Machine learning, deep learning, and artificial intelligence have become essential tools for handling and gaining insight from the enormous amounts of data that are being generated via high-performance computing, modern modeling and simulation, and instrument technology.

These methods organize data in a way that is both *systematic* (collected, processed, and stored methodically according to a standard practice) and *semantic* (unambiguous and logical, in both order and representation, so that relationships and biases can be easily explored). Once the data is organized, it can then be analyzed using powerful statistical methods.

HECC Data Science Team: Pilot Projects

HECC offers a range of services for researchers moving into advanced data science, using the latest technologies in statistics-based data analytics, machine learning, and deep learning. For general information, see HECC Data Science Services.

The Data Science Team is currently working with NASA researchers on a number of projects, including:

- Carbon Nanotube Gas Sensor Using Neural Networks (PDF)
- Predicting Composition of Photo Voltaic Cells Using Neural Networks (PDF)

For more information about working with our team, please contact us at dataanalytics@nas.nasa.gov.

Getting Started with Machine Learning

In this section of the Knowledge Base, we provide information that can help you get started with a machine learning environment and short tutorials describing how to use the machine learning tools available on NAS systems. Please check back often, as new articles will become available soon.

Once you have a NAS account, you will have access to software and infrastructure for your machine learning project. If you are a new user, be sure to read through our New User Orientation and complete the first-time login process.

Conda Environments

We provide multiple conda environments that include basic machine learning packages, as well as common image processing and natural language processing packages, for your machine learning projects.

The tables below show currently available conda environments that can be used with nodes containing either x86_64-Based CPUs (Table 1) or aarch64-based CPUs (Table 2). All of the environments include GPU support.

Notes:

- The conda environments contain many of the same software programs, but the package versions may be different. Also, the environments may contain additional packages to the ones listed here. To learn more, see Using Conda Environments for Machine Learning.
- The environments are provided via software modules in HECC system directories. For general information about NAS-provided software modules, see Software Directories and Using Software Modules.

Table 1. Compatible Hardware: Nodes with x86_64-Based CPUs

Environment Name	Packages Included	
horovod	jupyterlab	pillow
	mpi4py	horovod
	pandas	tensorflow
mayavi	mayavi	pandas
	jupyterlab	pillow
	matplotlib	scikit-learn
pyt2_4	dask	astropy
	jupyterlab	pytorch 2.4
	matplotlib	pytorch lightning 2.3.3
	pandas	scikit-learn
pyt2_5	dask	astropy
	jupyterlab	pytorch 2.5.1
	matplotlib	pytorch lightning 2.4
	pandas	scikit-learn

pyt2_6	dask jupyterlab matplotlib pandas	astropy pytorch 2.6 pytorch lightning 2.5 scikit-learn
pyt2_7	dask jupyterlab matplotlib pandas	astropy pytorch 2.7.0+cu118 pytorch lightning 2.5.1.post0 scikit-learn
doppler_cal	dask matplotlib pandas sunpy pillow	astropy pytorch 2.7.1 pytorch lightning 2.5.2 scikit-learn
rpnet	pandas jupyterlab matplotlib rpnet	pillow scikit-learn tensorflow
exoplnt_nas	pandas jupyterlab matplotlib pillow	scikit-learn tensorflow astropy scipy
tf2_15	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.15
tf2_16	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.16
tf2_19	dask pandas jupyterlab matplotlib nltk	pillow scikit-learn sympy tensorflow 2.19
r3_6	jupyterlab r-caret r-dplyr r-essentials r-ggplot2 r-knitr r-markdown r-plyr	r-randomforest r-reticulate r-shiny r-tidyr r-tidyverse r-timedate r-xts r-zoo
r4_2	jupyterlab r-caret r-dplyr r-essentials r-ggplot2 r-knitr r-markdown r-plyr	r-randomforest r-reticulate r-shiny r-tidyr r-tidyverse r-timedate r-xts r-zoo
smartg	dask jupyterlab matplotlib	pandas pillow scikit-learn 1.0.1

dask_mpi	dask dask_mpi pandas jupyterlab	matplotlib pillow scikit-learn
dask_mpi_2025	dask dask_mpi pandas	jupyterlab matplotlib scikit-learn
jupyterlab	jupyterlab	
rapids_23_08	dask pandas jupyterlab matplotlib numba	pillow scikit-learn cudf cuml
ray	ray scikit-learn pandas matplotlib	jupyterlab dask pytorch 2.1.2
ray_openshift	ray pytorch	
spyder	spyder	

Table 2. Compatible Hardware: Nodes with aarch64-based CPUs (Grace Hopper)

Environment Name	Packages Included	
pyt2_4_gh	dask jupyterlab matplotlib pandas	astropy pytorch 2.4.1 pytorch lightning 2.5.0.post0 scikit-learn
pyt2_5_gh	dask jupyterlab matplotlib pandas	astropy pytorch 2.5 pytorch lightning 2.5 scikit-learn
pyt2_6_gh	dask jupyterlab matplotlib pandas	astropy pytorch 2.6 pytorch lightning 2.5 scikit-learn
pyt2_7_gh	dask jupyterlab matplotlib pandas	astropy pytorch 2.7.0.dev20250310+cu126 pytorch lightning 2.5.1.post0 scikit-learn
tf2_18_gh	dask pandas jupyterlab matplotlib nltk	pillow astropy scikit-learn sympy tensorflow 2.18
pyt_keras_gh	dask pandas jupyterlab matplotlib	astropy scikit-learn pytorch 2.5 pytorch lightning 2.5

rapids_24_12	dask pandas jupyterlab matplotlib numba	pillow scikit-learn cudf cuml
mamba_ssm	dask pandas jupyterlab matplotlib	pillow scikit-learn mamba-ssm pytorch 2.4.1

NVIDIA rapids can be found in the rapids24_12 conda environment. PyTorch versions 2.4 and 2.5 can be found in the pyt2_4_gh and pyt2_5_gh conda environments, respectively. PyTorch 2.5 is also available in the pyt_keras_gh conda environment; this environment is also the host for Keras 3.7.

TensorFlow version 2.18 can be found in the tf2_18_gh conda environment. To use this version of TensorFlow across multiple nodes, you might need to set these environment variables in your code before calling TensorFlow, as follows:

```
os.environ["TF_USE_LEGACY_KERAS"] = "1"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

Or, you can set the environment variables at the command line before the code is run.

TensorFlow users who use the Keras API can use the pyt_keras_gh environment with PyTorch 2.5 as the backend. For most TensorFlow code bases, few or no changes to the code will be needed to use this environment. Contact NAS support if you need assistance in adapting TensorFlow code to run in this environment.

Note: Rendering graphics using OpenAI Gym is currently unavailable.

Packages

Miniconda version 4.12.0 is licensed from Anaconda, which directly impacts how environments can be cloned. Some of the previous cloning options cannot be used with the licensed Anaconda environments. For the new cloning instructions, see Managing and Installing Python Packages for Machine Learning.

Working with PBS

All NAS systems utilize the Portable Batch System (PBS) for batch job submission, job monitoring, and job management. To request examples of PBS scripts for Apache Spark and TensorFlow, please contact us at dataanalytics@nas.nasa.gov.

System Information

- Systems and Filesystems
- GPU nodes
- Current system status

Note: When you publish your results, please acknowledge the use of NAS resources.

Additional Resources

- HECC Webinars, including:
  - 2021 NAS Data Science Webinar
  - HECC Data Science Platform
- Machine Learning (Wikipedia)
- Machine Learning for Humans

WARNING: This article was recently updated with information about using conda environments on Grace Hopper nodes. Please note that because these nodes are fundamentally different from the other NAS node types, due to their Arm architecture, conda environments that were created to run on other NAS node types will ***not*** run on the Grace Hopper nodes. Therefore, you must create conda environments explicitly for the Grace Hopper nodes.

We provide multiple conda environments for you to use for your machine learning projects. The environments are available via the conda module in the /swbuild/analytix/tools/modulefiles directory.

All of the available environments include the basic machine learning packages, as well as common image processing and natural language processing packages. You can activate the environments via an interactive session or with a PBS batch job.

For a list of environments we currently provide, along with the packages they include, see Machine Learning: Overview.

Loading the Module and Activating the Environments in Interactive Mode

First, access and load the miniconda module, which provides access to the environment:

```
% module use -a /swbuild/analytix/tools/modulefiles
% module load miniconda3/v4
```

Then, activate one of the environments (where *env_name* is the name of the environment):

- For bash: % source activate *env_name*
- For csh: % source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *env_name*

Loading Conda Environments for Use with Grace Hopper Nodes

Because the Grace Hopper nodes are fundamentally different from other NAS node types, due to their Arm architecture, conda environments that were created to run on other NAS node types will ***not*** run on the Grace Hopper nodes.

You can load a conda environment compatible with Grace Hopper nodes using the gh2 module. To load this module:

```
% module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/gh2
```

For a list of conda environments available for use with the Grace Hopper nodes, see Table 2 in Machine Learning: Overview.

Useful Conda Commands

After loading the module, you will have access to conda commands, including:

% conda info --envs
: Shows available environments.

% conda list -n *env_name*
: Shows installed packages within an environment.

(*env_name*)% conda deactivate
: Deactivates an environment after loading.

For more detailed documentation, see the Conda website.

Using a PBS Batch Job to Activate an Environment

You can activate your machine learning environment, run your program, and deactivate the environment in a PBS script. For example:

```
#!/bin/bash -x
#PBS -l select=1:model=mil_a100:ncpus=64:ngpus=4
#PBS -l place=scatter:excl
#PBS -q gpu_normal

cd $PBS_O_WORKDIR

module purge

# load the module and environment
module use -a /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate env_name

# run python script
python test_cnn.py

# deactivate environment
conda deactivate
```

```
# end of script
```

For more detailed information about running your PBS job on the GPU nodes, see Requesting GPU Resources.

Requesting New or Missing Packages

If you need to use a specific package that is not currently installed in any of the environments, please send an email to support@nas.nasa.gov. In your email, provide the name of the package and request a ticket be opened with the Data Science team. We will help install the package into the existing environments or create a new environment if needed. You can also install your own packages or create your own conda environment.

You have two options to install your own Python packages in our machine learning environment:

- Use the pip tool to install them directly
- Build your own conda environment

Consider the benefits and disadvantages of each method, described below, before choosing which works best for you.

Using the pip Tool to Install Packages

Using the pip tool to install packages is easier than building your own conda environment and takes less space in your home directory.

The packages installed with pip will be applied to all environments that you load and will take precedence over the installed versions of packages within those environments—possibly causing version errors, depending on the differences. If the packages do not show up in other environments, make sure the Python versions in the loaded environment and the base environment match. Packages are Python-version-dependent; therefore, to ensure you can use the packages with the environment you want, load the environment before using pip.

The packages will be installed in the $HOME/.local/lib/... directory.

Complete the following steps to install packages on a PFE:

1. Load the miniconda3 module:

   ```
   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4
   ```

2. Use the pip command to install the packages:

   pfe20 % pip install --user *package_name*

Note: Grace Hopper GPU-compatible versions for TensorFlow are not currently available in Pip. To install PyTorch for the Grace Hopper nodes, you will need to specify the GPU-compatible version you wish to install. For example, to install PyTorch 2.6 by using: pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126.

Building Your Own Conda Environment

Building your own conda environment gives you the control to manage and install your own packages, and they will be less likely to have version errors than the pip-installed packages. The easiest way to create your own environment is to clone an existing conda environment into your own directory, then modify it.

Creating an environment can take up a significant portion of your disk quota, depending on the packages installed. To ensure that you can use your conda environment properly, please familiarize yourself with all the basic conda commands.

Before You Begin: Install a conda Token

The conda environments are supported under an Anaconda site license. Anaconda has also implemented "Block" restrictions for NASA users with nasa.gov email addresses accessing the public repository named repo.anaconda.com. This change is part of NASA's organizational policy to ensure ALL NASA users utilize the appropriate premium repository services, repo.anaconda.cloud.

TIP: If you are receiving a "CondaHTTPError: HTTP 429 TERMS OF SERVICE RATE LIMIT EXCEEDED ..." or a similar error, your usage of the repo.anaconda.com repository has been blocked to comply with NASA policy.

In order to create a conda environment and be in compliance with the NASA policy, we require all users to install a conda token prior to creating an environment. Complete these steps to acquire and install a token:

1. From your NASA email account, send a request for a NASA Ames conda token to dataanalytics@nas.nasa.gov with the following details:
   - Project GID.
   - Full name.
   - Project title.
   - Project principal investigator (PI).
   - Expected project start and end dates.
   - Expected software packages to be used.

   Note: At this time we are not issuing tokens to non-NASA email addresses. You can still use our prebuilt conda environments, and/or send in a support ticket to have any required packages installed. You can find instructions on how to use pip to customize a conda environment in the Using the pip Tool to Install Packages section of this article.

2. If approved, you will receive an email, "Invitation to the NASA Ames organization on Anaconda Nucleus."
   Important: Emails with incomplete information, and/or sent from a non-NASA email address, will not receive a response.
3. Create an Anaconda Nucleus account using the email you received. Click the **Join Organization** button and finish creating an account using the email address that the invitation was sent to.

4. In a couple hours you will receive an email from Anaconda Nucleus with your new token and instructions on how to install it.

To activate your token on a NAS system:

1. Log on to a PFE.
2. Load the HECC Data Science miniconda module as follows:

3. module use -a /swbuild/analytix/tools/modulefiles
   module load miniconda3/v4

4. Run the conda token set command with token you received in the email:
   conda token set *token_string_from_email*
5. Verify the token is linked by using the conda info command. In the channel URLs, there should be:
   https://repo.anaconda.cloud/repo/main/...
   instead of:
   https://repo.anaconda.com/pkgs/main/...
   In addition, you can check whether https://repo.anaconda.cloud/repo/main appears in your .condarc file.

You should now have access to the commercial anaconda.cloud repo. Environments cloned from one of the Data Science conda environments or created from one of the Data Science YAML files will download and install packages from this repo.

If you have questions about this procedure or require assistance, please contact us by sending an email to support@nas.nasa.gov.

Creating a Conda Environment from a Clone

Cloning with the conda clone Command

You can clone existing NAS conda environments using the --clone flag. This environment can then be further customized.

Complete these steps to build your own conda environment by cloning an existing environment:

1. Load the miniconda3 module:

   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4

   TIP: If you don't want to fill up your home directory, you can put the environment in your/nobackup directory. To do this, set the CONDA_ENVS_PATH environment variable to point to a .conda/envs folder in /nobackup, as shown below. The folder is created the first time you run the conda command.
   - For bash: % export CONDA_ENVS_PATH=/nobackup/$USER/.conda/envs
   - For csh: % setenv CONDA_ENVS_PATH /nobackup/$USER/.conda/envs
2. Export the path to the target directory where you want to put the packages for cloning:

   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/*target_directory*

3. Clone an existing conda environment into a new folder. In this example, we create a new environment with no extra packages installed by cloning the base environment:

   pfe20 % conda create --name *my_env* --clone base

Your environment has been created.

Cloning from a YAML File

If you want to create a custom conda environment from a conda environment that is no longer being supported, you can recreate that environment from a YAML file. A list of YAML files for each of the NAS-provided conda environments can be found in the /swbuild/analytix/tools/miniconda3_220407_yml/ and /swbuild/analytix/tools/miniconda3_241216_yml/ directories. For example:

- pyt2_4_gh_env.yml
- pyt2_5_gh_env.yml
- pyt2_6_gh_env.yml
- tf2_18_gh_env.yml
- dask_mpi_env.yml
- horovod_env.yml
- jupyterlab_env.yml
- tf2_12_env.yml
- pyt1_11_env.yml
- pyt1_12_env.yml
- pyt2_0_env.yml

For more information about YAML files, see the conda documentation.

Complete these steps to build a conda environment from a YAML file:

1. Load the miniconda3 module:

   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4

2. Export the path to the target directory where you want to put the packages for installation:

   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/.conda/pkgs

3. Copy the YAML file you want to use to your local directory:

   pfe20 % cp /swbuild/analytix/tools/miniconda3_220407_yml/*filename*.yml *filename*.yml.

4. Open the file, then change the entry for name to a name for your new environment.

   Note: This step is required because keeping the default name may cause conflicts in cloning the environment.

5. At the prompt, enter the conda create command to create the environment:

   pfe20 % conda env create -f *filename*.yml

6. Activate the new conda environment and add the conda environment path to LD_LIBRARY_PATH:
   source activate *my_env*
   conda env config vars set LD_LIBRARY_PATH=$CONDA_PREFIX/lib:$LD_LIBRARY_PATH

   Note: For TensorFlow environments you may also need to update the CUDNN_PATH:

   CUDNN_PATH=$(dirname $(python -c "import nvidia.cudnn;print(nvidia.cudnn.__file__)"))
   conda env config vars set LD_LIBRARY_PATH=$CUDNN_PATH/lib:$LD_LIBRARY_PATH

   Other paths may also need updating. Please contact NAS support for assistance if you have issues in cloning environments from the yml files.

Creating a Fresh Conda Environment

Another option is to create a fresh conda environment. Complete these steps to do so:

1. Load the miniconda3 module:

   pfe20 % module use -a /swbuild/analytix/tools/modulefiles
   pfe20 % module load miniconda3/v4

2. Export the path to the target directory where you want to put the packages for installation:

   pfe20 % export CONDA_PKGS_DIRS=/nobackup/$USER/.conda/pkgs

3. At the prompt, enter the conda create command to create the environment:

   pfe20 % conda create -n *my_env* python=3.x

Using Your Custom Environment

To use your custom environment, follow these steps:

1. Load the miniconda3 module:

   pfe20% module use -a /swbuild/analytix/tools/modulefiles
   pfe20% module load miniconda3/v4

2. Activate your environment:

   (for bash)
   pfe20% source activate *my_env*

   (for csh)
   pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *my_env*

   Note: If you created your environment in your /nobackup directory, use:

   (for bash)
   pfe20% export CONDA_ENVS_PATH=/nobackup/$USER/.conda/envs
   pfe20% source activate *my_env*

   (for csh)
   pfe20% setenv CONDA_ENVS_PATH /nobackup/$USER/.conda/envs
   pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *my_env*

3. Install any other packages you want by using the conda install command:

   *(my_env)*pfe20% conda install *package_name package_name* ...

Your newly installed packages are ready to use. After you finish using your environment, you can deactivate it by using theconda deactivate command.

If you have any issues, please contact us at support@nas.nasa.gov.

For more information about NAS-provided conda environments, see Using Conda Environments for Machine Learning.

