DevOps Part A Submission

- Name: Jeyakumar Sriram
- Class: DAAA/FT/2B/01
- Admin No: 2214618

> If you are a teacher and feel bored please read Docs/DeepLearning.md where I documented the nightmares I went through during this project. Also look at the beautiful README I created, it looks nicer on GitLab

Objective:

I will be using the 128x128 CNN model for my DevOps project. To make it easier to quickly glance the important information I have made this section.

Development Environment Setup

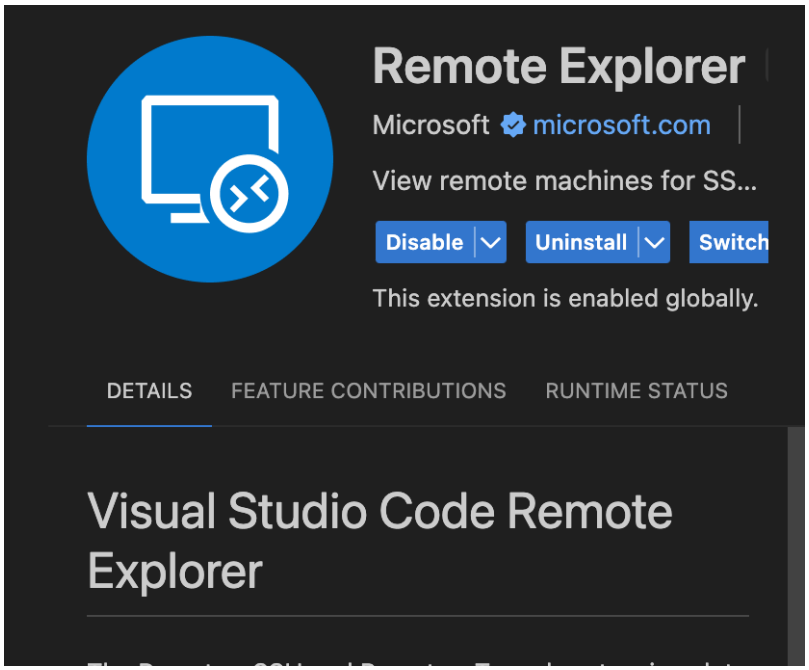
- Ensure that you have Docker installed, if not download it at this website
- Once you are done, run the following commands

```
docker pull python:3.9.18
docker run -it -v /var/run/docker.sock:/var/run/docker.sock --name
CNN_Server python:3.9.18 sh -c "apt-get update ; apt-get install docker.io -
y ; bash
```

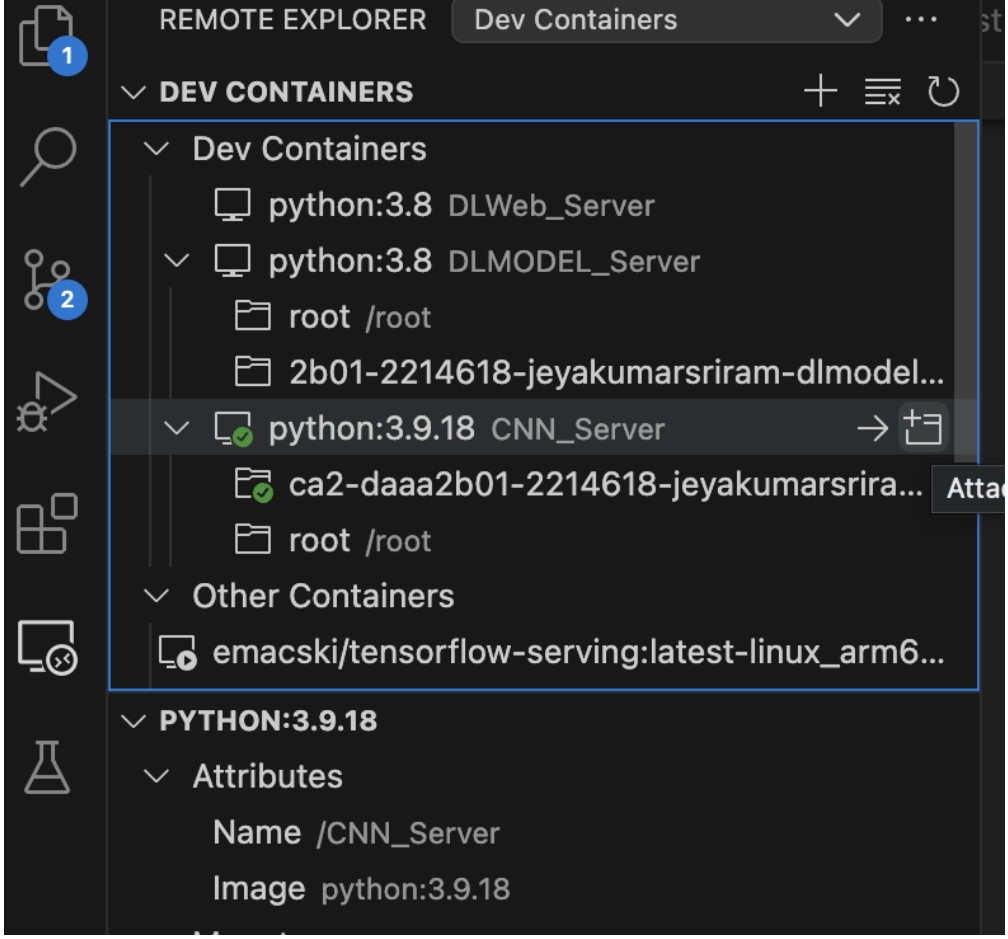
> Note: If you are using Mac or Linux, these commands will work instantly, but if you are using Windows, you might have to enable Windows Subsystem for Linux before proceeding

To make sure your docker works inside the development container, try running `docker info`

- Check your Docker Desktop to see if the CNN_Server container is running
- Open up Visual Studio Code and install "Remote Explorer" extension if you don't have it



- Navigate to the Explorer tab and click on the icon as shown below



Configuring Development Environment

- Run the sequence of commands below to download the project directory inside your development container and configure python with all the required packages.

```
git clone https://gitlab.com/4618-devops/ca2-daaa2b01-2214618-
jeyakumarsriram-dl.git
python -m venv env
source ./env/bin/activate
cd /ca2-daaa2b01-2214618-jeyakumarsriram-dl
python -m pip install tensorflow
pip install matplotlib seaborn numpy pytest
```

- To train the models, you would need the dataset too. it is recommended to use the dataset from this website

Or alternatively if you have the Kaggle API acces you can run

```
kaggle datasets download -d moustacheman/vegetable-images
```

- After which, you should see a zip file, unzip it and place it into your project directory. Ensure it is named "Vegetable Images" if not change the `ROOT` variable below.

