BEST PRACTICES FOR FRAMEWORKS AND SCRIPTS

DG-08865-001_v001 | July 2019

Best Practices
TABLE OF CONTENTS

Chapter 1. Frameworks General Best Practices........................................................... 1
  1.1. Extending Containers.....................................................................................1
  1.2. Datasets And Containers.............................................................................1
Chapter 2. Frameworks Best Practices...................................................................... 3
  2.1. Kaldi..........................................................................................................3
  2.4. Keras And Containerized Frameworks........................................................... 7
    2.4.1. Adding Keras To Containers.................................................................8
    2.4.2. Creating Keras Virtual Python Environment........................................9
    2.4.3. Using Keras Virtual Python Environment With Containerized Frameworks...11
    2.4.4. Working With Containerized VNC Desktop Environment.......................13
Chapter 3. Scripts.................................................................................................. 16
  3.1. DIGITS..................................................................................................... 16
  3.2. NVCaffe....................................................................................................16
  3.3. TensorFlow...............................................................................................21
  3.4. Keras.......................................................................................................21
Chapter 1.
FRAMEWORKS GENERAL BEST PRACTICES

As part of DGX-2, DGX-1, and DGX Station, NVIDIA makes available tuned, optimized, tested, and ready to run nvidia-docker containers for the major deep learning frameworks. These containers are made available via the NGC container registry, nvcr.io, so that you can use them directly or use them as a basis for creating your own containers.

This section presents tips for efficiently using these frameworks. For best practices regarding how to use Docker, see Docker And Container Best Practices. To get started with NVIDIA containers, see Preparing To Use NVIDIA Containers.

1.1. Extending Containers

There are a few general best practices around the containers (the frameworks) in nvcr.io. As mentioned earlier, it’s possible to use one of the containers and build upon it (extend it). By doing this, you are in a sense fixing the new container to a specific framework and container version. This approach works well if you are creating a derivative of a framework or adding some capability that doesn’t exist in the framework or container.

However, if you extend a framework understand that in a few months time, the framework will have likely changed. This is due to the speed of development of deep learning and deep learning frameworks. By extending a specific framework, you have locked the extensions into that particular version of the framework. As the framework evolves, you will have to add your extensions to these new versions, increasing your workload. If possible, it’s highly recommended to not tie the extensions to a specific container but keep them outside. If the extensions are invasive, then it is recommended to discuss the patches with the framework team for inclusion.

1.2. Datasets And Containers

You might be tempted to extend a container by putting a dataset into it. But once again, you are now fixing that container to a specific version. If you go to a new version of
a framework or a new framework you will have to copy the data into it. This makes keeping up with the fast paced development of frameworks very difficult.

**A best practice is to not put datasets in a container. If possible also avoid storing business logic code in a container.** The reason is because by storing datasets or business logic code within a container, it becomes difficult to generalize the usage of the container.

Instead, one can mount file systems into a container that contain only the desired data sets and directories with business logic code to run. Decoupling the container from specific datasets and business logic enables you to easily change containers, such as framework or version of a container, without having to rebuild the container to hold the data or code.

The subsequent sections briefly present some best practices around the major frameworks that are in containers on the container registry (**nvcr.io**). There is also a section that discusses how to use Keras, a very popular high-level abstraction of deep learning frameworks, with some of the containers.
Chapter 2.  
FRAMEWORKS BEST PRACTICES

The following sections present some best practices in regard to running the frameworks that NVIDIA provides as part of the NGC Registry or with the DGX-2, DGX-1, or DGX Station. The examples may refer to older containers but they are just examples to illustrate a point.

2.1. Kaldi

The Kaldi Speech Recognition Toolkit project began in 2009 at Johns Hopkins University with the intent of developing techniques to reduce both the cost and time required to build speech recognition systems. While originally focused on ASR support for new languages and domains, the Kaldi project has steadily grown in size and capabilities, enabling hundreds of researchers to participate in advancing the field. Now the de-facto speech recognition toolkit in the community, Kaldi helps to enable speech services used by millions of people every day.

For information about the optimizations and changes that have been made to Kaldi, see the Deep Learning Frameworks Release Notes.

2.2. NVCaffe

NVCaffe™ can run using the DIGITS container or directly via a command line interface. Also, a Python interface for NVCaffe called pycaffe is available.

When running NVCaffe via the command line or pycaffe use the nvcr.io/nvidia/caffe:17.05 or later container. You can use the run_caffe_mnist.sh script as an example that uses the MNIST data and the LeNet network to perform training via the NVCaffe command line. In the script, the data path is set to /datasets/caffe_mnist. You can modify the path to your desired location. To run, you can use the following commands:

```
./run_caffe_mnist.sh
# or with multiple GPUs use -gpu flag: "-gpu=all" for all gpus or
#    comma list.
```
This script demonstrates how to orchestrate a container, pass external data to the container, and run NVCAFFE training while storing the output in a working directory. Read through the `run_caffe_mnist.sh` script for more details. It is based on the MNIST training example.

The Python interface, pycaffe, is implemented via `import caffe` in a Python script. For examples of using pycaffe and the Python interface, refer to the test scripts.

A description of orchestrating a Python script with Docker containers is described in the `run_tf_cifar10.sh` script.

