Making Docker, Conda and Jupyter play well together

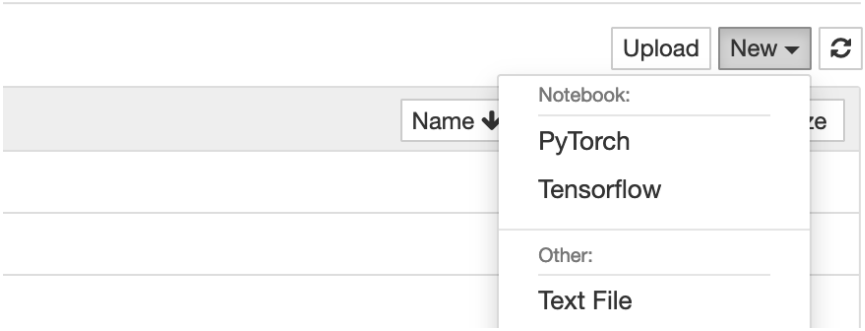
Build a Docker image running a multi-kernel Jupyter notebook server in fifteen minutes

Borun Chowdhury Ph.D. Dec 26, 2020 · 5 min read ★

Motivation

Docker and Docker-Compose are great utilities that support the microservice paradigm by allowing efficient containerization. Within the python ecosystem the package manager Conda also allows some kind of containerization that is limited to python packages. Conda environments are especially handy for data scientists working in jupyter notebooks that have different (and mutually exclusive) package dependencies.

However, due to the peculiar way in which conda environments are setup, getting them working out of the box in Docker, as it were, is not so straightforward. Furthermore, adding kernelspecs for these environments to jupyter is another very useful but complicated step. This article will clearly and concisely explain how to setup Dockerized containers with Jupyter having multiple kernels.



To gain from this article you should already know the basics of Docker, Conda and Jupyter. If you don't and would like to there are excellent tutorials on all three on their websites.

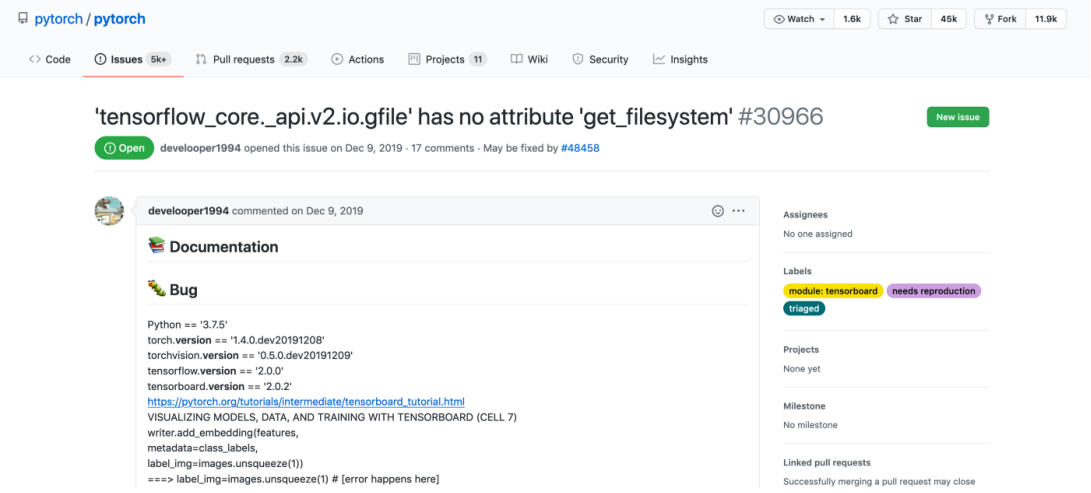
Introduction

I like Docker in the context of data science and machine learning research as it is very easy for me to containerize my whole research setup and move it to the various cloud services that I use (my laptop, my desktop, GCP, a barebones cloud we maintain and AWS).

Requiring multiple conda environments and associated jupyter kernels is something one often needs in Data Science and Machine Learning. For instance when dealing with Python 2 and Python 3 code or when transitioning from Tensorflow 1 to Tensorflow 2.

One such issue came up recently for me. I work with TensorFlow and PyTorch both for deeplearning and till now I had them both installed in the same conda environment in

my docker image. However, turns out that installing tensorflow 2.x breaks tensorboard for pytorch and the "solution" seems to be to not install tensorflow in the same environment as pytorch.



To reproduce it just try running the tutorial notebook mentioned in the image above after installing tensorflow 2.

So, in this case the solution is either of the following

1. Two dockerized containers with one having tensorflow 2 and the other pytorch.
2. One container with two environments that give two kernels in jupyter.

The second one seems more elegant. Nevertheless, the standard way of creating a conda environment and activating it requires an interactive sessions and that is not possible when building a docker image.

The Solution

In this article I quickly describe what needs to be done using a simple example and the actual code can be found in my repository. In fact my repository has the actual code that can be used to run the jupyter notebook mentioned above here but that has a lot of other steps that can distract away from the main topic of this article which is how to make Docker and Conda play well together. So I have also made another toy example here that just helps explain this particular point.