After all these steps, you should be ready to train some models!

Model Training

Run the following cells sequentially. Ensure that the kernl you selected is Venv

In [ ]:
```
# Basics
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import os
import time

# ML
import tensorflow as tf
from tensorflow.keras.layers import *

# Miscellaneous
import warnings
warnings.filterwarnings("ignore")

# ML
import tensorflow as tf
import tensorflow.keras.backend as K

# Modelling
from tensorflow.keras.layers import *

tf.__version__
```

Modify the `ROOT` if your data directory is different.

In [ ]:
```
ROOT = "./Vegetable Images"
CLASSES = ['Broccoli','Capsicum','Bottle_Gourd','Radish','Tomato','Brinjal','Pumpkin'
tfClasses = tf.constant(CLASSES)
TRAIN = tf.data.Dataset.list_files(f"{ROOT}/train/*/*")
TEST = tf.data.Dataset.list_files(f"{ROOT}/test/*/*")
VAL = tf.data.Dataset.list_files(f"{ROOT}/validation/*/*")
dataCats = ["train","test","validation"]
```

We will be performing some preprocessing on the images. If you are using a different dataset, edit this part as needed.

In [ ]:
```
classCounts = []
for cat in dataCats:
    class_distro = [len(tf.data.Dataset.list_files(f"{ROOT}/{cat}/{i}/*")) for i in C
    classCounts.append(class_distro)

train_distro = classCounts[0]

targetSize = max(train_distro)
additional_needed = [targetSize-i for i in train_distro]

aug_train = []
for i,v in enumerate(CLASSES):
    path = f'{ROOT}/train/{v}'
    imgNeeded = additional_needed[i]
    images = os.listdir(path)[:imgNeeded]
    aug_train.extend([path + "/" + i for i in images])
train_data_aug = tf.data.Dataset.from_tensor_slices(aug_train)

def createPreprocessor(imgSize):
    def processing(path):
        label = tf.strings.split(path , os.path.sep)
        one_hot = label[-2] == tfClasses
        label = tf.argmax(one_hot)
        label = tf.one_hot(label, depth=len(CLASSES))
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)
        img = tf.image.resize(img, [imgSize, imgSize])
        img = tf.image.rgb_to_grayscale(img)
        img = img / 255.0
```

```
        return img , label
    return processing

def augmentation(image,label):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.rot90(image)
    image = tf.image.random_brightness(image, max_delta=0.2)
    image = tf.image.random_contrast(image, lower=0.8, upper=1.2)
    return image , label

def configure_for_performance(ds):
  ds = ds.cache()
  ds = ds.shuffle(buffer_size=1000)
  ds = ds.batch(32)
  ds = ds.prefetch(buffer_size=tf.data.AUTOTUNE)
  return ds

train_ori_large = TRAIN.map(createPreprocessor(128))
train_aug_large = TRAIN.map(createPreprocessor(128)).map(augmentation)
train_large_ds = train_ori_large.concatenate(train_aug_large)

test_large_ds = TEST.map(createPreprocessor(128))
val_large_ds = VAL.map(createPreprocessor(128))

data = train_large_ds.concatenate(test_large_ds).concatenate(val_large_ds)
data = configure_for_performance(data)

train_ori_small = TRAIN.map(createPreprocessor(31))
train_aug_small = TRAIN.map(createPreprocessor(31)).map(augmentation)
train_small_ds = train_ori_small.concatenate(train_aug_small)

test_small_ds = TEST.map(createPreprocessor(31))
val_small_ds = VAL.map(createPreprocessor(31))

data_small = train_small_ds.concatenate(test_small_ds).concatenate(val_small_ds)
data_small = configure_for_performance(data_small)
```

We will be defining our 2 models here. One accepts a 128x128 size image input while the other takes a 31x31 input. Both will only use grayscale images as specified in the previous assignment brief.

In [ ]:

```
reg_cnn_model = tf.keras.Sequential([
    Conv2D(16, 3, activation='relu', input_shape=(31, 31, 1)),
    BatchNormalization(),
    MaxPooling2D(),

    Conv2D(32, 3, activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    Conv2D(64, 3, activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(15, activation='softmax')
])

reg_cnn_model_large = tf.keras.Sequential([
    Conv2D(32, 3, activation='relu', input_shape=(128, 128, 1)),
    BatchNormalization(),
    MaxPooling2D(),
```

```
    Conv2D(64, 3, activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    Conv2D(128, 3, activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    Conv2D(256, 3, activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    Flatten(),

    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(15, activation='softmax')
])
```

This is a summary of the inputs of the various layers of the CNN models. The number of parameters in each is also mentioned. Note that models with higher parameters will take longer to train.

In [ ]:
```
reg_cnn_model.summary()
reg_cnn_model_large.summary()
```

We will be saving the models using the SavedModel format rather than h5 or .keras in order to ensure compatibility for Tensorflow Serving. The model training begins now, estimated wait time is 20 minutes. So have a nap or a small meal while it cooks!

