Deep Learning Projects with Docker Containers

Seri Lee*

Seoul National University
sally20921@snu.ac.kr

January 12, 2021

I. Why Containers?

Containers are lightweight software packages that run in isolation on the host computing environment. Using containerized environments for research is helpful because they are easy to use, reproducible, and keep primary system clean. You can easily set up a full functioning multi-container application using Docker Compose. They also allow you to set up ready-to-go Jupyter notebooks with Tensorboard tacking with one command. In this post, I will describe how to set up and deploy containers for deep learning in your system, and how to extend basic images for a specific task.

II. Starting with Containers

We will now create two containers-one for Jupyter Notebook and another for Tensorboard. We start with the Tensorboard container because it is very small and easy to understand. Here is the full Dockerfile for it:

```
FROM python:3.8-slim-buster
RUN pip3 install tensorboard
```

You can save this file as `tensorboard.Dockerfile`, build the container and run it interatively.

```
docker build -t dl-tensorboard:latest -f tensorboard.Dockerfile .
docker run --rm -it dl-tensorboard:latest /bin/bash
```

and the `tensorboard` will be available to run. But it will not run on its own, it needs a directory where the data is stored.

The next part is building an image with the Jupyter Notebook. First of all, we need a GPU support in our containers. Luckily, NVIDIA got us covered with their 'Container Toolkit'. Another very useful project from NVIDIA is a set of pre-built images with CUDA installed. With these images, you don't need to do a thing related to CUDA setup. NIVDIA provides three flavors of their images-`base`, `runtime`, and `devel`. Our option is runtime because it includes cuDNN.

The next interesting point is that we are going to perform a multi-stage build of our image. We will have a base image that encapsulates all the logic related to setting things up, and the final image that adds finishing touches depending on the task in hand. This logic will allow us to have

*A thank you or further information

a base image resting in the Docker cache so there will be no need to download PyTorch everytime when we add a new package.

Since we use NVIDIA CUDA image as a base, we will need to install Python and some required tools. Also, due to the known issue, it is not possible to run a Notebook binary in the container as-is. We need to add the `--init` flag when running the container.

Finally, we can have a look at the actual Dockerfile for the Jupyter Notebook image:

```
# NVIDIA CUDA image as a base
# We also mark this image as "jupyter-base"
FROM nvidia/cuda:10.2-runtime AS jupyter-base

WORKDIR /

# install python and its tools
RUN apt update && apt install -y --no-install-recommends \
    git \
    build-essential \
    python3-dev \
    python3-pip \
    python3-setuptools
RUN pip3 -q intall pip --upgrade

# install all the basic packages
RUN pip3 install \
    jupyter \
    numpy pandas \
    torch torchvision \
    tensorboardX
```

III. Services

The final step of creating the environment is combining both images into a complete multi-container application with the help of Docker Compose. This tool allows to create a YAML file that contains a description of services that form the application. Ther is a workaround that enables usage of GPUs in Compose services. Let's look into the `docker-compose.yml` file:

```
version: "3.8"
services:
    tensorboard:
        image: dl-tensorboard
        build:
            context: ./
            dockerfile: tensorboard.Dockerfile
        ports:
            - ${TENSORBOARD_PORT}:${TENSORBOARD_PORT}
        volumes:
            - ${ROOT_DIR}:/jupyter
        command:
            [
```

```
            "tensorboard",
            "--logdir=${TENSORBOARD_DIR}",
            "--port=${TENSORBOARD_PORT",
            "--bind-all"
        ]

jupyter-server:
    image: dl-jupyter
    init: true
    build:
        context: ./
        dockerfile: jupyter.Dockerfile
    device_requests:
        - capabilities:
            - "gpu"
    env_file:./.env
    ports:
        - ${JUPYTER_PORT}:${JUPYTER_PORT}
    volumes:
        -${ROOT_DIR}:/jupyter
    command:
    [
        "jupyter",
        "notebook",
        "--ip=*",
        "--port'${JUPYTER_PORT}",
        "--allow-root",
        "--notebook-dir=${JUPYTER_DIR}",
        '--NotebookApp.token=${JUPYTER_TOKEN}'
    ]
```

We specify what ports from teh host machine will be exposed to the container. Docker compose takes these variables from a special file name `.env` wher the userlists all variables that have to be used during compose. This allows you to have a single place where you store all parts of the compose script that can differ from environment to environment. The `.env` file looks like this:

```
ROOT_DIR=/path/to/application/root/directory/host
JUPYTER_PORT=42065
JUPYTER_TOKEN=somepassword
JUPYTER_DIR=/jupyter/notebooks/container
TENSORBOARD_PORT=45175
TENSORBOARD_DIR=/jupyter/runs/container
```

This `ROOT_DIR` is mounted to both containers as `/jupyter`.

The `command` section of the service definition contains a set of arguments that combined form a command that will be executed on the container start-up. In the end, we get the following structure of the working directory:

```
.env
docker-compose.yml
```

```
jupyter.Dockerfile
tensorboard.Dockerfile
```

With all set and done, you can finally run `docker-compose up -d` inside our working directory.