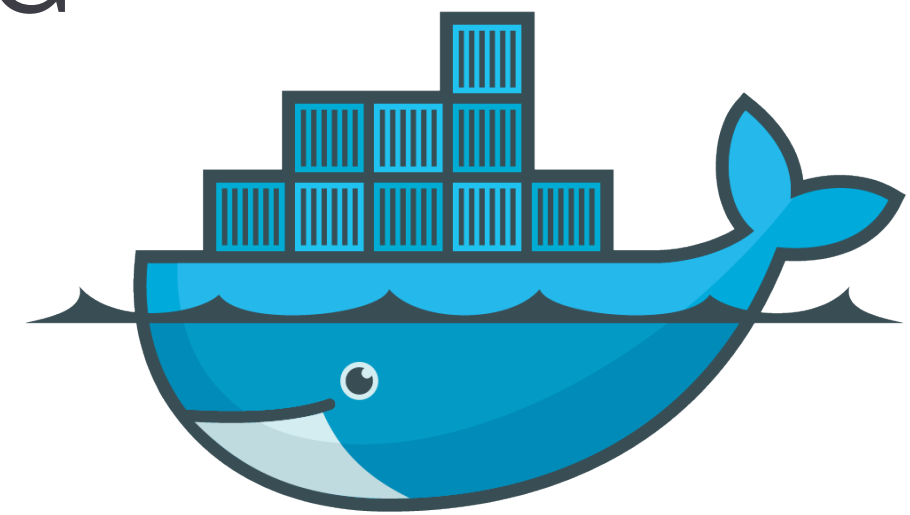
HANDS-ON WITH DOCKER FOR DEEP LEARNING

Taming the whale

01/12/2016



docker

Marc-André Gardner

Université Laval

Why Docker?

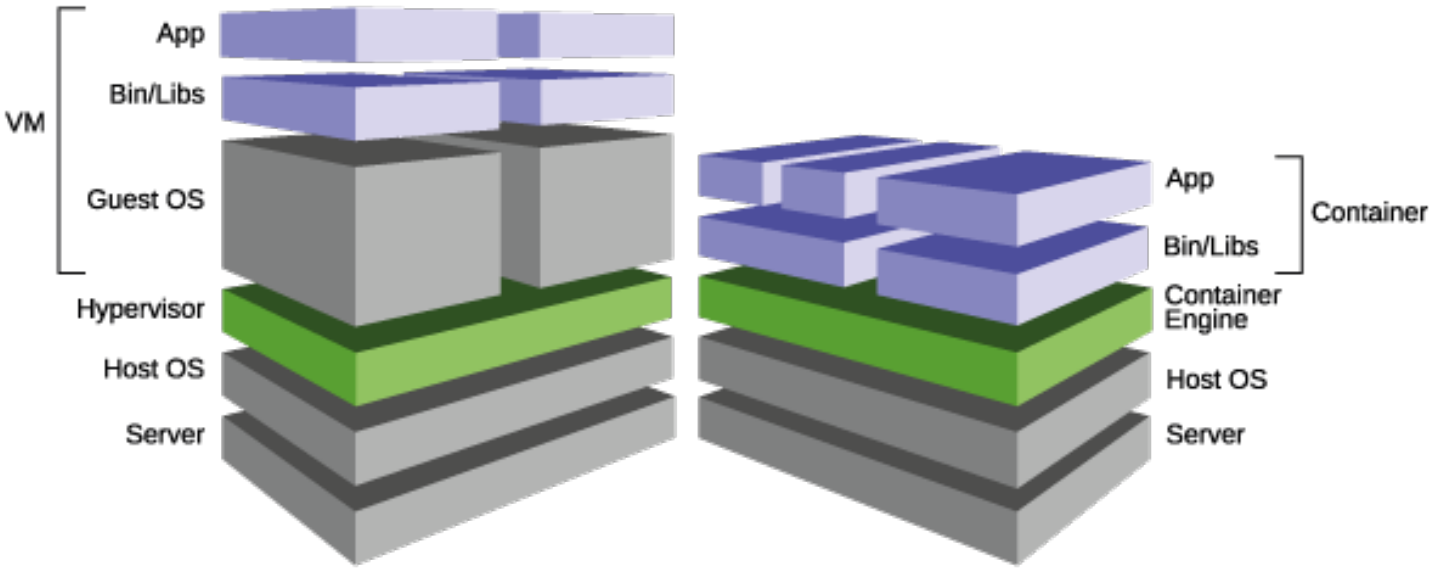
Let's play with deep learning!

