively you can enable the graphics-drivers PPA. Currently, it supports Ubuntu 18.04 LTS, 17.10, 17.04, 16.04 LTS, and 14.04 LTS operating systems (still under testing phase):

```
add-apt-repository ppa:graphics-drivers/ppa
apt update
apt install nvidia-390
```

Verify the nvidia-driver installation

Launch the command:

```
nvidia-smi
```

Output:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.30                 Driver Version: 390.30                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40m          Off  | 00000000:82:00.0 Off |                    0 |
| N/A   21C    P8    18W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Mesos slave configuration

```
export MESOS_MASTER="zk://<master>:2181/mesos"
export MESOS_ZK="zk://<master>:2181/mesos"
export MESOS_EXECUTOR_REGISTRATION_TIMEOUT="10mins"
export MESOS_CONTAINERIZERS="mesos,docker"
export MESOS_LOG_DIR="/var/log/mesos"
export MESOS_IP="<agent-ip>"
export MESOS_WORK_DIR="/var/lib/mesos"
export MESOS_HOSTNAME="<agent-hostname>"
export MESOS_ISOLATION="docker/runtime,filesystem/linux,cgroups/devices,gpu/nvidia"
export MESOS_IMAGE_PROVIDERS="docker"
```

(Re)start the mesos agent. In the log you will see the available GPU as resource offered by the agent node:

Feb 26 23:10:07 hpc-09-02 mesos-slave[1713]: I0226 23:10:07.155503 1745 slave.cpp:533] Agent resources: gpus(*):1; cpus(*):40; mem(*):256904; disk(*):3489713; ports(*):[31000-32000]

Testing GPU support in Mesos

Verify that Mesos is able to launch a task consuming GPUs:

```
mesos-execute --master=mesos-m0.recas.ba.infn.it:5050 --name=gpu-test  --docker_
↪image=nvidia/cuda       --command="nvidia-smi"       --framework_capabilities="GPU_
↪RESOURCES"       --resources="gpus:1"
I0305 15:22:38.346174  4443 scheduler.cpp:188] Version: 1.5.0
I0305 15:22:38.349104  4459 scheduler.cpp:311] Using default 'basic' HTTP␣
↪authenticatee
I0305 15:22:38.349442  4462 scheduler.cpp:494] New master detected at master@172.20.0.
↪38:5050
Subscribed with ID 6faa9a75-d48b-4dc6-96ee-73c35997706b-0017
Submitted task 'gpu-test' to agent 'd33d527c-8d1f-4e53-b65d-e2b2c67c0889-S2'
Received status update TASK_STARTING for task 'gpu-test'
  source: SOURCE_EXECUTOR
Received status update TASK_RUNNING for task 'gpu-test'
  source: SOURCE_EXECUTOR
Received status update TASK_FINISHED for task 'gpu-test'
  message: 'Command exited with status 0'
  source: SOURCE_EXECUTOR
```

Look into the task sandbox. The stdout should report the following:

```
Marked '/' as rslave
Prepared mount '{"flags":20480,"source":"\/var\/lib\/mesos\/slaves\/d33d527c-8d1f-
↪4e53-b65d-e2b2c67c0889-S2\/frameworks\/6faa9a75-d48b-4dc6-96ee-73c35997706b-0017\/
↪executors\/gpu-test\/runs\/5ebbfaf3-3b8b-4c32-9337-740a85feef75","target":"\/var\/
↪lib\/mesos\/provisioner\/containers\/5ebbfaf3-3b8b-4c32-9337-740a85feef75\/
↪backends\/overlay\/rootfses\/e56d62ea-4334-4582-a820-2b9406e2b7f8\/mnt\/mesos\/
↪sandbox"}'
Prepared mount '{"flags":20481,"source":"\/var\/run\/mesos\/isolators\/gpu\/nvidia_
↪390.30","target":"\/var\/lib\/mesos\/provisioner\/containers\/5ebbfaf3-3b8b-4c32-
↪9337-740a85feef75\/backends\/overlay\/rootfses\/e56d62ea-4334-4582-a820-
↪2b9406e2b7f8\/usr\/local\/nvidia"}'
Prepared mount '{"flags":20513,"target":"\/var\/lib\/mesos\/provisioner\/containers\/
↪5ebbfaf3-3b8b-4c32-9337-740a85feef75\/backends\/overlay\/rootfses\/e56d62ea-4334-
↪4582-a820-2b9406e2b7f8\/usr\/local\/nvidia"}'
Changing root to /var/lib/mesos/provisioner/containers/5ebbfaf3-3b8b-4c32-9337-
↪740a85feef75/backends/overlay/rootfses/e56d62ea-4334-4582-a820-2b9406e2b7f8
Mon Mar  5 14:23:41 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.30                 Driver Version: 390.30                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40m          Off  | 00000000:82:00.0 Off |                    0 |
| N/A   21C    P8    18W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Testing Chronos patch for GPU support

Patch available on github: https://github.com/mesos/chronos/pull/810

Patch compilation

The following steps are needed to test the patch:

1. Get the patched code:

```
git clone https://github.com/mesos/chronos.git -b v3.0.2
cd  chronos
git fetch origin pull/810/head:chronos
git checkout chronos
```

2. Compile:

```
docker run -v `pwd`:/chronos --entrypoint=/bin/sh maven:3-jdk-8 -c "\
curl -sL https://deb.nodesource.com/setup_7.x | bash - \
&& apt-get update && apt-get install -y --no-install-recommends nodejs \
&& ln -sf /usr/bin/nodejs /usr/bin/node \
&& cd /chronos \
&& mvn clean \
&& mvn versions:set -DnewVersion=3.0.2-1 \
&& mvn package -DskipTests"
```

The jar **chronos-3.0.2-1.jar** will be created in the folder ./target/

3. Create the docker image **Dockerfile:**

```
FROM ubuntu:16.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF && \
echo deb http://repos.mesosphere.io/ubuntu trusty main > /etc/apt/sources.list.d/
↪mesosphere.list && \
apt-get update && \
apt-get -y install --no-install-recommends mesos openjdk-8-jre-headless && \
apt-get clean && \
rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
ADD chronos-3.0.2-1.jar /
COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]
```

File entrypoint.sh:

~~~
``` syntaxhighlighter-pre
#!/bin/sh
CMD="java -Xmx512m -cp chronos-3.0.2-1.jar org.apache.mesos.chronos.scheduler.Main"
# Parse environment variables
for k in `set | grep ^CHRONOS_ | cut -d= -f1`; do
    eval v=\$$k
    CMD="$CMD --`echo $k | cut -d_ -f2- | tr '[:upper:]' '[:lower:]'` $v"
done
# authentication
PRINCIPAL=${PRINCIPAL:-root}
if [ -n "$SECRET" ]; then
    touch /tmp/secret
    chmod 600 /tmp/secret
    echo -n "$SECRET" > /tmp/secret
    CMD="$CMD --mesos_authentication_principal $PRINCIPAL --mesos_authentication_
↪secret_file /tmp/secret"
fi
~~~

(continues on next page)

(continued from previous page)

```
echo $CMD
if [ $# -gt 0 ]; then
    exec "$@"
fi
exec $CMD
```
```

Start the patched Chronos Framework:

Using the docker image described above you can run Chronos as follows:

```
docker run --name chronos -d --net host --env-file /etc/chronos/.chronosenv chronos:3.
↪0.2_gpu
```

with the following environment:

```
LIBPROCESS_IP=172.20.0.38
CHRONOS_HOSTNAME=172.20.0.38
CHRONOS_HTTP_PORT=4400
CHRONOS_MASTER=zk://172.20.0.38:2181/mesos
CHRONOS_ZK_HOSTS=zk://172.20.0.38:2181
CHRONOS_ZK_PATH=/chronos/state
CHRONOS_MESOS_FRAMEWORK_NAME=chronos
CHRONOS_HTTP_CREDENTIALS=admin:******
CHRONOS_ENABLE_FEATURES=gpu_resources
```

Testing

Approach: submit a batch-like job that uses the tensorflow docker image, downloads the code available here and runs the convolutional network example

```
apt-get update; apt-get install -y git

git clone https://github.com/aymericdamien/TensorFlow-Examples

cd TensorFlow-Examples/examples/3_NeuralNetworks;

time python convolutional_network.py
```

The test is based on the tutorial provided by mesosphere DC/OS

Two different versions of the tensorflow docker image will be used in order to verify the correct execution of the job regardless of the version of CUDA and cuDNN used to build the binaries inside the docker image:

Test #1:

Job definition:

```
{
  "name": "test-gpu",
  "command": "cd $MESOS_SANDBOX && /bin/bash gpu_demo.sh",
  "shell": true,
  "retries": 2,
  "cpus": 4,
```

(continues on next page)
```

(continued from previous page)

```
  "disk": 256,
  "mem": 4096,
  "gpus": 1,
  "uris": [
    "https://gist.githubusercontent.com/maricaantonacci/
↪1a7f02903513e7bba91f451e0f4f5ead/raw/78c737fd0e2a288a2040c192368f6c4ecf8eb88a/gpu_
↪demo.sh"
  ],
  "environmentVariables": [],
  "arguments": [],
  "runAsUser": "root",
  "container": {
    "type": "MESOS",
    "image": "tensorflow/tensorflow:latest-gpu"
  },
  "schedule": "R/2018-03-05T23:00:00.000Z/PT24H"
}
```

The job is correctly run. The following relevant info were retrieved from the stderr file in the job sandbox:

```
Cloning into 'TensorFlow-Examples'...
/usr/local/lib/python2.7/dist-packages/h5py/__init__.py:36: FutureWarning: Conversion␣
↪of the second argument of issubdtype from `float` to `np.floating` is deprecated.␣
↪In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpLLDDDs
2018-03-05 14:38:59.059890: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1212]␣
↪Found device 0 with properties:
name: Tesla K40m major: 3 minor: 5 memoryClockRate(GHz): 0.745
pciBusID: 0000:82:00.0
totalMemory: 11.17GiB freeMemory: 11.09GiB
2018-03-05 14:38:59.059989: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312]␣
↪Adding visible gpu devices: 0
2018-03-05 14:38:59.496393: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993]␣
↪Creating TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 10757␣
↪MB memory) -> physical GPU (device: 0, name: Tesla K40m, pci bus id: 0000:82:00.0,␣
↪compute capability: 3.5)
2018-03-05 14:39:23.210323: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312]␣
↪Adding visible gpu devices: 0
2018-03-05 14:39:23.210672: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993]␣
↪Creating TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 260␣
↪MB memory) -> physical GPU (device: 0, name: Tesla K40m, pci bus id: 0000:82:00.0,␣
↪compute capability: 3.5)

real    0m32.394s
user    0m35.192s
sys 0m12.204s
I0305 15:39:25.630180  4680 executor.cpp:938] Command exited with status 0 (pid: 4716)
```

Test #2

Job definition:

```
{
  "name": "test2-gpu",
```

(continues on next page)

(continued from previous page)

```
  "command": "cd $MESOS_SANDBOX && /bin/bash gpu_demo.sh",
  "shell": true,
  "retries": 2,
  "cpus": 4,
  "disk": 256,
  "mem": 4096,
  "gpus": 1,
  "uris": [
    "https://gist.githubusercontent.com/maricaantonacci/
↪1a7f02903513e7bba91f451e0f4f5ead/raw/78c737fd0e2a288a2040c192368f6c4ecf8eb88a/gpu_
↪demo.sh"
  ],
  "environmentVariables": [],
  "arguments": [],
  "runAsUser": "root",
  "container": {
    "type": "MESOS",
    "image": "tensorflow/tensorflow:1.4.0-gpu"
  },
  "schedule": "R/2018-03-05T23:00:00.000Z/PT24H"
}
```

As you can see, the only difference wrt Test#1 is the docker image: here we are using the tag 1.4.0-gpu of the tensorflow docker image that has been built using a different CUDA and cuDNN version.

