Technische Universität Berlin

Fakultät IV

Big Data Engineering (DAMS Lab) Group

Sentinel Building Segmentation

Architecture of Machine Learning Systems

Summer Semester 2024

submitted by

Sultan Numyalai

Matriculation number: 461650

and

Baldo Brendel

Matriculation number: 480043

and

Asier Serrano Aramburu

Matriculation number: 0502058

Date of submission: 08.07.2024

Contents

1 Installation guide

It is recommended to run the project on a machine with a Linux/UNIX operating system.

1.1 Clone the project repo

git clone git@github.com:asicoderOfficial/Sentinel-2A-Segmentation.git

cd Sentinel-2A-Segmentation

1.2 Install Jupyter Notebook and Poetry via the terminal

To install Jupyter Notebook and Poetry, run:

pip install jupyter pip install poetry

1.3 Installation of other requirements

To install the dependencies, run poetry install

1.4 Adding an .env file to the project

In the project config dir create a .env.local file with the following SH_CLIENT_ID, SH_CLIENT_SECRET variables.

1.5 Running the data pipeline

poetry run python -m `data_pipeline`, then enter the path to the `data_pipeline.yml` configuration file.

1.6 Running the experiment

`poetry run python -m ml_pipeline`, then enter the path to the `ml_pipeline.yml` config file.

2 Task 1

The solution for task 1 is shown in the jupyter notebook located at the file *notebooks/task1.ipynb*.

After loading the credentials from the *config/.env.local* file into the *SHConfig* class, the sentinel API parameters are set to then create the bounding box for Berlin. For extracting buildings, we use *osmnx* library, as we faced many issues with *pyrosm*, and the SentinelHub API is used to extract the images without labeling. Hence, we create 2 different classes, which will also be reused in Task 2:

- *src/data_acquisition/osm/OSM*: wraps up the calls to the osmnx library.
  Using the *buildings()* method, we obtain the features from the specified bounding box (in OSM standards, this is: north, south, east, west) in a *GeoDataframe* setting *tags='building':True*, and project with the parameter *to_latlong=True* (very important to avoid rotations between SentinelHub and OSM resulting data).

  What we get is an RGBA image, for which we only select the first channel, and create a binary mask (pixels >= 50 will be cataloged with a 1 as buildings, and the rest 0). This is because, when there are zones with more buildings, these values were higher, and by experimentation we realized 50 was a good enough heuristic.

  Then, buildings can be both visualized at the moment, and saved for later with the specified dimensions.

- *src/data_acquisition/sentinel_hub_image_downloader/SentinelHubImageDownloader*:
  wraps up the *SentinelHubRequest* call to the SentinelHub API.

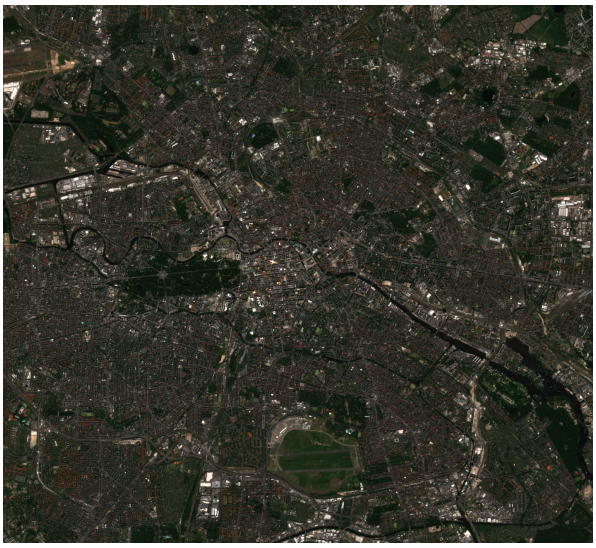
  All the logic is in the *download_image* method. We set parameters such as the time interval, the evalscript, maximum cloud coverage allowed and the bounding box (in Sentinel standard: west, south, east, north). It downloads the raw image without labels and can do so in .png (for RGB) and .tiff (for more channels) formats, depending on the specified extension and evalscript.

First, we show the image size to the user, as it is important to know, because SentinelHub only allows for images with width and height < 2.500 pixels for the free version.