- Ok, let's try Tensorflow!
  - But not r0.12! I need r0.10!
  - Ok but r0.10 is incompatible with cuDNN R5, I need cuDNN r3!
  - Hum right but then I cannot use CUDA 8.0
  - But my brand new GeForce 1080 needs that driver version!
- You know what, let's start with this Caffe model instead.
  - Fine, but now I need Boost 1.55.
  - And protobuf, hdf5lib and lmdb
  - And glog
  - What is glog?
  - Ok everything is installed, can I *finally* run something?
  - Not so fast, the code you want to use is built on an old, never released, caffe dev branch
  - (3 hours browsing the Internet, struggling to retrieve this branch)
  - Surprise! This branch does not support cuDNN R5 either!
- Hmm what about using Torch instead? Let's install it
  - Ok, first I must install Lua
  - Damn, its OpenBLAS is conflicting with the one I installed for numpy earlier
  - Wait, I have built the Torch executables on another similar workstation.
  - HAHA! Your graphics cards do not share the exact same streaming multiprocessors version (whatever that means)! Now everything crash...

After a few hours like that...

What is Docker?

- Docker allows the deployment of *software containers*
- Each container is *standalone*, and has its own:
  - Filesystem
  - Librairies
  - Configuration files
- Each container is also *isolated* from the others and the host
- The only shared part is the kernel
- Can be seen as a high-level virtual machine



What is useful with Docker?

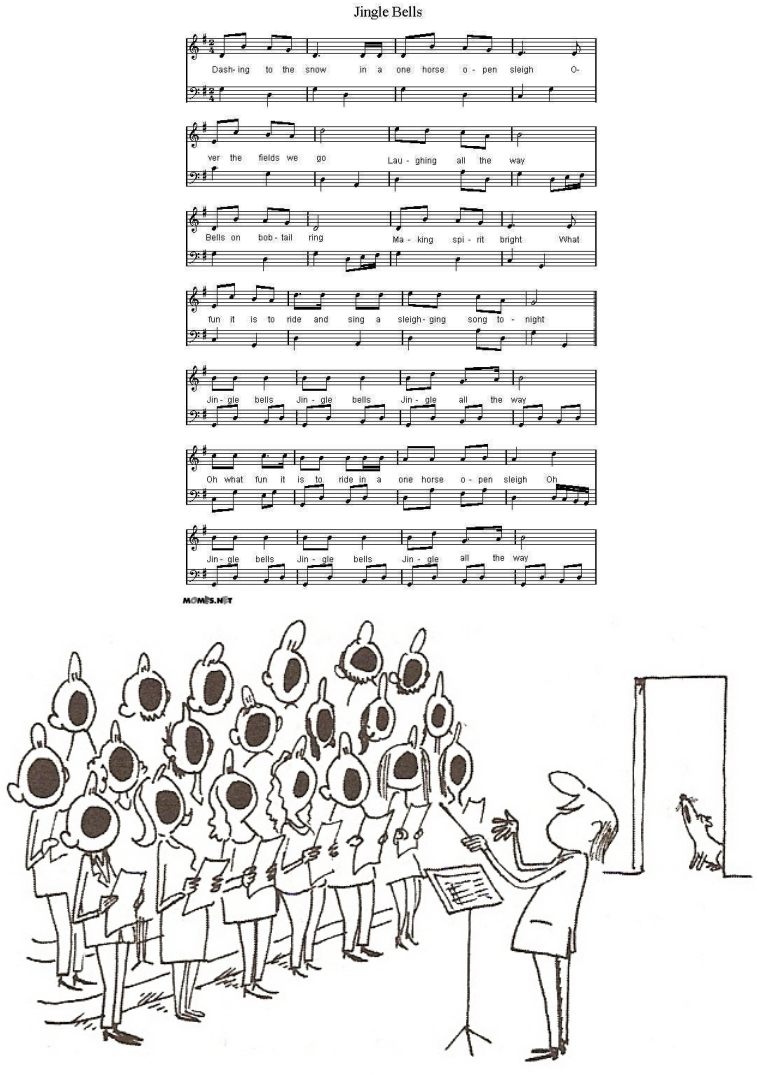
- Docker allow us to see the whole system as a code repository:
  - You can *commit* any change you made to the system
  - If there is an error, you can *revert* them
    - Also, this error will not affect the host system
  - If you want to run the *exact same* system configuration after 6 months without using it, you can
  - You can *bundle* your system, and then:
    - You can *backup it*
    - You can *transfer it* to another computer
    - You can *share it* with others
  - You can retrieve actual systems from online repositories
  - It is *your* system: in the Docker container, you are the indisputed master!
  - You can run more than one system at the same time
- Since the only shared part is the Linux kernel, you may run different distributions (Ubuntu, Fedora, Centos, etc.) on the same machine at the same time!