Also in this case the job is correcly run:

```
Cloning into 'TensorFlow-Examples'...
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpomIcq9
2018-03-05 16:36:24.518455: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your␣
↪CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.
↪1 SSE4.2 AVX
2018-03-05 16:36:25.261578: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030]␣
↪Found device 0 with properties:
name: Tesla K40m major: 3 minor: 5 memoryClockRate(GHz): 0.745
pciBusID: 0000:82:00.0
totalMemory: 11.17GiB freeMemory: 11.09GiB
2018-03-05 16:36:25.261658: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120]␣
↪Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla K40m, pci bus␣
↪id: 0000:82:00.0, compute capability: 3.5)
2018-03-05 16:36:52.299346: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120]␣
↪Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla K40m, pci bus␣
↪id: 0000:82:00.0, compute capability: 3.5)

real    0m35.273s
user    0m37.828s
sys 0m13.164s
I0305 17:36:54.803642  7405 executor.cpp:938] Command exited with status 0 (pid: 7439)
```

Additional note:

Running the job without gpu (using the image tensorflow/tensorflow:latest and "gpus": 0) we got for the same script:

```
real   2m15.647s
user    22m33.384s
sys 15m51.164s
```

Testing GPU support in Marathon

In order to enable GPU support in Marathon, you need to start the framework with the commandline option –enable_features=gpu_resources (or using the env variable MARATHON_ENABLE_FEATURES):

Start Marathon Framework:

```
docker run -d --name marathon --net host --env-file /etc/marathon/.marathonenv ␣
↪indigodatacloud/marathon:1.5.6
```

with the following enviroment:

```
LIBPROCESS_IP=<mesos-master-ip>
MARATHON_HOSTNAME=<mesos-master-fqdn/ip>
MARATHON_HTTP_ADDRESS=<mesos-master-ip>
MARATHON_HTTP_PORT=8080
MARATHON_MASTER=zk://<mesos-master>:2181/mesos
MARATHON_ZK=zk://<mesos-master>:2181/marathon
MARATHON_FRAMEWORK_NAME=marathon
MARATHON_HTTP_CREDENTIALS=admin:******
MARATHON_ENABLE_FEATURES=gpu_resources
```

Test:

The following application has been submitted to Marathon:

```
{
  "id": "tensorflow-gpus",
  "cpus": 4,
  "gpus": 1,
  "mem": 2048,
  "disk": 0,
  "instances": 1,
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "tensorflow/tensorflow:latest-gpu"
    }
  }
}
```

Running tensorflow docker container

1) Using “DOCKER” containerizer on a Mesos cluster without GPUs

Submit to Marathon the following application:

```
{
  "id": "/tensorflow-app",
  "cmd": "PORT=8888 /run_jupyter.sh --allow-root",
  "cpus": 2,
  "mem": 4096,
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "tensorflow/tensorflow:latest",
```

(continues on next page)

(continued from previous page)

```
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8888,
          "hostPort": 0,
          "servicePort": 10000,
          "protocol": "tcp"
        }
      ],
      "privileged": false,
      "forcePullImage": true
    }
  },
  "env": {
    "PASSWORD": "s3cret"
  },
  "labels": {
    "HAPROXY_GROUP": "external"
  }
}
```

Then you can access the service through the cluster LB on port 10000 (servicePort)

2) Using "MESOS" containerizer on a Mesos cluster with GPUs

```
{
 "id": "tensorflow-gpus",
 "cpus": 4,
 "gpus": 1,
 "mem": 2048,
 "disk": 0,
 "instances": 1,
 "container": {
   "type": "MESOS",
   "docker": {
     "image": "tensorflow/tensorflow:latest-gpu"
   }
 },
 "portDefinitions": [
       {"port": 10000, "name": "http"}
  ],
 "networks": [ { "mode": "host" } ],
 "labels":{
   "HAPROXY_GROUP":"external"
 },
 "env": {
   "PASSWORD":"s3cret"
 }
}
```

Then you can access the service through the cluster LB on port 10000.

If the "port" field in portDefinitions is set to 0 then Marathon will assign a random service port (that you can know with a GET request to /v2/apps/app-name)

References

Enabling open-id connect authentication

Mesos/Marathon/Chronos do not support open-id connect authentication natively.

A very simple solution is to front the mesos cluster with an Apache server that itself is capable of negotiating authentication for users.

The following configuration can be used to setup a reverse proxy that uses the module ***mod_auth_openidc***:

```
ServerName mesos.example.com

<VirtualHost *:443>
  ServerName mesos.example.com

  LoadModule auth_openidc_module /usr/lib/apache2/modules/mod_auth_openidc.so

  OIDCClaimPrefix                   "OIDC-"
  OIDCResponseType                  "code"
  OIDCScope                         "openid email profile"
  OIDCProviderMetadataURL           https://iam.deep-hybrid-datacloud.eu/.well-known/
↪openid-configuration
  OIDCClientID                      332e618b-d3bf-440d-aea1-6da2823aaece # replace with␣
↪your client ID
  OIDCClientSecret                  ****                                 # replace with␣
↪your client secret
  OIDCProviderTokenEndpointAuth     client_secret_basic
  OIDCCryptoPassphrase              ****                                 # replace with␣
↪your passphrase
  OIDCRedirectURI                   https://mesos.example.com/mesos/redirect_uri

  OIDCOAuthVerifyJwksUri "https://iam.deep-hybrid-datacloud.eu/jwk"

  <Location /mesos>
    AuthType openid-connect
    Require valid-user
    LogLevel debug
  </Location>

 <Location /marathon>
    AuthType oauth20
    Require valid-user
    LogLevel debug
    RequestHeader set Authorization "Basic YWRtaC46bTNzb3NNLjIwMTY="
 </Location>

 <Location /chronos>
    AuthType oauth20
    Require valid-user
    LogLevel debug
    RequestHeader set Authorization "Basic YWRtaZ46bTNzb3NDLjIwMTY="
 </Location>

  ProxyTimeout 1200
  ProxyRequests Off
  ProxyPreserveHost Off
```

(continues on next page)

(continued from previous page)

```
  ProxyPass /mesos/ http://172.20.30.40:5050/
  ProxyPassReverse /mesos/ http://172.20.30.40:5050/

  ProxyPass /marathon/ http://172.20.30.40:8080/
  ProxyPassReverse /marathon/ http://172.20.30.40:8080/

  ProxyPass /chronos/ http://172.20.30.40:4400/
  ProxyPassReverse /chronos/ http://172.20.30.40:4400/

  RemoteIPHeader X-Forwarded-For

  ## Logging
  ErrorLog "/var/log/apache2/proxy_mesos_error_ssl.log"
  ServerSignature Off
  CustomLog "/var/log/apache2/proxy_mesos_access_ssl.log" combined

  ## SSL directives

  SSLProxyEngine on
  SSLEngine on
  SSLCertificateFile      "/etc/letsencrypt/live/mesos.example.com/fullchain.pem"
  SSLCertificateKeyFile   "/etc/letsencrypt/live/mesos.example.com/privkey.pem"
</VirtualHost>
```

Note that Line 30 is needed if you have enabled basic HTTP authentication to protect your endpoints (in the example above, username/password authentication has been enable for Marathon).

In this case you need to add the Authorization header in the request to the backend. The hash can be computed with the following python script:

```
import base64
hash = base64.b64encode(b'user:password')
```

Once the proxy is up and running you can contact the cluster API endpoints using the IAM (open-id connect) token:

Marathon API endpoint: https://mesos.example.com/marathon

Chronos API endpoint: https://mesos.example.com/chronos

For example:

```
curl -H "Authorization: bearer $IAM_ACCESS_TOKEN" -X GET https://mesos.example.com/
↪marathon/v2/apps
```

If you want to allow users to access also the Web interfaces of Marathon and Chronos, then add the following configuration:

```
  <Location /marathon-web>
    AuthType openid-connect
    Require valid-user
    LogLevel debug
    RequestHeader set Authorization "Basic YWRtaC46bTNzb3NNLjIwMTY="
  </Location>

  <Location /chronos-web>
    AuthType openid-connect
    Require valid-user
    LogLevel debug
```

(continues on next page)

(continued from previous page)

```
    RequestHeader set Authorization "Basic YWRtaZ46bTNzb3NDLjIwMTY="
  </Location>

  ProxyPass /marathon-web/ http://172.20.30.40:8080/
  ProxyPassReverse /marathon-web/ http://172.20.30.40:8080/

  ProxyPass /chronos-web/ http://172.20.30.40:4400/
  ProxyPassReverse /chronos-web/ http://172.20.30.40:4400/
```

The Web UIs will be accessible at the following urls:

Marathon Web UI: https://mesos.example.com/marathon-web/

Chronos Web UI: https://mesos.example.com/chronos-web/

2.1.2 Kubernetes

DEEP : Installing and testing GPU Node in Kubernetes - CentOS7

Introduction

The **manual** procedure for installation and configuration od a Kubernetes cluster is provided. The cluster is composed by a Master node and one Worker node

Cluster Status

Cluster Name: KubeDeep

Kubectl components

```
# kubectl get componentstatuses
NAME                  STATUS    MESSAGE              ERROR
controller-manager    Healthy   ok
scheduler             Healthy   ok
etcd-0                Healthy   {"health": "true"
```

```
# kubectl get nodes
NAME        STATUS    ROLES     AGE       VERSION
Node1_GPU   Ready     <none>    13d       v1.10.0
```

Worker GPU specifications

```
# nvidia-smi
Mon Apr  2 23:13:37 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.30                 Driver Version: 390.30                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40m          On   | 00000000:02:00.0 Off |                    0 |
| N/A   30C    P8    20W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K40m          On   | 00000000:84:00.0 Off |                    0 |
| N/A   32C    P8    20W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Tests

Test #1 - Simple vector-add (CUDA8)

This CUDA Runtime API sample is a very basic sample that implements element by element vector addition. The examples uses CUDA8 driver.

```
#cat vector-add.yaml
apiVersion: v1
kind: Pod
metadata:
 name: vector-add
spec:
 restartPolicy: OnFailure
 containers:
    - name: cuda-vector-add
```

(continues on next page)

(continued from previous page)

```
     # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/
↪Dockerfile
     image: "k8s.gcr.io/cuda-vector-add:v0.1"
     resources:
       limits:
         nvidia.com/gpu: 1
```

```
# kubectl apply -f vector-add.yaml
pod "vector-add" created

# kubectl get pods --show-all
NAME                   READY     STATUS       RESTARTS   AGE
vector-add       0/1       Completed   0      4s
```

Test #2 - Simple vector-add with different CUDA driver

This CUDA Runtime API sample is a very basic sample that implements element by element vector addition. The examples uses two Docker images with different version of CUDA driver. To complete the test, a new Docker image with CUDA driver version 9 has been built and uploaded in a private repo.