Then, we proceed with the subtasks with the Berlin map:

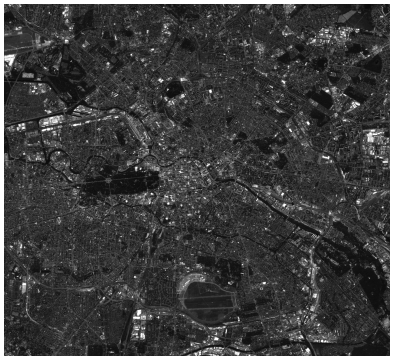
- Buildings visualization: show only the buildings in white with a black background (binary), using the OSM class.
- Sentinel raw satellite image visualization: using the SentinelHubImageDownloader class, we use 3 different evalscripts for RGB, IRB and grayscale, and show the images to the user with *imageio*.
- Overlapping buildings: we show in yellow the buildings over the grayscale image.

((a)) Buildings from Open Street Maps

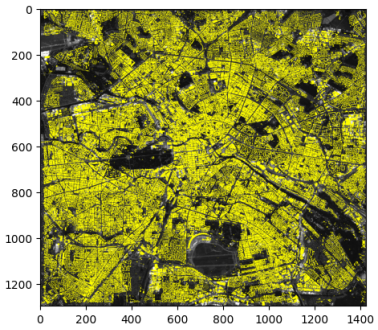


((b)) RGB Bands from Sentinel 2

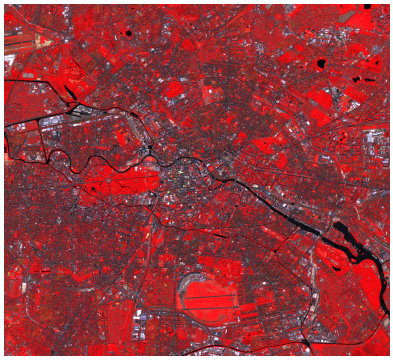
Figure 1: Latitude North: 52.574409 South: 52.454927 and Longitude West: 13.294333, East: 13.500205



((a)) Single Band



((b)) Overlapping



((c)) IRB

Figure 2: Comparison of RGB, Red, and Single Band Images

3 Task 2

For solving task 2, the following files are used: , , , . Briefly describing them:

- *data_pipeline.py*: This is the main file, which uses the rest. It is the pipeline that extracts, cleans, creates the patches and stores the data merged.
- *config/data_pipeline.yml*: This is where all parameters are specified, so that the data pipeline knows which cities to download data from, what is training and what is testing data, etc. It is easily customizable by the user and does not require changing the source code.
- *notebooks/task2.ipynb*: This notebook was created to perform a simple yet insightful Exploratory Data Analysis (EDA) to better assess which train/val split strategy to follow.
- *src/ml/data/split.py*: A specific file that contains the logic to perform this train/val split logic, named as *density* in the yml file.

3.1 The data pipeline file.

3.2 The yml file.

3.2.1 Functionalities

- Download: to download the data and store the original images data, as well as their respective patches.
- Merge: to take all patches of the same size of all the cities in a specified directory and concatenate them along the first axis.

They are separated to give the user the possibility to first download and then merge easily.

3.2.2 Cities

They are specified in the *cities* array, and divided into *train* and *test* sets. For each, the north, south, east and west coordinates of the bounding box has to be specified. As an example for Berlin city (used for testing):

```
cities:
- name: Berlin
  north: 52.574409
  south: 52.454927
  west: 13.294333
  east: 13.500205
  set: test
```

3.2.3 Sentinel API

Specify the parameters for the request, which are *resolution*, *max_cloud_coverage* and *evalscript* (the later one is specified as a string and decoded in a constant dict that references them).

```
sentinel:
  resolution: 10
  max_cloud_coverage: 5
  evalscript: ALL
```

3.2.4 Patches creation and storage

In patches, we can specify size, stride and the augmentations to perform, which can be both individual augmentations, and composed. Additionally, a saving directory is indicated.

```
patches:
     - size: 32
       stride: 6
       augmentations:
         - name: random_horizontal_flip
           parameters:
             p: 0.5
         - name: random_rotation
           parameters:
             degrees: 90
         - name: compose
           augmentations:
             - name: random_horizontal_flip
               parameters:
                 p: 0.5
             - name: random_rotation
               parameters:
                 degrees: 90
     - size: 64
       stride: 12
       augmentations: {}
     - size: 128
       stride: 24
       augmentations: {}
 patches_save_dir: data
```