In [ ]:
```
es = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='loss', factor=0.5, patience

reg_cnn_model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

modelHist = reg_cnn_model.fit(
      data_small, epochs=25, callbacks=[es,reduce_lr]
    )

reg_cnn_model.save("./cnn_small/1", save_format="tf")

reg_cnn_model_large.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

modelHist2 = reg_cnn_model_large.fit(
      data, epochs=15, callbacks=[es,reduce_lr]
    )

reg_cnn_model_large.save("./cnn_large/1", save_format="tf")
```

For your convenience, I have written some code to see the difference in speed and accuracy between the two models, As you can see the larger model is slightly slower but more accurate.

In [ ]:
```
# Measure prediction time for final_31
start_time = time.time()
prediction_31 = reg_cnn_model.evaluate(configure_for_performance(test_small_ds))
end_time = time.time()
time_taken_31 = end_time - start_time

# Measure prediction time for final_128
start_time = time.time()
prediction_128 = reg_cnn_model_large.evaluate(configure_for_performance(test_large_ds
end_time = time.time()
time_taken_128 = end_time - start_time

# Print the prediction times
print(f"Prediction time for final_31: {time_taken_31} seconds")
print(f"Prediction time for final_128: {time_taken_128} seconds")
```

Model Serving

Ensure that you have `cnn_large/` and `cnn_small/` directories in your project's root folder.

- Run this command to pull the tensorflow serving docker image

`docker pull tensorflow/serving`
If you are using a M Series Macbook, use this command instead to ensure compatibility

`docker pull emacski/tensorflow-serving:latest-linux_arm64`
These are two tensorflow models that we will be serving using the same container. Since we are not serving just one model, we cannot use the standard command for tensorflow/serving.

- Create a `model_config.config` file in the root of your project directory.

> For convenience, consider copying your `model_config.config` , `cnn_large/` and `cnn_small/` files to a easily accessible place like `C://production` in windows and `~/production` in Unix based systems.

- Add the following YAML code to your config

```
model_config_list:
  config:
    name: "large"
    base_path: "/models/cnn_large"
    model_platform: "tensorflow"
  config:
    name: "small"
    base_path: "/models/cnn_small"
    model_platform: "tensorflow"
```

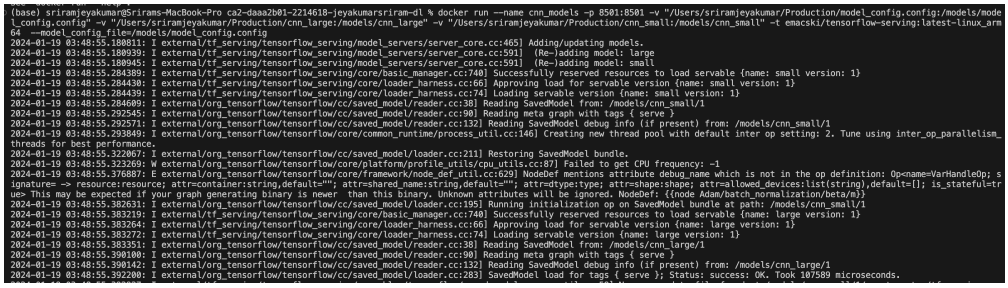
- Run the following command to serve your models. Replace the string with your system's information before running the command

```
docker run --name cnn_models -p 8501:8501 \
    -v
"/Users/sriramjeyakumar/Production/model_config.config:/models/model_config.con
\
    -v "/Users/sriramjeyakumar/Production/cnn_large:/models/cnn_large" \
    -v "/Users/sriramjeyakumar/Production/cnn_small:/models/cnn_small" \
    -t emacski/tensorflow-serving:latest-linux_arm64 --
model_config_file=/models/model_config.config
```

> You should change `emacski/tensorflow-serving:latest-linux_arm64` to `tensorflow/serving` if you are not using a M Series Mac

Here is the breakdown of the command above: - We map the port 8501 to our localhost's port 8501 so we can make requests to it - We mount 3 different files: Our two model directories and the config file - We use the serving image to create the container - We specify to TF Serving the config file

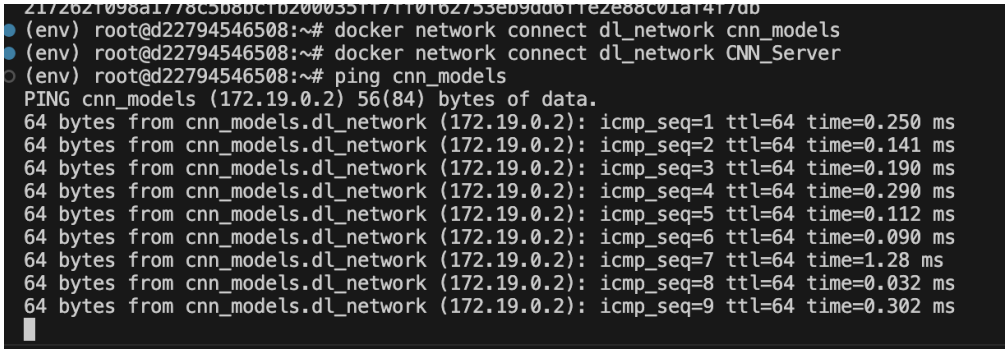
You should expect to see something like this when it runs



Also run thw following commands to connect your development container to the Tensorflow Server so you have call its api from within your development environment.

```
apt-get install iputils-ping
docker network create dl_network
docker network connect dl_network cnn_models
docker network connect dl_network CNN_Server
```