The typical Jupyter installation includes more than 125 packages. If any one of these packages has been compromised, any account running Jupyter could become compromised. Keep in mind that even if the initial installation does not include any compromised packages, you might need to install additional packages—which could have been compromised. Or, you might run a Jupyter "notebook," obtained from an internet repository, that contains hidden malware; a Jupyter notebook has exactly the same access to files and other system resources as the user who started the notebook server. While we do not know of any active exploits against Jupyter at this time, this is a serious risk.

WARNING: We recommend you do not use Jupyter, or similar software, if your account has access to any ITAR, proprietary, or non-public data or programs.

Steps for Securely Setting Up Jupyter Notebook

Before you use Jupyter Notebook on NAS systems, you must complete these steps.

Notes:

- You only need to do these steps once (the first time you use Jupyter).
- Some of the command lines are too long to be formatted as one line, so they are broken with a backslash (\).

Before You Begin: Make sure you have set up SSH passthrough.

1. Log into a PFE and load the miniconda3 module from the /swbuild/analytix directory:

   ```
   pfe20% module use -a /swbuild/analytix/tools/modulefiles
   pfe20% module load miniconda3/v4
   ```

2. To access Jupyter commands, activate a conda environment with Jupyter installed. All provided environments other than the base environment will work. For a list of other environments you can use, see Machine Learning: Overview. We suggest using the latest tensorflow or pytorch environment. In this example, we use the jupyterlab environment:

   ```
   (For bash) pfe20% source activate jupyterlab
   (For csh) pfe20% source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh jupyterlab
   ```

3. Set a password for your Jupyter notebooks, using both the jupyter notebook and the jupyter server commands:

   ```
   (jupyterlab)% jupyter notebook password
   (jupyterlab)% jupyter server password
   ```

   Two JSON password files will be written to your $HOME/.jupyter directory. Both commands are needed to ensure that the secure setup is compatible with older environments. In the future, Jupyter will move to using only jupyter server commands, so using both commands will also help guarantee future compatibility. The password used for both commands should be the same to reduce troubleshooting steps, but make sure the password follows the same security standards as other NAS passwords..

   Note: The default "token" authentication is not as secure as a password if you plan to share the notebook with others.

4. Make the Jupyter configuration information visible only to you:

   ```
   (jupyterlab)% chmod 700 $HOME/.jupyter
   ```

5. Protect Jupyter from request forgery by creating a self-signed certificate for your Jupyter HTTPS server:

   ```
   (jupyterlab)% openssl req -x509 -config \
   /swbuild/analytix/tools/jupyter_setup/jupyter_ssl.conf \
   -newkey rsa -out $HOME/.jupyter/jupyter.pem -days 1000
   ```

6. Copy the default NAS configuration files to your private configuration directory. These configuration files allow you to connect to the Jupyter server from your local web browser. The following command line copies two different configuration files (jupyter_server_config.py and jupyter_notebook_config.py) to your .jupyter directory:

   ```
   (jupyterlab)% cp /swbuild/analytix/tools/jupyter_setup/jupyter_*_config.py \
   $HOME/.jupyter/
   ```

7. Verify that the following files are present in your $HOME/.jupyter directory:

   ```
   (jupyterlab)% ls -l $HOME/.jupyter/
   -rw------- 1    103 Jun 17 14:35 jupyter_notebook_config.json
   -rw------- 1    103 Jun 17 14:35 jupyter_server_config.json
   -rw------- 1   1704 Jun 17 14:35 jupyter.key
   -rw------- 1   1155 Jun 17 14:35 jupyter.pem
   -rw------- 1  29875 Jun 17 14:35 jupyter_notebook_config.py
   -rw------- 1  57082 Jun 17 14:35 jupyter_server_config.py
   ```

   There might be additional files in the directory, but it must contain at least the six listed here.

8. Deactivate the environment:

   ```
   (jupyterlab)% conda deactivate
   ```

The setup is done. To clear any environment variables, log out of the PFE and log back in. You can now use Jupyter Notebook by

following the instructions in Using Jupyter Notebook for Machine Learning Development.

IMPORTANT: You must run Jupyter on a node dedicated to you alone. You can do this either by running aninteractive PBS session (for CPU or GPU jobs) or by reserving a dedicated compute node (for CPU jobs only). Running Jupyter on any other system, including an secure front end (SFE), Lou front end (LFE), or a non-dedicated front-end system, is prohibited.

Jupyter Notebook, an open-source application based on Interactive Python (IPython), is a useful tool for interactively exploring science data.

WARNING: You must access Jupyter through a Unix socket; accessing Jupyter using the default web server is prohibited. Alternatively, you can run a notebook from the command line by converting it to a Python script.

Steps for Using Jupyter Notebook for Machine Learning Development

Before You Begin: There are security issues to consider when running these tools on a multi-user system. ;You must complete the steps in Secure Set Up for Using Jupyter on NAS systems before you use Jupyter Notebook for the first time.

Running Jupyter on a secure front end (SFE), Lou front end (LFE), or any other non-dedicated front-end system (such as PFE, AFE, CFE, CGHFE) is prohibited.

These steps describe how to use a PBS interactive job to set up a notebook running on one NVIDIA A100 GPU node. If you do not want to use a GPU node, you can reserve a dedicated compute node and use SSH to connect to the node. You can run Jupyter from any node type.

Notes:

- The steps use the jupyterlab conda environment as an example. For a list of other environments you can use, see Machine Learning: Overview.
- Some of the command lines are too long to be formatted as one line, so they are broken with a backslash (\).

To use Jupyter Notebook:

1. Start a PBS interactive session:

   ```
   cfe01 % qsub -I -q gpu_devel -lselect=1:model=mil_a100:mem=64GB:ncpus=16:ngpus=1 \
   -l walltime=1:00:00
   ```

2. On the compute node, load the miniconda3 module from the /swbuild/analytix directory:

   ```
   r101i2n6 % module use -a /swbuild/analytix/tools/modulefiles
   r101i2n6 % module load miniconda3/v4
   ```

   where r101i2n6 is the name of a node (the one you use will have a different name). Be sure to note the name of the compute node, as you will need it to access the Jupyter notebook on your local machine in step 5.
3. Activate a conda environment to use the proper libraries. For example, to activate the jupyterlab environment:
   - For bash: r101i2n6 % source activate jupyterlab
   - For csh: r101i2n6 % source \ /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh jupyterlab
4. Start a Jupyter server that can be accessed by your local workstation:

   ```
   (jupyterlab)r101i2n6 % jupyter lab --sock /tmp/user.sock --no-browser
   ```

   If you get errors due to missing SSL files, try using the following (on one command line):

   ```
   (jupyterlab)r101i2n6 % jupyter lab --sock /tmp/user.sock --no-browser  \
   --certfile=~/.jupyter/jupyter.pem --keyfile ~/.jupyter/jupyter.key
   ```

   Note: If setting the Jupyter socket to /tmp/user.sock does not allow your browser to connect to the Jupyter server, then try setting the socket to a different directory, such as $HOME/jupyter_socket/user.sock.

5. On your local workstation, open a terminal and type the following (on one command line) to connect by SSH to the compute node:

   ```
   your_local_system% ssh -l username -o "StrictHostKeyChecking ask" \
   -L 8080:/tmp/user.sock -o ProxyJump=sfe,cfe01 r101i2n6
   ```

   Note: You must specify the PFE you used to launch the compute node, and the name of the compute node. In this example, we specify cfe01 and r101i2n6. You might also need to specify the SFE. For *username*, use your NAS username. If you get a hostname error when trying to connect, try the command without including -l *username*.

6. You can now access JupyterLab from a browser on your workstation by connecting to https://localhost:8080/ or https://127.0.0.1:8080 You should see the JupyterLab landing page with a prompt for a password in your browser.

   Note: Currently, users are experiencing issues using some web browsers that do not allow you to use your self-signed certificate with JupyterLab on NAS systems. For this reason, we recommend that you use Firefox to access JupyterLab. If you encounter this issue while using Firefox, complete these steps to resolve the issue:

   1. Go to: **Firefox Settings > Privacy & Security**, then search for "certificates".
   2. Click **View Certificates** then click the **Servers** tab. Click **Add Exception** to add localhost server 127.0.0.1:8080. If the exception already exists, remove it then add it again.

   This should allow the notebook to access port 8080.

7. On the JupyterLab page, log in using the password you set in the Secure Setup procedure. You should now be able to access your files and launch an interactive notebook instance.

For more information about using this software, see Project Jupyter.

NAS currently supports three versions of the R machine learning platform:

- R Versions 3.6 and 4.2 are provided in the r3_6 and r4_2 NAS conda environments, respectively.
- R Version 4.1 and 4.2 are provided as a NAS software package.

The versions in conda come preinstalled with many of the popular R packages used in data science and machine learning, including JupyterLab. Although the version provided as a NAS software package does not include JupyterLab, you can configure JupyterLab to work with this version of R by following the instructions below.

IMPORTANT: You must run Jupyter on a node dedicated to you alone. You can do this either by running aninteractive PBS session (for CPU or GPU jobs) or by reserving a dedicated compute node (for CPU jobs only). Running Jupyter on any other system, including an secure front end (SFE), Lou front end (LFE), or a non-dedicated PFE, is prohibited.

Before You Begin: Complete all the steps in the following articles:

- Secure Setup for Using Jupyter Notebook for Machine Learning Development on NAS Systems
- Using Jupyter Notebook for Machine Learning Development on NAS Systems

Complete the following steps to install R and set up JupyterLab to use it:

1. Load the following NetBSD Packages Collection (pkgsrc) module and start R (version 4.1):

   ```
   pfe25% module load pkgsrc/2022Q1-rome
   pfe25% R
   ```

   Note: A newer version of R (4.2) is available in pkgsrc/2023Q3.

2. Install IRkernel in R:

   ```
   > install.packages("IRkernel")
   ```

3. Run IRkernel in R:

   ```
   > IRkernel::installspec()
   ```

4. Copy the IRkernel directory to your Jupyter data directory. For example, if your IRkernel directory is in$HOME/R/x86_64-redhat-linux-gnu-library/4.1, do:

   ```
   > cp -R $HOME/R/x86_64-redhat-linux-gnu-library/4.1/IRkernel $HOME/.local/share/jupyter
   ```

5. Change the R path (the first "R") in the $HOME/.local/share/jupyter/IRkernel/kernelspec /kernel.json file to: /nasa/pkgsrc/toss4/2022Q1-rome/bin/R *(or other R)*
6. Load Jupyter and run R.

TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking of experiment metrics such as loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more.

This article explains how to quickly get started with TensorBoard. You will need to load and activate a NAS conda environment that includes TensorFlow.

Before You Begin: In order to use the methods described in this article, you must set up SSH Passthrough.

Creating Data for TensorBoard

Within your TensorFlow application there is a piece of code that does the model fitting, as shown below. Here, we create a TensorBoard callback:

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback= tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
```

