AMLS Report

Dimitrij Schulz

TU Berlin, dmtschulz@gmail.com

CCS CONCEPTS • Computing methodologies~Machine learning~Learning paradigms~Supervised learning~Supervised learning by classification • Computing methodologies~Artificial intelligence~Computer vision~Computer vision problems~Object detection • Computing methodologies~Artificial intelligence~Computer vision~Computer vision problems~Image segmentation

A summary of how to run the code:
Open the file 'AMLS_2024_solution.ipynb'. Start by running the first cell to import all the necessary libraries and modules. Execute cell after cell. For 'connect_to_openeo' you must authenticate on the Copernicus page. After all the TIF files have been downloaded from the Copernicus provider you have to rename the downloaded TIF files into 'City_name.tif'. So, each city from the "city_names" list should have a corresponding TIF file in a city folder, e.g. acquisition_alignment/openeo_cities/Berlin/Berlin.tif, acquisition_alignment/openeo_cities/Bremen/Bremen.tif and so on for all 11 cities. After train, validation and test data has been generated you can specify learning parameters in 'params' variable and adjust BATCH_SIZE. After BaselineCNN has been trained you can go back and adjust parameters. Modify model_id to not override the trained model, if you want to train with other parameters. With more than one dictionary in the params list, you can train multiple models simultaneously.
After training go to the next cell to evaluate the model on the test data. If you trained more than one model, don't forget to adjust the model_id = params[**i**]["model_id"] to evaluate concrete model. Next train/evaluate the UNet model. At the end you chose a transformer, for example "train_transformsHF" and pass it to CustomTensorDataset() as transform parameter to apply data augmentation during training. Train and evaluate the model with augmentations.

1.1 Data Acquisition and Alignment

In this sub-task, we will create a pipeline to collect data for the model based on latitude and longitude coordinates. First, we'll download OpenStreetMap (OSM) files and create map projections of the buildings contained in these files.

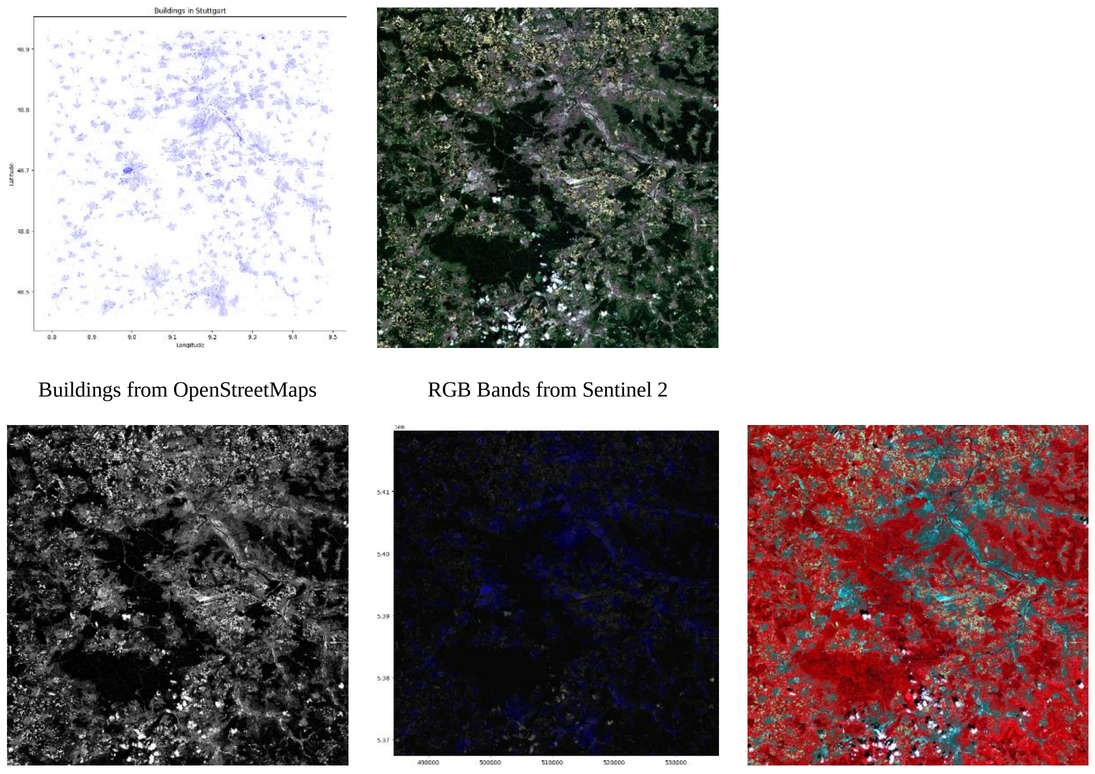
As a first step, we will define 10 major cities for training and 1 (Berlin) for testing. The cities are Berlin, Koeln, Frankfurt, Stuttgart, Dortmund, Duesseldorf, Bremen, Leipzig, Dresden, Muenchen, Hamburg. Using Pyrosm, we will download an OSM file for each city into a folder called pyrosm_cities.