Dockerfile

The docker file contains the following main steps:

- Start with Ubuntu
- Download and install miniconda

```
RUN wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-
x86_64.sh
RUN bash Miniconda3-latest-Linux-x86_64.sh -b -p /miniconda
ENV PATH=$PATH:/miniconda/condabin:/miniconda/bin
```

- Install jupyter in the base environment
- Create two more environments and add them to jupyter kernel list. This is a non-trivial step as one has to run `ipykernel install` in the new environment. Usually one would do that by doing `conda init` and then activate the new environment but this requires starting a new bash shell which we cannot do here. This is thus handled in a different way by running commands in the appropriate shells in the following manner.

```
RUN conda env create -f packages/environment_one.yml
SHELL ["conda","run","-n","one","/bin/bash","-c"]
RUN python -m ipykernel install --name kernel_one --display-name
"Display Name One"
RUN pip install -U -r packages/requirements_one.txt
```

- Add a new user and switch to her directory
- Perform `conda init` as well as make it so that by default a new bash session opens in a one of the newly created environments. This step is not necessary for the problem we described is worth noting in case we do want the shell to be conda friendly and launch into one of the extra environments

```
SHELL ["/bin/bash","-c"]
RUN conda init
RUN echo 'conda activate one' >> ~/.bashrc
```

- Expose the port on which jupyter notebook will listen and run it. **Note that in this example I am running a notebook with no authentication which is only for illustrative purposes. You should always turn on proper authentication.**

```
EXPOSE 8888
ENTRYPOINT ["jupyter", "notebook", "--no-browser","--ip=0.0.0.0","--
NotebookApp.token=''","--NotebookApp.password=''"]
```

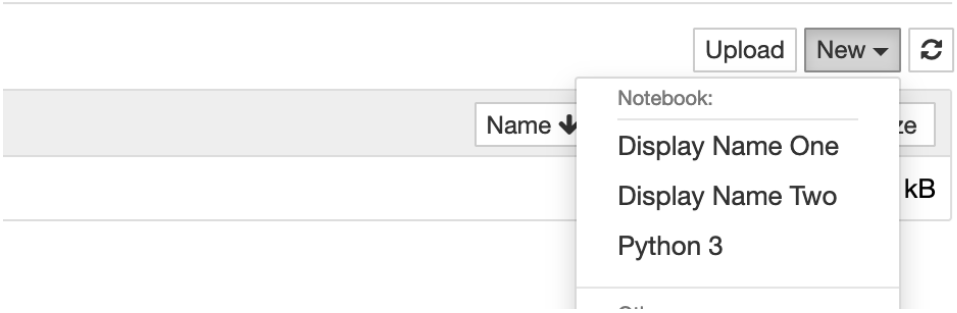
Docker-Compose

Finally I find it very useful to put all my microservices behind the reverse proxy traefik. This has the additional advantage of being able to turn on SSL on all services without individual configuration. In this toy example there is only one microservice but when we have many it is useful to turn them on with prefixes and here is why I used an entry point instead of command in the dockerfile above. I can now complete the command by asking jupyter to start the notebook server while listening on a different path where traefik will redirect the traffic

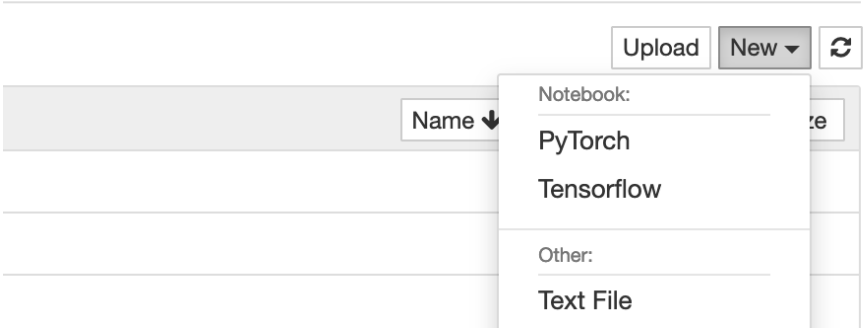
```
command: "--NotebookApp.base_url='multiple_conda_environments'"
```

Now the notebook can be accessed on https://myserver.com /multiple_conda_environment where you should replace myserver.com with your hostname.

Now when you run try to create a new notebook you should see a choice between the standard python3 and two additional kernels



If you go to the actual repository with the full code to run both tensorflow and pytorch you will find that I appropriated the python3 kernel to convert it into a tensorflow kernel and have changed its name to reflect that.



This is a better solution if you do not care about the extra default kernel floating around that is not going to be used.

Conclusion

Having your jupyter server run as a container is a must for every data scientist as it allows one to seamlessly **move their lab**, as it were, from one cloud to another.

Being able to have multiple kernels in jupyter is likewise a must for every data scientist as well as it allows one to work on multiple projects with mutually exclusive package dependencies.

Due to the way standard conda environments and their jupyter kernelspecs are installed, making the two play with each other is not straightforward. In this tutorial and the associated github repository I have explained how to make containerized jupyter installations that support multiple kernels.