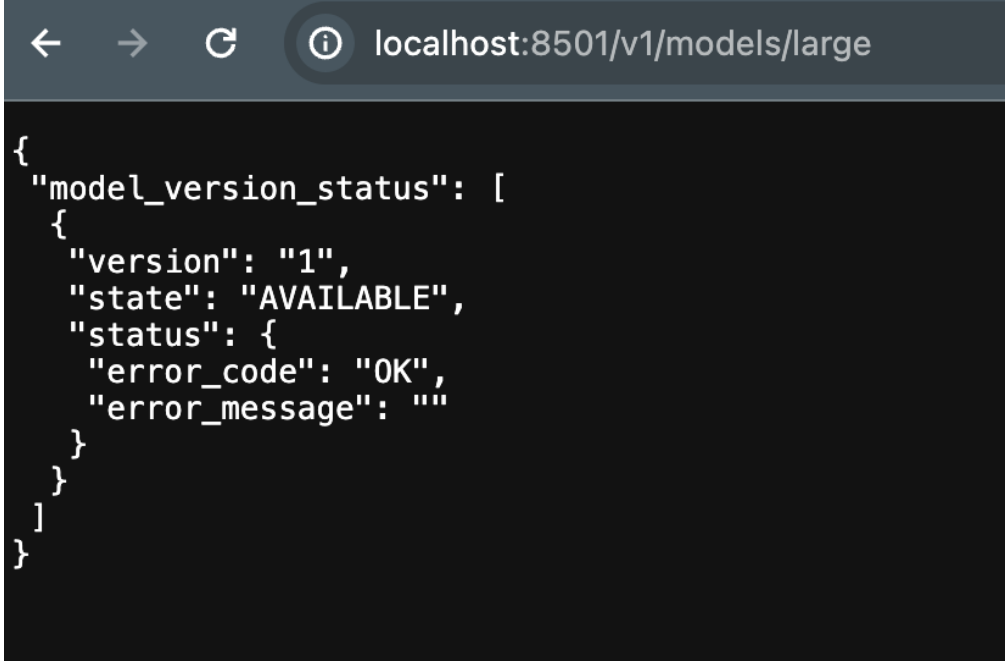
Try `ping cnn_models` and you should see something like this

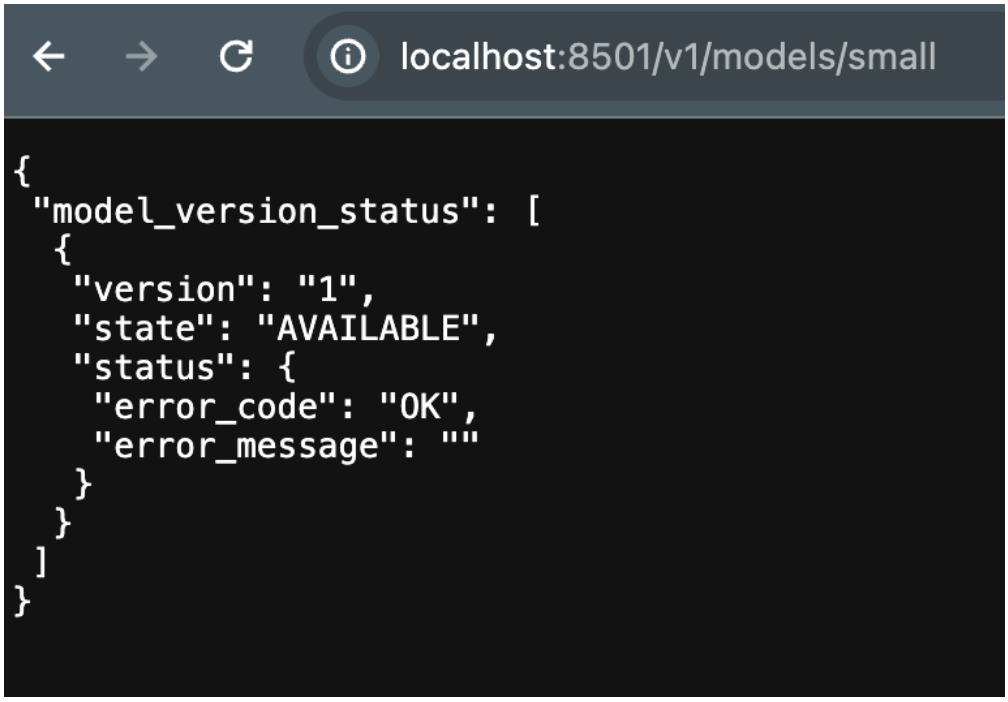


Checking if TensorFlow Serving is Running

1. **Large Model:**
   - URL: http://localhost:8501/v1/models/large
   - This endpoint should provide information about the 128x128 TensorFlow model, including its status and configuration.
2. **Small Model:**
   - URL: http://localhost:8501/v1/models/small
   - This endpoint should provide information about the 31x31 TensorFlow model, including its status and configuration.

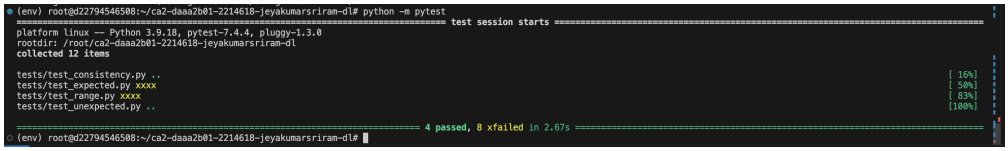
Open these URLs in your web browser or use tools like `curl` or `wget` in the command line to make HTTP requests. If TensorFlow Serving is running and the models are successfully loaded, you should see something like this





Also run this command that will use pytest to check various parts of our model. It will test the basic functionalities, test the range of the model as well as its consistency.

`python -m pytest`
You should expect to see something like this.



Congrats! 🥳 You have successfully served your models.

Now they are only accessible in your laptop. If you want to make it accessible everywhere, consider using a cloud solution like Render to host your model

Deployment

Before you do any deployment, you need your changes to be pushed to the cloud. But before you commit your changes so far into git, we have to address how you are storing your models.

Tensorflow models are not small and are more like static object which are not supposed to be trakced by git. So it is not the best idea to push your entire model onto github using the conventional way.

While there a probably better industry-standard ways to store your models, most of them are PAID 😭 . Since you my friend, have no cash to spare, I will give you a simpler solution that solves this problem. Git Large File Storage.

Git LFS basically allows git to take in your files but not trakc all the changes. When you push your repo to the platform like GitHub or GitLab, items marked with LFS wont be stored in the same place as the rest of your code. Instead it will be stored in a LFS server and a pointer to it will be stored in your main repo. Dont worry, when you push and pull, Git will automatically get your LFS items for you.