Next, we will load buildings and bounding boxes for each city into a dictionary by extracting the buildings from the OSM data files. Once the building data has been extracted, we will convert the buildings into a GeoDataFrame, including only polygons and multipolygons to maintain consistent entry types. Buildings on the boundaries of the OSM file will be excluded.

After obtaining the buildings, we will extract the boundaries from the OSM data files and save them as a list. This list represents the bounding boxes of the cities.

The next step is to download satellite images for the cities using the OpenEO Data Space Copernicus provider. We will set date intervals and cloud coverage percentages (6%) for this provider. With bounding boxes (bb) from the boundaries list for each city, we will pass the coordinates to the provider, specifying the collection ID (Sentinel-2 L2a), spatial extent (bb), temporal extent (2021-06-01T12:00:00Z to 2021-07-30T12:01:00Z), and four bands (RGB and near-infrared). We will then download the .tif image files. Since multiple .tif files may be downloaded for the specified period, I will manually choose the ones with the least cloud coverage or no clouds at all.

In the final step, we will plot the buildings from OpenStreetMap, overlay them onto the images with different bands, and create a visual representation of the buildings on the satellite images, as shown in Figure 1.



Buildings from OpenStreetMaps RGB Bands from Sentinel 2

Single Band Overlapping Buildings IRB

Figure 1: Data Acquisition and Alignment Plotting of Stuttgart: Latitude North: 48.93000030517578, South: 48.46000671386719 and Longitude West: 8.790005683898926, East: 9.499994277954102

1.2 Data Preparation

In this task, we will preprocess the input data to prepare it for model training and testing. One city is used for testing, eight cities for training, and two for validation. As previously mentioned, for each city, we have a TIF image file and building data from the OSM data frame. We open the image with the Rasterio library and first convert the buildings to the satellite coordinate reference system, which can be found in the metadata of the TIF file. Then, we create a mask for each city. These masks will act as labels during training, where the mask is a grayscale image with white pixels representing buildings and black pixels representing no buildings. After creating the mask, we save it as a TIF file. This process is repeated for all 11 cities. Now, for each city, we have a satellite image and a corresponding mask image with the buildings of this city as TIF files. The next steps involve dividing the images and masks for the eight training cities into small patches.

Additionally, we calculate the building distribution for the test city to ensure that the training and test data have a similar distribution of buildings. Specifically, the building pixel ratio for the test label is around 0.20 (20% of pixels are buildings on a mask). I set the tolerance to ±0.85, which essentially filters out training patches that almost lack buildings or contain too much building. Lower tolerance for patches is at ~0.03 and an upper tolerance is at ~0.37. This step is crucial because having a similar distribution of buildings in both the training and test datasets helps ensure that the model can generalize well. If the training data had significantly different building distributions than the test data, the model might perform poorly on test data i.e. real data. By maintaining a consistent distribution, we improve the model's ability to accurately detect buildings, leading to better overall performance and reliability.

The next step in creating the training set is to make small patches from the eight large images and masks of each city. I chose a patch size of 128x128 (width x height). In the last task, we can utilize data augmentation and reduce the patch size to 64 or 32. To create image patches and mask patches, we read the TIF image file and TIF mask file, set the patch size, and then go over the large image in patch-sized steps from left to right and top to bottom. It's worth mentioning that there was a small margin on the boundaries—a black area without buildings—so the patching began not from zero but from the patch size to skip the black area on the borders. During this process, we check the building distribution for each mask. If it's within our threshold, we add the patch to a list. After collecting all the patches, we convert the list to a NumPy array. Finally, we ensure that the shape of the image patches is [N, C, H, W] and for masks [N, H, W], where N: Number of Images, H: Image Height, W: Image Width, and C: Number of Channels. After that, we save the images and masks as NumPy files. We have 3932 images and masks patches for training.