An interactive session with NVCAFFE can be setup with the following lines in a script:

```bash
DATA=/datasets/caffe_mnist
CAFFEWORKDIR=$HOME/caffe_workdir
mkdir -p $DATA
mkdir -p $CAFFEWORKDIR/mnist
dname=${USER}_caffe
# Orchestrate Docker container with user's privileges
nvidia-docker run -d -t --name=$dname \
   -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \ 
   -e DATA=$DATA -v $DATA:$DATA \ 
   --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \ 
   --w $CAFFEWORKDIR nvcr.io/nvidia/caffe:17.05
# enter interactive session
docker exec -it $dname bash
# After exiting the interactive container session, stop and rm
#   container.
# docker stop $dname && docker rm $dname
```

In the script, the following line has options for Docker to enable proper NVIDIA® Collective Communications Library™ (NCCL) operation for running NVCAFFE with multiple GPUs.

```bash
--shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864
```

You can use the NVCAFFE command line or Python interface within the NVCAFFE container. For example, using the command line would look similar to the following:

```bash
caffe device_query -gpu 0 # query GPU stats. Use "-gpu all" for all gpus
caffe train help # print out help/usage
```

Using a Python interface would look similar to the following:

```python
# start python in container
>>> import caffe
>>> dir(caffe)
```

For more information about NVCAFFE, see NVCAFFE documentation.
2.3. DIGITS

DIGITS is a popular training workflow manager provided by NVIDIA. Using DIGITS, one can manage image data sets and training through an easy to use web interface for the NVCaife, Torch™, and TensorFlow frameworks.

For more information, see NVIDIA DIGITS, DIGITS source and DIGITS documentation.

2.3.1. Setting Up DIGITS

The following directories, files and ports are useful in running the DIGITS container.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS working directory</td>
<td>$HOME/digits_workdir</td>
<td>You must create this directory.</td>
</tr>
<tr>
<td>DIGITS job directory</td>
<td>$HOME/digits_workdir/jobs</td>
<td>You must create this directory.</td>
</tr>
<tr>
<td>DIGITS config file</td>
<td>$HOME/digits_workdir/digits_config_env.sh</td>
<td>Used to pass job directory and log file.</td>
</tr>
<tr>
<td>DIGITS port</td>
<td>5000</td>
<td>Choose a unique port if multi-user.</td>
</tr>
</tbody>
</table>

**Important** It is recommended to specify a list of environment variables in a single file that can be passed to the `nvidia-docker run` command via the `--env-file` option.

The `digits_config_env.sh` script declares the location of the DIGITS job directory and log file. This script is very popular when running DIGITS. Below is an example of defining these two variables in the simple bash script.

```bash
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
```

For more information about configuring DIGITS, see Configuration.md.

2.3.2. Running DIGITS

To run DIGITS, refer to the `run_digits.sh` script. However, if you want to run DIGITS from the command line, there is a simple `nvidia-docker` command that has most of the needed details to effectively run DIGITS.

```bash
$ mkdir -p $HOME/digits_workdir/jobs
$ NV_GPU=0,1 nvidia-docker run --rm -ti --name=${USER}_digits -p 5000:5000 \
```

You will have to create the `jobs` directory if it doesn’t already exist.
This command has several options of which you might need, but you may not need all of them. In the table below is a list of the parameters and their description.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV_GPU</td>
<td>Optional environment variable specifying GPUs available to the container.</td>
</tr>
<tr>
<td>--name</td>
<td>Name to associate with the Docker container instance.</td>
</tr>
<tr>
<td>--rm</td>
<td>Tells Docker to remove the container instance when done.</td>
</tr>
<tr>
<td>-ti</td>
<td>Tells Docker to run in interactive mode and associate tty with the instance.</td>
</tr>
<tr>
<td>-d</td>
<td>Tells Docker to run in daemon mode; no tty, run in background (not shown in the command and not recommended for running with DIGITS).</td>
</tr>
<tr>
<td>-p p1:p2</td>
<td>Tells Docker to map host port p1 to container port p2 for external access. This is useful for pushing DIGITS output through a firewall.</td>
</tr>
<tr>
<td>-u id:gid</td>
<td>Tells Docker to run the container with user id and group id for file permissions.</td>
</tr>
<tr>
<td>-v d1:d2</td>
<td>Tells Docker to map host directory d1 into the container at directory d2.</td>
</tr>
<tr>
<td>--env-file</td>
<td>Tells Docker which environment variables to set for the container.</td>
</tr>
<tr>
<td>--shm-size</td>
<td>This line is a temporary workaround for a DIGITS multi-GPU error you might encounter.</td>
</tr>
<tr>
<td>container</td>
<td>Tells Docker which container instance to run (for example, nvcr.io/nvidia/digits:17.05).</td>
</tr>
<tr>
<td>command</td>
<td>Optional command to run after the container is started. This option is not used in the example.</td>
</tr>
</tbody>
</table>

**Important** This is a very useful option because it allows you to store the data outside of the container.

After DIGITS starts running, open a browser using the IP address and port of the system. For example, the URL would be [http://dgxip:5000/](http://dgxip:5000/). If the port is blocked and an SSH tunnel has been setup (see DGX Best Practices), then you can use the URL [http://localhost:5000/](http://localhost:5000/).
In this example, the datasets are mounted to /digits_data (inside the container) via the option -v /datasets:/digits_data:ro. Outside the container, the datasets reside in /datasets (this can be any path on the system). Inside the container the data is mapped to /digits_data. It is also mounted read-only (ro) with the option :ro.

Important For both paths, it is highly recommended to use the fully qualified path name for outside the container and inside the container.

If you are looking for datasets for learning how to use the system and the containers, there are some standard datasets that can be downloaded via DIGITS.