Add the argument callbacks=[tensorboard_callback]), which accesses the function:

```
model.fit(train_ds,
    epochs=EPOCHS,
    steps_per_epoch=int(image_count/BATCH),
    verbose=verbose,
    callbacks=[tensorboard_callback])
```

Once your application is finished, you will see a directory called logs/.

Using TensorBoard

Complete these steps to use TensorBoard.

1. Start a PBS interactive section:

   ```
   pfe20 % qsub -I -q devel -lselect=1:ncpus=16:model=bro,walltime=1:00:00
   ```

   Note: You don't need a GPU node to look at the TensorBoard data.

2. On the compute node, load the miniconda3 module from the /swbuild/analytix directory:

   ```
   r601i1n1 % module use -a /swbuild/analytix/tools/modulefiles
   r601i1n1 % module load miniconda3/v4
   ```

   where r601i1n1 is the name of a compute node (the one you use will have a different name). Be sure to note the name of the compute node, as you will need it to access the TensorBoard on your local machine in step 5.

3. Activate a conda environment to use the proper libraries. For example, where *env_name* is the name of the environment:
   - For bash:
     r601i1n1 % source activate *env_name*
   - For csh (on one line):
     r601i1n1 % source /swbuild/analytix/tools/miniconda3_220407/bin/activate.csh *env_name*

4. Start a TensorBoard on the compute node:

   ```
   (env_name)r601i1n1 % tensorboard --logdir=logs
   Serving TensorBoard on localhost; to expose to the network,
   use a proxy or pass --bind_all
   TensorBoard 2.2.1 at http://localhost:6006/ (Press CTRL+C to quit)
   ```

5. On your local workstation, open a terminal and type the following (on one command line) to connect via SSH to the compute node:

   ```
   your_local_system% ssh -o "StrictHostKeyChecking ask" -L
   8080:localhost:6006 -o ProxyJump=sfe,pfe20 r601i1n1
   ```

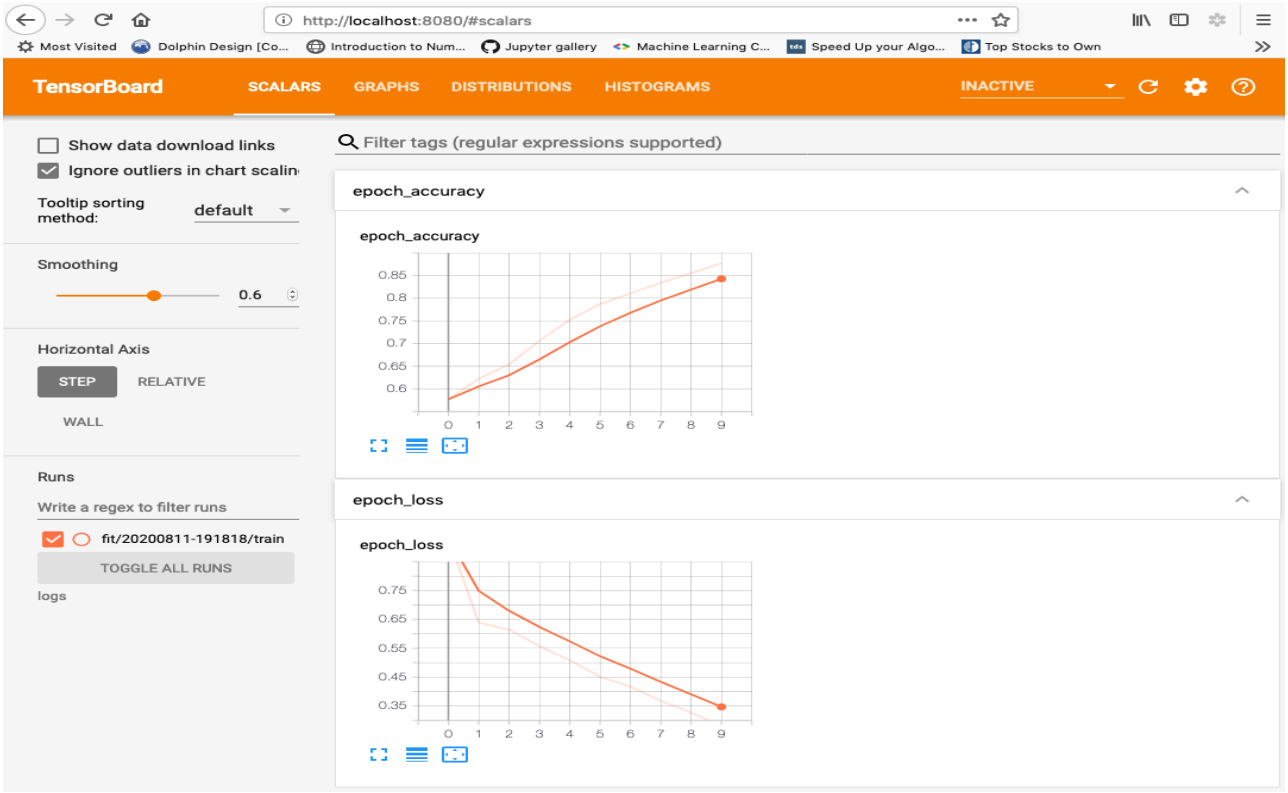
   Note: You must specify the PFE you used to launch the compute node, and the name of the compute node. In this example, we specify pfe20 and r601i1n1.

