02.07.2024

Sentinel Building Segmentation - Project Report

Laurids Radtke

This project is part of the module “Architecture of Machine Learning Systems” by Professor Matthias Boehm. To complete the module, it is obligatory to either contribute to Daphne or Apache SystemDS, or to complete this project. I chose to do this project as I thought it was the most interesting task. The project could be completed in teams of up to 3 people but I decided to do it alone so that I was forced to learn everything that is necessary to complete the project. Normally I like to work in teams but making the experience to be on my own on such a significant task was very valuable.
The goal of the project is to build a ML pipeline that downloads and prepares satellite data for training and then trains models on the data with the aim of classifying each pixel in a given image into the classes “contains a building” and “does not contain a building”. This can be seen as an image segmentation problem. This task was divided into four subtasks, 1. Data Acquisition and Alignment, 2. Data Preparation, 3. Modeling and Tuning and 4. Data Augmentation. The usage of any programming language as well as any open-source library was allowed to complete the project. As the programming language I chose Python as it is well suited for the task and I am also the most comfortable with it. The used libraries will be mentioned later. In this report I am going over every point, explaining what needed to be done and my solution for it.

The first task was to acquire the necessary training data and properly align it. That means satellite images need to be downloaded as well as information about the buildings present in the downloaded images to generate the labels for each pixel. That means the subtasks were downloading satellite images, downloading building location data, transferring this location data to labels for individual pixels (label mask) and then aligning the satellite images with the respective label masks so that every pixel is assigned with its correct label.
For this, the task description required us to use images of the Sentinel 2 satellite, which is an earth observation satellite from the ESA Copernicus mission, processed to stage 2a as well as Open Street Map (OSM) data for the building locations.
In my implementation I first download the OSM building data for a user defined city in the function get_osm_building_data(). For this I am using the Pyrosm library that can download OSM data as Protocolbuffer Binary Format (.pbf) files and extract them into GeoPandas dataframes. I am downloading the city data with the Pyrosm function get_data(). As this function downloads not only building data from buildings in the city borders but also from

surrounding regions, my next step is to extract the city border from the city of interest, which happens in the function get_city_boundary(). The extracted city border can be passed to the constructor of the Pyrosm OSM class, along with the file path to the .pbf file, to extract all buildings within the city boundaries. This building data is then saved as a pickle file in a designated directory. All data, the building data as well as the city border, are encoded as sets of geographical coordinates (EPSG:4326, also called World Geodetic System 1984/WGS84), one coordinate for each corner of a house or border.
In the next step the Sentinel Image data is downloaded in the function get_sentinel_image_data(). For this I am using the openEO library that allows users to connect to earth observation cloud backends and download data from there. First a connection to the backend is established, a data cube containing the desired data is specified with the openEO function load_collection() and then downloaded with download(). Important function arguments are the spatial extent, temporal extent and bands to download. The temporal extent is user defined and assumed to be just one day, resulting in the download of images at just one point in time. The user should also check (e.g. with the Copernicus Browser) for the presence of cloud cover as well as whether there is data available for the desired day at all.
The spatial extent is obtained from the extracted city border, where the northernmost point of the border provides the northern border of the image to be downloaded, the easternmost point of the city border provides the eastern border and so on. The bands to be downloaded can also be specified by the user but are assumed to be red, green and blue as well as visible and near-infrared (VNIR). The data is then downloaded as a .tiff file. After the download, the .tiff file is opened with the open() function of the rasterio library and the individual bands are extracted as numpy arrays. These are then clipped to 0-2500 and normalized to 0-255. I found that clipping to 0-2500 does preserve most of the information in the image while not making the image too dark when plotting. To generate the rgb image the red, green and blue bands are stacked. The same happens for the false-color image where VNIR, green and blue are stacked. The resulting images are then saved in a designated directory.
In the last step, the label masks are generated from the downloaded OSM data. For this the .tiff file as well as the pickle file with the OSM data is loaded. The .tiff file contains information about the coordinate reference system (CRS) of the image data, stored under the attribute “crs”. In this case the image data is in the Universal Transverse Mercator (UTM) coordinate system. With the GeoPandas function to_crs() the CRS of the building data (WGS84) is transformed to the CRS of the image data, UTM. In the next step an AffineTransformer object (part of the transform module in rasterio) is defined. The constructor is passed an affine transformation matrix that can be accessed under the .tiff file attribute “transform”.