Included in the DIGITS container is a Python script that can be used to download specific sample datasets. The tool is called digits.download_data. It can be used to download the MNIST data set, the CIFAR-10 dataset, and the CIFAR-100 dataset. You can also use this script in the command line to run DIGITS so that it pulls down the sample dataset. Below is an example for the MNIST dataset.

```
$ nvidia-docker run --rm -ti \
  -u $({id : -u}):$({id : -g}) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  --env-file=${HOME}/digits_workdir/digits_config_env.sh \
  -v /datasets:/digits_data \
  --entrypoint=bash \
  nvcr.io/nvidia/digits:17.05 \
  -c 'python -m digits.download_data mnist /digits_data/digits_mnist'
```

In the download example above, the entry point to the container was overridden to run a bash command to download the dataset (the -c option). You should adjust the datasets paths as needed.

An example of running DIGITS on MNIST data can be found here.

More DIGITS examples can be found here.

### 2.4. Keras And Containerized Frameworks

**Keras** is a popular Python frontend for TensorFlow, Theano, and Microsoft Cognitive Toolkit v.2.x release. Keras implements a high-level neural network API to the frameworks listed. Keras is not included in the containers in nvcr.io because it is evolving so quickly. You can add it to any of the containers if you like, but there are ways to start one of the nvcr.io containers and install Keras during the launch process. This section also provides some scripts for using Keras in a virtual Python environment.

Before jumping into Keras and best practices around how to use it, a good background for Keras is to familiarize yourself with virtualenv and virtualenvwrapper.

When you run Keras, you have to specify the desired framework backend. This can be done using either the $HOME/.keras/keras.json file or by an environment variable KERAS_BACKEND=<backend> where the backend choices are: theano, tensorflow, or cntk. The ability to choose a framework with minimal changes to the Python code makes Keras very popular.
There are several ways to configure Keras to work with containerized frameworks.

**Important** The most reliable approach is to create a container with Keras or install Keras within a container.

Setting up a container with Keras might be preferable for deployed containerized services.

**Important** Another approach that works well in development environments is to setup a virtual Python environment with Keras.

This virtual environment can then be mapped into the container and the Keras code can run against the desired framework backend.

The advantage of decoupling Python environments from the containerized frameworks is that given M containers and N environments instead of having to create $M \times N$ containers, one can just create $M + N$ configurations. The configuration then is the launcher or orchestration script that starts the desired container and activates the Keras Python environment within that container. The disadvantage with such an approach is that one cannot guarantee the compatibility of the virtual Python environment and the framework backend without testing. If the environment is incompatible then one would need to re-create the virtual Python environment from within the container to make it compatible.

### 2.4.1. Adding Keras To Containers

If you choose, you can add Keras to an existing container. Like the frameworks themselves, Keras changes fairly rapidly so you will have to watch for changes in Keras.

There are two good choices for installing Keras into an existing container. Before proceeding with either approach, ensure you are familiar with the Docker And Containers Best Practices guide to understand how to build on existing containers.

The first approach is to use the OS version of Python to install Keras using the Python tool *pip*.

```
# sudo pip install keras
```

Ensure you check the version of Keras that has been installed. This may be an older version to better match the system OS version but it may not be the version you want or need. If that is the case, the next paragraph describes how to install Keras from source code.

The second approach is to build Keras from source. It is recommended that you download one of the releases rather than download from the master branch. A simple step-by-step process is to:

1. Download a release in `.tar.gz` format (you can always use `.zip` if you want).
2. Start up a container with either TensorFlow, Microsoft Cognitive Toolkit v2.x, or Theano.
3. Mount your home directory as a volume in the container (see Using And Mounting File Systems).
4. Navigate into the container and open a shell prompt.
5. Uncompress and untar the Keras release (or unzzip the .zip file).
6. Issue cd into the directory.

```
# cd keras
# sudo python setup.py install
```

If you want to use Keras as part of a virtual Python environment, the next section will explain how you can achieve that.

### 2.4.2. Creating Keras Virtual Python Environment

Before jumping into Keras in a virtual Python environment, it’s always a good idea to review the installation dependencies of Keras. The dependencies are common for data science Python environments, NumPy, SciPy, YAML, and h5py. It can also use cuDNN, but this is already included in the framework containers.

You will be presented with several scripts for running Keras in a virtual Python environment. These scripts are included in the document and provides a better user experience than having to do things by hand.

The venvfns.sh script is a *master script*. It needs to be put in a directory on the system that is accessible for all users, for example, it could be placed in `/usr/share/virtualenvwrapper/`. An administrator needs to put this script in the desired location since it has to be in a directory that every user can access.

The setup_keras.sh script creates a *py-keras* virtual Python environment in `~/.virtualenvs` directory (this is in the user’s home directory). Each user can run the script as:

```
$./setup_keras.sh
```

In this script, you launch the `nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04` container as the local user with your home directory mounted into the container. The salient parts of the script are below:

```
dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \
   -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
   nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
```

**Important** When creating the Keras files, ensure you have the correct privileges set when using the `-u` or `--user` options. The `-d` and `-t` options daemonize the container process. This way the container runs in the background as a daemon service and one can execute code against it.

You can use `docker exec` to execute a snippet of code, a script, or attach interactively to the container. Below is the portion of the script that sets up a Keras virtual Python environment.

```
docker exec -it $dname \
```

---

www.nvidia.com
Best Practices for Frameworks and Scripts
bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
mkvirtualenv py-keras
pip install --upgrade pip
pip install keras --no-deps
pip install PyYaml
# pip install -r /path/to/requirements.txt
pip install numpy
pip install scipy
pip install ipython'

If the list of Python packages is extensive, you can write a requirements.txt file listing those packages and install via:

```bash
pip install -r /path/to/requirements.txt --no-deps
```

This particular line is in the previous command, however, it has been commented out because it was not needed.