The next step is to create a validation set. This is done by choosing the last two cities, Hamburg and Munich, for validation and cropping a region from these cities. For simplicity, I opened the TIF files of the two cities in QGIS, chose a point, and cropped an area of height and width relative to the point. The height and width of the cropped area are the same as the height and width of the test city. After cropping the city, I create a mask with buildings for the cropped area and save the mask as a grayscale TIF file. This process is repeated for both cities. Similarly to the training set, I load the images and masks for both cities into a NumPy array and save them as NumPy files with the same dimensions as the training set.

The image and the buildings mask for the test city has been created in the beginning of this task. Its image and mask have been also saved as a NumPy file.

At this stage we have a disjoint train, validation and test set.

1.3 Modeling and Tuning

For the third task we first load the train patches, validation patches and test data from previously saved NumPy files, convert them into PyTorch tensors and send them to GPU or CPU device, depending on which one is available, so in the future during training the model, train and validation data would be on the same device. We created TensorDataset and DataLoader for training and validation.

The first model is the BaselineCNN model, a convolutional neural network (CNN) designed to detect the presence or absence of a building in each pixel of an image. It begins by accepting an input with a specified number of color channels, indicated by the in_channels parameter.

The network consists of four convolutional layers. The first layer applies 32 filters of size 3x3 with padding of 1 to maintain the input size, followed by a ReLU activation function to introduce non-linearity. This is followed by a second convolutional layer that uses 64 filters of size 3×3, also with padding of 1, and another ReLU activation function.

The third convolutional layer increases the number of filters to 128, again with a 3×3 kernel size and padding of 1, followed by a ReLU activation. The final layer is a convolutional layer with a single filter of size 1×1 and no padding, which reduces the depth of the feature maps to a single output channel. Unlike the previous layers, this final layer does not apply a ReLU activation, as it needs to produce raw output values suitable for binary classification tasks, where each pixel's output indicates the presence (1) or absence (0) of a building.

The increasing filter sizes in the convolutional layers allow the model to capture various hierarchical features. The initial layers detect low-level features like edges and textures, while deeper layers capture more complex structures such as shapes and patterns related to buildings.

In the forward pass, the input x is sequentially processed through each convolutional layer, with ReLU activations applied after each of the first three layers. The final output is produced by the last convolutional layer without further activation, resulting in a single-channel output map indicating building presence on a pixel-by-pixel basis.

For the loss I chose nn.BCEWithLogitsLoss() function. This loss combines a sigmoid layer and the binary cross-entropy loss in one single class. The sigmoid layer maps the raw output values from the final convolutional layer to a probability range of 0 to 1.

The binary cross-entropy part of the loss function then measures the difference between the predicted probability and the actual binary label (0 or 1) from the ground truth mask.

The primary metrics used for validation are the validation loss and Intersection over Union (IoU). The validation loss is calculated as the average of the loss values over all samples in the validation dataset. IoU measures the overlap between the predicted and ground truth masks, providing insight into the accuracy of the model's pixel-wise predictions.

The ML Pipeline works as follows: First train_models function is called, which can train more than one model with different parameters in parallel. Inside, create_model function is called and an instance of the model is created based on provided parameters. Next, the train function is called which contains a usually PyTorch train loop, call of the validation function and other functions used for model, plots and metrics savings.

The model is then evaluated on the test image to get an accuracy on unseen data. Predicted pixel values are compared with ground truth ones. Table 1 summarizes the results of the BaselineCNN. Hyper-parameter tuning is performed after 1 trial. Figure 2 represents a plot with train vs validation loss for the first trial i.e. without hyperparameter-tuning. Figure 3 represents the test image, the predicted and ground truth mask for the first trial with an accuracy of 0.8355.

Table 1: Hyper-parameters of the BaselineCNN.