What Docker is not?

- Docker does not ease the librairies installation process
  - Except for the fact that you may always take one step back if you made a mistake
- Docker is not a *data* backup solution: keep your datasets and your results safe!

Images vs Containers

- An image is like a music score
- A container is what is *executing* this score (namely, an orchestra)
- We can copy the score and give it to another orchestra
- In the Docker world, we may also pause the orchestra, record it (so to make a new score) or terminate it at any moment.
  - Do not try this in real life



Images vs Containers

- A Docker **image** contains a *filesystem state*
- You can (and should) **tag** an image
  - Multiple tags can be added
- A Docker **container** contains a *running system state*
- A **Dockerfile** is a text file containing the instructions needed to *build* a Docker

Docker images

- An image is identified by its image ID, like `4201baf875aa`
- One can assign a repository and a **tag** to an image:
  `$ docker tag 4201baf875aa ubuntu-cudnn:caffe`
  - The name must only contains lowercase letters, digits, _ and -
  - Usually, an image name is made of two parts, separated by a colon, like “repository:tag”
- An image can be loaded from a local file:
  `$ docker load -i /path/to/my/imagefile.tar`
- Conversely, it can be saved to a local file:
  `$ docker save -o /path/to/my/imagefile.tar imageid`
- An image can also be pulled from a remote server:
  `$ docker pull repository:tag`
- You can view the currently loaded images plus some info about them:

```
$ docker images
REPOSITORY    TAG    IMAGE ID      CREATED  SIZE
ubuntu-cudnn  caffe  4201baf875aa  6 weeks  4.874 GB
```

Docker containers

- Once we have an image, we can execute it in a **container**:

  ```
  $ docker run IMAGE COMMAND
  ```

- Usually, we just want to run a terminal in our image so we can start our task ourselves:

  ```
  $ docker run IMAGE /bin/bash
  ```

- Hey! Nothing happened, the command just returns!
  - You have to ask Docker to behave like a TTY, or the shell will think that it is parsing a file and exit after reaching EOF

    ```
    $ docker run -it IMAGE /bin/bash
    ```

- You may list the containers using:

```
$ docker ps -a
CONTAINER ID IMAGE  COMMAND      CREATED         STATUS     NAMES
fc655d9d3f6e ubuntu "/bin/bash" 3 seconds ago   Up 2 sec   dreamy_dijkstra
37a636090f3f ubuntu "/bin/bash" 3 minutes ago  Exited (0) awesome_poitras
```

- A container remains even after being closed!
  - To avoid that, use the -rm option:

    ```
    $ docker run -it --rm IMAGE /bin/bash
    ```

More fun with containers

- A container can be **exported**:

  ```
  $ docker export -o /path/to/my/file.tar CONTAINER
  ```

  - Where CONTAINER is either the container ID or the name of the container
  - **Not to be confounded with** saving an image!
- The produced tarball may then be imported

  ```
  $ docker import /path/to/my/file.tar
  ```
- You can pause an entire container at any moment:

  ```
  $ docker pause CONTAINER
  $ docker unpause CONTAINER
  ```
- The changes made in a container are *temporary*; when you exit, the underlying image stays unaffected
  - If an orchestra ruins a song, it does not destroy the song itself!
- But if you *want* your modifications to be permanent:

  ```
  $ docker commit -m MESSAGE CONTAINERID [IMAGEID]
  ```
- This will create a new image