```
# cat cuda8-vector-add.yaml
apiVersion: v1
kind: Pod
metadata:
 name: cuda8-vector-add
spec:
 restartPolicy: OnFailure
 containers:
    - name: cuda-vector-add
     # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/
↪Dockerfile
     image: "k8s.gcr.io/cuda-vector-add:v0.1"
     resources:
       limits:
         nvidia.com/gpu: 1

# cat cuda9-vector-add.yaml
apiVersion: v1
kind: Pod
metadata:
 name: cuda9-vector-add
spec:
 restartPolicy: OnFailure
 containers:
    - name: cuda-vector-add
     image: <private repo>/deep/cuda-vector-add:v0.2
     resources:
       limits:
         nvidia.com/gpu: 1
```

```
# kubectl apply -f cuda9-vector-add.yaml -f cuda8-vector-add.yaml
pod "cuda9-vector-add" created
pod "cuda8-vector-add" created
```

(continues on next page)

(continued from previous page)

```
# kubectl get pods --show-all
NAME                    READY     STATUS       RESTARTS   AGE
cuda8-vector-add        0/1       Completed    0      2s
cuda9-vector-add        0/1       Completed    0      2s
```

Test #3 - Simple BinomialOption with different CUDA driver

This sample evaluates fair call price for a given set of European options under binomial model. To complete the test, two new Docker images with CUDA8 and CUDA9 has been built and uploaded in a private repo. The test will take some seconds and GPU engage can be shown

```
# cat cuda8-binomialoption.yaml
apiVersion: v1
kind: Pod
metadata:
 name: cuda8-binomialoption
spec:
 restartPolicy: OnFailure
 containers:
    - name: cuda8-binomilaoption
     image:  <private_repo>/deep/cuda-binomialoption:v0.1
     resources:
       limits:
         nvidia.com/gpu: 1

# cat cuda9-binomialoption.yaml
apiVersion: v1
kind: Pod
metadata:
 name: cuda9-binomialoption
spec:
 restartPolicy: OnFailure
 containers:
    - name: cuda9-binomialoption
     image: <private_repo>/deep/cuda-binomialoption:v0.2
     resources:
       limits:
         nvidia.com/gpu: 1
```

```
# kubectl apply -f cuda8-binomialoption.yaml -f cuda9-binomialoption.yaml
pod "cuda8-binomialoption" created
pod "cuda9-binomialoption" created

# kubectl get pods --show-all
NAME                    READY     STATUS       RESTARTS   AGE
cuda8-binomialoption    1/1     Running     0          2s
cuda9-binomialoption    1/1     Running     0          2s

# kubectl get pods --show-all
NAME                    READY     STATUS       RESTARTS   AGE
cuda8-binomialoption    1/1     Running     0          22s
cuda9-binomialoption    1/1     Running     0          22s

# kubectl get pods --show-all
```

(continues on next page)

(continued from previous page)

```
NAME                      READY     STATUS      RESTARTS   AGE
cuda8-binomialoption    0/1     Completed   0      1m
cuda9-binomialoption    0/1     Completed   0      1m

# nvidia-smi
Mon Apr  2 23:35:17 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.30                 Driver Version: 390.30                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40m          On   | 00000000:02:00.0 Off |                    0 |
| N/A   31C    P0    63W / 235W |     80MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K40m          On   | 00000000:84:00.0 Off |                    0 |
| N/A   33C    P0    63W / 235W |     80MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      3385      C   ./binomialOptions                              69MiB |
|    1      3369      C   ./binomialOptions                              69MiB |
+-----------------------------------------------------------------------------+
```

Test #4 - Job Scheduling features

Tests highlithing the features of the Kubernetes scheduler. Default schedule policies are used (FIFO).

Submission of a bunch of different cuda jobs with different running time.

- Parrec (1h)
- Cuda8-binomialoption.yaml (5 min)
- Cuda9-binomialoption.yaml (5 min)
- Cuda8-vector-add.yaml (few sec)
- Cuda9-vector-add.yaml (few sec)

The parrec job has been launched as first job. One GPU has been engaged by the job; the other is still available for other jobs.

```
# kubectl apply -f parrec.yaml
pod "parrec" created

# kubectl get pods -o wide
NAME      READY     STATUS    RESTARTS   AGE      IP             NODE
parrec    1/1       Running   0          22s      172.30.0.52    gpu-node-01
```

Other jobs have been submitted in the following order:

```
# kubectl apply -f cuda8-binomialoption.yaml -f cuda9-binomialoption.yaml -f cuda8-
↪vector-add.yaml -f cuda9-vector-add.yaml
pod "cuda8-binomialoption" created
```

(continues on next page)

(continued from previous page)

```
pod "cuda9-binomialoption" created
pod "cuda8-vector-add" created
pod "cuda9-vector-add" created

# kubectl get pods -o wide
NAME                   READY     STATUS    RESTARTS   AGE     IP            NODE
cuda8-binomialoption   1/1       Running   0          4s      172.30.0.53   gpu-node-
↪01
cuda8-vector-add       0/1       Pending   0          4s      <none>        <none>
cuda9-binomialoption   0/1       Pending   0          4s      <none>        <none>
cuda9-vector-add       0/1       Pending   0          4s      <none>        <none>
parrec                 1/1       Running   0          1m      172.30.0.52   gpu-node-
↪01
```

The “cuda8-binomialoption” is running, the other are in the FIFO queue in pending state. After completion, the other job will be running in the same order they have been submitted.

```
# kubectl get pods -o wide

NAME                   READY     STATUS    RESTARTS   AGE     IP            NODE
cuda8-binomialoption   1/1       Running   0          31s     172.30.0.53   gpu-node-
↪01
cuda8-vector-add       0/1       Pending   0          31s     <none>        <none>
cuda9-binomialoption   0/1       Pending   0          31s     <none>        <none>
cuda9-vector-add       0/1       Pending   0          31s     <none>        <none>
parrec                 1/1       Running   0          2m      172.30.0.52   gpu-node-
↪01

# kubectl get pods -o wide
NAME                   READY     STATUS      RESTARTS   AGE     IP            NODE
cuda8-binomialoption   0/1       Completed   0          49s     172.30.0.53   gpu-
↪node-01
cuda8-vector-add       0/1       Pending     0          49s     <none>        <none>
cuda9-binomialoption   0/1       Pending     0          49s     <none>        <none>
cuda9-vector-add       0/1       Pending     0          49s     <none>        <none>
parrec                 1/1       Running     0          2m      172.30.0.52   gpu-
↪node-01

# kubectl get pods -o wide
NAME                   READY     STATUS              RESTARTS   AGE     IP             ␣
↪NODE
cuda8-binomialoption   0/1       Completed           0          1m      172.30.0.53    ␣
↪gpu-node-01
cuda8-vector-add       0/1       Pending             0          1m      <none>
↪<none>
cuda9-binomialoption   0/1       ContainerCreating   0          1m      <none>         ␣
↪gpu-node-01
cuda9-vector-add       0/1       Pending             0          1m      <none>
↪<none>
parrec                 1/1       Running             0          2m      172.30.0.52    ␣
↪gpu-node-01

# kubectl get pods -o wide
NAME                   READY     STATUS      RESTARTS   AGE     IP            NODE
cuda8-binomialoption   0/1       Completed   0          1m      172.30.0.53   gpu-
↪node-01
```

(continues on next page)

(continued from previous page)

```
cuda8-vector-add       0/1       Pending     0          1m       <none>        <none>
cuda9-binomialoption   1/1       Running     0          1m       172.30.0.54   gpu-
↪node-01
cuda9-vector-add       0/1       Pending     0          1m       <none>        <none>
parrec                 1/1       Running     0          2m       172.30.0.52   gpu-
↪node-01

# kubectl get pods -o wide
NAME                   READY     STATUS      RESTARTS   AGE      IP            NODE
cuda8-binomialoption   0/1       Completed   0          2m       172.30.0.53   gpu-
↪node-01
cuda8-vector-add       0/1       Completed   0          2m       172.30.0.55   gpu-
↪node-01
cuda9-binomialoption   0/1       Completed   0          2m       172.30.0.54   gpu-
↪node-01
cuda9-vector-add       0/1       Pending     0          2m       <none>        <none>
parrec                 1/1       Running     0          3m       172.30.0.52   gpu-
↪node-01

# kubectl get pods -o wide
NAME                   READY     STATUS      RESTARTS   AGE      IP            NODE
cuda8-binomialoption   0/1       Completed   0          2m       172.30.0.53   gpu-
↪node-01
cuda8-vector-add       0/1       Completed   0          2m       172.30.0.55   gpu-
↪node-01
cuda9-binomialoption   0/1       Completed   0          2m       172.30.0.54   gpu-
↪node-01
cuda9-vector-add       0/1       Completed   0          2m       172.30.0.56   gpu-
↪node-01
parrec                 1/1       Running     0          4m       172.30.0.52   gpu-
↪node-01
```

Access PODs from outside the cluster

To access PODs from outside the cluster it can be possible following different procedures that strictly depend on the usecase and (cloud) providers.

NodePort**,** ***hostNetwork***, ***hostPort***, ***LoadBalancer*** and ***Ingress*** features of Kubernetes can be adopted as described in the following Reference:

http://alesnosek.com/blog/2017/02/14/accessing-kubernetes-pods-from-outside-of-the-cluster/

For example puroposes, the **Test #5** example will describe and use the ***NodePort*** procedure as the cluster is defined as 1 Master and 1 Worker both with routable IPs.

Test #5 - Long running service with NodePort

Prerequisites

1. Kubernetes Node with routable IP
2. Port range dynamically selected as from the “kubernetes-apiservice.service” configuration file
3. Nginx replica 2; V. 1.13.12 - latest (as from the YAML files)

Yaml files related to nginx deployment and nginx service

```
# cat ngnix.deploy.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: nginx-example
 namespace: default
 labels:
    app: nginx
spec:
 replicas: 2
 strategy:
    type: RollingUpdate
    rollingUpdate:
     maxSurge: 1
     maxUnavailable: 0
 template:
    metadata:
     labels:
       app: nginx
       name: nginx
    spec:
     containers:
     - image: nginx:latest
       name: ingress-example
       ports:
       - name: http
         containerPort: 80
       readinessProbe:
         httpGet:
           path: /
           port: 80
           scheme: HTTP
       livenessProbe:
         httpGet:
           path: /
           port: 80
           scheme: HTTP
         initialDelaySeconds: 5
         timeoutSeconds: 1
```

```
# cat ngnix.svc.yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 namespace: default
spec:
 type: NodePort
 ports:
 - name: http
    port: 80
    targetPort: 80
    protocol: TCP
 selector:
    app: nginx
```

Creation of nginx POD and nginx service. The following commands will return the Node hostname and the port associated to the nginx.