Trial	Epochs	Batch Size	C. Channels	Optimizer	Learning Rate	Weight Decay	Early Stopping (patience, delta)	Avg Val. Loss	IoU
1	10	32	4	Adam	0.001	0	(4, 0.05)	0.3463	0.29723
2	4	32	4	Adam	0.0005	0	(1, 0.05)	0.4447	0.23119
3	4	16	4	Adam	0.0008	0	(1, 0.05)	0.3214	0.30085
4	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3134	**0.37869**
5	4	8	4	Adam	0.0009	6e-4	(1, 0.05)	0.3121	0.33891
6	4	8	3	Adam	0.0008	5e-4	(1, 0.05)	0.3665	0.14472
x	30	4	4	Adam	0.00001	5e-4	(5, 0.005)	0.4175	0.32484

Adaption of Hyper-parameters: The blue highlighted cells indicate the changed hyper parameter for the given trial.

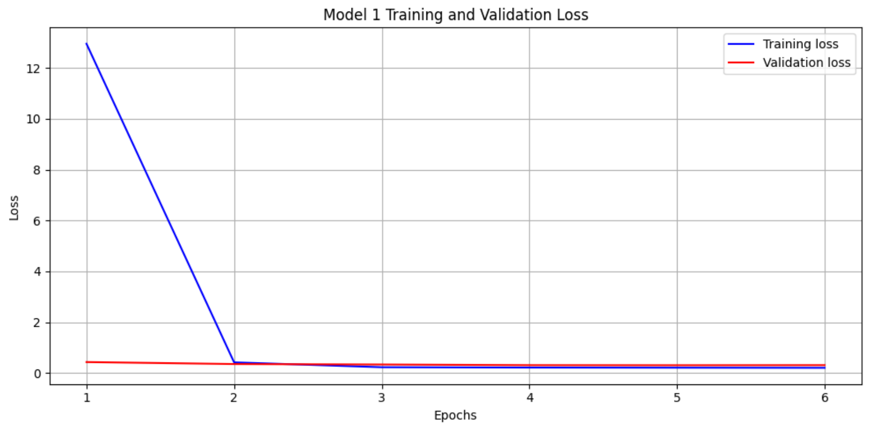


Figure 2: Train vs validation loss of the BaselineCNN. 1 Trial. Indicating fast convergence and almost plateau after epoch 2. Slightly overfitted.

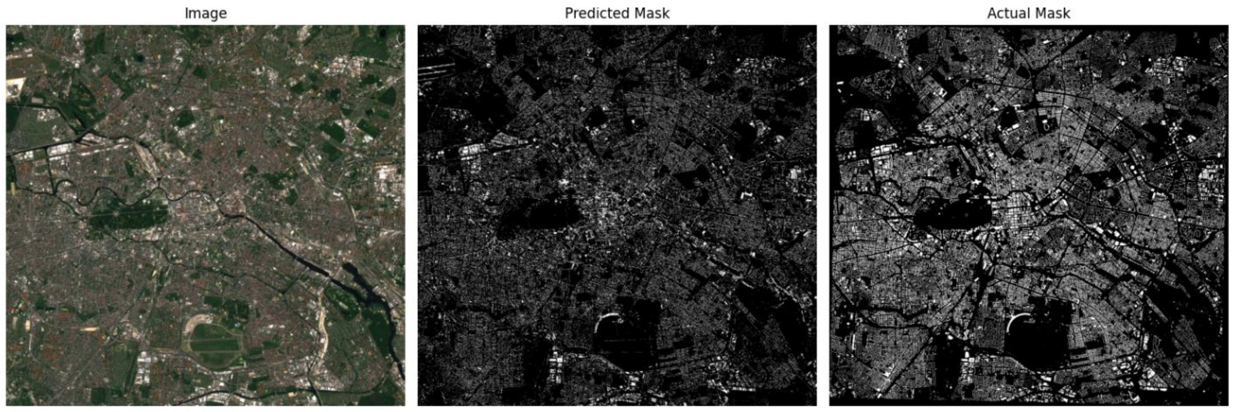


Figure 3: Result of the first trial of the BaselineCNN model. Test image, predicted mask and ground truth mask.

