ML-based Building Detection on Satellite Imagery

Benjamin Hilliger (383579)
Technische Universität Berlin
b.hilliger@campus.tu-berlin.de

Joscha Bisping (401424)
Technische Universität Berlin
joscha.l.bisping@tu-berlin.de

Christian Stubbe (478387)
Technische Universität Berlin
c.stubbe@campus.tu-berlin.de

***Abstract*—This exercise report describes our solution for the Sentinel Building Exercise. Our baseline convolutional neural network is able to detect buildings on Sentinel-2 satellite imagery with a Binary Cross Entropy Loss of 0.369 after hyperparameter tuning for the Berlin region. In addition, we test a U-Net model architecture that performs with a Binary Cross Entropy Loss of 0.3259 on the same region after hyperparameter tuning. By applying data augmentation techniques, we show that the performance of the base model can be increased to a Binary Cross Entropy Loss of 0.351. This project report describes the pipeline design, model building, tuning, and data augmentation techniques used to achieve these results. First, we provide a summary of the given tasks. Then we describe our solution for the data acquisition pipeline and the data processing pipeline. Finally, we will describe the model creation, tuning, and the impact of the data augmentation techniques on the models. This ML pipeline can have an impact on urban planning, disaster management, and environmental monitoring by providing automated building detection from satellite imagery.**

I. Exercise Recap

The Sentinel Building Exercise is a component of the *Architecture of Machine Learning Systems* course offered by Prof. Matthias Böhm at TU Berlin during the summer term of 2024. The primary objective of this exercise is to develop a machine learning (ML) model capable of detecting buildings in satellite imagery. The specific task is to construct a data acquisition and preparation pipeline, train an ML model to classify the presence of buildings in satellite images, optimize the model through hyperparameter tuning, and apply data augmentation techniques to enhance model quality.

The satellite imagery utilized in the project were acquired in the Sentinel-2 Copernicus program. Sentinel-2 is a constellation of two identical satellites in the same orbit that can be used to photograph land and coastal areas. The Copernicus program is financially supported by the Member States of the European Union, the European Space Agency (ESA), the European Organization for the Exploitation of Meteorological Satellites (EUMETSAT), the European Centre for Medium-Range Weather Forecasts (ECMWF), EU agencies and Mercator Ocean International[1].

The exercise mandates that training data for this project must be obtained from Open Street Maps[2]. Open Street Maps is a community-maintained project that provides a geographic database. The Open Street Maps database contains location vectors for buildings around the world. The data quality in the database in Open Street Maps depends on the geographic region, with the highest quality in countries like Germany, UK and USA [1].

The model is required to be a convolutional neural network (CNN) with the first three layers having a width of three kernels and a single step. The task further specifies that each CNN layer should increase the number of channels to 32, 64, and 128. The final CNN layer must produce a single-valued channel. In addition, either U-Net, SegNet, FastFCN, or Gated-SCNN must be tested as another model architecture. We chose the U-Net model architecture.

Finally, the task requires the application of data augmentation techniques such as rotation, reflections, and noise to improve the robustness and accuracy of the model. The impact of these augmentation techniques on the model performance must be evaluated and documented in this report. During this step, the hyperparameters of the model must remain unchanged.

This project report describes our work on the exercise and explains the architectural decisions we made in pipeline design and model building. We also provide statistics on model performance, describe the impact of hyperparameter tuning, and evaluate data augmentation techniques.

II. How To Run the Code

The source code is available on GitHub[3]. A README.md provides instructions how to run the code. The dependencies can be installed from requirements.txt. The pipeline and model creation can be run from main.ipynb. We recommend a machine with a GPU, at least 24 GB RAM and additionally 16 GB swap memory available. We provide the processed images from Sentinel in tubCloud for a streamlined replication of our model results[4].

III. Solution Architecture

The solution follows the structure given in the exercise description. Each subtask (data acquisition, data preparation, modeling and tuning, and data augmentation) is available in its own directory. The starting point for code execution and execution monitoring is a Jupyter notebook named main.ipynb. This file calls all functions that are part of the data pipeline, model creation, tuning, and data augmentation. In the utilities folder the file plot_utils.py provides functions for plotting model results and data acquisition results. We used Python

[1] https://sentinels.copernicus.eu
[2] https://www.openstreetmap.org
[3] https://github.com/christianstubbe/architecture-of-ml-systems
[4] https://tubcloud.tu-berlin.de/s/2ryzsBY27bSztoL

as the programming language for this exercise because it is the industry-standard programming language for machine learning.