Inside a container

- Starting a container with `/bin/bash` as executable target gives you a prompt:

```
$ docker run -it --rm ubuntu /bin/bash
root@87132209cb86:/#
```

- Wait, I have root access?
- Yes, but only in the context of this very container; you cannot access to anything outside it!
  - You cannot even see the other processes running on the host

```
root@87132209cb86:/# ps aux
USER  PID %CPU %MEM VSZ    RSS   STAT START  COMMAND
root  1   0.0  0.0  18248  3220  Ss   21:23  /bin/bash
root  15  0.0  0.0  34424  2824  R+   21:32  ps aux
```

- Now you may install everything you need, as a super-user!
- And if you break something?
  - Just exit and restart the container
  - As long as you did not *commit* anything, the changes you made are temporary

Peaking outside: using Docker volumes

- The isolation of each container is great, but what if I want to access outside files?
- One solution could be to put and commit them in the image itself
  - Not very interesting for big datasets (use a lot of space)
  - Not interesting either for code, as you have to make an update from the code repo each time you are starting a new container...
- Another better way: Docker **volumes**
- Use the option `-v path:mountpoint:permissions`
  - `path` is the path on your host system you want to share
  - `mountpoint` is the path of the mountpoint on the container. If it does not exist in the image, Docker will create it.
  - `permissions` is either `ro` (read-only) or `rw` (read and write)

```
$ docker run -it --rm -v $HOME/mycode:/mnt/code:ro ubuntu /bin/bash
root@87132209cb86:/# ls /mnt/code
createdataset.py  deepsn_batch.py  observeLoss.lua train.py
```

- You can mount multiple volumes (provide many `-v` arguments)

Using a GPU

- We would like to use GPUs inside Docker containers.
- In itself, it is not hard :

  ```
  $ docker run --devices=/dev/nvidia*:/dev/nvidia* ...
  ```

- But then you must install everything in your image
- Since the nVidia driver and CUDA are specific to each GPU architecture, we are in for some fun...
- Fortunately, nVidia has our back covered with `nvidia-docker`:

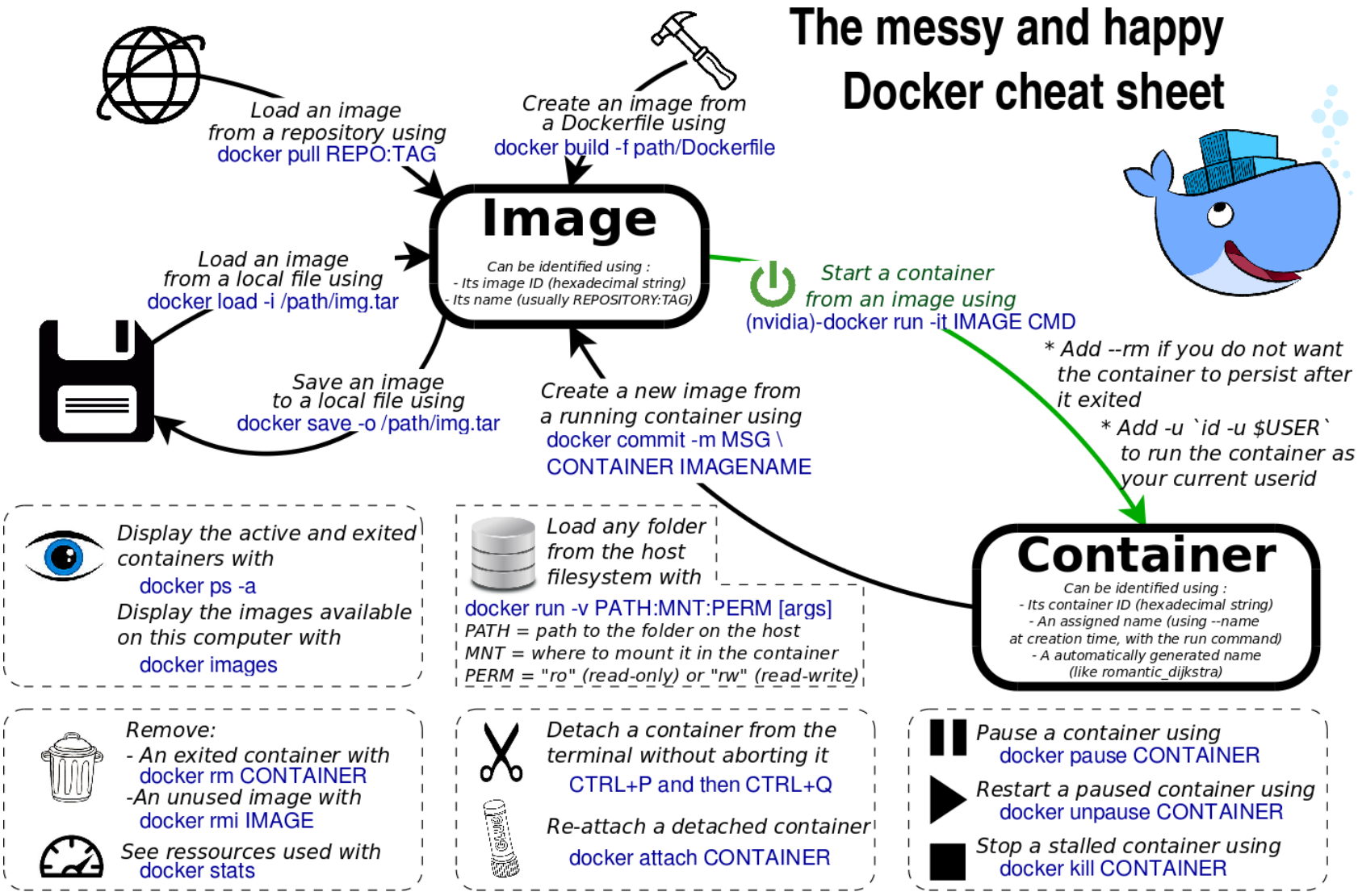
  ```
  $ nvidia-docker run -it --rm IMAGE /bin/bash
  ```