After x attempts to tune the hyperparameters to improve the performance, the hyperparameter tuning process was stopped. Despite many hyperparameter adjustments, the validation loss on the last epoch didn't go lower than ~0.3. Figure 4 depicts train and validation loss for the xth attempt with an accuracy of 0.8413 for test data.

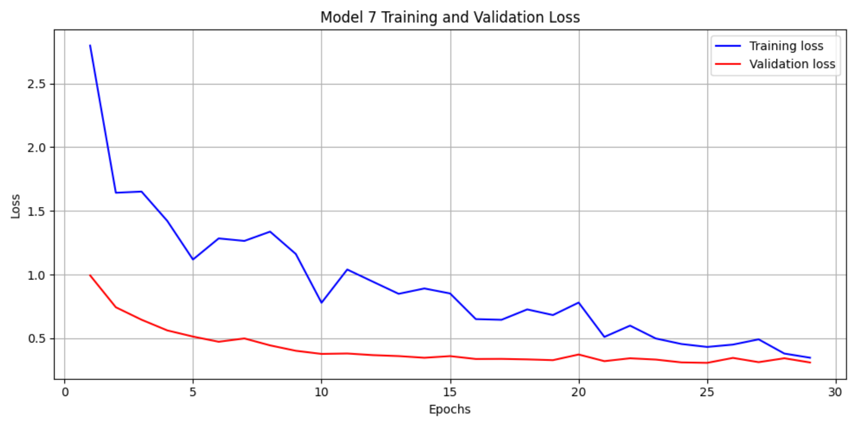


Figure 4: Not overfitted model but still not the best one.

Finally, a model with the greatest IoU value has been evaluated on the test data and got an accuracy of 0.8452. Figure 5 represents train vs. validation loss and Figure 6 the evaluation on the test data.

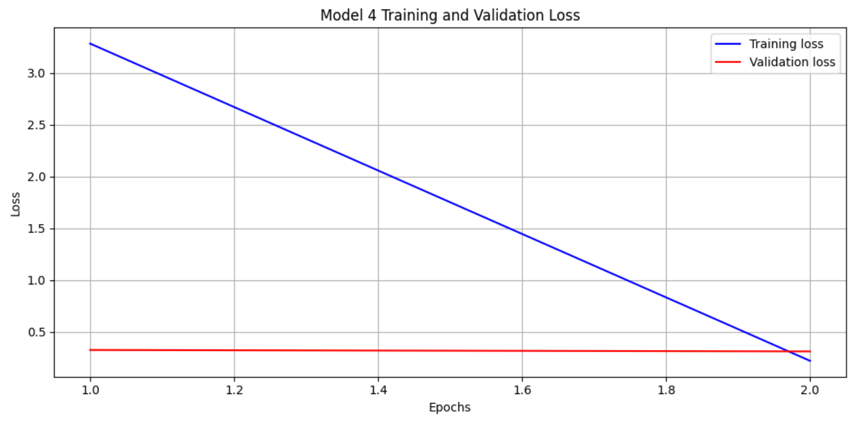


Figure 5: Train vs validation loss of the 4th BaselineCNN. Slightly overfitted with triggered early stopping at epoch 2.

To prevent such a drastically falling of the train loss, escape overfitting and make the model to learn slower and smoother, we could reduce the learning rate.