6. You can now access JupyterLab from a browser on your workstation by connecting to:

   ```
   http://localhost:8080/ or http://127.0.0.1:8080
   ```

   Note: If your browser objects to the self-signed certificate used by this instance of TensorBoard, allow it to continue.

Once you have completed these steps, you will get this interactive screen to create customized measurements and views of your data:



References

For more information, see the following documentation:

- Get Started with TensorBoard
- TensorBoard Tutorial

TensorFlow uses strategies to make distributing neural networks across multiple devices easier. The strategy used to distribute TensorFlow across multiple nodes is multiworkermirroredstrategy, which is slightly more complicated to implement than other strategies like mirroredstrategy.

Using MultiWorkerMirroredStrategy

The basic steps to use multiworkeredmirroredstrategy on HECC resources are:

1. Request multiple nodes in your PBS script.
2. Create an .sh script that can load the miniconda module and run the Python script with proper command line inputs.
3. Set the TF_CONFIG environment variable in the .py file to point to the other nodes that you have requested and chose a port to use for communication.
4. Use tf.data.shard() to make sure that that training data is properly distributed between each node.
5. Use ssh to log into each node individually and run the .sh script, passing the node information to each node to run as a background process.
6. Run the chief worker node after all the other nodes have been started.

For more information on this strategy, see TensorFlow's website: Distributed Training with TensorFlow and Multi-Worker Training with Keras.

Notes:

- This strategy is still considered 'experimental' by TensorFlow, so the usage might not be stable.
- Currently, the method is scalable for CPU nodes, but not GPU nodes.
- Sleeping in the for loop is used to help guarantee that the worker nodes are started and waiting for the chief node. If the chief node starts before the worker nodes, the code might hang.

We have prepared the following three files to demonstrate the method of running multiple CPU nodes over TensorFlow:

PBS Script
: Requests resources and calls an .sh file.

Shell Script (run_vgg.sh)
: Starts the TensorFlow Python code.

Python Script (vgg_dist.py)
: Sample Python code to perform training in TensorFlow.

Each of these files is shown below.

PBS Script

Note: This script has been tested using bash and may require changes in order to work with csh. Please contact Support if you encounter any issues.

```
#!/bin/bash
#PBS -l select=3:ncpus=16:model=bro
#PBS -q normal
#PBS -l walltime=1:00:00
#PBS -j oe
#PBS -o node_3_test.log
#PBS -N multinodetest

#change to nobackup
#cd $PBS_O_WORKDIR
BASE=$(pwd)

echo "The base node is:"
hostname
printf "\n\n"

#get node information from nodefile
NODES=($( cat $PBS_NODEFILE | uniq ))
NUM_OF_NODES=${#NODES[@]}
#worker number for other nodes
C=1

# for each node that's not the current node
for node in ${NODES[@]}
do
  if [[ $node != $(eval hostname) ]]
  then
    # ssh into each node and run the .sh script with node info
    # run in background
    ssh $node "$BASE/run_vgg.sh $C ${NODES[*]}" &
    C=$(($C+1))
```

```
    sleep 2
  fi
done

#run the main worker node
$BASE/run_vgg.sh 0 ${NODES[*]}

echo "Done with PBS"
```

run_vgg.sh Script

```
#!/bin/bash
#load module command
source /usr/local/lib/global.profile

#cd to code location
cd /nobackup/$USER
BASE=/nobackup/$USER/vgg_dist

echo "On node: "
hostname

module purge
module -a use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
source activate conda_environment

echo "loaded conda env"

# Set environments for CPU
setenv KMP_AFFINITY "granularity=fine,verbose,compact,1,0"
setenv KMP_SETTINGS 1
setenv OMP_NUM_THREADS 16     # Sandy has 16 CPUs per node
setenv KMP_BLOCKTIME 30

#run python script with inputs from this .sh script
python $BASE/vgg_dist.py $@
conda deactivate
```

vgg_dist.py Script

Note: Three of the lines in this script are too long to be formatted as one line, so they are broken with a back slash (\).

```
import os
import json
import sys
import time
import pathlib
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, AveragePooling2D, \
MaxPool2D, Flatten, Dense, BatchNormalization, Dropout, Activation
#set seeds for testing
tf.random.set_seed(22)
base_dir = os.path.dirname(os.path.abspath(__file__))
# ---------- Set up TF_CONFIG ---------- #
# load the index and node info from the command line
index = int(sys.argv[1])
verbose = index < 1
nodes=[]
# node names for each node, append a port # to the names
for i in range(2,len(sys.argv)):
    nodes.append(sys.argv[i] + ':2001')
# set TF_CONFIG variable that MultiWorkerMirroredStrategy needs
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': nodes
    },
    'task': {'type': 'worker', 'index': index}
})
print(os.environ['TF_CONFIG'])
'''
verbose = 1
index = 0
'''
#if(verbose):
    #tf.debugging.set_log_device_placement(True)
#set up strategies
```

```
#strategy = tf.distribute.OneDeviceStrategy(device='/device:CPU:0')
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy \
(tf.distribute.experimental.CollectiveCommunication.AUTO)
#strategy = tf.distribute.MirroredStrategy()
EPOCHS = 5
#batch size per worker on each shard of the dataset
BATCH = 256
LEARNING_RATE = 0.001
IMG_WIDTH, IMG_HEIGHT = 224, 224
AUTOTUNE = tf.data.experimental.AUTOTUNE
train_dir = base_dir + '/data/train'
train_dir = pathlib.Path(train_dir)
image_count = len(list(train_dir.glob('*/*.jpg')))
CLASS_NAMES = np.array([item.name for item in train_dir.glob('*') \
if item.name != "LICENSE.txt"])
def get_label(file_path):
    # convert the path to a list of path components
    parts = tf.strings.split(file_path, '/')
    # The second to last is the class-directory
    return parts[-2] == CLASS_NAMES
def decode_img(img):
    # convert the compressed string to a 3D uint8 tensor
    img = tf.image.decode_jpeg(img, channels=3)
    # Use `convert_image_dtype` to convert to floats in the [0,1] range.
    img = tf.image.convert_image_dtype(img, tf.float32)
    # resize the image to the desired size.
    return tf.image.resize(img, [IMG_WIDTH, IMG_HEIGHT])
def process_path(file_path):
    label = get_label(file_path)
    # load the raw data from the file as a string
    img = tf.io.read_file(file_path)
    img = decode_img(img)
    return img, label
def prepare_for_training(ds, cache=True, shuffle_buffer_size=1000, \
num_workers=1, index=0):
    # This is a small dataset, only load it once, and
    # keep it in memory.
    # use `.cache(filename)` to cache preprocessing work for
    # datasets that don't fit in memory.
    ds.shard(num_workers, index)
    if cache:
        if isinstance(cache, str):
            ds = ds.cache(cache)
        else:
            ds = ds.cache()
    ds = ds.shuffle(buffer_size=shuffle_buffer_size)
    # Repeat forever
    ds = ds.repeat()
    ds = ds.batch(BATCH)
    # `prefetch` lets the dataset fetch batches in the background while
    # the model is training.
    ds = ds.prefetch(buffer_size=AUTOTUNE)
    return ds
#list_ds = tf.data.Dataset.list_files(str(train_dir/'*/*'))
#labeled_ds = list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
#train_ds = prepare_for_training(labeled_ds, num_workers=len(nodes), \
# index=index)
# imagegen = ImageDataGenerator(rescale=1./255)
# train = imagegen.flow_from_directory(directory='./data/train', \
# batch_size=BATCH, target_size=(224,224))
# val = imagegen.flow_from_directory(directory='./data/val', batch_size=BATCH, \
#target_size=(224,224))
def build_vgg_a():
    model = keras.Sequential()
    model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3), \
padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
    model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))
    model.add(Flatten())
    # added batch norm to keep weights down
    model.add(Dense(units=4096))
```

```
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(units=4096))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Dense(units=2, activation="softmax"))
    return model
#use the distributed strategy
with strategy.scope():
    list_ds = tf.data.Dataset.list_files(str(train_dir/'*/*'))
    labeled_ds = list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
    print(strategy.num_replicas_in_sync)
    train_ds = prepare_for_training(labeled_ds, \
num_workers=strategy.num_replicas_in_sync, index=index)
    opt = keras.optimizers.Adam(lr = LEARNING_RATE)
    model = build_vgg_a()
    #compile model
    model.compile(optimizer= opt,
        loss=keras.losses.categorical_crossentropy,
        metrics=['accuracy']
        )
if(verbose):
    model.summary()
start = time.time()
model.fit(train_ds,
    epochs=EPOCHS,
    steps_per_epoch=int(image_count/BATCH),
    verbose=verbose
    )
if(verbose):
  print('took ' + str((time.time()-start)/60) + ' minutes'
```

Cabeus, named after a lunar crater in the south polar region of the Moon, is a cluster system that supports HPC and machine learning/artificial intelligence (ML/AI) applications that can benefit from the GPU technology.