LFS Setup

For Windows, skip this section as it is installed by default for you, if it isn't for some reason, good luck!

For Macbook, you can run `brew install git-lfs`

But if you are working on this inside a Development container, you are in a linux environment. You will have to run more disgusting commands

```
curl -s https://packagecloud.io/install/repositories/github/git-
lfs/script.deb.sh | bash
apt-get install git-lfs
```

> We omitted the use of sudo here as its not needed for docker containers

Confirm installation using the command `git lfs`

Now, which files are so huge that we need Git LFS to track them? We you might think that it is the entire directories of `cnn_large/` and `cnn_small/` but NO. If you snoop around into these directories, most of the files basically contain metadata, like you model structure and whatnot. The real heavy files are those under `variables` subdirectory. So we will only use LFS on them. Now you should run the following commands.

```
git lfs install
git lfs track "*/*/variables/*"
git add .gitattributes
```

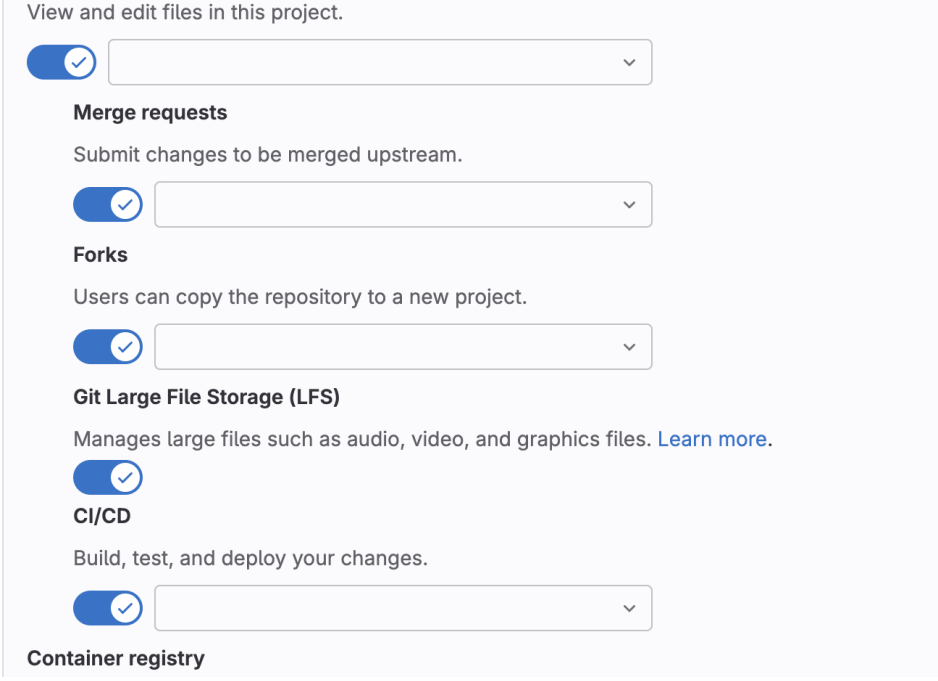
And that's it! You have setup LFS, now you just have to add and commit as usual. Whenever you create new models, remember to run the track command again.

If yyou are unsure if you have done this correctly, run

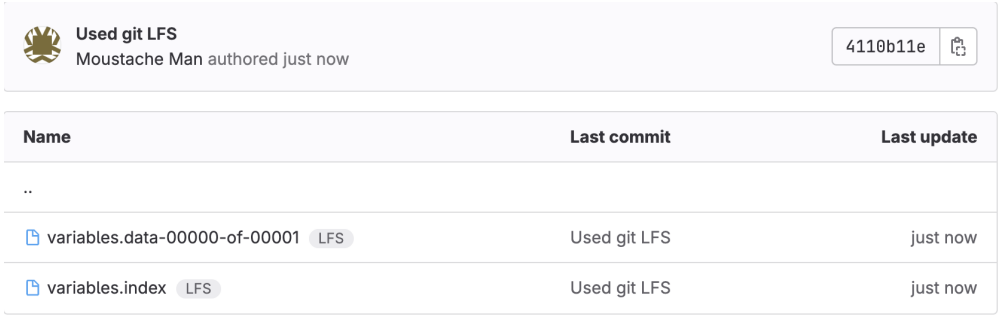
```
git lfs status
```

which will show the files that are tracked.

One more thing, under you GitLab ensure that the LFS option is selected like this