The exercise requires downloading satellite imagery and building location data for at least ten major cities. We have selected the following cities: Munich (Germany), Frankfurt (Germany), Hamburg (Germany), Aachen (Germany), Vienna (Austria), Paris (France), London (United Kingdom), Singapore (Singapore), Seoul (South Korea), Sidney (Australia), Bogotá (Colombia), Santiago (Chile), Montreal (Canada), Johannesburg (South Africa) and Cape Town (South Africa).

The choice was based on the following rationales:

- Open Street Maps provides the highest quality labeling for cities in Western Europe and North America [1]. So we choose half of our cities from these regions.
- We exclude all cities with more than 10 million inhabitants to reduce the hardware requirements for this pipeline.
- The exercise requires us to evaluate the model on satellite images of Berlin. Therefore, we have included four German cities that have similar building and layout characteristics to Berlin.
- For variety and to ensure that the model will work well in other regions, we include cities from all continents with large populations.
- To get the most training data, we select large cities in each continent that do not violate the previously defined constraints.

The Jupyter Notebook provides functionality to configure a logger and the logging level. By default, the logging level is set to ERROR. Logs are stored in the main.log file in the main directory. The logger is implemented in the folder utilities in the file plot_utils.py.

IV. Data Acquisition

The pipeline for the data collection subtask provides three core functions. First, it collects data from Open Street Maps and Sentinel satellite imagery for the cities specified in main.ipynb. Then it computes binary masks that contain information about whether or not a building is present at a given pixel in Open Street Maps. Finally, the pipeline provides functionality to plot and provide feedback to visually verify successful data collection for any given city.

The code for this part of the ML pipeline runs in the data_acquisition folder. The folder contains the file datahandler.py with the class DataHandler to provide a more structured and encapsulated approach. The class also allows for better debugging and testing. The functionality of the DataHandler class is documented in the figure 1. Table I contains an overview of the local files that are stored for each city as part of the execution of the data acquisition pipeline.

As described earlier, the first core functionality of the data acquisition pipeline is to retrieve Open Street Maps and Sentinel satellite imagery. During initialization, the DataHandler checks to see if a directory named *data* exists. If the directory does not exist, the DataHandler will create it. In the data directory, the pipeline creates a folder with the city name. If a directory for the given city already exists, the DataHandler checks for each city whether Open Street Map data and Sentinel imagery exist locally. If local data exists, the local data is used. The pipeline downloads Open Street Maps data using pyrosm, a Python library for reading and storing Open Street Maps in a protocolbuffer binary format[5]. After downloading the complete set of Open Street Maps for a given city, the buildings in the data are filtered out using the built-in get_buildings() function in pyrosm. It also creates a bounding box around the city to use as coordinates when downloading the satellite image. Both the protocol buffer binary data and the filtered buildings are stored in the subfolder for each city. The satellite images are downloaded from openEO, an API that unifies multiple Earth observation backends, such as Sentinel data, in a simplified and consistent way. An account on the openEO platform is required to retrieve Sentinel data and run this project[6]. OpeneoEO also provides functionality to automatically filter images with snow and cloud cover. The table II provides an overview of the settings used to retrieve the images. Based on the specified temporal extent over 2023, we used the built-in functionality of openEO to reduce the

File	Description
city.osm.pbf	Open Street Maps data
boundaries_mask.tif	Mask with pixels containing boundaries
bounds.pkl	Store bounding box
building_mask_dense.tif	Mask with pixels partly containing buildings
building_mask_sparse.tif	Mask with pixels fully containing buildings
buildings.geojson	Vectors of Open Street Maps buildings
job-results.json	openEO job information
openEO.tif	Satellite image from openEO