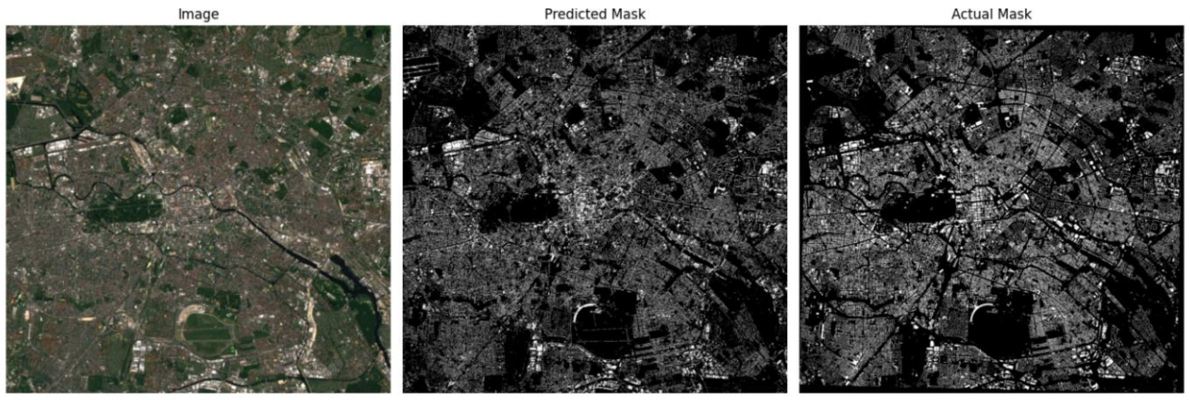


Figure 6: Result of the 4th trial of the BaselineCNN model. Test image, predicted mask and ground truth mask.

The second model is the UNet model adapted from [1]. Its also a CNN designed for image segmentation. It features an encoder-decoder architecture. The encoder consists of four blocks, each with two convolutional layers followed by ReLU activations and a max pooling layer, progressively doubling the number of filters (32, 64, 128, 256). At the bottleneck, two convolutional layers with 512 filters each capture high-level features. The decoder mirrors the encoder, using transposed convolutions to upsample the feature maps, with skip connections from the corresponding encoder layers to preserve spatial information. The final layer is a 1x1 convolution that produces a single-channel output segmentation map for pixel-wise building detection in this case. I only compare it with the best BaselineCNN trained with the same parameters i.e. model 4. Table 2 represents hyperparameters and Figure 7 represents train vs validation loss of this model.

Table 2: Hyper-parameters and validation of the UNet model.

Trial	Epochs	Batch Size	C. Channels	Optimizer	Learning Rate	Weight Decay	Early Stopping (patience, delta)	Avg Val. Loss	IoU
1	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3761	0.00045

Same hyper-parameters as in the 4th BaselineCNN model are used for UNet model trainig in a single run.

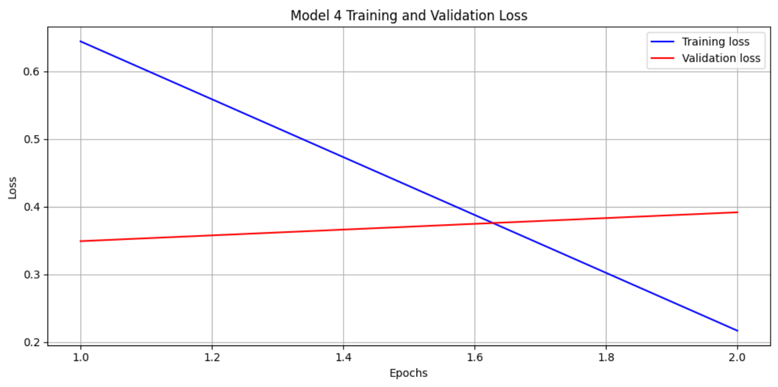


Figure 7: Train vs validation loss of the UNet model. Obvious overfitting.

The UNet model overfits drastically already after first epoch. The reason could be due to the too complex model architecture. It had an accuracy of 0.8235 on the test image. In order to prevent overfitting, one could reduce the number of layers and filters in order to simplify the model and then also perform hyper-parameter tuning, e.g. decrease learning rate to slow down the learning process and make the train loss curve more U shaped instead of a line.

1.4 Data Augmentation

In the last task, we will perform individual and compound data augmentation techniques. For this, I created a custom Dataset class which will perform a transformation chosen from the albumentations module. We train the baseline model using the best parameters from task 1.3, then validate and evaluate its performance with applied data augmentation.

Table 3: Best BaselineCNN model with data augmentation.

Data augmentation	Epochs	Batch Size	C. Channels	Optimizer	Learning Rate	Weight Decay	Early Stopping (patience, delta)	Avg Val. Loss	IoU
Horizontal Flip	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3271	0.25018
HF+VF	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3564	0.14412
Random Rot.	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3443	0.14451
RandomGridShuffle	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3189	**0.28195**
RGS+HF	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3235	0.24271
Crop (64x64)	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3239	0.25345
Crop (64x64)+RGS	4	8	4	Adam	0.0008	5e-4	(1, 0.05)	0.3407	0.21908

Each augmentation is performed at prob. of 0.5. Rotation has an limit of 180 degrees.