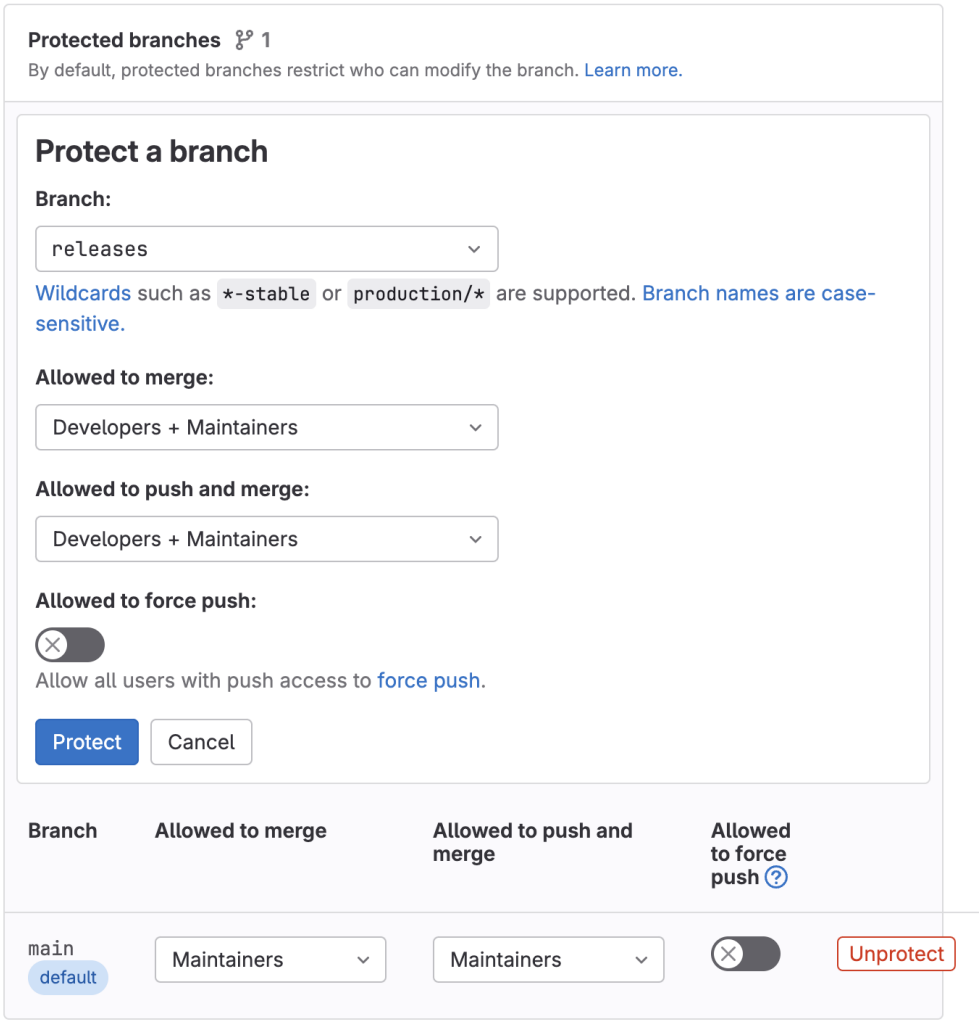
When you deployed to gitlab you will see a Tag like this to signifiy that it is under the LFS server



Deploying to Render

Ensure that you have committed all your changes to GitLab. You either create a new branch called `releases` like me or you can just use the main branch for deployment, whatever works for you. I'm treating my `main` branch more like a development branch while `releases` is the final branch for me. But again, whatever works for you.

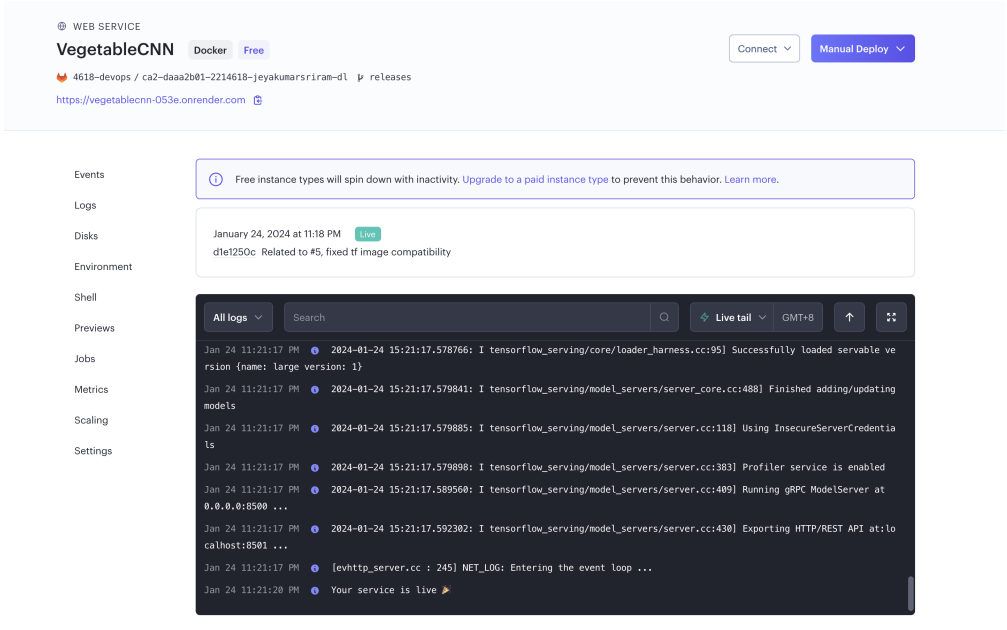
I would also suggest that you make `releases` a protected branch so it can't just be deleted liek this



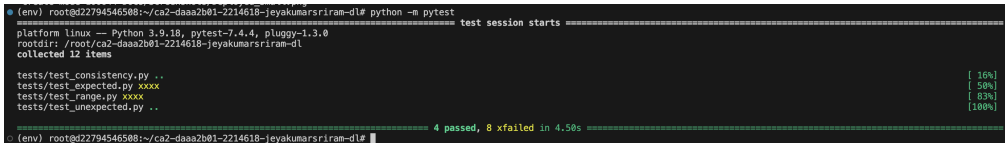
To deploy,

- Create an account on render and log in.
- After going to your dashboard, click "New" and select "Web Service" as the type of project you will be deploying
- Connect your GitLab to Render for easier deployment
- Fill in the name and make sure the Environment is Docker, set the render to trakc `releases` branch or whatever you want
- Click deploy and wait for the logs to say your service is live

You should see something like this if you model deployment was successful.



Now, edit the pytest/conftest.py file and replace the cnn_model:8501 with the url of your render application and rerun PyTest. You should expect to see something like this.



Setting Up CI/CD Pipeline

This is to basically make you like easier. Whenever you commit into the Releases branch, you want to make sure you code works before it deploys to the internet. Therefore I create a pipeline to test the code before deploying to the production server.