TABLE I
LOCAL FILES CREATED IN DATA ACQUISITION

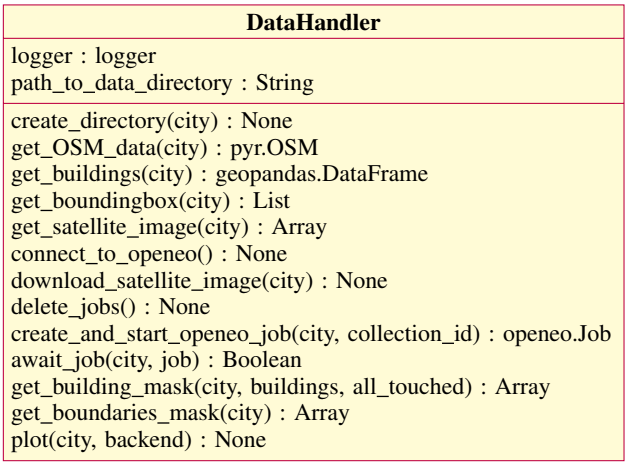

Fig. 1. DataHandler UML diagram

[5]https://pyrosm.readthedocs.io
[6]https://aai.egi.eu/registry/co_petitions/start/coef:327

data to the median pixel value in 2023.

Setting	Value
Temporal Extend	2023-01-01, 2023-12-31
Bands	02, 03, 04, 08, 11, 12, SCL
Max. Cloud Cover	30
Spatial Resolution	10
Collection	SENTINEL2_L2A

TABLE II
OPENEO SETTINGS

After retrieving the satellite imagery and the filtered buildings within the bounding box of the city, the pipeline creates two types of binary masks for the city: a dense mask and a sparse mask. Both masks contain binary information about whether or not a building is present at a given pixel. Pixels in the dense mask are stored with a positive value if the pixel contains at least parts of a building. On the other hand, the sparse mask stores a positive value only if the pixel contains a complete building. We distinguished between these two types of masks because of the high density of buildings in Berlin, the city used for model evaluation. Since the Sentinel imagery is downloaded at a resolution of 1 pixel per 10 meters, these two masks are very different. Both masks are stored as TIFF files in the city subfolder. When creating the model, we achieved a higher model quality with the dense mask.

The final core functionality of the data acquisition pipeline is to plot the images and buildings for the city to visually verify successful data acquisition. The exact plots are specified in the task description: a plot of individual bands, buildings, RGB images, IRB (Infrared, Red, and Blue), and overlapping buildings.

To verify that the satellite image and binary mask are correctly aligned, we also support plotting with a plotly backend. This has the advantage of making the image display interactive. Although this is a simplified plot, it allows you to zoom into different parts of the image and manually check that the satellite pixels and building masks are properly aligned.

V. Data Preparation

The data preparation pipeline preprocesses the building masks and satellite imagery acquired in IV into loaders for test, training, and validation tensors. The pipeline also checks the distribution statistics. This section explains and justifies the selection procedure and how the similarity of the distributions between the test, training and validation data is maintained.

Data preprocessing can be performed from main.ipynb by creating an instance of CityDataset defined in the data_preparation folder. Figure 2 shows a UML diagram of the class. As requested in the task description, CityDataset can create patches of any size. The patches of these images are filtered if the pixels are within the OSM boundaries of each city. Additionally, each patch must be covered by at least 10% buildings, since the goal of the model is to detect buildings. Areas that do not contain buildings, such as water, forests, and fields, are not relevant to the modeling task. Images with snow and clouds are already filtered out in the data acquisition pipeline using built-in functionality of the openEO datacube. During data preparation we also noticed that some values in the images are negative. We replaced all these values with zero values.

To select the training and validation data, we used the Stratified Shuffle Split from scikit learn. The test data will be the region of Berlin, as specified in the task description. The Stratified Shuffle Split function does not guarantee that the shuffles will be different. Therefore, we performed two additional manual checks to ensure the similarity of the distributions across the training and validation data. First, we ensure that each set contains a similar percentage of pixels containing buildings. Then we further ensure similarity by checking that the distribution of cities is the same, e.g. if 10% of the data in the training set is from Paris, then the validation set should also contain about 10% of data from Paris.