- No need to use nvidia-docker anywhere else (though it works)

Other useful tools

- `docker rm` deletes a *container* (if it is running, you must explicitly use the `-force` flag to kill it)
- `docker rmi` deletes an *image*
- You can **attach** yourself to any running container, just use:
  `$ docker attach CONTAINERID`
- To **detach** yourself without aborting the execution of the container, press Ctrl-p and then Ctrl-q
- If you *create* files on a volume, you will notice that they are all owned by root
  - It is not a huge problem since they are still readable and you may delete them from a container, but it is annoying
  - To avoid that, you may pass a userid to docker-run:
    ```
    $ docker run -u `id -u $USER` -it --rm ubuntu /bin/bash
    I have no name!@f7cb7055a581:/$
    ```
  - You get a weird prompt, but now your user on the host owns the file created by the container
- `docker stats` and `docker top` are useful to remotely check the status and resource usage of all containers

Cheat sheet


The messy and happy
Docker cheat sheet
Load an image
from a repository using
docker pull REPO:TAG
Create an image from
a Dockerfile using
docker build -f path/Dockerfile
Image
Can be identified using :
- Its image ID (hexadecimal string)
- Its name (usually REPOSITORY:TAG)
Load an image
from a local file using
docker load -i /path/img.tar
Save an image
to a local file using
docker save -o /path/img.tar
Create a new image from
a running container using
docker commit -m MSG \
CONTAINER IMAGENAME
Start a container
from an image using
(nvidia)-docker run -it IMAGE CMD
* Add --rm if you do not want
the container to persist after
it exited
* Add -u `id -u $USER`
to run the container as
your current userid
Container
Can be identified using :
- Its container ID (hexadecimal string)
- An assigned name (using --name
at creation time, with the run command)
- A automatically generated name
(like romantic_dijkstra)
Display the active and exited
containers with
docker ps -a
Display the images available
on this computer with
docker images
Load any folder
from the host
filesystem with
docker run -v PATH:MNT:PERM [args]
PATH = path to the folder on the host
MNT = where to mount it in the container
PERM = "ro" (read-only) or "rw" (read-write)
Remove:
- An exited container with
docker rm CONTAINER
-An unused image with
docker rmi IMAGE
See ressources used with
docker stats
Detach a container from the
terminal without aborting it
CTRL+P and then CTRL+Q
Re-attach a detached container
docker attach CONTAINER
Pause a container using
docker pause CONTAINER
Restart a paused container using
docker unpause CONTAINER
Stop a stalled container using
docker kill CONTAINER