GPU Hardware Summary

The node statistics are shown in two groups: nodes based on the x86_64 operating system (OS), and those based on the aarch64 OS.

Cabeus x86_64 GPU Nodes

- 12 racks
- 128 nodes
- 8,192 CPU cores + 1,769,472 double-precision GPU cores
- Total memory: 106 terabytes (65 TB from CPU host memory + 41 TB from GPU memory)
- Theoretical-double precision peak performance: 5.31 Petaflop/s (0.32 Pflop/s from CPUs + 4.99 Pflop/s from GPUs)

Cabeus aarch64 GPU Nodes

- 23 racks
- 538 nodes
- 38,736 CPU cores + 4,545,024 double-precision GPU cores
- Total memory: 258,240 GB (CPU) + 51,648 GB (GPU)
- Theoretical-double precision peak performance: 2,147 Teraflops (CPU) + 17,998 Tflops (GPU)

GPU Nodes by Model

Each GPU compute node includes one CPU socket as the host and one or four GPU cards as the coprocessors. These compute nodes are grouped by the CPU processor models as listed below:

- **model=mil_a100**

  128 nodes total. Each node contains one AMD Milan CPU socket and four NVIDIA Amper A100 GPU cards, where the CPU and GPUs are connected via PCI Express 4.0 bus.

  The Milan CPU socket used in the `mil_a100` nodes are AMD EPYC 7763.

- **model=gh200**

  538 nodes total. Each node contains one NVIDIA GH200 Grace Hopper Superchip, which consists of one Grace CPU (Neoverse V2 core with ARM v9.0-A architecture) and one H100 GPU, connected by a high-bandwidth, memory-coherent NVLink-C2C interconnect.

Cabeus Processor, Memory, and Network Subsystems Statistics

	Node Details	
	mil_a100	**gh200**
Architecture	Apollo 6500 Gen10 Plus	Supermicro SuperServer ARS-111GL-NHR
Total # of Nodes	128	538
Host name	cb[01-05,08-09]n[01-12], cb[06-07]n[01-10], cb[10-12]n[01-08]	cgh[1-3,6-15,17-19,21-23]n[01-26], cgh[4,20]n[01-12], cgh16n[01-20]
	CPU Host	
Processors Model	64-core EPYC 7763	72-core Grace (Neoverse V2)
# of Physical CPU Cores/Node	64	72
CPU-Clock	2.45 GHz	3.465 GHz
Maximum Double Precision Floating Point Operations per cycle per core	16	16
Total CPU Double Precision Flops/Node	2,508 GFlops	3.99 TFlops
Host Memory/Node	512 GB (8 x 64 GB DDR4)	480 GB (LPDDR5X)
	GPU	
GPU Architecture	Ampere	Hopper
Compute Capability**	cc80	cc90
Device Name	NVIDIA A100-SXM4-80GB	NVIDIA GH200 480GB

Clock Rate (Base/Boost)	1,275 MHz/1,410 MHz	(n/a)/1,980 MHz
# of coprocessors/Node	4	1
# of Streaming Multiprocessors (SMs)/coprocessor	108	132
# of Single Precision CUDA Cores/coprocessor	6,912	16,896
# of Double Precision CUDA Cores/coprocessor	3,456	8,448
# of Tensor Cores/coprocessor	432	528
Maximum double precision floating point operations per cycle per CUDA core	2	2
Total GPU Double Precision Flops/Coprocessor (Base/Boost) using Double Precision CUDA Cores	8.8 TFlops /9.7 TFlops	33.4 TFlops
Total GPU Double Precision Flops/Node (Base/Boost) using Double Precision CUDA Cores	35.2 TFlops/38.8 TFlops	33.4 TFlops
L2 Cache	40 MB	60 MB
Global Memory/coprocessor	80 GB (HBM2e)	96 GB (HBM3)
Memory Clock Rate	1,593 MHz	2,619 MHz
Memory Bus Width	5,120 bits	6,144 bits
Memory Bandwidth	2,039 GB/s (if using SXM form factor; 1,935 GB/s if using PCI form factor)	4,000 GB/s
GPU-to-GPU Communication	with NVLink 3.0: 600 GB/s	N/A
CPU-to-GPU Communication	PCIe 4.0 with 31.5 GB/s bandwidth for each direction	NVLink-C2C with 900 GB/s bi-directional
	Inter-node Network	
IB Device on Node	1 HDR 200 Gb 2-port card + 1 GbE 4-port card	NDR 200 Gb 1-port
	Local Disk	
SSD/node***	raw (2 x 1.92 TB SSD drives + 2 x 3.84 TB NVMe SSD) or (6 x 2.93 TB NVMe SSD)	960 GB Samsung NVMe M.2

** The compute capability identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. Do not confuse the compute capability number with the CUDA version.

*** Not yet available for public usage, pending configuration decisions.

For more hardware details, access a GPU node through a PBS session and run one of the following commands:

For CPU info:

```
cat /proc/cpuinfo
```

For host memory info:

```
cat /proc/meminfo
```

Note: For the gh200 nodes with the unified memory feature,
the total memory shown in the output of /proc/meminfo is the sum
of the CPU memory and the GPU memory.

For GPU info:

```
/usr/bin/nvidia-smi -q
```

or load a nvhpc module and run the command pgaccelinfo or nvaccelinfo.

For example, for systems with x86_64 architecture:

```
module avail nvhpc
---------- /nasa/modulefiles/toss4 ----------

nvhpc-nompi/22.3 nvhpc-nompi/23.11 nvhpc/23.5 nvhpc/24.1
nvhpc-nompi/23.5 nvhpc-nompi/24.1 nvhpc/23.7 nvhpc/openmpi-3.1.5
nvhpc-nompi/23.7 nvhpc/22.3 nvhpc/23.11 nvhpc/openmpi-4.0.5

module load nvhpc/23.11
```

For example, for systems with aarch64 architecture:

```
module avail nvhpc
module load nvhpc/24.11

pgaccelinfo
or
nvaccelinfo
```

Introduction

Large language models (LLMs) are an AI/ML innovation that has the potential to enhance scientific discovery. One way to use LLMs in the NAS environment is through the Hugging Face application programming interface (API).

Hugging Face

Hugging Face is a platform for machine learning, comprised of multiple python packages that can help train, deploy, and host open-source and licensed machine learning models.

The Hugging Face Hub hosts over 120,000 pretrained models, datasets, and demo spaces that are all open source and publicly available. To help manage and use these pretrained models at NAS, a shared space is used to access a selection of these models curated by NAS experts. For more information, see Hugging Face and Hugging Face Hub.

To access Hugging Face, we have added the appropriate Hugging Face libraries to the available PyTorch and TensorFlow conda environments. When these conda environments are activated, the following environment variables are set:

```
export HF_HUB_CACHE = /nobackupp27/analytix/hugging_face/hub
export HF_HUB_OFFLINE = 1
```

These variables set the default location where Hugging Face stores and accesses models. You can access shared models and turn off network connectivity, which is required in order to run Hugging Face at NAS.

TIP: You can also access the hub cache from your own conda environment with Hugging Face installed, if you set the above environment variables before loading Hugging Face.

The following open-source models are currently available in the NAS environment:

Purpose	Model
Masked Language Modelling	google-bert/bert-base-uncased google-bert/bert-base-cased FacebookAI/roberta-base microsoft/deberta-v3-base distilbert/distilbert-base-uncased distilbert/distilbert-base-cased
Translation	google-t5/t5-base
Text Generation	openai-community/gpt2 meta-llama/Meta-Llama-3-8B meta-llama/Meta-Llama-3-70B meta-llama/Meta-Llama-3-70B-Instruct google/gemma-7b
Speech Recognition	openai/whisper-large-v3

To load these models, use the transformers library. You can load the models with the pipeline function for easy inference, or you can load them with the trainer function for fine-tuning with your own dataset. For more information on using the transformers library, see:

- Hugging Face Transformers Pipeline
- Hugging Face Transformers Fine Tuning

To request more open source models be added to this repository, send email to our Data Analytics team with the requested model.

The two deep learning frameworks supported on NAS systems, TensorFlow and PyTorch, are capable of training using multiple GPUs on the same node. In TensorFlow, the MirroredStrategy function is the best option for training using multiple GPUs on a single node. In PyTorch, the DataParallel function is the best option.

Using TensorFlow MirroredStrategy

Using the TensorFlow MirroredStrategy framework is relatively straightforward, with only slight modifications needed to existing Python code. Complete these steps to set it up.