```
# kubectl apply -f ngnix.deploy.yaml -f ngnix.svc.yaml
deployment.extensions "nginx-example" created
service "nginx" created
# kubectl get pods
NAME                              READY     STATUS      RESTARTS   AGE
nginx-example-78847794b7-8nm8t    0/1     Running       0          11s
nginx-example-78847794b7-n8nxs    0/1     Running       0          11s

# kubectl get pods
NAME                              READY     STATUS      RESTARTS   AGE
nginx-example-78847794b7-8nm8t    1/1     Running       0          30s
nginx-example-78847794b7-n8nxs    1/1     Running       0          30s

# kubectl get svc
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)      AGE
nginx         NodePort    192.168.0.130   <none>      80:30916/TCP    51s
```

Test of nginx

```
# curl http://gpu-node-01:30916
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
...
```

Delete one of the two PODs from nginx

```
# kubectl delete pod nginx-example-78847794b7-8nm8t
pod "nginx-example-78847794b7-8nm8t" deleted
```

A new POD is creating while the old POD is getting deleted. No service downtime is registered from the user

```
# kubectl get pods
NAME                              READY     STATUS        RESTARTS   AGE
nginx-example-78847794b7-6gvnn    0/1     Running         0          4s
nginx-example-78847794b7-8nm8t    0/1     Terminating     0          12m
nginx-example-78847794b7-n8nxs    1/1     Running         0          12m

# kubectl get pods
NAME                              READY     STATUS      RESTARTS   AGE
nginx-example-78847794b7-6gvnn    0/1     Running       0          12s
nginx-example-78847794b7-n8nxs    1/1     Running       0          12m

# kubectl get pods -o wide
NAME                              READY     STATUS      RESTARTS   AGE       IP          ␣
↪   NODE
nginx-example-78847794b7-6gvnn    1/1     Running       0          28s       172.30.0.61␣
↪   gpu-node-01
nginx-example-78847794b7-n8nxs    1/1     Running       0          12m       172.30.0.59␣
↪   gpu-node-01
```

Delete the seconf POD. The internal POD IP is changing, but the public endpoint is the same.

```
# kubectl delete pod nginx-example-78847794b7-n8nxs
pod "nginx-example-78847794b7-n8nxs" deleted
```

(continues on next page)

(continued from previous page)

```
# kubectl get pods -o wide
NAME                                READY     STATUS        RESTARTS   AGE     IP           ␣
↪    NODE
nginx-example-78847794b7-2szlv   0/1      Running         0          4s      172.30.0.
↪62    gpu-node-01
nginx-example-78847794b7-6gvnn   1/1      Running         0          50s     172.30.0.
↪61    gpu-node-01
nginx-example-78847794b7-n8nxs   0/1      Terminating     0          13m     172.30.0.
↪59    gpu-node-01
# kubectl get pods -o wide
NAME                                READY     STATUS       RESTARTS   AGE     IP           ␣
↪  NODE
nginx-example-78847794b7-2szlv   1/1      Running        0          13s     172.30.0.62␣
↪  gpu-node-01
nginx-example-78847794b7-6gvnn   1/1      Running        0          59s     172.30.0.61␣
↪  gpu-node-01
```

Changing the version of nginx with an older version (v. 1.12).

```
# cat ngnix.deploy.yaml
apiVersion: extensions/v1beta1
...
     containers:
     - image: nginx:1.12
```

Apply changes. New PODs are created while old PODs are deleted. No nginx downtime is registered from the user. Public endpoint and port remain unchanged.

```
# kubectl apply -f ngnix.deploy.yaml -f ngnix.svc.yaml
deployment.extensions "nginx-example" configured
service "nginx" unchanged

# kubectl get pods
NAME                                READY     STATUS       RESTARTS   AGE
nginx-example-5d9764f848-8kc8b   0/1      Running        0          9s
nginx-example-78847794b7-2szlv   1/1      Running        0          2m
nginx-example-78847794b7-6gvnn   1/1      Running        0          2m

# kubectl get pods -o wide
NAME                                READY     STATUS        RESTARTS   AGE     IP           ␣
↪    NODE
nginx-example-5d9764f848-8kc8b   1/1      Running         0          23s     172.30.0.
↪63    gpu-node-01
nginx-example-5d9764f848-xwr77   1/1      Running         0          7s      172.30.0.
↪64    gpu-node-01
nginx-example-78847794b7-6gvnn   0/1      Terminating     0          3m      172.30.0.
↪61    gpu-node-01

# kubectl get pods -o wide
NAME                                READY     STATUS       RESTARTS   AGE     IP           ␣
↪  NODE
nginx-example-5d9764f848-8kc8b   1/1      Running        0          54s     172.30.0.63␣
↪  gpu-node-01
nginx-example-5d9764f848-xwr77   1/1      Running        0          38s     172.30.0.64␣
↪  gpu-node-01
```

References

The following guides have been followed for the Installation and Configuration of Kubernetes cluster - a detailed step-by-step guide will be provided soon

Kubernetes Installation/Configuration docs

https://github.com/kelseyhightower/kubernetes-the-hard-way

GPU Node - CentOS7.4

Install docker

https://docs.docker.com/install/linux/docker-ce/centos/#set-up-the-repository

Driver nvidia-install repo

https://developer.nvidia.com/cuda-downloads?target_os=Linux

Driver Nvidia-install driver

http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#post-installation-actions

Install NVIDIA-docker repo

https://nvidia.github.io/nvidia-docker/

Install-NVIDIA docker

https://github.com/NVIDIA/nvidia-docker/wiki/Installation-(version-2.0)#prerequisites

Installing GPU node and adding it to Kubernetes cluster

This is a guide on how to install a GPU node and join it in a running Kubernetes cluster deployed with kubeadm. The guide was tested on a Kubernetes cluster v1.9.4 installed with kubeadm. The cluster nodes are KVM virtual machines deployed by OpenStack. VMs are running Ubuntu 16.04.4 LTS. The node with GPU has a single NVIDIA K20m GPU card.

Step-by-step guide

1. We start with a blank node with a GPU. This is the node, we would like to join in our Kubernetes cluster. First, update the node and install graphic drivers. The version of the drivers has to be at least 361.93. We have installed version 387.26 and CUDA Version 8.0.61. Drivers and CUDA installation is not a part of this guide.

   NVIDIA drivers information Expand source

```
ubuntu@virtual-kubernetes-gpu-2:~$ nvidia-smi
Wed Mar 14 08:52:53 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 387.26                 Driver Version: 387.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K20m          Off  | 00000000:00:07.0 Off |                    0 |
| N/A   30C    P0    53W / 225W |      0MiB /  4742MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

CUDA information Expand source

```
ubuntu@virtual-kubernetes-gpu-2:~$ cat /usr/local/cuda-8.0/version.txt
CUDA Version 8.0.61
```

2. The next step is to install Docker on the GPU node. Install Docker CE 17.03 from Docker's repositories for Ubuntu. Proceed with the following commands as a root user.

```
sudo apt-get update
sudo apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
sudo add-apt-repository \
   "deb https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") \
   $(lsb_release -cs) \
   stable"
sudo apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce␣
↪| grep 17.03 | head -1 | awk '{print $3}')
```

Docker installation test Expand source

```
root@virtual-kubernetes-gpu-2:~# docker --version
Docker version 17.03.2-ce, build f5ec1e2
root@virtual-kubernetes-gpu-2:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
```

(continues on next page)

(continued from previous page)

```
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

3. On the GPU node, add nvidia-docker2 package repositories, install it and reload Docker daemon configuration, which might be altered by nvidia-docker2 installation.

```
sudo curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \
  sudo apt-key add -
sudo curl -s -L https://nvidia.github.io/nvidia-docker/ubuntu16.04/amd64/nvidia-
↪docker.list | \
  sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update
sudo apt-get install -y nvidia-docker2
sudo pkill -SIGHUP dockerd
```

nvidia-docker2 GPU test Expand source

```
root@virtual-kubernetes-gpu-2:~# docker run --runtime=nvidia --rm nvidia/cuda␣
↪nvidia-smi
Unable to find image 'nvidia/cuda:latest' locally
latest: Pulling from nvidia/cuda
22dc81ace0ea: Pull complete
1a8b3c87dba3: Pull complete
91390a1c435a: Pull complete
07844b14977e: Pull complete
b78396653dae: Pull complete
95e837069dfa: Pull complete
fef4aadda783: Pull complete
343234bd5cf3: Pull complete
64e8786fc8c1: Pull complete
d6a4723d353c: Pull complete
Digest: sha256:3524adf9b563c27d9a0f6d0584355c1f4f4b38e90b66289b8f8de026a9162eee
Status: Downloaded newer image for nvidia/cuda:latest
Wed Mar 14 10:14:51 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 387.26                 Driver Version: 387.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K20m          Off  | 00000000:00:07.0 Off |                    0 |
| N/A   30C    P0    52W / 225W |      0MiB /  4742MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+
```

(continues on next page)

(continued from previous page)

```
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

4. Set nvidia-runtime as the default runtime for Docker on the GPU node. Edit the `/etc/docker/daemon.json` configuration file and set the "default-runtime" parameter to nvidia. This also allows us to ommit the –runtime=nvidia parameter for Docker.

```
{
    "default-runtime": "nvidia",
    "runtimes": {
        "nvidia": {
            "path": "/usr/bin/nvidia-container-runtime",
            "runtimeArgs": []
        }
    }
}
```

5. As a root user on the GPU node, add Kubernetes package repositories and install kubeadm, kubectl and kubelet. Then turn the swap off as it is not supported by Kubernetes.

```
apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
# turn off swap or comment the swap line in /etc/fstab
sudo swapoff -a
```

Specific version installation; e.g., 1.9.3-00 Expand source

```
# install aptitude, an interface to package manager
root@virtual-kubernetes-gpu-2:~# apt install aptitude -y

# show available kubeadm versions in the repositories
root@virtual-kubernetes-gpu-2:~# aptitude versions kubeadm
Package kubeadm:
p   1.5.7-00   kubernetes-xenial   500
p   1.6.1-00   kubernetes-xenial   500
p   1.6.2-00   kubernetes-xenial   500
...
p   1.9.3-00   kubernetes-xenial   500
p   1.9.4-00   kubernetes-xenial   500

# install specific version of kubelet, kubeadm and kubectl
root@virtual-kubernetes-gpu-2:~# apt-get install -y kubelet=1.9.3-00 kubeadm=1.9.
↪3-00 kubectl=1.9.3-00
```

6. On the GPU node, edit the /etc/systemd/system/kubelet.service.d/10-kubeadm.conf file add the following environment argument to enable DevicePlugins feature gate. If there is already Accelerators feature gate set , remove it.

```
Environment="KUBELET_EXTRA_ARGS=--feature-gates=DevicePlugins=true"
```

/etc/systemd/system/kubelet.service.d/10-kubeadm.conf Expand source

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/
↪bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_SYSTEM_PODS_ARGS=--pod-manifest-path=/etc/kubernetes/
↪manifests --allow-privileged=true"
Environment="KUBELET_NETWORK_ARGS=--network-plugin=cni --cni-conf-dir=/etc/cni/
↪net.d --cni-bin-dir=/opt/cni/bin"
Environment="KUBELET_DNS_ARGS=--cluster-dns=10.96.0.10 --cluster-domain=cluster.
↪local"
Environment="KUBELET_AUTHZ_ARGS=--authorization-mode=Webhook --client-ca-file=/
↪etc/kubernetes/pki/ca.crt"
Environment="KUBELET_CADVISOR_ARGS=--cadvisor-port=0"
Environment="KUBELET_CERTIFICATE_ARGS=--rotate-certificates=true --cert-dir=/var/
↪lib/kubelet/pki"
Environment="KUBELET_EXTRA_ARGS=--feature-gates=DevicePlugins=true"
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_SYSTEM_PODS_ARGS
↪$KUBELET_NETWORK_ARGS $KUBELET_DNS_ARGS $KUBELET_AUTHZ_ARGS $KUBELET_CADVISOR_
↪ARGS $KUBELET_CERTIFICATE_ARGS $KUBELET_EXTRA_ARGS
```

7. On the GPU node, reload and restart kubelet to apply previous changes to the configuration.

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

8. If not already done, enable GPU support on the Kubernetes master by deploying following Daemonset.

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.9/
↪nvidia-device-plugin.yml
```