3.3 The EDA notebook and train/val split based on density

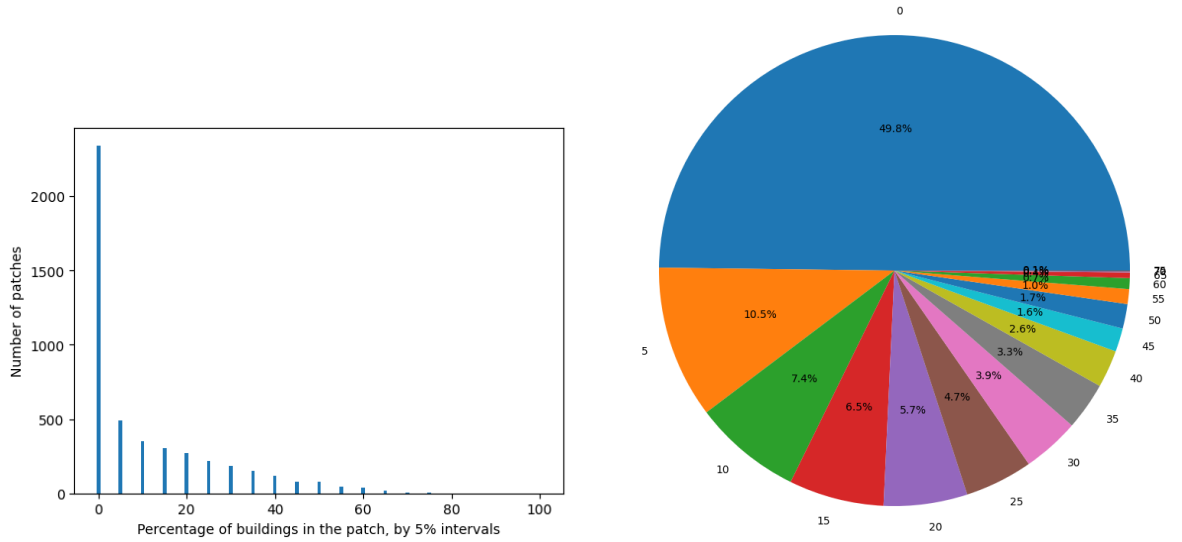
Following the intuition that there would be an imbalance in building numbers, we created a simple notebook that checks the distribution for all patch sizes.

This means that for each patch, we check the percentage of pixels cataloged as buildings (has a 1), and then group by intervals of 5%. So, we count how many, patches have from 0 to 5% of pixels corresponding to buildings, how many from 5 to 10%, etc. As well as some general statistics.

Hence, we code a strategy that, given the specified train/val split percentages, it picks random patches in a way that both sets have a very similar percentage of buildings with pixels, so that there is as little distribution shift as possible.

3.4 The data pipeline

- Load the *data_pipeline.yml* parameters.
- Download the data for each city:



((a)) Distribution of buildings for patch size 128 histogram

((b)) Distribution of buildings for patch size 128 pie chart

  - Extract the bounding boxes, and check if they are too big for the API call.
  - Download the Sentinel image and OSM buildings (labels).
  - Reshape the data to have a Pytorch shape like convention (C, H, W), instead of (W, H, C).
  - Save the original images so that if later different patches are needed, they can be extracted from the original images, avoiding a new download, in an *original_images*.
  - Create and store the patches: this logic is implemented in the *store_patches* function, which creates equal size squared patches with a given stride:
    * Checks the parameters specified, avoiding impossible situations such as having patches bigger than the image.
    * Pads the image: depending on the stride and the patch side size, it adds an equal size margin (padding) to both features and labels arrays of the images, so that finally all patches have the same size, and avoid cropping. This is done in a generalistic way, so that for any combination, the margin is correctly applied.
    * Select the patches that have no clouds: in SentinelHub, there is one channel named *CLM* which provides a cloud mask. If that channel contains some 1 for the selected patch, we discard it. After that, this channel is dropped, and only RGB is kept.
    * Apply the augmentations using the *AUGMENTATIONS_ENCODER*.
    * Save all patches of the same size processed together.

- Merge: traverse all patches of the same size and set (test or train) and concatenate on the first axis.

If *verbose* is set to True in the yml, logging will happen to show the user the progress of the download.