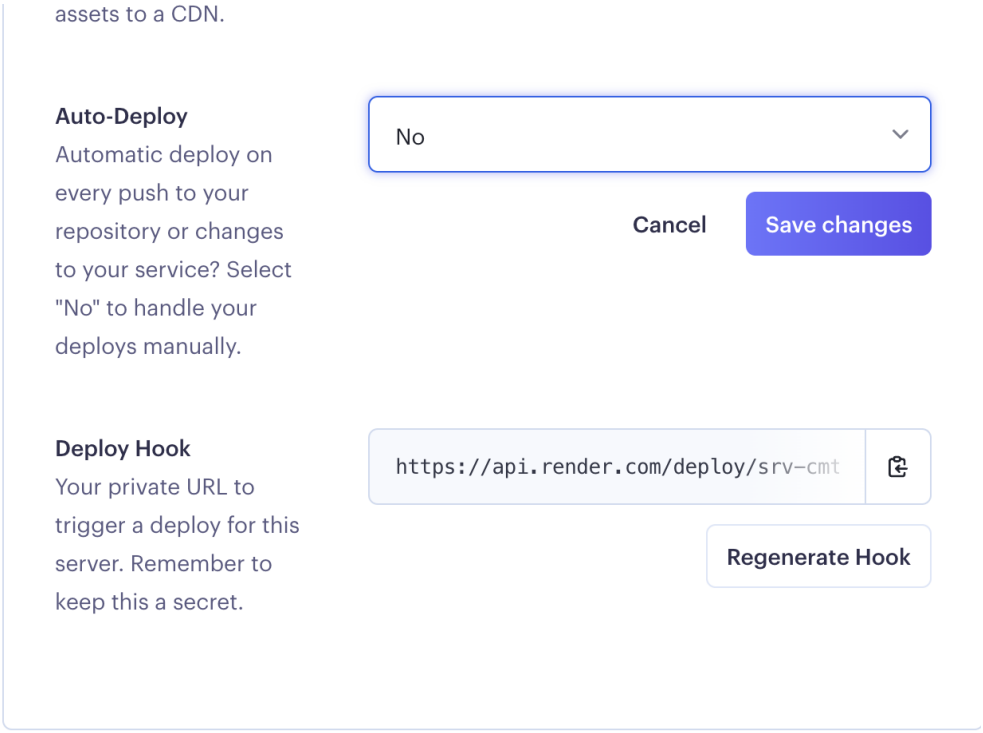
There are many ways to test a Tensorflow Serving app. I unsuccessfully tried running docker inside the CICD pipeline and test. Therefore, once again, I will show you a less fancy but crud way of solving this problem. Having two deployments: one for testing, one for production.

Let me explain the process to you. When a commit happens in the `releases` branch, the following will happen.

1. GitLab will trigger the redeployment of your code in a Test Server hosted on Render (separate from the one you made)
2. GitLab will run PyTest on that test server to confirm if it works as intended
3. If the PyTest returns a success code of 0, Gitlab will trigger the redeployment of your main production server

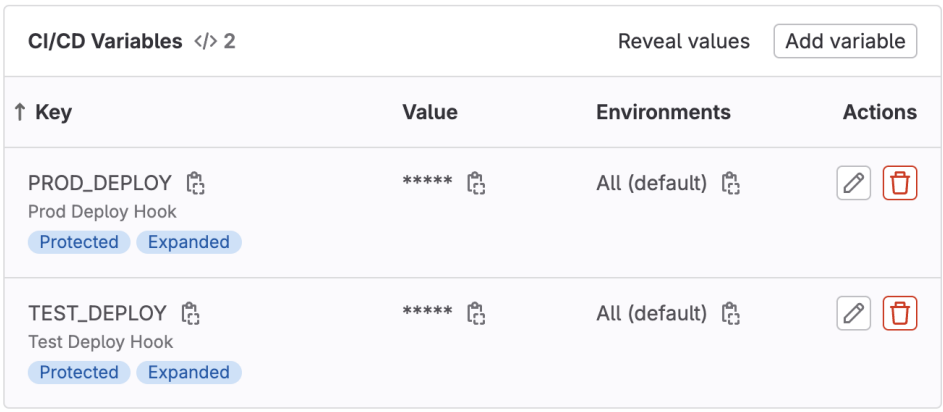
The configuration for the Test and Production server are exactly the same so you can test your code without messing up the main server. For this you would need to do a couple of things.

- Setup another render deployment called Test_Server. Follow the exactl same steps you followed just now but different name
- Go to the Settings of both Render deployments, scroll down and disable automatic redeploy. This is to make Render wait for you call before deploying something.



- Copy the Deploy Hook URL into a safe place

- In your gitlab project, go to Settings/CICD and expand the Variables tab. Under which you should add two variables using the two deploy hooks as shown below.



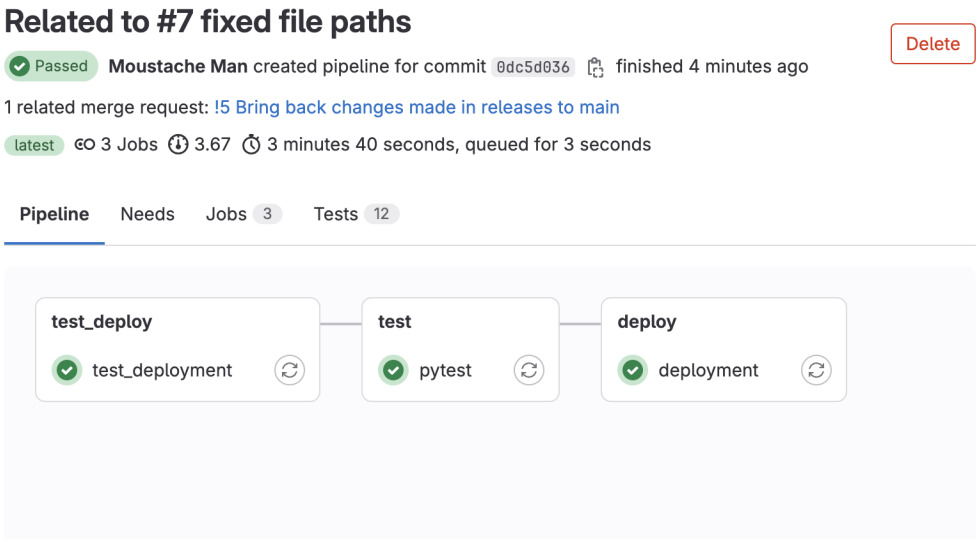
- Now edit your `tests/conftest` file and change the url to your Test Server's url - Create a `.gitlab-ci.yml` file in your project's root folder and put the following things in there.

```
stages:
  - test_deploy
  - test
  - deploy

test_deployment:
  stage: test_deploy
  script:
    - curl -s "$TEST_DEPLOY"
    - sleep 1m
  only:
    - releases

pytest:
  stage: test
  image: python:3.9.18
  script:
    - pip install -r requirements.txt
    - python -m pytest --junitxml=report.xml
    - TEST_CODE=$?
    - |
      if [ $TEST_CODE -eq 0 ]; then
        echo "Tests are all passed!"
      else
        echo "Tests failed"
        sleep 1m
        exit 1
      fi
  artifacts:
    when: always
    reports:
      junit: report.xml
  retry:
    max: 2
    when:
      - always
  only:
    - releases
  dependencies:
    - test_deployment

deployment:
```

```
  stage: deploy
  script:
    - curl -s "$PROD_DEPLOY"
  only:
    - releases
  dependencies:
    - pytest
```