The --no-deps option specifies that dependencies of packages should not be installed. It is used here because by default installing Keras will also install Theano or TensorFlow.

Important On a system where you don’t want to install non-optimized frameworks such as Theano and TensorFlow, the --no-deps option prevents this from happening.

Notice the line in the script that begins with bash -c .... This points to the script previously mentioned (venvfn.sh) that needs to be put in a common location on the system. If some time later, more packages are needed, one can relaunch the container and add those new packages as above or interactively. The code snippet below illustrates how to do so interactively.

```bash
dname=${USER}_keras
nvidia-docker run --name=$dname -d -t -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME 
nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
sleep 2  # wait for above container to come up
docker exec -it $dname bash
```

You can now log into the interactive session where you activated the virtual Python environment and install what is needed. The example below installs h5py which is used by Keras for saving models in HDF5 format.

```bash
source ~/.virtualenvs/py-keras/bin/activate
pip install h5py
deactivate
exit
```

If the installation fails because some underlying library is missing, one can attach to the container as root and install the missing library.

The next example illustrates installing the python-dev package which will install Python.h if it is missing.

```bash
$ docker exec -it -u root $dname 
bash -c 'apt-get update && apt-get install -y python-dev # anything else...'
```
The container can be stopped or removed when you are done using the following command.

```
$ docker stop $dname && docker rm $dname
```

### 2.4.3. Using Keras Virtual Python Environment With Containerized Frameworks

The following examples assume that a *py-keras venv* (Python virtual environment) has been created per the instructions in the previous section. All of the scripts for this section can be found in the Scripts section.

The run_kerastf_mnist.sh script demonstrates how the Keras *venv* is enabled and is then used to run the Keras MNIST code `mnist_cnn.py` with the default backend TensorFlow. Standard Keras examples can be found here. Compare the run_kerastf_mnist.sh script to the run_kerasth_mnist.sh that uses Theano. There are primarily two differences:

1. The backend container `nvcr.io/nvidia/theano:17.05` is used instead of `nvcr.io/nvidia/tensorflow:17.05`.
2. In the code launching section of the script, specify `KERAS_BACKEND=theano`. You can run these scripts as:

```
$./run_kerasth_mnist.sh  # Ctrl^C to stop running
$./run_kerastf_mnist.sh
```

The run_kerastf_cifar10.sh script has been modified to accept parameters and demonstrates how one would specify an external data directory for the CIFAR-10 data. The `cifar10_cnn_filesystem.py` script has been modified from the original `cifar10_cnn.py`. The command line example to run this code on a system is the following:

```
$./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
```

The above assumes the storage is mounted on a system at `/datasets/cifar`.

**Important** The key takeaway is that running some code within a container involves setting up a launcher script.

These scripts can be generalized and parameterized for convenience and it is up to the end user or developer to write these scripts for their custom application or their custom workflow.

For example:

1. The parameters in the example script were joined to a temporary variable via the following:

   ```
   function join { local IFS="$1"; shift; echo "$*"; }
   script_args=$(join : "$@")
   ```

2. The parameters were passed to the container via the option:

   ```
   -e script_args="$script_args"
   ```
3. Within the container, these parameters are split and passed through to the computation code by the line:

```
python $cifarcode ${script_args//:/ }
```

4. The external system NFS/storage was passed as read-only to the container via the following option to the launcher script:

```
-v /datasets/cifar:/datasets/cifar:ro
```

and by

```
--datadir=/datasets/cifar
```

The `run_kerasf_cifar10.sh` script can be improved by parsing parameters to generalize the launcher logic and avoid duplication. There are several ways to parse parameters in bash via `getopts` or a custom parser. One can write a non-bash launcher as well as using Python, Perl, or something else.

The `run_keras_script` script implements a high-level parameterized bash launcher. The following examples illustrate how to use it to run the previous MNIST and CIFAR examples above.

```
# running Tensorflow MNIST
./run_keras_script.sh \
   --container=nvcr.io/nvidia/tensorflow:17.05 \
   --script=examples/keras/mnist_cnn.py

# running Theano MNIST
./run_keras_script.sh \
   --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
   --script=examples/keras/mnist_cnn.py

# running Tensorflow Cifar10
./run_keras_script.sh \
   --container=nvcr.io/nvidia/tensorflow:17.05 --backend=tensorflow \
   --datamnt=/datasets/cifar \
   --script=examples/keras/cifar10_cnn_filesystem.py \
   --epochs=3 --datadir=/datasets/cifar

# running Theano Cifar10
./run_keras_script.sh \
   --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
   --datamnt=/datasets/cifar \
   --script=examples/keras/cifar10_cnn_filesystem.py \
   --epochs=3 --datadir=/datasets/cifar
```

**Important** If the code is producing output that needs to be written to a filesystem and persisted after the container stops, that logic needs to be added.

The examples above show containers where their home directory is mounted and is "writeable". This ensures that the code can write the results somewhere within the user's home path. The filesystem paths need to be mounted into the container and specified or passed to the computational code.
These examples serve to illustrate how one goes about orchestrating computational code via Keras or even non-Keras.

**Important** In practice, it is often convenient to launch containers interactively, attach to them interactively, and run code interactively.