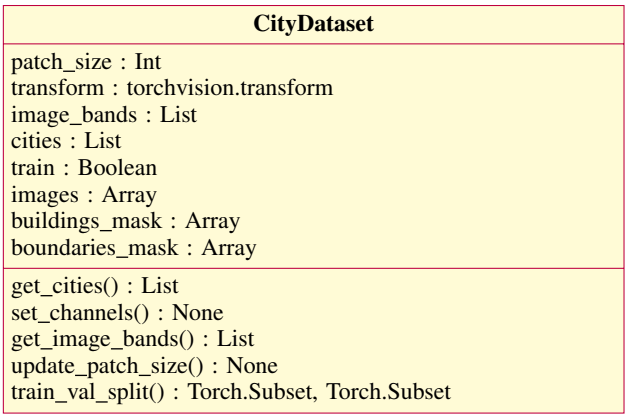

Fig. 2. CityDataset UML diagram

VI. Modeling and Tuning

Our model building and tuning code is organized in the model folder. The folder consists of three subfolders: base, u-net, and lightning. For both the base and u-net models, we used PyTorch Lightning, a framework that organizes PyTorch code to make it more readable and flexible. During hyperparameter tuning, we tested Adam and AdamW as optimizers in addition to stochastic gradient descent. We found that AdamW performed best on both the base model and the U-Net model (see table IV and table V). Along with the optimizer selection, we chose different bands as input data for the model.

As requested in the exercise description, we implemented the base model as a simple four-layer convolutional neural network (CNN). The architecture consists of three convolutional layers with kernel sizes of 3x3 and increasing channel depths of 32, 64, and 128, respectively, followed by a final convolutional layer with a 1x1 kernel that produces a single output channel to predict building presence. This model uses ReLU activation functions and padding to preserve spatial dimensions. For the output layer, we used a sigmoid activation function to output values between zero and one. As a loss

function, we used Binary Cross Entropy (BCE) loss because our problem is a binary classification problem [2].

First, we trained the base model without any hyperparameter tuning. After training to 100 epochs, the model performed with a BCE loss of 0.384 on the test dataset. Out of the maximum 100 epochs, training was automatically stopped after 21 epochs. The table III shows the training settings. To ensure the distribution of the training and test data, we checked the mean, std, min and max percentages of the labels in the training, validation and all data.

Setting	Value
Max epochs	100
Early stopping patience	2
Loss	BCE loss
Batch size	32
Patch size	32x32
Num Workers	20
Validation set	10% of dataset
Training samples	43038
Validation samples	4783
Number of channels	6
Len of train dataloader	1345
Len of val dataloader	150
Initial loss by tuner	0.0016143585568264868

TABLE III
MODEL TRAINING SETTINGS

After training without any hyperparameter tuning, we used a tuner to find an optimal initial learning rate of 0.00161. Then we trained with three different optimizers and five randomly selected channels. Three to six channels were chosen per model. Table IV shows the results of the hyperparameter tuning. Based on these results, we chose Adam as the optimizer for our model. Figure 3 visually shows the building detection results for Berlin for the base model with a threshold of 0.5.

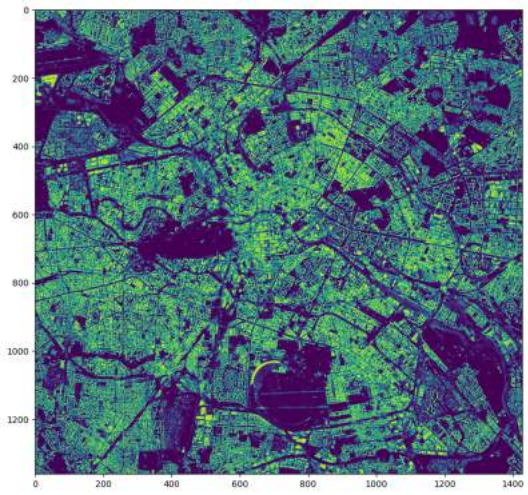

Fig. 3. Base Model Results for Berlin

Bands Used	Optimizer	BCE Loss
1, 2, 3, 4, 5, 6	Adam	0.369
1, 2, 3, 4, 5, 6	AdamW	0.4656
1, 2, 3, 4, 5, 6	SGD	0.4483
1, 2, 4, 6	Adam	0.4041
1, 2, 4, 6	AdamW	0.3899
1, 2, 4, 6	SGD	0.5196
1, 4, 5, 6	Adam	0.4041
1, 4, 5, 6	AdamW	0.3919
1, 4, 5, 6	SGD	0.4648
2, 5, 6	Adam	0.4492
2, 5, 6	AdamW	0.4615
2, 5, 6	SGD	0.4922
3, 4, 5, 6	Adam	0.4157
3, 4, 5, 6	AdamW	0.3933
3, 4, 5, 6	SGD	0.4707