This pipeline basically does what I said. To point out a few things, the testing stage will retry twice before quitting and with a 1 minite delay inbetween. This delay is included bcause something Render takes very long to deploy and a delay in render deployment shouldn't cause the pipeline to fail. I will also save the PyTest in a report.xml file so that you can view it later. To trigger the redeployment, all you have to do is send a GET request to the Deploy Hook you copied. This can be done using curl. To assess the success of PyTest, we will be utilising the status code. 0 means all tests have passed. Only in that scenario we will deploy the application, if not we will just stop the pipeline using `exit 1`.

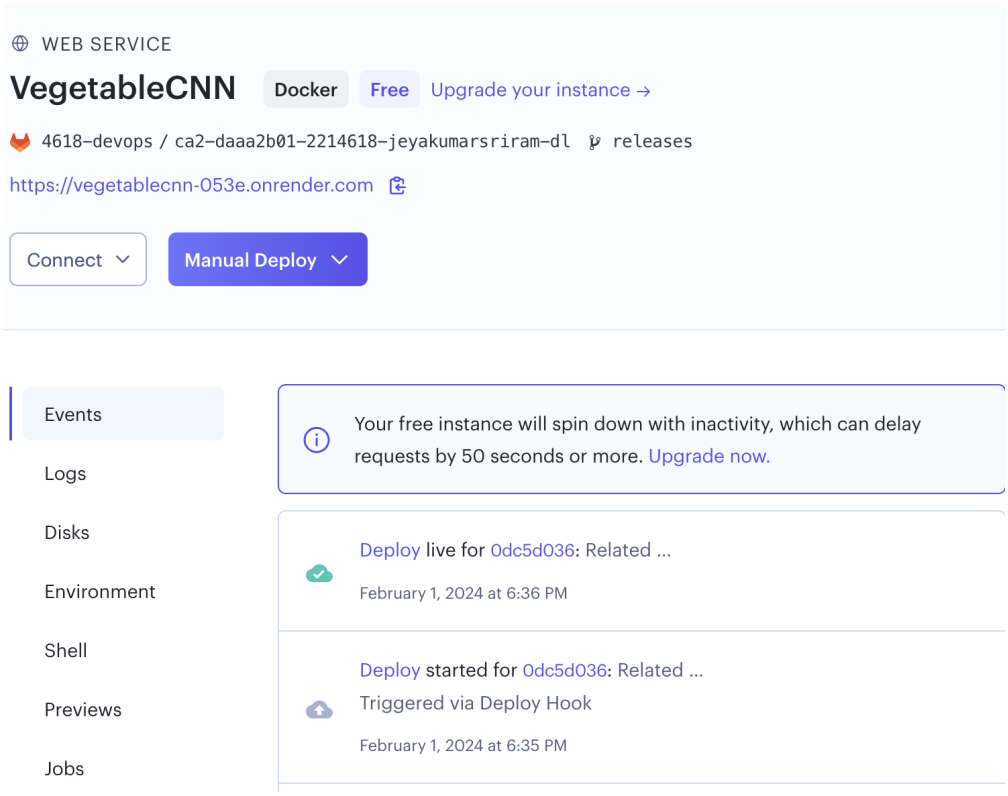
Now you can try to make a commit in the `releases` branch. To see your pipeline running go to Build/Pipelines in your GitLab interface. You should seem something like this



And if you click on the test stage, you can view the results like this

```
[notice] A new release of pip is available: 23.0.1 -> 23.3.2
[notice] To update, run: pip install --upgrade pip
$ python -m pytest --junitxml=report.xml
============================= test session starts ==============================
platform linux -- Python 3.9.18, pytest-7.4.4, pluggy-1.3.0
rootdir: /builds/4618-devops/ca2-daaa2b01-2214618-jeyakumarsriram-dl
collected 12 items
tests/test_consistency.py ..                                             [ 16%]
tests/test_expected.py xxxx                                              [ 50%]
tests/test_range.py xxxx                                                 [ 83%]
tests/test_unexpected.py ..                                              [100%]
- generated xml file: /builds/4618-devops/ca2-daaa2b01-2214618-jeyakumarsriram-dl/report.xml -
======================== 4 passed, 8 xfailed in 7.47s ========================
$ TEST_CODE=$?
$ if [ $TEST_CODE -eq 0 ]; then # collapsed multi-line command
Tests are all passed!
Uploading artifacts for successful job                                   00:02
Uploading artifacts...
report.xml: found 1 matching artifact files and directories
WARNING: Upload request redirected                location=https://gitlab.com/api/v4/jobs/6071108398/artifacts?artifact_format=gzip&artifact_type=junit new-url=https://gitlab.com
WARNING: Retrying...                              context=artifacts-uploader error=request redirected
Uploading artifacts as "junit" to coordinator... 201 Created  id=6071108398 responseStatus=201 Created token=glcbt-65
Cleaning up project directory and file based variables                   00:01
Job succeeded
```

If you move over to render you will see the logs mentioned that it was redeployed like this



That's it! You are done!