During these interactive sessions, it is easier to (automate via helper scripts) debug and develop code. An interactive session might look like the following sequence of commands typed manually into the terminal:

```
# in bash terminal
dname=mykerastf
nvidia-docker run --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
nvcr.io/nvidia/tensorflow:17.05
docker exec -it $dname bash
# now interactively in the container.
source ~/.virtualenvs/py-keras/bin/activate
source ~/.venvsh
enablevenvglobalsitepackages
./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# change some parameters or code in cifar10_cnn_filesystem.py and run again
./run_kerastf_cifar10.sh --aug --epochs=2 --datadir=/datasets/cifar
disablevenvglobalsitepackages
exit # exit interactive session in container
docker stop $dname && docker rm $dname # stop and remove container
```

### 2.4.4. Working With Containerized VNC Desktop Environment

The need for a containerized desktop varies depending on the data center setup. If the systems are set up behind a login node or a head node for an on-premise system, typically data centers will provide a VNC login node or run X Windows on the login node to facilitate running visual tools such as text editors or an IDE (integrated development environment).

For a cloud based system (NGC), there may already be firewalls and security rules available. In this case, you may want to ensure that the proper ports are open for VNC or something similar.

If the system serves as the primary resource for both development and computing, then it is possible to setup a desktop-like environment on it via containerized desktop. The instructions and `Dockerfile` for this can be found [here](#). Notice that these instructions are primarily for DGX-2 and DGX-1, but should work for the DGX Station.

You can download the latest release of the container to the system. The next step is to modify the `Dockerfile` by changing the `FROM` field to be:

```
FROM nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
```
This is not an officially supported container by the NVIDIA DGX product team, in other words, it is not available on nvcr.io and was provided as an example of how to setup a desktop-like environment on a system for convenient development with eclipse or sublime-text (suggestion, try visual studio code which is very like sublime text but free) or any other GUI driven tool.

The build_run_dgxdesk.sh example script is available on the GitHub site to build and run a containerized desktop as shown in the Scripts section. Other systems such as the DGX Station and NGC would follow a similar process.

To connect to the system, you can download a VNC client for your system from RealVnc, or use a web-browser.

=> connect via VNC viewer hostip:5901, default password: vncpassword
=> connect via noVNC HTML5 client: http://hostip:6901/?password=vncpassword

2.5. MXNet

MXNet™ is part of the Apache Incubator project. The MXNet library is portable and can scale to multiple GPUs and multiple machines. MXNet is supported by major public cloud providers including Amazon Web Services (AWS) and Azure Amazon; who have chosen MXNet as its deep learning framework of choice at AWS. It supports multiple languages (C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl, Wolfram Language).

NVIDIA includes a release of MXNet as well. You can read the release notes here. NVIDIA also has a page in the GPU Ready Apps catalog for MXNEMXNetT that explains how you can build it outside of the container registry (nvcr.io). It also presents some test results for MXNet.

To get started with MXNet, the NVIDIA Deep Learning Institute (DLI) has some courses that utilize MXNet.

2.6. PyTorch

PyTorch™ is designed to be deeply integrated with Python. It is used naturally as you would use NumPy, SciPy and scikit-learn, or any other Python extension. You can even write the neural network layers in Python using libraries such as Cython and Numba. Acceleration libraries such as NVIDIA’s cuDNN and NCCL, along with Intel MKL are included to maximize performance.

NVIDIA has a release of PyTorch as well. You can read the release notes here. There is also a good blog that discusses recursive neural networks using PyTorch.

2.7. TensorFlow

An efficient way to run TensorFlow on the GPU system involves setting up a launcher script to run the code using a TensorFlow Docker container. For an example of how to
run CIFAR-10 on multiple GPUs on system using `cifar10_multi_gpu_train.py`, see TensorFlow models.

If you prefer to use a script for running TensorFlow, see the `run_tf_cifar10.sh` script in the Scripts Best Practices section. It is a bash script that you can run on a system. It assumes you have pulled the Docker container from the `nvcr.io` repository to the system. It also assumes you have the CIFAR-10 data stored in `/datasets/cifar` on the system and are mapping it to `/datasets/cifar` in the container. You can also pass arguments to the script such as the following:

```
$.run_tf_cifar10.sh --data_dir=/datasets/cifar --num_gpus=8
```

The details of the `run_tf_cifar10.sh` script parameterization is explained in the Keras section of this document (see Keras And Containerized Frameworks). You can modify the `/datasets/cifar` path in the script for the site specific location to CIFAR data. If the CIFAR-10 dataset for TensorFlow is not available, then run the example with writeable volume `-v /datasets/cifar:/datasets/cifar` (without `ro`) and the data will be downloaded on the first run.

If you want to parallelize the CIFAR-10 training, basic data-parallelization for TensorFlow via Keras can be done as well. Refer to the example `cifar10_cnn_mgpu.py` on GitHub.