1. Create a MirroredStrategy instance:

   ```
   mirrored_strategy = tf.distribute.MirroredStrategy()
   ```

2. Create, compile, and fit the model in the scope of the MirroredStrategy:

   ```
   with mirrored_strategy.scope():
     model = Model()
     model.compile(loss = ..., optimizer = ..., ...)
     model.fit(x_train, y_train, batch_size, epochs, ...)
   ```

   This approach allows TensorFlow to effectively use all of the GPUs attached to the node. If you want to use only a few of the GPUs in a node to train a model, then you must assign the GPUs to the MirroredStrategy, as follows:

   ```
   mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
   ```

   The model will be trained using only the assigned GPUs.

Using PyTorch DataParallel

The basic principle behind PyTorch DataParallel is to allocate the data and model to the GPUs, rather than the CPUs. Complete these steps to set it up.

1. Create the device object:

   ```
   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
   ```

2. Move the model to the device:

   ```
   model = nn_model()
   model = nn.DataParallel(model)
   model.to(device)
   ```

3. Assign the data to the GPUs by applying the `*.cuda()` function to the torch tensor. For example:

   ```
   gpu_data = torch.tensor(data).cuda()
   ```

The model can now be trained on the GPUs using existing PyTorch code.

Multiple GPU Node Training Using PyTorch Lightning

PyTorch Lightning is an API for the PyTorch training loop. PyTorch Lightning also has functionality to train deep learning models using multiple GPU nodes. Setting up this feature on NAS systems can be complex. Complete these steps to set it up.

1. In your Python code, update the PyTorch Lightning trainer for multi-node uses.
   - For older versions of PyTorch Lightning (those in the NAS pyt1_12 and pyt1_13 conda environments) do:

     ```
     trainer = pl.Trainer(gpus=number_gpus_on_each_node,
                          num_nodes=number_of_nodes,
                          strategy="ddp",
                          .... other_flags)
     ```

   - For newer versions of PyTorch Lightning, do:

     ```
     trainer = pl.Trainer(accelerator='gpu',
                          devices=number_gpus_on_each_node,
                          num_nodes=number_of_nodes,
                          strategy="ddp",
                          .... other_flags)
     ```

2. Now, you need to set up the cluster environment so Lightning can do the rest. To do so, you must first collect information about the environment.
   - Collect the node names by using the following bash script:

     ```
     NODES=($( cat $PBS_NODEFILE | uniq ))
     NUM_OF_NODES=${#NODES[@]}
     ```

- Collect the following information: the port on the master node that will be used for communication (MASTER_PORT - this can be any open unused port); the total number of GPUs over all the nodes (WORLD_SIZE); and the rank of the current node (NODE_RANK). You also need to obtain the IP address of the master node (MASTER_ADDR); to do this, ping the master node:

  ```
  ping ${NODES[0]} -c 2 > master_ip.txt
  ```

  The resulting text file will contain the IP address and other information:

  ```
  PING r101i0n6.p4.nas.nasa.gov (10.150.39.109) 56(84) bytes of data.
  64 bytes from r101i0n6.p4.nas.nasa.gov (10.150.39.109): icmp_seq=1 ttl=64 time=0.017 ms
  64 bytes from r101i0n6.p4.nas.nasa.gov (10.150.39.109): icmp_seq=2 ttl=64 time=0.031 ms

  --- r101i0n6.p4.nas.nasa.gov ping statistics ---
  2 packets transmitted, 2 received, 0% packet loss, time 1053ms
  rtt min/avg/max/mdev = 0.017/0.024/0.031/0.007 ms
  ```

  This information must be parsed to collect the IP address, 10.150.39.107. You can use Python to do the parsing. For example:

  ```
  #This code parses ping output text to get host address
  #
  with open('master_ip.txt', 'r') as f:
      ip = f.readlines()

  with open('ip.txt', 'w') as f:
      f.write(ip[0].split('(')[1].split(')')[0])
  ```

  The resulting file contains the IP address, which you can then assign to the MASTER_ADDR environment variable in the next step.

3. Assign the environment variables in your PBS script, using the information you gathered in step 2:

   ```
   export MASTER_PORT=12018
   export MASTER_ADDR=`cat ip.txt`
   export WORLD_SIZE=$TOTAL_NUM_GPS
   ```

   where TOTAL_NUM_GPS = the total number of GPUs across all nodes.

   Note: For GH200 nodes, change the line `export MASTER_ADDR=`cat ip.txt`` to `export MASTER_ADDR=$(eval hostname)`

4. Pass the cluster information to each node and run your Python code on each node. One way to do this is to run a for loop in you PBS script:

   ```
   #the rank of the node
   C=1

   # for each node that's not the current node
   for node in ${NODES[@]}
   do
     if [[ $node != $(eval hostname) ]]
     then
       # ssh into each node and run the .sh script with node info
       ssh $node "$BASE/./setup_workers.sh $MASTER_PORT $MASTER_ADDR $WORLD_SIZE $C" &
       C=$(($C+1))
       sleep 2
     fi
   done
   ```

   where setup_workers.sh is the shell script that will prepare the other nodes and run your Python code.

   Note: For GH200 nodes, change the line `if [[ $node != $(eval hostname) ]]` to `if [[ $node != $(eval hostname --long) ]]`

5. In the shell script, do the following:

   - Change directory to the code location:

     ```
     cd /path_to_code_location
     ```

   - Load the data science conda environments. This example uses the pyt2_0 conda environment:

     ```
     #load module command
           source /usr/local/lib/global.profile

     #load module and activate conda environment
     module purge
     module use -a /swbuild/analytix/tools/modulefiles
     module load miniconda3/v4
     source activate pyt2_0
     ```

   - Set up the cluster environment:

     ```
     export MASTER_PORT=$1
     ```

```
export MASTER_ADDR=$2
export WORLD_SIZE=$3

#setup gpu
export CUDA_VISIBLE_DEVICES=0,1,2,3
#for GH200 setting CUDA_VISIBLE_DEVICES in not necessary when using PyTorch Lightning


# set NCCL to use ip-over-infiniband rather than default RoCE,
#since our infiniband does not have RoCE
export NCCL_IB_DISABLE=1
```

- Run the Python code:

  ```
  NODE_RANK=$4 python your_python_code.py
  ```

6. Back in your PBS script, you need to run the same commands (except for the MASTER_PORT, MASTER_ADDR, and WORLD_SIZE environment variable you've already created):

   ```
   #setup GPUs
   export CUDA_VISIBLE_DEVICES=0,1,2,3

   # set NCCL to use ip-over-infiniband rather than default RoCE,
   #since our infiniband does not have RoCE
   export NCCL_IB_DISABLE=1

   module purge
   module use -a /swbuild/analytix/tools/modulefiles
   module load miniconda3/v4
   source activate pyt2_0

   NODE_RANK=0 python your_python_code.py
   ```

Related Resources

- Distributed training with TensorFlow
- Multi-worker training with Keras
- PyTorch Tutorial: Data Parallelism
- Run on an On-Prem Cluster (Intermediate Training)

Dask is a Pythonic distributed programming framework that enables development of scalable Python code for big data. You can think of Dask as a Python library for parallel computing, and also as a tool to scale computational libraries such as Numpy, Pandas, and Scikit-Learn.

To use Dask, load and activate one of our conda environments for machine learning that has the Dask software package installed.

TIP: For multi-node applications, we recommend using the dask_mpi conda environment.

This article provides a very basic overview of how Dask parallelizes common Python operations. For detailed information, see the Dask website.

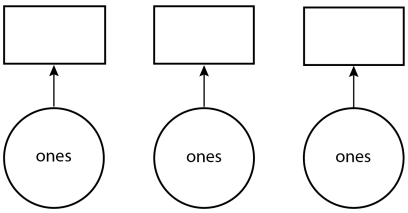
Parallelizing Python Operations

This section provides three different examples of parallelization in Dask.

Example 1

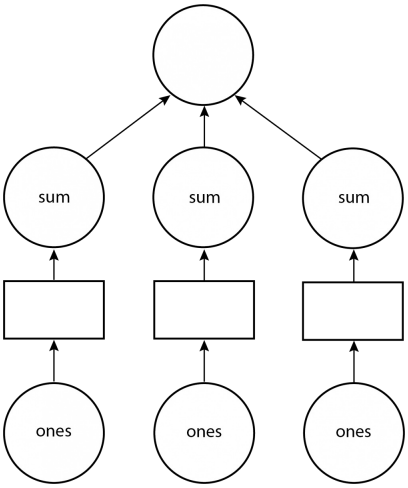
To create a one-dimensional array with entries equal to unity:

```
import dask.array as da
x = da.ones(15, chunks=(5,))
```



To sum the array:

```
x.sum
```

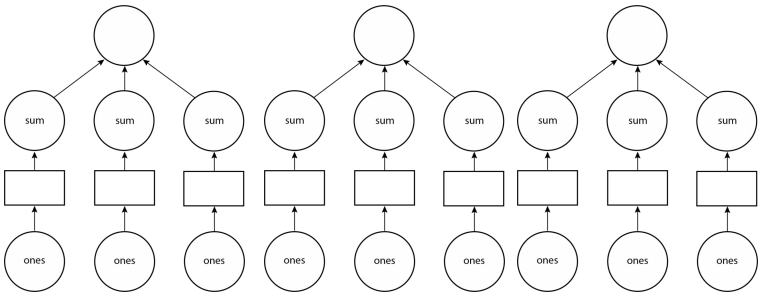


There are three groups, each with five chunks of data (3x5) to be summed in parallel.

