LiveEO MLOps Engineer Challenge

this report is a summary of my work. It contains instructions for running the code, explanations of the methods used and possible future features and functionalities that can be added.

**Code exploration and understanding:**

This code allows to train and evaluate a neural network in order to detect buildings in the provided images implemented using the PyTorch Lightning library (an open-source Python library that provides a high-level interface to PyTorch). It contains the data pre-processing and transformation and the implementations scripts.

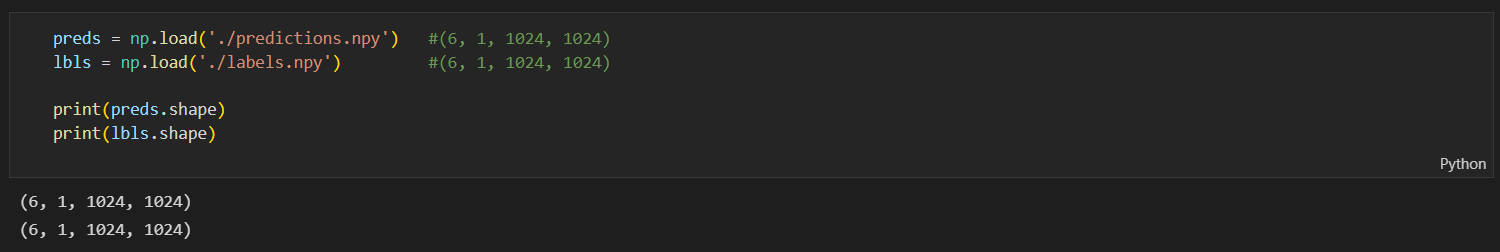
To run inference on test images, we need to run this command:

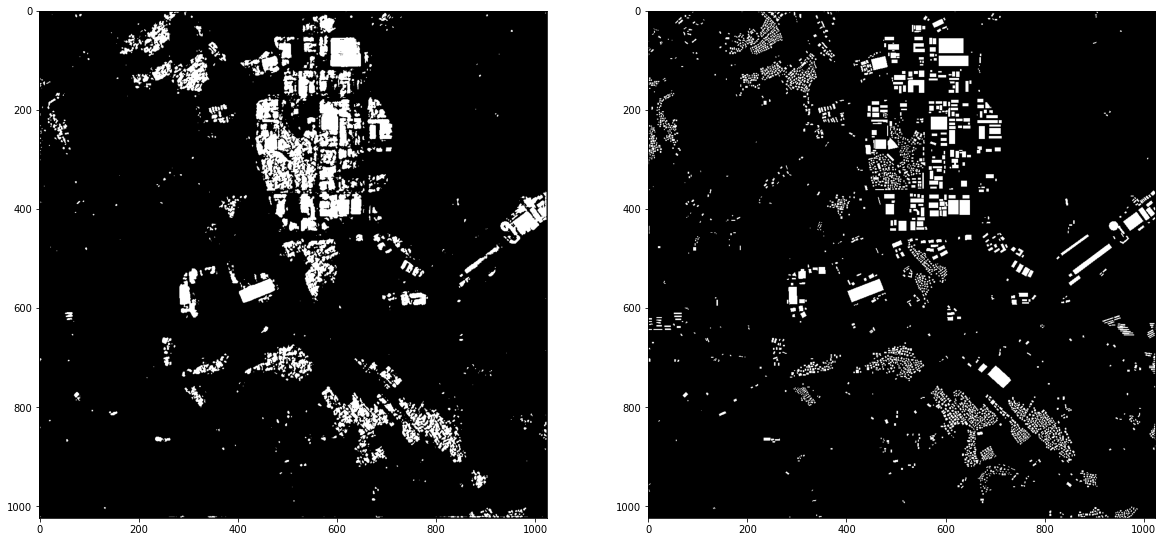
python ./scripts/main.py --ckpt\_path './trained\_models/best\_model.ckpt'

The execution mode and the path to the test images are set by default.

This allows to generate and save 2 files containing UNet predictions and its corresponding ground truth masks:

* labels.npy
* predictions.npy

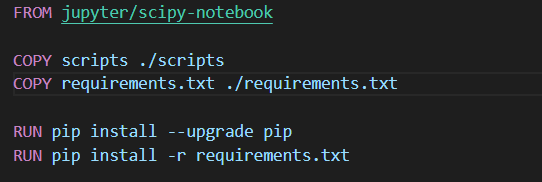


****

The test images are split into testset and trainingset with a test size of 0.1. We only have 6 images for test.

**Setting up the docker image:**

I created a simple Dockerfile with the jupyter/scipy-notebook image as the base image, copied scripts and requirements.txt and installed requirements.