A description of orchestrating a Python script with Docker containers is described in the `run_tf_cifar10.sh` script.
Chapter 3.
SCRIPTS

3.1. DIGITS

3.1.1. run_digits.sh

#!/bin/bash
# file: run_digits.sh
mkdir -p $HOME/digits_workdir/jobs

EOF

nvidia-docker run --rm -ti --name=${USER}_digits -p 5000:5000 \
-u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
--env-file=${HOME}/digits_workdir/digits_config_env.sh \
--shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
nvcr.io/nvidia/digits:17.05

3.1.2. digits_config_env.sh

# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log

3.2. NVCaffe

3.2.1. run_caffe_mnist.sh

#!/bin/bash
# file: run_caffe_mnist.sh
basedir="$(cd "$(dirname "$BASH_SOURCE[0]""")" && pwd)"

function join { local IFS="$1"; shift; echo "$*"; }

# arguments to passthrough to caffe such as "-gpu all" or "-gpu 0,1"
script_args="$(join : $@)"

DATA=/datasets/caffe_mnist
CAFFEWORKDIR=$HOME/caffe_workdir

mkdir -p $DATA
mkdir -p $CAFFEWORKDIR/mnist

# Backend storage for Caffe data.
BACKEND="lmdb"

dname=${USER}_caffe

# Orchestrate Docker container with user's privileges
nvidia-docker run -d -t --name=$dname \
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
    -e DATA=$DATA -v $DATA:$DATA \
    -e BACKEND=$BACKEND -e script_args="$script_args" \
    --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
    -w $CAFFEWORKDIR nvcr.io/nvidia/caffe:17.05

sleep 1 # wait for container to come up

download and convert data into lmdb format.
docker exec -it $dname bash -c '
    pushd $DATA
    for fname in train-images-idx3-ubyte train-labels-idx1-ubyte t10k-images-idx3-ubyte t10k-labels-idx1-ubyte ; do
        if [ ! -e $DATA/$fname ]; then
            wget --no-check-certificate http://yann.lecun.com/exdb/mnist/$fname.gz
gunzip $fname.gz
        fi
    done
    popd

    TRAINDIR=$DATA/mnist_train_${BACKEND}
    if [ ! -d "$TRAINDIR" ]; then
        convert_mnist_data \
        $DATA/train-images-idx3-ubyte $DATA/train-labels-idx1-ubyte
        $TRAINDIR --backend=$BACKEND
    fi

    TESTDIR=$DATA/mnist_test_${BACKEND}
    if [ ! -d "$TESTDIR" ]; then
        convert_mnist_data \
        $DATA/t10k-images-idx3-ubyte $DATA/t10k-labels-idx1-ubyte
        $TESTDIR --backend=$BACKEND
    fi

# SETUP CAFFE NETWORK TO TRAIN/TEST/SOLVER

name: "LeNet"

layer {
    name: "mnist"
type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "$DATA/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}

layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "$DATA/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}

layer {
  name: "ip2"
  type: "InnerProduct"
bottom: "ip1"
top: "ip2"
param {
  lr_mult: 1
}
param {
  lr_mult: 2
}
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
EOF

cat <<EOF > ${CAFFEWORKDIR}/mnist/lenet_solver.prototxt
# The train/test net protocol buffer definition
net: "mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The Learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "mnist/lenet"
# solver mode: CPU or GPU
solver_mode: GPU
EOF
3.3. TensorFlow

3.3.1. run_tf_cifar10.sh

#!/bin/bash
# file: run_tf_cifar10.sh
# run example:
# ./run_tf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
# ./run_tf_cifar10.sh --help 2>/dev/null
_basedir="$(cd "$(_dirname "$(_BASH_SOURCE[0]))")" & & pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
cifarcode=${_basedir}/examples/tensorflow/cifar/cifar10_multi_gpu_train.py
# cifarcode=${_basedir}/examples/tensorflow/cifar/cifar10_train.py

function join { local IFS="\"$1\""; shift; echo "$*"; }

script_args=$(join : "$@"

dname=${USER}_tf

nvidia-docker run --name=$dname -d -t -
 --shmem-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
 -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
 -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
 -e cifarcode=$cifarcode -e script_args=""$script_args""
 nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname bash -c 'python $cifarcode ${script_args//:/ }'

docker stop $dname & & docker rm $dname

3.4. Keras

3.4.1. venvfns.sh

#!/bin/bash
# file: venvfns.sh
# functions for virtualenv
[[ "${BASH_SOURCE[0]}" == "${0}" ]] & & \

www.nvidia.com
Best Practices for Frameworks and Scripts
Scripts

```
echo Should be run as: source "$0" && exit 1

enablevenvglobalsitepackages() {
    if ! [ -z ${VIRTUAL_ENV+x} ]; then
        _libpypath=$(dirname $(python -c 
            "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
        if ! [[ "${_libpypath}" == *"$VIRTUAL_ENV"* ]]; then
            return # VIRTUAL_ENV path not in the right place
        fi
        no_global_site_packages_file=${_libpypath}/no-global-site-packages.txt
        if [ -f $no_global_site_packages_file ]; then
            rm $no_global_site_packages_file;
            echo "Enabled global site-packages"
        else
            echo "Global site-packages already enabled"
        fi
    fi
}

disablevenvglobalsitepackages() {
    if ! [ -z ${VIRTUAL_ENV+x} ]; then
        _libpypath=$(dirname $(python -c 
            "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
        if ! [[ "${_libpypath}" == *"$VIRTUAL_ENV"* ]]; then
            return # VIRTUAL_ENV path not in the right place
        fi
        no_global_site_packages_file=${_libpypath}/no-global-site-packages.txt
        if ! [ -f $no_global_site_packages_file ]; then
            touch $no_global_site_packages_file
            echo "Disabled global site-packages"
        else
            echo "Global site-packages were already disabled"
        fi
    fi
}

3.4.2. setup_keras.sh

#!/bin/bash
# file: setup_keras.sh

dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \ 
    nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
docker exec -it -u root $dname \
    bash -c 'apt-get update && apt-get install -y virtualenv virtualenvwrapper'
docker exec -it $dname \
    bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
    mkvirtualenv py-keras
    pip install --upgrade pip
    pip install keras --no-deps
    pip install PyYAML
    pip install numpy
    pip install scipy
    pip install ipython'
docker stop $dname && docker rm $dname
```