A possible result of running this pipeline would be:

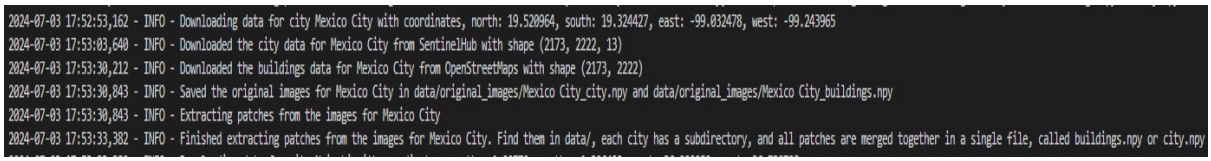


Figure 4: Logging example of downloading Berlin city data.

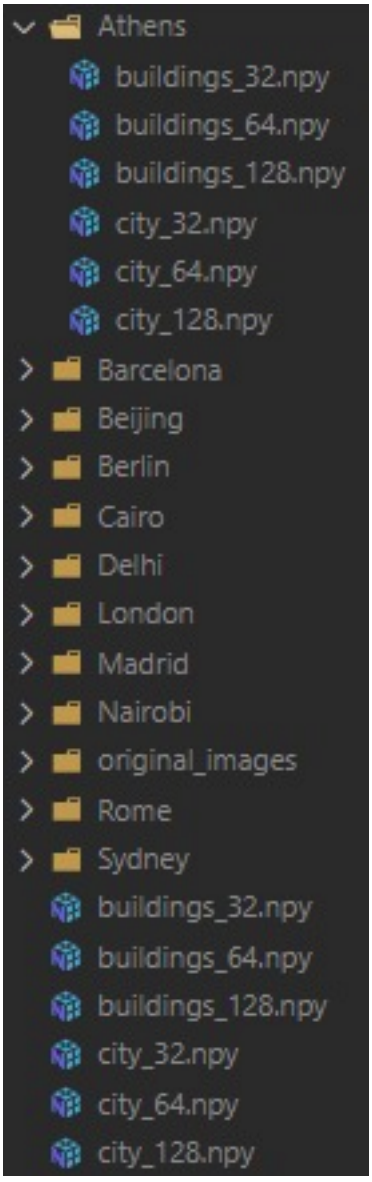


Figure 5: Example of training data.

4 Task 3

Task 3 employs the `ml_pipeline.py` and `ml_pipeline.yml`, together with the respective subfiles imported from the *src* directory.

4.1 The yml file

The yml file guarantees an easily adjustable configuration of experiments. It allows the configuration of general parameters, the model to be used, loss and optimizer to be used, sets parameters for the train, validation and test datasets and logging.
As an example:

```
# General configuration
device: cuda
seed: 42
verbose: true
```

```
# Model
model:
  name: UNet
  parameters:
    in_channels: 3
    out_channels: 1
loss:
  name: haussdorf
  parameters: {}
optimizer:
  name: Adam
  parameters:
    lr: 0.001
    weight_decay: 0.0001
```

```
# Training
train:
  mode:
    name: density
    perc: [0.7]
  data_dir: data/train
  patches_sizes: [128]
  hyperparameters:
    mode: manual
    batch_size: [32]
    epochs: [1]
    lr: [0.1, 0.001]
  early_stopping:
    delta: 0.01
    patience: 3
```

```
# Validation
val:
  metrics:
    - name: Dice
  threshold: 0.5
```

```
# Test
test:
  data_dir: data/test
  threshold: 0.5
```

```
# Logging and plotting
logging:
  dir: logs
```

Important options to note:

- *train mode*: specifies how to split the data between training and validation sets. If specified as *density*, the previously mentioned algorithm is run to assure no distribution shift. Else, *random* is specified and a normal sklearn split is performed.
- *model*: it specifies which model to use, either *Baseline* or *U-Net* model (implemented by us from scratch).
- *loss*: which loss function to use. Additionally to Pytorch's *BCEWithLogitsLoss* function, we implemented *Dice* and *Hausdorff* loss functions. The latter one is coded so that false negatives are heavily penalized. This is because we believe that is the biggest problem this challenge faces.
- *early_stopping*: we implemented early stopping to reduce overfitting risk and make training faster. Both *patience* and *delta* parameters can be tuned in the yml file.
- *logging*: where to store the logs produced during training and testing.