TABLE IV
BASE MODEL HYPERPARAMETER TUNING RESULTS

In addition to the base model, we implemented another model based on the U-Net architecture, which is well suited for segmentation tasks due to its encoder-decoder structure and skip connections [3]. The encoder part of the U-Net progressively reduces spatial dimensions while increasing feature depth, and the decoder part restores the original dimensions while merging features from corresponding encoder layers.

We also initially trained the U-Net model without any hyperparameter tuning and with the same settings as the base model (III), except for the initial learning rate set by the tuner. For the U-Net model, the tuner set the initial learning rate to 0.00085. Training for the U-Net model stopped after nine epochs with a validation BCE loss of 0.354, a training BCE loss of 0.339, and a testing BCE loss of 0.354.

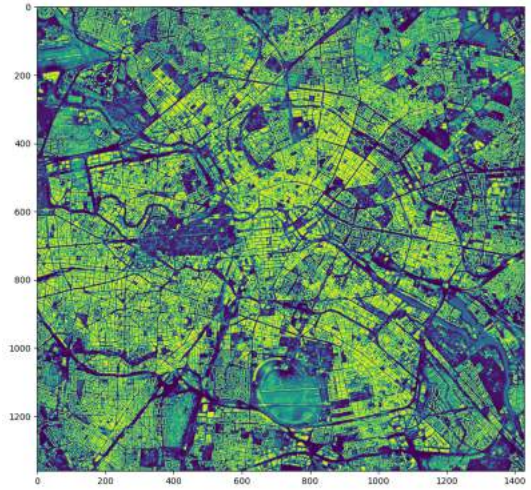

Fig. 4. U-Net Model Results for Berlin

Hyperparameter tuning improved the performance of the U-Net model. We used the same strategy as for the base model: three different optimizers and five randomly selected channels,

Bands Used	Optimizer	BCE Loss
1, 2, 3, 4, 5, 6	Adam	0.3259
1, 2, 3, 4, 5, 6	AdamW	0.3627
1, 2, 3, 4, 5, 6	SGD	0.6267
1, 2, 4, 6	Adam	0.3192
1, 2, 4, 6	AdamW	0.3542
1, 2, 4, 6	SGD	0.3311
1, 4, 5, 6	Adam	0.3405
1, 4, 5, 6	AdamW	0.3427
1, 4, 5, 6	SGD	0.3295
2, 5, 6	Adam	0.2455
2, 5, 6	AdamW	0.4157
2, 5, 6	SGD	0.9228
3, 4, 5, 6	Adam	0.3327
3, 4, 5, 6	AdamW	0.336
3, 4, 5, 6	SGD	3

TABLE V
U-NET MODEL HYPERPARAMETER TUNING RESULTS

with three to six channels to be selected per model. The results of the hyperparameter tuning can be seen in table V. Figure 4 visually shows the building detection results for Berlin for the U-Net model with a threshold of 0.5.

In conclusion, both the base CNN and U-Net model showed a slight improvement with hyperparameter tuning. Based on the hyperparameter tuning results, Adam should be selected as the optimizer and all input channels of the Sentinel images should be selected for optimal model performance. The U-Net model performs slightly better than the base model.

VII. DATA AUGMENTATION

Data augmentation techniques are used in ML pipelines to improve model performance and generalization by enlarging the training dataset based on augmenting the existing data. This includes transformations such as rotation, scaling, and mirroring of existing data. By introducing variability, data augmentation can reduce overfitting and improve model robustness, especially in scenarios with limited data [4].