9. For the simplicity, generate a new token on the Kubernetes master and print the join command.

```
ubuntu@virutal-kubernetes-1:~$ sudo kubeadm token create --print-join-command
kubeadm join --token 6e112b.a598ccc2e90671a6 KUBERNETES_MASTER_IP:6443 --
↪discovery-token-ca-cert-hash␣
↪sha256:863250f81355e64074cedf5e3486af32253e394e939f4b03562e4ec87707de0a
```

10. Go back to the GPU node and use the printed join command to add GPU node into the cluster.

```
ubuntu@virtual-kubernetes-gpu-2:~$ sudo kubeadm join --token 6e112b.
↪a598ccc2e90671a6 KUBERNETES_MASTER_IP:6443 --discovery-token-ca-cert-hash␣
↪sha256:863250f81355e64074cedf5e3486af32253e394e939f4b03562e4ec87707de0a
[preflight] Running pre-flight checks.
    [WARNING FileExisting-crictl]: crictl not found in system path
[discovery] Trying to connect to API Server "KUBERNETES_MASTER_IP:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://
↪KUBERNETES_MASTER_IP:6443"
[discovery] Requesting info from "https://KUBERNETES_MASTER_IP:6443" again to␣
↪validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate␣
↪validates against pinned roots, will use API Server "KUBERNETES_MASTER_IP:6443"
[discovery] Successfully established connection with API Server "KUBERNETES_
↪MASTER_IP:6443"
```

(continues on next page)

(continued from previous page)

```

This node has joined the cluster:
* Certificate signing request was sent to master and a response
  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
```

11. Run following command to see the GPU node (virtual-kubernetes-gpu-2) status on the cluster.

```
ubuntu@virutal-kubernetes-1:~$ kubectl get nodes
NAME                        STATUS     ROLES     AGE       VERSION
virtual-kubernetes-gpu      Ready      <none>    1d        v1.9.4
virtual-kubernetes-gpu-2    NotReady   <none>    13s       v1.9.4
virutal-kubernetes-1        Ready      master    5d        v1.9.4
virutal-kubernetes-2        Ready      <none>    5d        v1.9.4
virutal-kubernetes-3        Ready      <none>    5d        v1.9.4
```

12. After a while, the node is ready.

```
virtual-kubernetes-gpu-2   Ready     <none>    7m        v1.9.4
```

13. Now we have 2 GPU nodes ready in our Kubernetes cluster. We can label the recently added node (virtual-kubernetes-gpu-2) with the accelerator type by running following command on the master.

```
kubectl label nodes virtual-kubernetes-gpu-2 accelerator=nvidia-tesla-k20m
```

14. To check nodes for accelerator label, run kubectl get nodes -L accelerator on Kubernetes master.

```
ubuntu@virutal-kubernetes-1:~/kubernetes$ kubectl get nodes -L accelerator
NAME                        STATUS    ROLES     AGE       VERSION   ACCELERATOR
virtual-kubernetes-gpu      Ready     <none>    1d        v1.9.4    nvidia-tesla-
↪k20m
virtual-kubernetes-gpu-2    Ready     <none>    24m       v1.9.4    nvidia-tesla-
↪k20m
virutal-kubernetes-1        Ready     master    5d        v1.9.4
virutal-kubernetes-2        Ready     <none>    5d        v1.9.4
virutal-kubernetes-3        Ready     <none>    5d        v1.9.4
```

15. To test the GPU nodes, go to the master and create a file with the following content and execute it.

gpu-test.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-
↪cuda/Dockerfile
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1 # requesting 1 GPU per container
```

(continues on next page)

(continued from previous page)

```
  nodeSelector:
    accelerator: nvidia-tesla-k20m # or nvidia-tesla-k80 etc.
```

```
ubuntu@virutal-kubernetes-1:~/kubernetes$ kubectl create -f gpu-test.yml
pod "cuda-vector-add" created
ubuntu@virutal-kubernetes-1:~/kubernetes$ kubectl get pods -a
NAME                            READY     STATUS      RESTARTS   AGE
cuda-vector-add                 0/1       Completed   0          19s
```

2.1.3 OpenStack nova-lxd

OpenStack nova-lxd installation via Ansible

This document provide step by step deployment of Openstack site with nova-lxd via Openstack Ansible. That would allows site admins to avoid obstacles and pitfalls during deployment and create a Openstack site with nova-lxd for testing and development in a short time without studying extensive documentation of Openstack Ansible.

This documentation also show that nova-lxd is supported (although not perfectly, see steps 10-11 for fixing LXD configuration) in mainstream Openstack deployment tool beside Ubuntu-specific Juju charms. Ubuntu 18.04 and LXD 3.0 are also supported (instead of 16.04 / LXD 2.0 used in Juju).

Comparison between Openstack Ansible and Juju/conjure-up

Juju is Ubuntu- specific, and using Ubuntu distro packages for installing Openstack. Openstack Ansible is distro-neutral and by default it uses source code from github for Openstack installation (configurable for distro packages, however). For testing and development, installation via source code has advantages using latest codes, however, for production sites, distro packages are more stable, especially when Ubuntu offers up to 5-year software maintenance in comparison with 18-month from Openstack.

Other differences: Juju uses LXD containers for all Openstack services and has better integration with Ubuntu ecosystem (MAAS, LXD). Openstack Ansible use LXC containers for services on master (like nova-api, keystone, glance, cinder) or directly baremetal for services on workers (like nova-compute, cinder-volume).

Openstack Ansible offers wide possibilities of customization, e.g. method of installation (distro package vs source), based Linux distro (RedHat/CenOS, openSUSE and Ubuntu), selection of services to be installed. Beside Openstack installation, it also does many other tasks for security hardening and high availability (e.g. haproxy/Galera). As the result, it is more complex and need more time for deployment. (Un)fortunately, Openstack Ansible has very extensive documentation that is useful but may require time to study. See [1] and [2] from references for more information.

Installing a All-in-One Openstack site with nova-lxd via Openstack Ansible

This installation takes rather long time (totally around ~2h according to disk performance, network connection and list of installed services). Fast disk and network connection is strongly recommended because of intensive disk operation during installation and large amount of files downloaded from repositories

1. Install Ubuntu Bionic/ Create VM with vanilla Ubuntu Bionic (at least 16GB RAM, 80GB disk). So far do not init/configure the LXD service, Ansible script may report error like “storage already exist”.
2. Optional: Update all packages, then reboot
3. Cloning Openstack-Ansible

```
    # git clone https://git.openstack.org/openstack/openstack-ansible \
        /opt/openstack-ansible
    # cd /opt/openstack-ansible
```

1. Optional: Choose version/branch if needed

```
    # git tag -l
    # git checkout master
    # git describe --abbrev=0 --tags
```

1. Bootstrap Ansible (about 6min in the test)

```
    # scripts/bootstrap-ansible.sh
```

1. Default scenario is without CEPH. Select CEPH scenario if you want and bootstrap AOI (about 6min in the test)

```
    # export SCENARIO='ceph'
    # scripts/bootstrap-aio.sh
```

1. Setting hypervisor to LXD:

Edit file "/etc/openstack_deploy/user_variables.yml". Add a line "nova_virt_type: lxd" into it.

```
     nova_virt_type: lxd
```

1. Examine list of services to be installed in "/etc/openstack_deploy/conf.d/" Copy more services from "/opt/openstack-ansible/etc/openstack_deploy/conf.d/" to "/etc/openstack_deploy/conf.d/" as needed. Remember to change the filename extension from aio to yml. For example, CEPH scenario is without Horizon and we want to add it.

```
    # cp /opt/openstack-ansible/etc/openstack_deploy/conf.d/horizon.yml.aio \
        /etc/openstack_deploy/conf.d/horizon.yml
```

1. Start the core installation (in 3 big steps). It takes long time (about 15min + 30min + 30min in the test), so you may want to use tmux or screen terminal if using SSH on unreliable network.

```
    # cd /opt/openstack-ansible/playbooks
    # openstack-ansible setup-hosts.yml
    # openstack-ansible setup-infrastructure.yml
    # openstack-ansible setup-openstack.yml
```

During execution of the first command "openstack-ansible setup-hosts.yml", it is possible that you have timeout error during LXC caching if you have slow disk and/or network connection. In this case, please increase the value of "lxc_cache_prep_timeout" in "/etc/ansible/roles/lxc_hosts/defaults/main.yml" and re-execute the command.

The test "TASK [os_tempest : Execute tempest tests]" in the last command "openstack-ansible setup-openstack.yml" will fail. Ignore it and continue to next steps.

1. The Ansible script create a storage with the name "default" and driver "dir" for LXD, however, it does not work (btrfs or zfs required). Now create a new storage for LXD with btrfs with other name, e.g. "nova". A simplest way is to run "lxd init" and configure storage via it. Do not touch networks/bridges or other configuration.

```
# lxd init
```

1. Setting LXD storage for nova: Edit file /etc/nova/nova.conf, add the following section into the file. For the pool option, use the name of storage created in the previous step, e.g. "nova"

```
[lxd]
allow_live_migration = True
pool ="nova"
```

Restart nova-compute service:

```
# systemctl restart nova-compute
# systemctl status nova-compute
```

1. Installation is now completed, however, some post-installation configurations are needed before starting the first VM. Refer to [3] for more information. The post-installation configuration can be done via CLI or via Horizon dashboard. The following steps show the configuration via Horizon.
2. Get the IP address of load balancer from "external_lb_vip_address" in "/etc/openstack_deploy/openstack_user_config.yml" file. Use i"ptables" for IP forwarding to get the dashboard from your PC, e.g. :

```
# iptables -t nat -A PREROUTING -p tcp -m tcp --dport 8080 -j DNAT --to-
↪destination external_lb_vip_address:443
```

Also remember do open firewall for the chosen port (8080 in the example).

1. Open Horizon dashboard in your browser as https://ip_address_of_host:8080. Log in using "admin" user and password stored in "keystone_auth_admin_password" item in "/etc/openstack_deploy/user_secrets.yml" file.
2. In Horizon, do the steps for configuring Openstack network: create a new private network, create a private subnet for the private network, create a router to connect private subnet to existing public network, open ports in security groups (e.g. port 22 for SSH). Also import SSH public key from "~/.ssh/id_rsa.pub" on the host.
3. Creating images does not work in default Horizon installed by Ansible, you must change Horizon setting or use command-line to create image. Use "lxc-attach" command to get into "aio1_utility_container_xxxxxx" container, load Openstack credential, download a LXD image from repository and add it to glance:

```
# lxc-ls
# lxc-attach aio1_utility_container_xxxxxxxx
# cd
# source openrc
# wget http://cloud-images.ubuntu.com/xenial/current/xenial-server-cloudimg-amd64-
↪root.tar.gz
# glance image-create --name xenial-lxd --disk-format raw --container-format bare␣
↪--file xenial-server-cloudimg-amd64-root.tar.gz
```

1. Return to Horizon, create a new VM with the newly added "xenial-lxd" image. Remember to no create a new volume. Allocating a floating IP and assign it to the VM. From command line on the host, try to connect to the VM via ssh.
2. Success.

Notes:

- CEPH volume is still not attachable to VM by defaults, some additional work required.

References

1. https://docs.openstack.org/openstack-ansible/latest/user/aio/quickstart.html

2. https://docs.openstack.org/project-deploy-guide/openstack-ansible/latest/
3. https://docs.openstack.org/openstack-ansible/latest/admin/openstack-firstrun.html

Deploying OpenStack environment with nova-lxd via DevStack

This document is a step-by-step guide of OpenStack with nova-lxd deployment via DevStack on a single machine (All-in-One). The guide was tested on a host running Xenial or Bionic. The host has pre-installed following libraries:

- Python 2.7
- PyLXD 2.2.7

Installation steps

1. Create a new host running Xenial or Bionic with Python 2.7
2. Install the newest version of a library PyLXD 2.2.7

   (a) Install pip

```
$ sudo apt update
$ sudo apt install python-pip
```

   (a) Set up your environment variable `bash LC_ALL`

```
$ export LC_ALL="en_US.UTF-8"
$ export LC_CTYPE="en_US.UTF-8"
$ sudo dpkg-reconfigure locales
```

   (a) Download and install the library PyLXD

```
$ git clone https://github.com/lxc/pylxd.git
$ cd pylxd/
$ sudo pip install .
```

3. *Optional step:* Install ZFS

```
$ sudo apt install lxd zfsutils-linux
```

1. *Optional step:* If you need to install a different LXD/LXC version, execute the following steps:

   (a) Uninstall LXD and LXC (delete configuration and data files of LXD/LXC and it's dependencies)

```
$ sudo apt-get purge --auto-remove lxd lxc
```

   (a) Install the wanted version of LXD/LXC:

   - LXD/LXC 3.0.1 on a host running Xenial

```
$ sudo apt-get purge --auto-remove lxd lxc
```

   - the newest version

```
$ sudo snap install lxd
```

if you wish to install LXD 3.0 and then only get bugfixes and security updates. If running staging systems, you may want to run those on the candidate channels, using `bash--channel=candidate` and `bash--channel=3.0/candidate` respectively.