4.2 The ML pipeline workflow

After loading the yml values, the machine learning pipeline iteratively launches a different experiment sequentially for all possible combinations of the specified parameters' values:

- Patch size
- Train percentage

- Batch size
- Number of epochs
- Learning rate

The rest of the specified settings, such as optimizer, model, and verbose, are common to all experiments of the same execution.
For each experiment, a unique ID is created, together with its specific folder, which contains the training logs, model checkpoints every 5 epochs, the final model checkpoint, a yml with the experiment specified parameters, and most importantly, both csv and pngs which show for each epoch: metrics for validation and training, loss function and training time. This gives us the ability to easily inspect each setting and decide the best model.

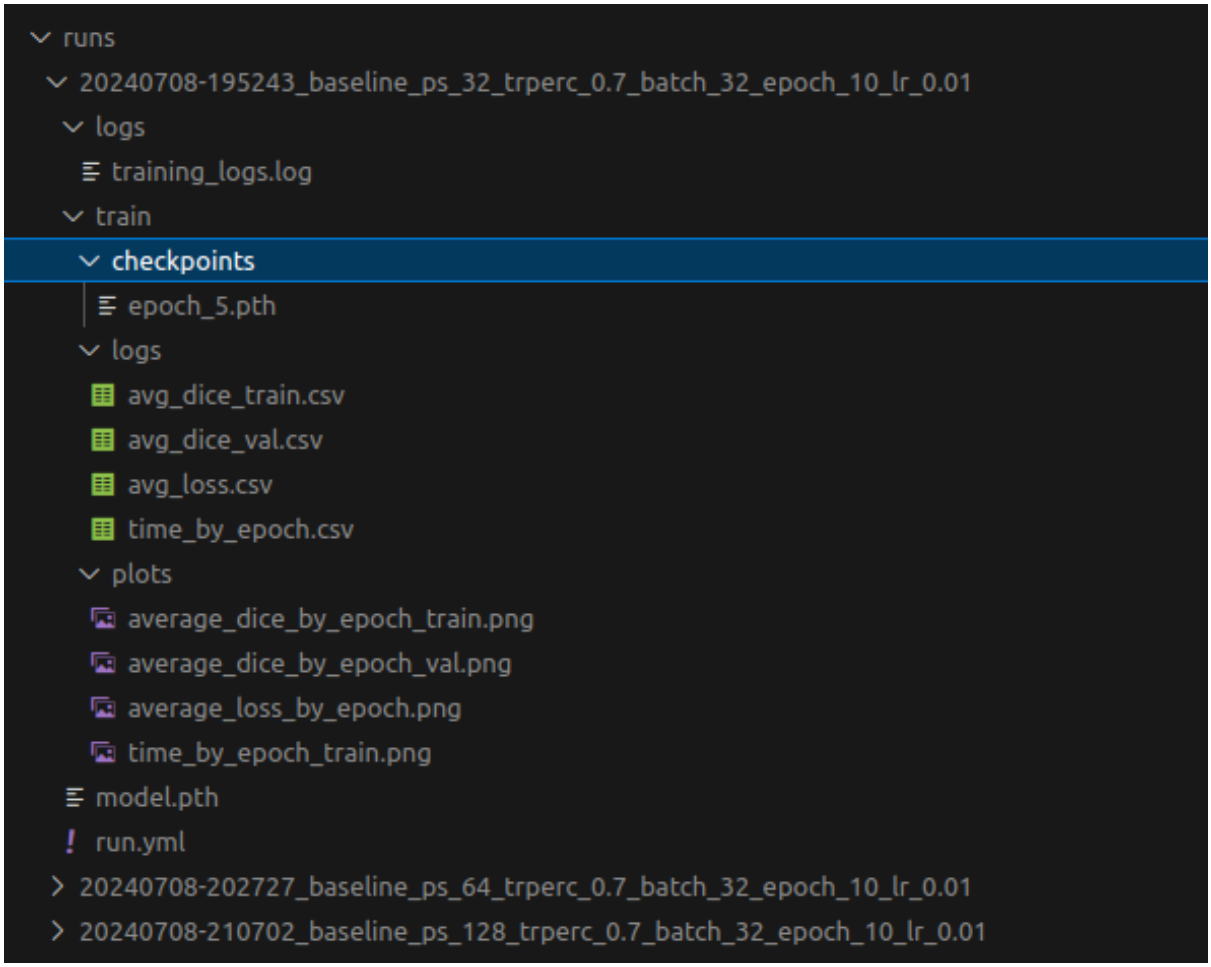


Figure 6: Produced folder contents for each experiment.