3.4.2. setup_keras.sh

#!/bin/bash
# file: setup_keras.sh

dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \ 
    nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
docker exec -it -u root $dname \
    bash -c 'apt-get update && apt-get install -y virtualenv virtualenvwrapper'
docker exec -it $dname \
    bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
    mkvirtualenv py-keras
    pip install --upgrade pip
    pip install keras --no-deps
    pip install PyYAML
    pip install numpy
    pip install scipy
    pip install ipython'
docker stop $dname && docker rm $dname
3.4.3. run_kerastf_mnist.sh

#!/bin/bash
# file: run_kerastf_mnist.sh
_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$basedir
mnistcode=$(basedir)/examples/keras/mnist_cnn.py

dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \\
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \\
    -w $workdir -e mnistcode=$mnistcode \\
    nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \\
    bash -c 'source ~/.virtualenvs/py-keras/bin/activate \\
    source ~/venvfnsh.sh \\
    enablevenvglobalsitepackages \\
    python $mnistcode \\
    disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname

3.4.4. run_kerasth_mnist.sh

#!/bin/bash
# file: run_kerasth_mnist.sh
_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$basedir
mnistcode=$(basedir)/examples/keras/mnist_cnn.py

dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \\
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \\
    -w $workdir -e mnistcode=$mnistcode \\
    nvcr.io/nvidia/theano:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \\
    bash -c 'source ~/.virtualenvs/py-keras/bin/activate \\
    source ~/venvfnsh.sh \\
    enablevenvglobalsitepackages \\
    KERAS_BACKEND=theano python $mnistcode \\
    disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname

3.4.5. run_kerastf_cifar10.sh
#!/bin/bash
# file: run_kerastf_cifar10.sh

# run example:
# ./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
# ./run_kerastf_cifar10.sh --help 2>/dev/null

_basedir="$(cd "$(_basedir=$(dirname "$BASH_SOURCE[0]")")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=_basedir
kerascode=${_basedir}/examples/keras/cifar10_cnn_filesystem.py

function join { local IFS=$"\$"; shift; echo "\$"; }

script_args=$(join : "$@"

dname=${USER}_keras

nvidia-docker run --name=$dname -d -t \
   -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
   -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
   -e kerascode=$kerascode -e script_args=""script_args""
   nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
   bash -c 'source ~/.virtualenvs/py-keras/bin/activate 
   source ~/venvfnsvs.sh 
   enablevenvglobalsitepackages 
   python $kerascode ${script_args//:/ } 
   disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname

3.4.6. run_keras_script

#!/bin/bash
# file: run_keras_script.sh

_basedir="$(cd "$(_basedir=$(dirname "$BASH_SOURCE[0]")")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=_basedir

function join { local IFS=$"\$"; shift; echo "\$"; }

container="nvcr.io/nvidia/tensorflow:17.05"
backend="tensorflow"
script=''

datamnt=''

usage() {
   cat <<EOF
Usage: $0 [ -h|--help ] [ --container=container ] [ --script=script ]
     [ --remain_args ]

   Sets up a keras environment. The keras environment is setup in a
   virtualenv and mapped into the docker container with a chosen
   --backend. Then runs the specified --script.

   --container - Specify desired container. Use "=" equal sign.

   EOF
}
Default: ${container}

--backend - Specify the backend for Keras: tensorflow or theano.
Default: ${backend}

--script - Specify a script. Specify scripts with full or relative
    paths (relative to current working directory). Ex.:
    --script=examples/keras/cifar10_cnn_filesystem.py

--datamnt - Data directory to mount into the container.

--<remain_args> - Additional args to pass through to the script.

-h|--help - Displays this help.

EOF

remain_args=()

while getopts ":h-" arg; do
  case "${arg}" in
  h ) usage
     exit 2
     ;;
  - ) [[ $OPTIND -ge 1 ] && optind=$(expr $OPTIND - 1 ) || optind=$OPTIND
     eval _OPTION="\$optind"
     OPTARG=$(echo $_OPTION | cut -d'=' -f2)
     OPTION=$(echo $_OPTION | cut -d'=' -f1)
     case $OPTION in
      --container ) larguments=yes; container="$OPTARG" ;;
      --script ) larguments=yes; script="$OPTARG" ;;
      --backend ) larguments=yes; backend="$OPTARG" ;;
      --datamnt ) larguments=yes; datamnt="$OPTARG" ;;
      --help ) usage; exit 2 ;;
      --* ) remain_args+=($_OPTION) ;;
     esac
     OPTIND=1
     shift
     ;;
  esac
done

script_args="$(join : ${remain_args[@]})"

dname=${USER}_keras

# formulate -v option for docker if datamnt is not empty.
mntdata=$([ ! -z "$(datamnt/" ] ] && echo "-v $(datamnt):$(datamnt):ro" )
nvidia-docker run --name=$dname -d -t \
    -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
    $mntdata -w $workdir \
    -e backend=$backend -e script=$script -e script_args="$script_args" \
    $container

sleep 1 # wait for container to come up
docker exec -it $dname \
    bash -c 'source ~/.virtualenvs/py-keras/bin/activate 
    source ~/.venvfnas.sh 
    enablevenvglobalsitepackages 
    KERAS_BACKEND=$backend python $script ${script_args//:/ } 
    disablevenvglobalsitepackages'
3.4.7. cifar10_cnn_filesystem.py