Despite a few data augmentation attempts the 4 trial of the BaselineCNN from task 1.3 could not be outperformed. The best result was RandomGridShuffle augmentation which segmented the input image into 2x2 grid with 50% probability. Despite a few attempts to combine it with other data augmentation techniques the results could not be improved. Although, the accuracy on the test data for the best IoU score is 0.8415. Figures 8 and 9 demonstrate train vs. validation loss and the result on the test data with the RandomGridShuffle (RGS) augmentation respectively.

Other data augmentation techniques like distortion have not been performed because they would not add any useful data into the train set. Buildings have distinct and often rigid structural characteristics. Distortions can alter these characteristics, leading to unrealistic representations of buildings. For instance, a rectangular building might appear skewed or curved, which does not reflect real-world scenarios. Data augmentations aim to expose the model to a variety of scenarios that it might encounter in the real world. Since augmentations like shearing do not typically represent real-world variations of buildings, they introduce unrealistic examples into the training set, potentially harming the model's performance on real data.

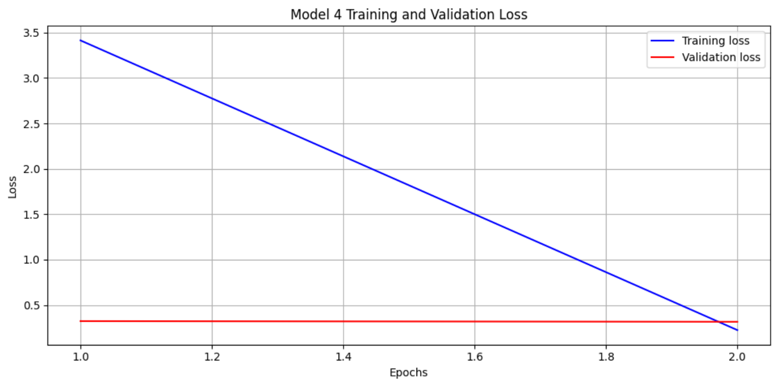


Figure 8: Train vs validation loss of the model with RGS augmentation. Slightly overfitted.

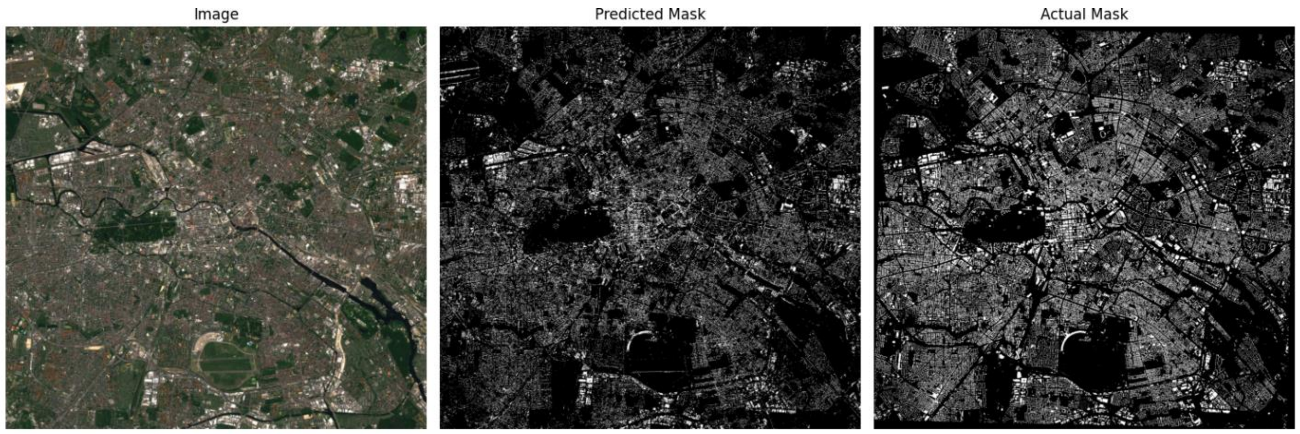


Figure 9: Result on test image of the best model with RGS as augmentation.

REFERENCES

[1] https://towardsdatascience.com/cook-your-first-u-net-in-pytorch-b3297a844cf3