Augmentation Technique	Val Loss	Train Loss	Test Loss
Base Performance	0.384	0.375	0.369
Individual Augmentations			
Color Jitter (CJ)	0.385	0.383	0.357
Flip, Rotate X and Y (FRR)	0.396	0.386	0.351
Compound Augmentations			
CJ, FRR	0.389	0.384	0.362
CJ, 2x Random FRR	0.391	0.389	0.368

TABLE VI
DATA AUGMENTATION TECHNIQUES FOR THE BASE MODEL

To improve our model, we implemented two data augmentation techniques (Table VI). Color Jitter (CJ) is based on torchvision.transforms.ColorJitter, which randomly changes the brightness, contrast, saturation, and hue of an image. However, we have modified torchvision.transforms.ColorJitter to support six channels on the image. Flip, Rotate X, Rotate Y (FRR) randomly rotates, randomly flips horizontally, or randomly flips vertically the image. Random(FRR) randomly selects either Flip, Rotate X, or Rotate Y. Not selecting any of these three data augmentation techniques is also an option.

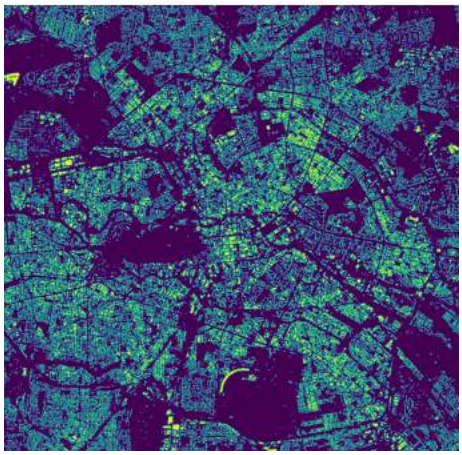

Fig. 5. Base Model Results for Berlin with Color Jitter and 2x Random Flip, Rotate X and Rotate Y. Threshold 0.9

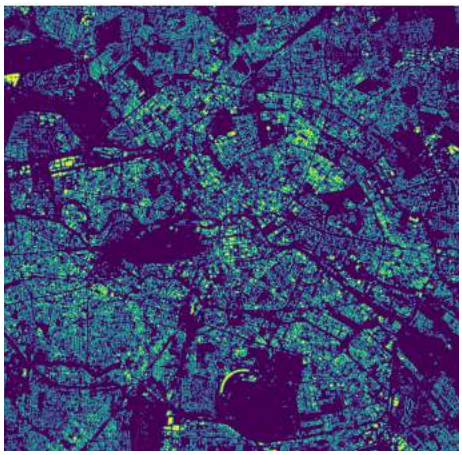

Fig. 6. Base Model Results for Berlin without Data Augmentation Threshold 0.9

We tested the impact of the data augmentation techniques individually and as a compound combination. Table VI shows that the data augmentation techniques individually and in combination slightly improve the model performance. Verifying these results by looking at the visual model results in Figure 5 and Figure 6, we can see that the model detects buildings with more confidence when data augmentation is applied.

Due to computational constraints, we only applied data augmentation to the base model. We expect to get similar results with data augmentation on the U-Net model.

VIII. CONCLUSION

This exercise report described an ML pipeline capable of detecting buildings from Sentinel-2. It explored the model

architecture of a basic CNN and a U-Net based model. It also provided statistics on the impact of data augmentation techniques on the model. To further optimize this ML pipeline, we see five approaches: First, there is potential to further improve the existing model by exploring additional data augmentation techniques. Next, the selection of cities used as training data could be systematically explored, with the potential to significantly increase the number of cities in the training data. Then, the binary building mask used to decide whether a given pixel is a building pixel could be fine-tuned by changing from a binary mask to a mask that contains a percentage value of how much of the pixel is covered by buildings. Finally, with additional computing power, more complex models could be trained.

IX. Contributions

The source code and this report were written in equal parts by Joscha Bisping, Benjamin Hilliger, and Christian Stubbe.

References

[1] S. S. Sehra, J. Singh, and H. S. Rai, "A systematic study of openstreetmap data quality assessment," in *2014 11th International Conference on Information Technology: New Generations*, 2014, pp. 377–381.

[2] U. Ruby and V. Yendapalli, "Binary cross entropy with deep learning technique for image classification," *Int. J. Adv. Trends Comput. Sci. Eng*, vol. 9, no. 10, 2020.

[3] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[4] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, 2019.