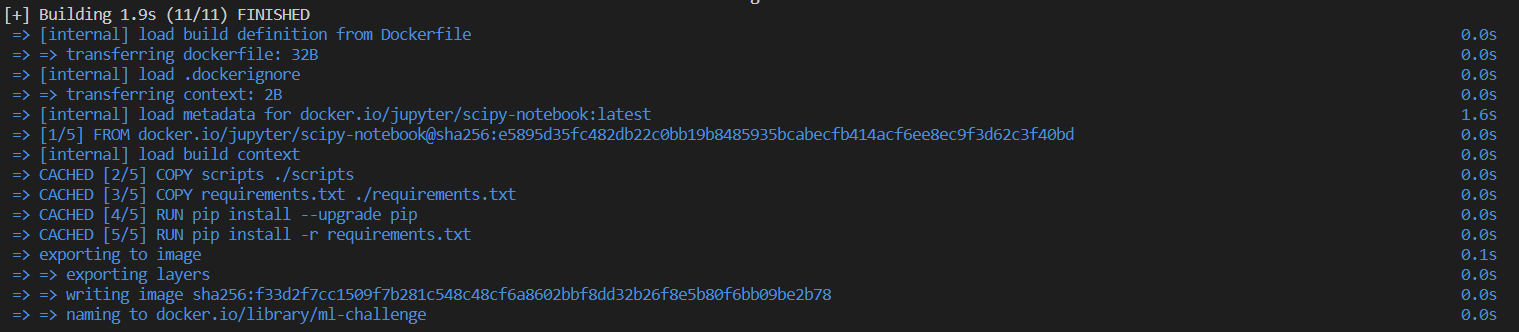


The jupyter/scipy-notebook is a Jupyter Notebook scientific Python stack and it includes popular packages from the scientific Python ecosystem. It’s a ready-to-run Docker image.

In order to build the image, we run the following command in our terminal:

docker build -t ml-challenge -f Dockerfile .

The output is the following:



To execute inference test runs and save the outputs using docker, we run the following command:

docker run --name mlops\_challenge --ipc=host -it ml-challenge python ./scripts/main.py --ckpt\_path './trained\_models/best\_model.ckpt' --num\_workers 0

I’ve added downloads.py script that allows to download images and checkpoints saved on google drive (since I don’t have access to another cloud storage service).

**Setting up the CI/CD pipeline:**

GitHub Actions offers a complete and powerful set of functionalities. It is as easy as adding a YAML file and putting it in the.github/workflows/ directory to add Actions to a repository. The Action will become active as soon as the YAML file is committed and pushed.

I created a YAML file allowing to:

* Lint the code when new commits are pushed to GitHub

The Linter will check the code syntax and provide instructions on how to clean it. I’ve used Flake8.

* Build and push a Docker image to the Docker Hub when new commits are pushed to GitHub

Pushing an image to the Docker Hub requires adding the following variables:

* DOCKERHUB\_USERNAME (username of your Docker Hub account)
* DOCKERHUB\_PASSWORD (password for your Docker Hub account)

as GitHub Repository Secrets. Such values are stored in an encrypted form by GitHub and are safe to use with Actions.

The tag\_with\_sha element will tag an image with the SHA of the Git commit from which the image was built.

**Future possible features and functionalities:**

1. Execute unit testing every time a commit is pushed to GitHub.

Some test example cases:

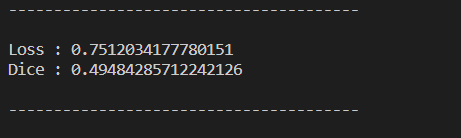
* Ensure that the data has the right format (that the data loader is working correctly)
* Ensure that the image have its correct corresponding mask
* Run a training step and compare the weight before and after to ensure that they are updated
* The model’s output has the correct shape

1. We can use CML functions to automate development workflows including model training and evaluation and comparing ML experiments.

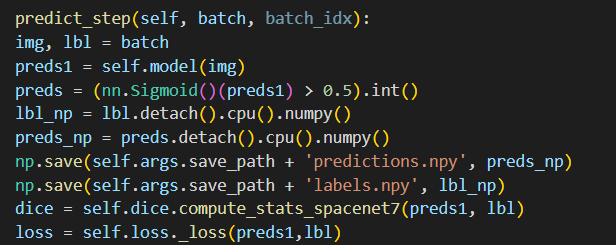
CML can help train and evaluate models and then generate a visual report with results and metrics automatically on every pull request.

**Setting up evaluation metrics:**

Running inference with docker will log this output:



We could extract the evaluation metrics as follow:



**Model versioning:**

I set up MLflow locally as well as a database-backend store using PostgreSQL to register models. I used the mlflow.pytorch module which provides an API for logging and loading PyTorch models.

I used auotlog method. Autologging is only supported for PyTorch Lightning models (which is our case).

In order to run the MLflow server, we run the following command in our terminal:

mlflow server --backend-store-uri postgresql://mlflow:mlflow@localhost/mlflow\_db1 --default-artifact-root /mlruns -h 0.0.0.0 -p 8000

It will show two main sections:

* Experiments: Where we will find different projects, where each of these can have multiple runs and each run contains all logged informations (parameters, metrics, etc.)
* Models: Containing all models that have been registered

The local database mlflow\_db1 I’ve set up will contain data related to the models registered

Each experiment is associated with an experiment ID. This is important because MLflow will create a new folder with this ID under mlruns which will contain the models and artifacts generated in each run.

Training the model will allow us to track the parameters, metrics, and artifacts and get everything logged to mlflow.