Example 2

To sum a two-dimensional array:

```
import dask.array as da
x = da.ones((15,15), chunks=(5,5))
x.sum(axis=0)
```



The 3x3 groups are summed in parallel.

Example 3

To add an array to its transpose:

```
import dask.array as da
x = da.ones((15,15), chunks=(5,5))
x + x.T
```

This is done by obtaining the transposes in parallel before adding the original matrix.

Parallelizing Existing Systems

Dask can scale existing codebases with minor changes. For example, consider the following code:

```
results = []

for x in A:
  for y in B:
    if x < y:
      results.append(f(x,y))
    else :
      results.append(g(x,y))
```

Suppose there is no parallelization for the work at `f` and `g`, because it does not look like a big array or dataframe. You can use the dask.delayed function to parallelize the existing codebase:

```
import dask
results = []

for x in A:
  for y in B:
    if x < y:
      results.append(dask.delayed(f)(x,y))
    else :
      results.append(dask.delayed(g)(x,y))

Results = dask.compute(results)
```

Running Across Multiple Nodes

To run Dask across multiple nodes, load the `mpi-hpe/mpt` module and use the `dask_mpi conda` environment:

```
PBS r429i5n7~> module use /swbuild/analytix/tools/modulefiles
PBS r429i5n7~> module load miniconda3/v4
PBS r429i5n7~> module load mpi-hpe/mpt
PBS r429i5n7~> source activate dask_mpi
```

At the command prompt on the master node, run `dask-mpi`:

```
PBS r429i5n7~> mpirun -np 20 dask-mpi --worker-class distributed.Worker --scheduler-file
~/dask_sch/sched.json & your_python_script
```

where `~/dask_sch` is the path to the Dask scheduler file, `sched.json`.

In your Python set the Dask Client scheduler file path to the path of the Dask scheduler file (`sched.json`) that was created when `dask-mpi` was run at the command prompt:

```
import dask
from dask.distributed import Client

#initialize dask client
path_to_sched = '/dask_sch/sched.json'
client = Client(scheduler_file=path_to_sched)
```

Dask will now run across the nodes, and the scheduler information can be found in the`~/dask_sch/sched.json` directory.

You can use this sample PBS script:

```
##!/bin/bash
#PBS -q devel -l select=25:ncpus=4:model=has
#PBS -l walltime=01:00:00
#PBS -j oe
#PBS -N air_dask_test

#set environment
module use /swbuild/analytix/tools/modulefiles
module load miniconda3/v4
module load mpi-hpe/mpt
source activate dask_mpi

#clear and create directory for dask scheduler
if [ -d dask_sch ]
then
   echo "removing old scheduler directory"
   rm -r dask_sch
   echo "creating new scheduler directory"
   mkdir dask_sch
else
   echo "creating scheduler directory"
   mkdir dask_sch
fi

#run program
mpirun -np 20 dask-mpi --worker-class distributed.Worker --scheduler-file
$(pwd)/dask_sch/sched.json & your_python_script
```

The What-If Tool (WIT) is a helpful visualization tool that aids in analyzing models and datasets. The tool automatically visualizes datasets and allows you to compare different models and see how specific inputs change the output. WIT can visualize both classification and regression models, and gives various visualized statistics like receiver operating characteristic (ROC) curves, confusion matrices, and error calculations, depending on the type of model.

WIT is included in the latest versions of TensorFlow environments installed on NAS systems. To use WIT, you must first set up Jupyter Notebook for your NAS environment.

IMPORTANT: You must run Jupyter on a node dedicated to you alone. You can do this either by running an interactive PBS session (for CPU or GPU jobs) or by reserving a dedicated compute node (for CPU jobs only). Running Jupyter on any other system, including an secure front end (SFE), Lou front end (LFE), or a non-dedicated PFE, is prohibited.

Note: WIT only works on Broadwell or newer nodes.

Starting a Jupyter Notebook

You can access WIT through an interactive Jupyter notebook. To use WIT, first start a Jupyter notebook by using the steps described in Using Jupyter Notebook for Machine Learning Development on NAS Systems—with one change in Step 4:

- In Step 4, instead of loading a JupyterLab server, load a Jupyter notebook server, as WIT does not work in JupyterLab. The command line becomes:

  (*env_name*)r313i0n0 % jupyter notebook --no-browser

The other steps remain the same.

Configuring the Notebook to Use WIT

Once you load a Jupyter notebook in your local browser, you can configure it to use WIT. Complete these steps:

1. In a Python cell of a notebook, load the proper libraries:

   ```
   from witwidget.notebook.visualization import WitConfigBuilder
   from witwidget.notebook.visualization import WitWidget
   ```

2. Enable your dataset to work properly in the visualizer by converting your data to one of the following data structures:
   - tf.train.Example
   - tf.train.SequenceExample protos

   For more information on these data structures, see this documentation on the TensorFlow website.
3. Create the tool by building a WitConfigBuilder, which specifies the data and model to be used for the visualization. The simplest way to use any model is with a custom prediction function:

   ```
   config_builder = WitConfigBuilder(data).set_custom_predict_fn(model)
   ```

   If you want to compare two models, you can add:

   ```
   config_builder=
   WitConfigBuilder(data).set_custom_predict_fn(model).set_custom_compare_predict_fn(model2)
   ```

   where data is a list of tf.train.Example or tf.train.SequenceExample protos, and model is any function used for prediction. The tool directly calls this function for inference with the data provided.
4. Start an interactive window by creating a WitWidget using the following configuration:

   ```
   a = WitWidget(config_builder, height=tool_height_in_px)
   ```

   where config_builder is the WitConfigBuilder object created above, and height is the height (in pixels) of the interactive window opened in the notebook. We recommend setting the height at 1000, depending on the size of your browser window.

Completing these steps will display the interactive window in the notebook when you run the cell. The window will have three tabs at the top:

- **Tab 1:** visualizes all the data.
- **Tab 2:** visualizes the model results.
- **Tab 3:** visualizes data statistics by features.

Additional Resources

- For more information and in-depth examples, see the WIT website .
- To learn more about what you can do with WIT, review the notebooks provided by the developers.

When you work in JupyterLab, you may find it convenient to customize the JupyterLab interface by adding a conda environment to the list of available kernels. To accomplish this, complete the procedures described below.

Note: Some of the command lines below are too long to be formatted as one line, so they are broken with a backslash (\).

Step 1: Activate the Conda Environment

1. Launch an interactive job on a Cabeus compute node, such as mil_a100 or gh200:

   ```
   cfe01 % qsub -I -q gpu_devel -lselect=1:model=mil_a100:mem=64GB:ncpus=16:ngpus=1 \
   -l walltime=1:00:00
   ```

2. Load the appropriate miniconda module:
   - On the A100 nodes:

     ```
     module use -a /swbuild/analytix/tools/modulefiles
     module load miniconda3/v4
     ```

   - On the Grace Hopper nodes:

     ```
     module use -a /swbuild/analytix/tools/modulefiles
     module load miniconda3/gh2
     ```

3. View available conda environments:

   ```
   conda info -envs
   ```

4. Activate the environment you want to add to the JupyterLab kernel list:

   ```
   source activate conda_environment
   ```

   where *conda_environment* is environment name (for example, mamba_ssm).

Step 2: Add the Conda Environment to Available Kernels in JupyterLab

1. Register the conda environment with Jupyter:

   ```
   python -m ipykernel install --user --name=conda_environment \
   --display-name "Python (conda_environment)"
   ```

2. Start JupyterLab using a Unix socket:

   ```
   jupyter lab --sock /tmp/user.sock --no-browser
   ```

3. Open a new terminal and create an SSH tunnel from your local workstation:

   ```
   ssh -l username -o "StrictHostKeyChecking ask" \
   -L 8080:/tmp/user.sock -o ProxyJump=sfe,front_end compute_node
   ```

   where ***username*** is your NAS user name; ***front_end*** is the front end system you submitted from (for example, pfe22); and ***compute_node*** is the actual node name (for example, r101i2n6).

4. Access JupyterLab in your browser:
   Open http://localhost:8080
5. Select your conda kernel in JupyterLab:
   In JupyterLab, open or create a notebook; navigate to **Kernel > Change Kernel**; and select your conda environment (for example, Python ***conda_environment***).

Now you are all set to run JupyterLab with your custom conda environment.