#!/usr/bin/env python
# file: cifar10_cnn_filesystem.py
'''
Train a simple deep CNN on the CIFAR10 small images dataset.
'''

from __future__ import print_function
import sys
import os

from argparse import (ArgumentParser, SUPPRESS)
from textwrap import dedent

import numpy as np

# from keras.utils.data_utils import get_file
from keras.utils import to_categorical
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
import keras.layers as KL
from keras import backend as KB
from keras.optimizers import RMSprop

def parser_(desc):
    parser = ArgumentParser(description=dedent(desc))

    parser.add_argument('--epochs', type=int, default=200,
                        help='Number of epochs to run training for.')

    parser.add_argument('--aug', action='store_true', default=False,
                        help='Perform data augmentation on cifar10 set.')

    parser.add_argument('--datadir', default=SUPPRESS,
                        help='Data directory with Cifar10 dataset.')

    args = parser.parse_args()

    return args

def make_model(inshape, num_classes):
    model = Sequential()
    model.add(KL.InputLayer(input_shape=inshape[1:]))
    model.add(KL.Conv2D(32, (3, 3), padding='same'))
    model.add(KL.Activation('relu'))
    model.add(KL.Conv2D(32, (3, 3)))
    model.add(KL.Activation('relu'))
    model.add(KL.MaxPooling2D(pool_size=(2, 2)))
    model.add(KL.Dropout(0.25))
    model.add(KL.Conv2D(64, (3, 3), padding='same'))
    model.add(KL.MaxPooling2D(pool_size=(2, 2)))
    model.add(KL.Conv2D(64, (3, 3)))
    model.add(KL.Activation('relu'))
    model.add(KL.Conv2D(64, (3, 3)))
    model.add(KL.Activation('relu'))
    model.add(KL.Conv2D(64, (3, 3)))
    model.add(KL.Activation('relu'))
    model.add(KL.MaxPooling2D(pool_size=(2, 2)))
    model.add(KL.Dense(num_classes, activation='softmax'))

    return model
model.add(KL.Dropout(0.25))
model.add(KL.Flatten())
model.add(KL.Dense(512))
model.add(KL.Activation('relu'))
model.add(KL.Dropout(0.5))
model.add(KL.Dense(num_classes))
model.add(KL.Activation('softmax'))

return model

def cifar10_load_data(path):
    """Loads CIFAR10 dataset.
    # Returns
    Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
    """
    dirname = 'cifar-10-batches-py'
    # origin = 'http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
    # path = get_file(dirname, origin=origin, untar=True)
    path_ = os.path.join(path, dirname)
    num_train_samples = 50000
    x_train = np.zeros((num_train_samples, 3, 32, 32), dtype='uint8')
    y_train = np.zeros((num_train_samples,), dtype='uint8')
    for i in range(1, 6):
        fpath = os.path.join(path_, 'data_batch_' + str(i))
        data, labels = cifar10.load_batch(fpath)
        x_train[(i - 1) * 10000: i * 10000, :, :, :] = data
        y_train[(i - 1) * 10000: i * 10000] = labels
    fpath = os.path.join(path_, 'test_batch')
    x_test, y_test = cifar10.load_batch(fpath)
    y_train = np.reshape(y_train, (7, 1))
    y_test = np.reshape(y_test, (6, 1))
    if KB.image_data_format() == 'channels_last':
        x_train = x_train.transpose(0, 2, 3, 1)
        x_test = x_test.transpose(0, 2, 3, 1)
    return (x_train, y_train), (x_test, y_test)

def main(argv=None):
    """"
    main.__doc__ = __doc__
    argv = sys.argv if argv is None else sys.argv.extend(argv)
    desc = main.__doc__
    # CLI parser
    args = parser_(desc)
    batch_size = 32
    num_classes = 10
    epochs = args.epochs
    data_augmentation = args.aug
    datadir = getattr(args, 'datadir', None)

    # The data, shuffled and split between train and test sets:
    (x_train, y_train, (x_test, y_test) = cifar10_load_data(datadir) \
        if datadir is not None else cifar10.load_data()"""
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

callbacks = None

print(x_train.shape, 'train shape')
model = make_model(x_train.shape, num_classes)
print(model.summary())

# initiate RMSprop optimizer
opt = RMSprop(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

nsamples = x_train.shape[0]
steps_per_epoch = nsamples // batch_size

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True,
              callbacks=callbacks)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
datagen = ImageDataGenerator(
    # set input mean to 0 over the dataset
    featurewise_center=False,
    samplewise_center=False,  # set each sample mean to 0
    # divide inputs by std of the dataset
    featurewise_std_normalization=False,
    # divide each input by its std
    samplewise_std_normalization=False,
    zca_whitening=False,  # apply ZCA whitening
    # randomly rotate images in the range (degrees, 0 to 180)
    rotation_range=0,
    # randomly shift images horizontally (fraction of total width)
    width_shift_range=0.1,
    # randomly shift images vertically (fraction of total height)
    height_shift_range=0.1,
    horizontal_flip=True,  # randomly flip images
    vertical_flip=False)  # randomly flip images

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(x_train, y_train,
    batch_size=batch_size),
    steps_per_epoch=steps_per_epoch,
    epochs=epochs,
    validation_data=(x_test, y_test),
    callbacks=callbacks)

if __name__ == '__main__':
    main()
Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, DGX, DGX-1, DGX-2, and DGX Station are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2019 NVIDIA Corporation. All rights reserved.

www.nvidia.com