4.3 Experiments & results

We have tried, with both the baseline and U-Net models, all possible combinations of the parameters:

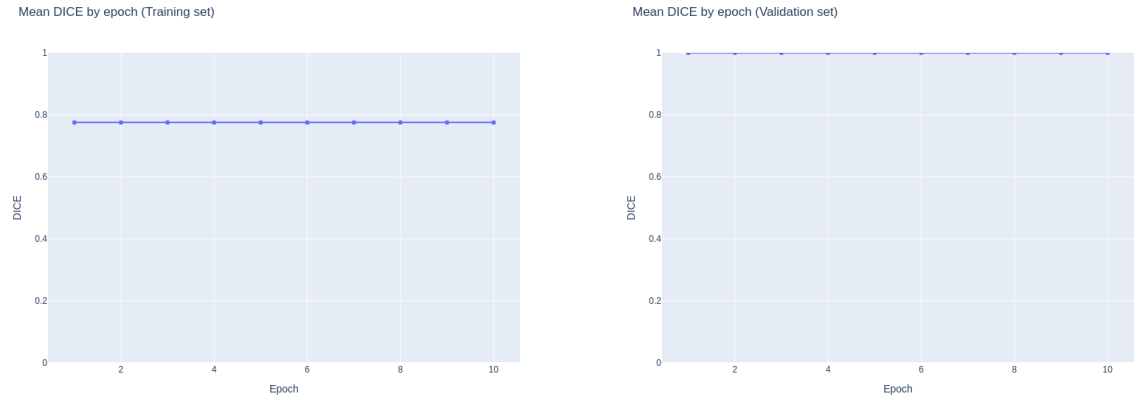
- Patch size: [32, 64, 128].
- Train percentage: [0.7, 0.8] (hence 0.3 and 0.2 for validation, and always with a density-based split mode).

- Batch size: [16, 32].
- Number of epochs: [10, 25, 50].
- Learning rate: [0.1, 0.01, 0.001]

Static parameters:

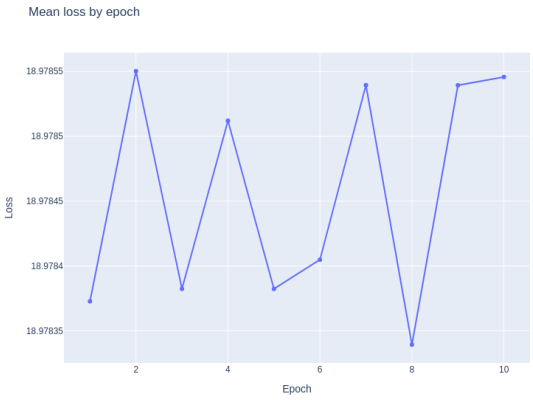
- Loss function: Hausdorff.
- Metric: Dice.
- Optimizer: Adam with a weight decay of 0.001.
- Dice threshold: 0.5.

So far, our results have been generally bad and weird. Hence, we here show some examples of runs, because we have always obtained somewhat the same outputs, no matter how many combinations we tried. At first, we thought the problem was the BCE loss function, so we implemented the other two mentioned, and still got the same results. Next, we experimented with multiple hyperparameters setups and 3 optimizers, but still got somewhat the same results, never reliable. So, we don't understand what was going on at the end.

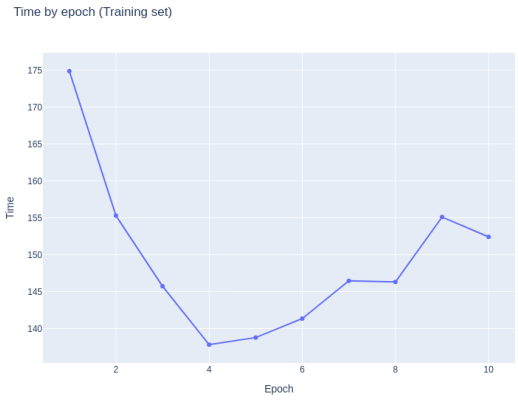


((a)) Average Dice by epoch in training ((b)) Average Dice by epoch in validation

Figure 7: Baseline model sample results for mean DICE by epoch.