```
$ sudo snap install lxd --channel=3.0
```

2. Configure LXD:

   (a) In order to use LXD, the system user must be a member of the 'lxd' user group.

   ```
   $ sudo adduser martin lxd
   $ newgrp lxd
   $ groups
   ```

   (b) LXD initialisation

   ```
   $ sudo lxd init
   ```

   The session below (LXD 3.0.1 with a zfs storage backend) is what was used to write this guide. Note that pressing Enter (a null answer) will accept the default answer (provided in square brackets).

   ```
   Would you like to use LXD clustering? (yes/no) [default=no]:
   Do you want to configure a new storage pool? (yes/no) [default=yes]:
   Name of the new storage pool [default=default]: lxd
   Name of the storage backend to use (btrfs, dir, lvm, zfs) [default=zfs]:
   Create a new ZFS pool? (yes/no) [default=yes]:
   Would you like to use an existing block device? (yes/no) [default=no]:
   Size in GB of the new loop device (1GB minimum) [default=15GB]:
   Would you like to connect to a MAAS server? (yes/no) [default=no]:
   Would you like to create a new local network bridge? (yes/no) [default=yes]:
   What should the new bridge be called? [default=lxdbr0]:
   What IPv4 address should be used? (CIDR subnet notation, “auto” or “none”)␣
   ↪[default=auto]:
   What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”)␣
   ↪[default=auto]: none
   Would you like LXD to be available over the network? (yes/no) [default=no]:
   Would you like stale cached images to be updated automatically? (yes/no)␣
   ↪[default=yes]
   Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]:
   ```

3. *Optional step:* Remove old LXD version to avoid conflict

```
$ sudo /snap/bin/lxd.migrate
```

1. *Optional step:* Increase the limit of number open files (only needed for larger tests). See https://lxd.readthedocs.io/en/latest/production-setup/
2. *Optional step:* Check configuration (as a user), if your configuration is correct

```
$ sudo /snap/bin/lxc storage list
$ sudo /snap/bin/lxc storage show default
$ sudo /snap/bin/lxc network show lxdbr0
$ sudo /snap/bin/lxc profile show default
```

9.*Optional step:* Run a test container in LXD (as a user), if LXD work correctly

```
$ sudo lxc launch ubuntu:16.04 u1
$ sudo lxc exec u1 ping www.ubuntu.com
```

(continues on next page)

(continued from previous page)

```
$ sudo lxc stop u1
$ sudo lxc delete u1
```

1. Create a user “Stack” and add it to the ‘lxd’ user group

```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack
$ sudo usermod -G lxd -a stack
$ sudo su - stack
```

1. Download OpenStack installation scripts from DevStack repository

```
$ git clone https://git.openstack.org/openstack-dev/devstack
$ cd devstack
```

1. Create a **local.conf** file (a branch of a nova-lxd plugin is `bash stable/rocky`) as follows:

```
[[local|localrc]]

HOST_IP=127.0.0.1 # set this to your IP
FLAT_INTERFACE=ens2 # change this to your eth0

ADMIN_PASSWORD=devstack
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
SERVICE_TOKEN=$ADMIN_PASSWORD

## run the services you want to use
ENABLED_SERVICES=rabbit,mysql,key
ENABLED_SERVICES+=,g-api,g-reg
ENABLED_SERVICES+=,n-cpu,n-api,n-crt,n-obj,n-cond,n-sch,n-novnc,n-cauth,placement-api,
↪placement-client
ENABLED_SERVICES+=,neutron,q-svc,q-agt,q-dhcp,q-meta,q-l3
ENABLED_SERVICES+=,cinder,c-sch,c-api,c-vol
ENABLED_SERVICES+=,horizon

## disabled services
disable_service n-net

## enable nova-lxd
enable_plugin nova-lxd https://git.openstack.org/openstack/nova-lxd stable/rocky
```

1. Start installation of an OpenStack environment (it will take a 15 - 20 minutes, largely depending on the speed of your internet connection. Many git trees and packages will be installed during this process.)

```
$ ./stack.sh
```

1. Configuration of `bash nova-compute`: In order for a lxd storage pool to be recognized in nova, the `bash/etc/nova/nova-cpu.conf` file needs to have `bash [lxd] section` containing the following lines:

```
[lxd]
allow_live_migration = True
pool = {{ storage_pool }}
```

Restart nova-compute service:

```
$ systemctl restart devstack@n-cpu.service
$ systemctl status devstack@n-cpu.service
```

1. *Optional step:* if your OpenStack installation is set to incorect repository, execute the following commands (adding a Rocky cloud archive):

```
$ sudo rm /etc/apt/sources.list.d/{{Bad_archive}}.list
$ sudo add-apt-repository cloud-archive:rocky
```

1. *Optional step:* Use IP forwarding to get access to the dashboard from outside by executing the following command on the host where the whole OpenStack environment with nova-lxd is installed:

```
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to "{{IP_of_
↪horizon}}:80"
```

where IP_of_horizon is the IP address of the dashboard that is given when the installation finishes (in the format of 10.x.x.x, e.g. 10.110.236.154)

1. Log into the dashboard (http://{{IP_address_of_host}}:8080/horizon), with "admin_domain" domain, "admin" user and "devstack" password, where ip_address_of_host is the IP of the host machine, where the whole OpenStack environment is installed (e.g. http://147.213.76.100:8080/horizon)
2. Create a new VM (Launch instance) in Compute->Instances. **Do not create a new volume (choose NO for new volume)**, and add only internal network.
3. In Network->Security group, add new ingress rules for ICMP (ping) and TCP port 22 (SSH) to default security group.
4. Allocate a new floating IP from Network -> Floating IPs and assign to the VM.
5. From host machine, try ssh to floating IP of VM

```
$ ssh ubuntu@{{Floating_ip}}
```

Handy commands:

Notes:

- CEPH volume is still not attachable to VM by defaults, some additional work required.

References

1. https://discuss.linuxcontainers.org/t/lxd-3-0-0-has-been-released/1491
2. https://docs.jujucharms.com/devel/en/tut-lxd
3. https://docs.openstack.org/devstack/latest/
4. https://github.com/openstack/nova-lxd/blob/master/devstack/local.conf.sample
5. https://wiki.ubuntu.com/OpenStack/CloudArchive

Installing nova-lxd with Juju

This is instruction of deploying Openstack with nova-lxd on single machine (All-in-One) for testing and deployment. Tested on Ubuntu Xenial and Bionic. Whole process of installation would take around 2h. Be careful with the LXD setting, leave default values when possible to avoid later problems. ("none" to IPv6, "lxdbr0" for bridge and "default" for name of LXD storage.

Installation

1. Create/Install a new machine with Ubuntu 16/18.04 and adequate performance (16GB RAM needed)
2. Optional: Update all packages (as root)

```
# apt update && apt dist-upgrade -y && apt autoremove -y
```

1. Install LXD (version 3.x required):

```
# sudo snap install lxd
```

1. Configure LXD:

```
# /snap/bin/lxd init
```

Say none to IPv6. If there a empty disk volume (block storage) attached, use it as existing block storage for LXD without mounting/formatting, otherwise use dir (with name of the storage as "default" for both cases). For other options, leave default value.

1. Optional: Add current user to LXD group (for using LXD as user later)

```
# sudo usermod -a -G lxd $USER
# newgrp lxd
```

1. Optional: remove old LXD version to avoid conflict

```
# sudo /snap/bin/lxd.migrate
```

1. Optional: increase the limit of number open files (only needed for larger tests). See https://lxd.readthedocs.io/en/latest/production-setup/

```
# echo fs.inotify.max_queued_events=1048576 | sudo tee -a /etc/sysctl.conf
# echo fs.inotify.max_user_instances=1048576 | sudo tee -a /etc/sysctl.conf
# echo fs.inotify.max_user_watches=1048576 | sudo tee -a /etc/sysctl.conf
# echo vm.max_map_count=262144 | sudo tee -a /etc/sysctl.conf
# echo vm.swappiness=1 | sudo tee -a /etc/sysctl.conf
# sudo sysctl -p
```

Also update default profile for improving network connnection

```
# lxc profile device set default eth0 mtu 9000
```

1. Optional: Check configuration (as user), if your configuration is correct

```
# /snap/bin/lxc storage list
# /snap/bin/lxc storage show default
# /snap/bin/lxc network show lxdbr0
# /snap/bin/lxc profile show default
```

1. Optional: Run a test container in LXD (as user), if LXD work correctly

```
# lxc launch ubuntu:16.04 u1
# lxc exec u1 ping www.ubuntu.com
# lxc stop u1
# lxc delete u1
```

1. Install juju:

```
# sudo snap install juju --classic
```

1. Install conjure-up

```
# sudo snap install conjure-up --classic
```

1. Optional: Start tmux terminal (to avoid unwanted termination in the case of network disruption)

```
# tmux
```

1. Start conjure-up (in tmux terminal if tmux is used):

```
# conjure-up
```

Choose Openstack with Nova-LXD, localhost. Other options can be left as default (lxdbr0 network bridge, "default" storage pool, ~/.ssh for SSH key), then deploy

1. Go to have a coffee for about 45-90 min (depending on performance of host machine). The installation with deploy nova-lxd with relevant services (keystone, glance, cinder, horizon). Remember the IP address of horizon dashboard from the output (e.g. http://10.110.236.154/horizon).
2. Use IP forwarding to get access to dashboard from outside by executing the following command on the host where whole openstack with nova-lxd is installed:

```
# sudo iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to "IP_of_
↪horizon:80"
```

where IP_of_horizon is the IP address of the dashboard that is given when the installation finishes (in format of 10.x.x.x, e.g. 10.110.236.154)

1. Log into the dashboard (http://ip_address_of_host:8080/horizon), with "admin_domain" domain, "admin" user and "openstack" password, where ip_address_of_host is the IP of host machine, where whole Openstack is installed (e.g. http://147.213.76.100:8080/horizon)
2. Create a new VM (Launch instance) in Compute->Instances. Do not create new volume (choose NO for new volume), and add only internal network.
3. In Network->Security group, add new ingress rules for ICMP (ping) and TCP port 22 (SSH) to default security group.
4. Allocate a new floating IP from Network -> Floating IPs and assign to the VM.
5. From host machine, try ssh to floating IP of VM

```
# ssh ubuntu@floating_ip
```

Notes

- Ceph and Cinder are installed together with other Openstack services, however attaching block storage does not work. According to https://lists.gt.net/openstack/dev/64776, it should require some additional work.

- Although Ubuntu Bionic with LXD 3.0 was used as base OS on the host, in LXD containers are Ubuntu Xenial with LXD 2.0

OpenStack nova-lxd testing configuration

As the nova-lxd plugin in OpenStack is still experimental, we need to deploy and test its current status to find out the working configuration before implementing new features

Testing of nova-lxd with different software configurations

OpenStack with nova-lxd has been deployed by different methods: OpenStack DevStack, JuJu, and OpenStack Ansible. Various combinations with OpenStack/LXD and base OS version have been tested to find out the working configuration of OpenStack nova-lxd for production.

Since nova-lxd plugin and OpenStack DevStack (deployment of OpenStack Cloud by Python scripts) have extremely shallow documentation, the first step was finding out the best starting configuration for development. We reached the following results (without advanced post-configuration efforts):

The hosts have the following static parameters:

There are two other deployment options for OpenStack alongside DevStack which are the following: Juju, and Ansible. The main advantage of these two approaches is automated deployment (with a post-configuration) of an OpenStack Cloud environment. The main differences between the deployment approaches are the configuration of an OpenStack environment, and LXD/LXC support. Juju supports LXD/LXC 2.0.11. It creates inside a Xenial/Bionic host another virtual layer by an LXD daemon from the host, and so Xenial with LXD/LXC 2.0.11 is installed in a container. All in all, it doesn't support higher versions of LXD/LXC which supports GPUs.

On the other hand, Ansible supports LXD/LXC 3.0.1. However, the situation about a nova-lxd plugin integration is the same, but the configuration of an OpenStack environment is different. According to Alex Kavanagh (one of the maintainers for the nova-lxd plugin) suggestions, the plugin has to be post-configured within a nova-compute configuration that is not documented in any official sources. The configuration file has to contain the following lines:

```
[DEFAULT]
compute_driver = nova_lxd.nova.virt.lxd.LXDDriver

[lxd]
allow_live_migration = True
pool = {{ storage_pool }}
```

A problem with the post-configuration is that it has to be performed in a different way within an OpenStack Cloud environment deployed by DevStack. The environment has a dedicated configuration file nova-cpu.conf. The other approaches create the standard deployment of an OpenStack Cloud environment, and so the configuration of the nova-lxd plugin is performed by editing of a nova.conf file.

Working configuration

According to the performed test mentioned above, we chose OpenStack Ansible repository for deployment of an OpenStack Cloud. The main reason is that it deploys a standard OpenStack Cloud with LXD/LXC 3.0.1 which supports GPUs.

2.1.4 uDocker

uDocker new GPU implementation

The use of NVIDIA GPUs for scientific computing requires the deployment of proprietary drivers and libraries. In order to use the GPUs inside containers, the devices associated to the GPU have to be visible inside the container. Furthermore, the driver has to be installed in the image and the version has to match the one deployed on the guest host. This turns the Docker images un-shareable and the image must be built locally for each host. The alternative is to have an image for each version of the driver, which is un-manageable since at each update, many images would have to be built. The *uDocker* released at the end of the Indigo-Datacloud project does not have such features, as such in order to use GPUs, the image has to have the NVIDIA drivers and libraries matching the host system. On the other hand, it is not necessary to pass the NVIDIA devices to the *uDocker* container since they are visible inside the container, in the same way a non-privileged user can use those devices in the host system.

The work performed during the first months of the DEEP-HybridDataCloud project, was to implement such automatism. The development is available in the "devel" branch of the official GitHub repository [1], and scheduled for the first release of DEEP-HybridDataCloud software stack at the end of October 2018. The libraries and drivers deployed in the host system are made available to the containers. This version has been tested under several conditions and by several users and use cases. The tests performed in the framework of DEEP-HybridDataCloud project WP3 and WP4 are described in the following.

Test and evaluation of new implementation

The following tests and benchmark tools were developed to test performance of the python code packed in a Docker container and executed by means of *uDocker* [2]. We compare *uDocker* performance with baremetal execution and via Singularity 2.5.2-dist [3]. The benchmark tools are based on official Tensorflow Docker images from the Docker Hub [4] and deep learning python scripts publicly available at GitHub [5]. All scripts implement convolutional neural networks (CNN) but of different architecture: AlexNet, GoogLeNet, Overfeat, VGG [6], and one based on Tensorflow example for the MNIST dataset [7]. The latter processes MNIST data placed inside the Docker container while others process synthetic data generated on-the-fly. We adapted the scripts for more recent Tensorflow versions and homogenized the scripts to have a 'burn-in' phase, measure total runtime, mean time per batch and its standard deviation. In all tests the same version of the python scripts and corresponding Docker images, tagged as '181004' at both GitHub and Docker Hub [7] are used. The tests comprise the following:

1. They are executed on GPU nodes of ForHLR II cluster [8], where each of the nodes contains four 12-core Intel Xeon processors E7-4830 v3 (Haswell), 1 TB of main memory, 4x960 GB local SSDs, 4 NVIDIA GeForce GTX980 Ti graphic cards. Operating System is RedHat Enterprise Linux 7.5, CUDA Toolkit 9.0.176 and NVIDIA Driver 384.81 are installed system-wide, cudnn 7.0.5 is installed in the user's $HOME directory. The test are performed with Python version 2.7.5.
2. For baremetal performance tests, two Tensorflow GPU versions 1.5.0 and 1.8.0 are installed in separate virtual environments via pip installation tool.
3. For *uDocker* tests, we build two Docker images based on the same Tensorflow versions, 1.5.0 and 1.8.0. The python scripts and MNIST data are stored inside the images. In both Docker images CUDA Toolkit is 9.0.176, cudnn version is 7.0.5, and Python is 2.7.12. We use *uDocker* (devel branch) to pull images from the Docker Hub. To run the containers, F3 (Fakechroot) mode is set with –nvidia flag (to add NVIDIA libraries and binaries).
4. For Singularity tests, Docker images built for *uDocker* tests where converted to Singularity images and uploaded to ForHLR II. Singularity version 2.5.2-dist was used. The containers are executed with –nv flag for NVIDIA support.
5. In all tests, the CNN scripts with synthetic data are executed for 1000 batches, therefore the mean time per batch is averaged over 1000 steps, the MNIST script is run for 20000 steps.

The results of the tests are shown in Figure 1, where we normalize the mean runtime per batch to the baremetal case and Tensorflow 1.8.0. Error bars are scaled by the mean time per batch for baremetal and Tensorflow 1.8.0. The tests do not indicate any penalty in executing the CNN scripts in either container technology *uDocker* or Singularity in comparison with baremetal within the statistical uncertainty. They may even suggest that running the scripts inside containers is slightly faster than in baremetal, which could be connected to caching of data locally at the node in case of containers but needs to be better understood. Tensorflow 1.5.0 tends to be a bit faster than Tensorflow 1.8.0 for synthetic data but slower when real MNIST dataset is processed. This might be interpreted as improved I/O performance in Tensorflow 1.8.0 comparing to 1.5.0.

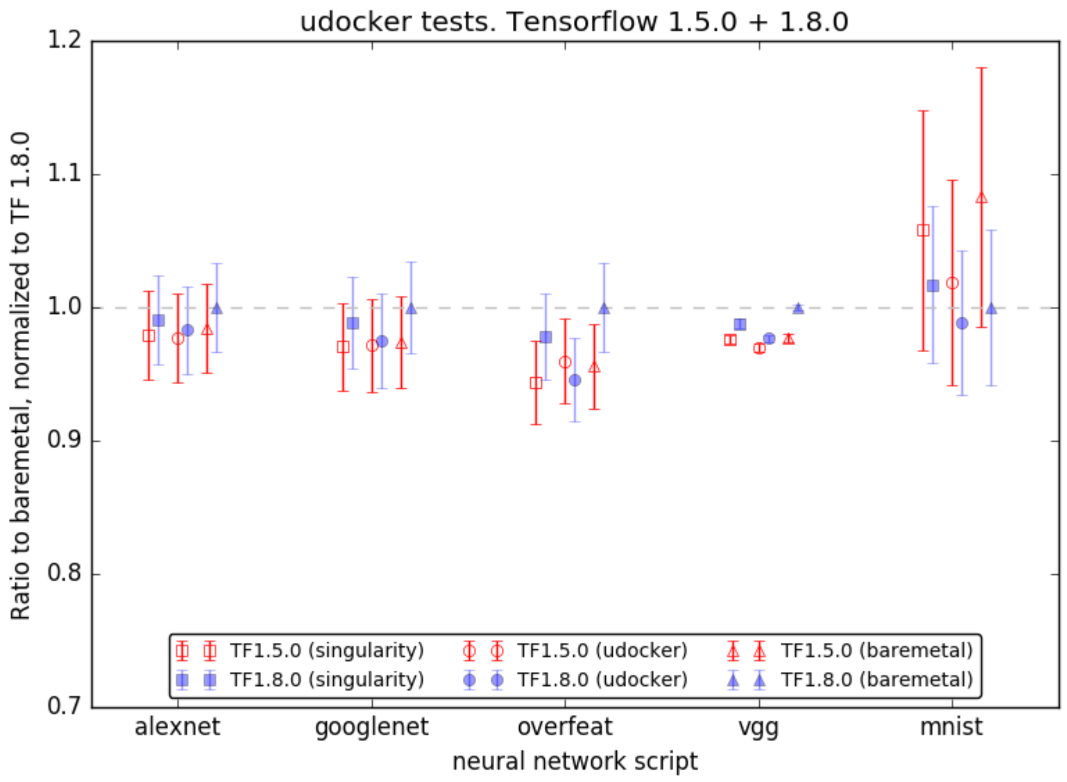


Figure 1: *uDocker* performance tests using Tensorflow 1.5.0 and 1.8.0 in comparison with Baremetal installation in a user directory and Singularity. Lower values indicate better performance.

One job per node is executed in these tests, i.e. only one GPU card of the node is used and three other cards are not involved. *uDocker* however allows to pass environment settings inside containers, therefore making it possible to define which GPU card to use through providing the CUDA_VISIBLE_DEVICES parameter. It was possible to submit a job to our batch system such that in one job we can instantiate 4 *uDocker* containers where each container runs on a separate GPU card of the same node. This test does not indicate a degradation of performance in terms of total runtime and mean runtime per batch but shows significant increase in the uncertainty of the runtime per batch, especially in the case of the MNIST dataset.

As the tests suggest, it is of interest to study the behavior of processing large datasets from containers. Therefore we plan to extend our scripts to establish training on e.g. CIFAR datasets [9]. Such training may also require multi-GPU training so that we can preform tests to access all GPU cards on one node from single container. If our use-cases show interest, we may add other neural network architectures to the tests, such as Long Short-Term Memory (LSTM) or Generative Adversarial Networks (GAN).

9-Oct-2018

References

1. uDocker official GitHub repository: https://github.com/indigo-dc/udocker
2. Jorge Gomes, Emanuele Bagnaschi, Isabel Campos, MarioDavid, Luís Alves, João Martins, João Pina, Alvaro López-García, PabloOrviz, "Enabling rootless Linux Containers in multi-user environments: The udocker tool", Computer Physics Communications, Volume 232, 2018, Pages 84-97, ISSN 0010-4655, https://doi.org/10.1016/j.cpc.2018.05.021.
3. G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," PLoS ONE, 2017.
4. Docker Hub tensorflow/tensorflow, https://hub.docker.com/r/tensorflow/tensorflow/
5. convnet-benchmarks: https://github.com/soumith/convnet-benchmarks/tree/master/tensorflow , Tensorflow MNIST example: https://github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/mnist/mnist_deep.py
6.
   - AlexNet: Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks (2012), http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
   - GoogLeNet: C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich: Going Deeper with Convolutions (2015), In Computer Vision and Pattern Recognition (CVPR) [http://arxiv.org/abs/1409.4842]
   - Overfeat: Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., & Lecun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. In International Conference on Learning Representations (ICLR2014), CBLS, April 2014 [http://openreview.net/document/d332e77d-459a-4af8-b3ed-55ba, http://arxiv.org/abs/1312.6229]
   - VGG: Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556 (2014), http://arxiv.org/abs/1409.1556
7. tf-benchmarks GitHub: https://github.com/vykozlov/tf-benchmarks/tree/181004, Docker Hub: https://hub.docker.com/r/vykozlov/tf-benchmarks/tags/ , tags 181004-tf150-gpu and 181004-tf180-gpu
8. Computational resource ForHLR II available at Karlsruhe Institute of Technology, https://wiki.scc.kit.edu/hpc/index.php/Category:ForHLR
9. CIFAR-10 and CIFAR-100 datasets, https://www.cs.toronto.edu/~kriz/cifar.html

2.1.5 Miscelaneous

GPU sharing with MPS

From [1], MPS is a runtime service designed to let multiple MPI processes using CUDA to run concurrently on a single GPU. A CUDA program runs in MPS mode if the MPS control daemon is running on the system.

When CUDA is first initialized in a program, the CUDA driver attempts to connect to the MPS control daemon. If the connection attempt fails, the program continues to run as it normally would without MPS. If however, the connection attempt to the control daemon succeeds, the CUDA driver then requests the daemon to start an MPS server on its behalf.

If there's an MPS server already running, and the user id of that server process matches that of the requesting client process, the control daemon simply notifies the client process of it, which then proceeds to connect to the server. If there's no MPS server already running on the system, the control daemon launches an MPS server with the same user id (UID) as that of the requesting client process. If there's an MPS server already running, but with a different user id than that of the client process, the control daemon requests the existing server to shutdown as soon as all its clients are done. Once the existing server has terminated, the control daemon launches a new server with the user id same as that of the queued client process.

The MPS server creates the shared GPU context, manages its clients, and issues work to the GPU on behalf of its clients. An MPS server can support upto 16 client CUDA contexts at a time. MPS is transparent to CUDA programs, with all the complexity of communication between the client process, the server and the control daemon hidden within the driver binaries.

From [2], the Volta architecture introduced new MPS capabilities. Compared to MPS on pre-Volta GPUs, Volta MPS provides a few key improvements:

- Volta MPS clients submit work directly to the GPU without passing through the MPS server.
- Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.
- Volta MPS supports limited execution resource provisioning for Quality of Service (QoS).

How to use MPS service

Start MPS service:

```
# nvidia-cuda-mps-control -d
```

Stop MPS service:

```
# echo quit | nvidia-cuda-mps-control
```

Testing environment

- HW: Virtual machine on IISAS-GPU cloud, flavor gpu1cpu6 (6 cores, 24GB RAM, 1 GPU Tesla K20m)
- SW: Ubuntu 16.04, latest nvidia driver and CUDA (nvidia driver version 410.48, CUDA 10.0.130)

Test 1. Test with CUDA native sample nbody, without nvidia-cuda-mps service

a. Single CUDA process

```
#./nbody -benchmark -numbodies=512000

number of bodies = 512000
512000 bodies, total time for 10 iterations: 29438.994 ms
= 89.047 billion interactions per second
= 1780.930 single-precision GFLOP/s at 20 flops per interaction
```

b. Two processes at the same time:

```
# ./nbody -benchmark -numbodies=512000

512000 bodies, total time for 10 iterations: 52418.652 ms
= 50.010 billion interactions per second
= 1000.194 single-precision GFLOP/s at 20 flops per interaction
```

Performance of each process reduced to about 1/2 due to parallel execution (overall GPU performance is the same). nvidia-smi shows both processes using GPU at the same time. No GPU conflicts detected.

Test 2. Test with CUDA native sample nbody, with nvidia-cuda-mps service

a. Single CUDA process:

Same performance as without MPS server.

b. Two processes with different user IDs:

The second process is blocked (waiting without termination), it starts computation only the first process is terminated. Performance is the same as without MPS server.

In both case, nvidia-smi indicates nvidia-cuda-mps-server is using GPU, not the nbody process.

c. Two processes with the same user ID:

Both processes will run in parallel. Performance will be evenly divided between processes, like without MPS service (Test 1b).

Test 2 has been repeated with nbody commands placed inside Docker containers instead of baremetal, the same behavior. Note that Docker set user ID at root by default.

Test 3. Test with Docker using mariojmdavid/tensorflow-1.5.0-gpu image, without nvidia-cuda-mps service

Command used in the test:

```
# sudo docker run --runtime=nvidia --rm -ti mariojmdavid/tensorflow-1.5.0-gpu /home/
↪tf-benchmarks/run-bench.sh all
```

a. Single container:

all tests passed.

b. Two containers in parallel:

the second container shows different error messages according to the running benchmarks. Some error message is rather clear like that

```
2018-10-03 13:58:33.135064: W tensorflow/core/framework/op_kernel.cc:1198] Resource␣
↪exhausted: OOM when allocating tensor with shape[1000] and type float on /
↪job:localhost/replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc
```

Some error messages are rather internal

```
2018-10-03 13:51:00.160626: W tensorflow/stream_executor/stream.cc:1901] attempting␣
↪to perform BLAS operation using StreamExecutor without BLAS support
Traceback (most recent call last):
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 221, in <module>
    tf.app.run()
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/platform/app.py",␣
↪line 124, in run
    _sys.exit(main(argv))
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 217, in main
    run_benchmark()
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 210, in run_benchmark
    timing_entries.append(time_tensorflow_run(sess, grad, "Forward-backward"))
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 136, in time_tensorflow_run
    _ = session.run(target_op)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/client/session.py",␣
↪line 895, in run
    run_metadata_ptr)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/client/session.py",␣
↪line 1128, in _run
    feed_dict_tensor, options, run_metadata)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/client/session.py",␣
↪line 1344, in _do_run
    options, run_metadata)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/client/session.py",␣
↪line 1363, in _do_call
    raise type(e)(node_def, op, message)
tensorflow.python.framework.errors_impl.InternalError: Blas GEMM launch failed : a.
↪shape=(128, 9216), b.shape=(9216, 4096), m=128, n=4096, k=9216
         [[Node: affine1/affine1/MatMul = MatMul[T=DT_FLOAT, transpose_a=false,␣
↪transpose_b=false, _device="/job:localhost/replica:0/task:0/device:GPU:0"](Reshape,␣
↪affine1/weights/read)]]
Caused by op u'affine1/affine1/MatMul', defined at:
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 221, in <module>
    tf.app.run()
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/platform/app.py",␣
↪line 124, in run
    _sys.exit(main(argv))
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 217, in main
    run_benchmark()
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 181, in run_benchmark
    last_layer = inference(images)
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 120, in inference
    affn1 = _affine(resh1, 256 * 6 * 6, 4096)
  File "/home/tf-benchmarks/benchmark_alexnet.py", line 76, in _affine
    affine1 = tf.nn.relu_layer(inpOp, kernel, biases, name=name)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/ops/nn_impl.py",␣
↪line 272, in relu_layer
    xw_plus_b = nn_ops.bias_add(math_ops.matmul(x, weights), biases)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/ops/math_ops.py",␣
↪line 2022, in matmul
    a, b, transpose_a=transpose_a, transpose_b=transpose_b, name=name)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/ops/gen_math_ops.py",
↪ line 2516, in _mat_mul
    name=name)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/framework/op_def_
↪library.py", line 787, in _apply_op_helper
    op_def=op_def)
```

(continues on next page)

(continued from previous page)

```
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/framework/ops.py",␣
↪line 3160, in create_op
    op_def=op_def)
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/framework/ops.py",␣
↪line 1625, in __init__
    self._traceback = self._graph._extract_stack()  # pylint: disable=protected-access
InternalError (see above for traceback): Blas GEMM launch failed : a.shape=(128,␣
↪9216), b.shape=(9216, 4096), m=128, n=4096, k=9216
         [[Node: affine1/affine1/MatMul = MatMul[T=DT_FLOAT, transpose_a=false,␣
↪transpose_b=false, _device="/job:localhost/replica:0/task:0/device:GPU:0"](Reshape,␣
↪affine1/weights/read)]]
```

Test 4. Test with Docker using mariojmdavid/tensorflow-1.5.0-gpu image, with nvidia-cuda-mps service

Option "–ipc=host" required for connecting MPS service (full command "sudo docker run –runtime=nvidia –rm –ipc=host -ti mariojmdavid/tensorflow-1.5.0-gpu /home/tf-benchmarks/run-bench.sh all"), see https://github.com/NVIDIA/nvidia-docker/issues/419

Some tests passed but not all

```
/home/tf-benchmarks/run-bench.sh: line 78:   137 Aborted                 (core␣
↪dumped) python ${TFTest[$i]}
```

According to [3], only Tensorflow with version 1.6 and higher can support MPS.

Test 5. Test with Docker using vykozlov/tf-benchmarks:181004-tf180-gpu image, without and with nvidia-cuda-mps service

Tensorflow 1.8.0, GPU version, python 2, command:

```
sudo docker run --ipc=host --runtime=nvidia --rm -ti  vykozlov/tf-benchmarks:181004-
↪tf180-gpu  ./tf-benchmarks.sh all
```

a. without MPS service

all tests passed.

b. with MPS service

Only first part (forward) of each test passed, then the execution terminated (core dumped).

Identified reasons why Tensoflow does not work correctly with MPS

The reasons have been discussed in [3]:

- stream callbacks are not supported on pre-Volta MPS clients. Calling any stream callback APIs will return an error. (from MPS official document [4])
- But CUDA streams are used everywhere in Tensorflow

So Tensorflow will not work with MPS on old (pre-Volta) GPU.

Final remarks:

- Without MPS service, native CUDA samples can be executed in parallel and the GPU performance is divided among processes
- With MPS service, CUDA executions with different user IDs are serialized, one needs to wait until other finishes.
- CUDA processes with the same user ID can be executed in parallel.
- Tensorflow will not work with MPS on old (pre-Volta) GPU.
- Need to test on newer GPU cards (Volta)

References

1. http://manpages.ubuntu.com/manpages/xenial/man1/alt-nvidia-340-cuda-mps-control.1.html
2. https://docs.nvidia.com/deploy/mps/index.html
3. https://github.com/tensorflow/tensorflow/issues/9080
4. https://docs.nvidia.com/deploy/mps/index.html

CHAPTER 3

Indices and tables

- genindex
- search

Index