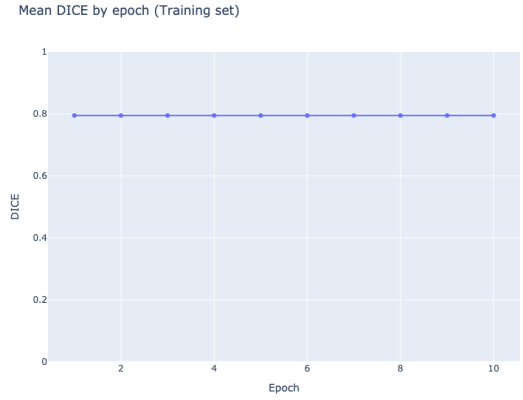


((a)) Average loss by epoch in training

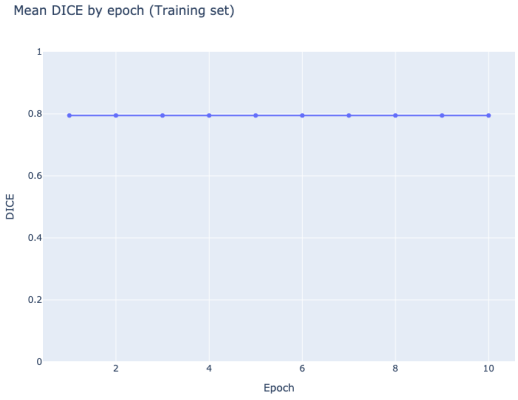


((b)) Train time by epoch

Figure 8: Baseline model sample results for loss and time in training.

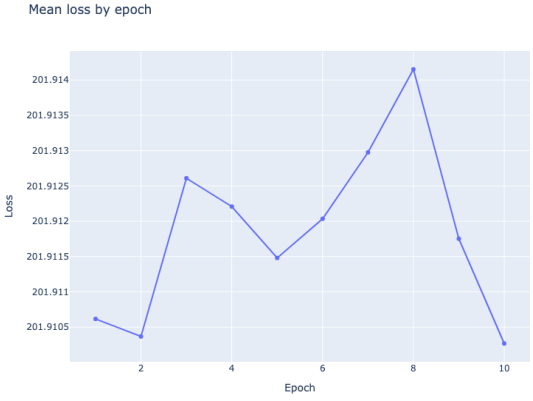


((a)) Average Dice by epoch in training



((b)) Average Dice by epoch in validation

Figure 9: U-Net model sample results for mean DICE by epoch.



((a)) Average loss by epoch in training

((b)) Train time by epoch

Figure 10: U-Net model sample results for loss and time in training.

5 Task 4

In the case of data augmentation, we try with both composed and individual augmentations, combining in the data pipeline, specially:

- Random rotations.
- Random horizontal and vertical flips.
- Color jitter.

We avoid transformations that change the shape of the image.
Augmentations are specified in the *data_pipeline.yml* file and therefore are applied during the data pipeline process, so the augmented patches are finally merged with the original ones, and no distinction is made during the train/val split.
The obtained results have been very similar, but we would rather use a model trained with augmented data, as it improves generalization power.

6 Conclusion

We implemented a complete ML architecture cycle: extract data, explore it, clean it, generate the dataset with patches, store it efficiently, load it fast, run an iterative ML training workflow and store the results of it.

Our focus points have been:

- To create 2 end-to-end pipelines that are easy to use, efficient and reliable.
- To produce high-quality code, that is clean and understandable by any new user, as well as easily extensible and well organized, with extensive and straightforward documentation.
- To provide the user with an easy to interact interface, which we achieved with the yml files, that allows anyone to easily understand and change the parameters for both pipelines.
- To provide a reliable dependency management system, for which we adopted poetry, a new manager, more sophisticated and less problematic than conda or pip virtual environments.
- To produce detailed and organized ML results reports, that not only clearly illustrate the training outcomes, but also allow the user to backtrack and load checkpoints of the model later.
- To have an informative system that lets the user knows how the pipelines are progressing, and which errors popped up later on, if any.

The implemented ML pipeline is adoptable and useful in practical applications like urban planning and development, environmental monitoring and land use changes.

While we did not achieve good results with the ML model, we firmly believe that from the architectural point of view, our work is spotless, user-friendly and extensible, which is at the end the main purpose of learning of the present course.