With this transformer the UTM coordinates of the building corners are translated to pixel coordinates in the image, using the rowcol() function of the transformer object. These coordinates are then converted into shapely Polygons and the label mask is finally created by rasterizing these Polygons with the rasterize() function of rasterio's feature module. Afterwards the label mask is saved and for visualizing purposes, a stacked image, showing the rgb image and the label mask stacked on top of each other, is generated and saved.

The next task was to arrange the just acquired and processed data in datasets suited for model training.
The first step here was to cut the big images that show whole cities as well as the label masks into smaller patches for training. This is done in the prepare_arrays() function. As the first function argument, a list of city names containing the cities to include in the dataset should be passed. For this to work, the city data for the desired cities must be already downloaded and prepared with the code from the previous step. Next the desired patch size must be passed. For my experiments, I decided to use a patch size of 64x64 px as suggested in the task sheet. Also I decided to just use RGB images and disregard the false-color images as well as individual bands. Otherwise I would not have been able to complete the project in time, as this would have increased the complexity especially in the experimentation phase considerably.
Firstly, prepare_arrays() loads the RGB image and the label mask. Then indices are generated to cut out the patches. With that the function loops through the indices and extracts the patches from the RGB images and label masks. The patches are stored into numpy arrays, which are then passed to a removal function called remove_patches(). In this function, patches that have more than 20% of white pixels are removed (cloud cover). A pixel is considered white when its grayscale intensity is over 95%. Furthermore, patches that do not have any labels associated with them are removed. As a result, commonly around 50% of patches are removed (e.g. Vienna 1656 total patches, 646 remaining patches).
This is done for every city and the remaining patches of every city are collectively stored in a feature and a label list respectively.
These two arrays are then returned and passed to build_final_arrays(). Here the data of the individual cities is merged, resulting in one feature numpy array and one label numpy array containing the data of all cities requested by the user. The feature array has the shape (num of patches, 64, 64, 3) and the label array has the shape (num of patches, 64, 64). The feature array is normalized to 0-1 and then the feature array and the label array are saved in a designated directory as numpy arrays.
In the next step, the train, test and validation splits are performed. For this I am using the scikit-learn function train_test_split(). First the feature and label arrays are splitted into train

and temporal datasets. The temporal datasets are then split into test and validation datasets. I have chosen 70% as a train dataset size and 15% respectively for the test and validation dataset size, as this is a common split. To ensure a balanced label distribution, I applied stratified sampling to the splitting processes. For this, the sums of the positive labels (contains a building) of each patch are binned into 100 bins, classifying the patches into 100 different classes. Passing this distribution over all patches to the train_test_split() functions ensures that all datasets contain a set of patches with a balanced label distribution.
After splitting, the resulting numpy arrays are stored in the dedicated dataset directory.

The next step was training the model and applying some techniques like hyperparameter tuning and data augmentation. I decided to use Pytorch for my implementation as I already know Tensorflow and wanted to get to know a new ML library.
The first step was to define a dataset class that wraps the training arrays. For this I implemented a class with the init() function for loading the numpy arrays saved by the preparation pipeline, the len() function for returning the length of the dataset as well as the getitem() function to return feature samples with the associated label mask. In the getitem() function I also implemented the possibility to apply on-the-fly data augmentation to the samples.
Next, I implemented the baseline model class, which I called PixelClassifier. The PixelClassifier consists of a modifiable number of stages. Each stage contains a convolutional layer (torch.nn.Conv2d) and a dropout layer (torch.nn.Dropout2d) with modifiable dropout probability. After each stage the Relu function is applied and in the end the sigmoid function is applied to output a probability for each pixel.
For comparison to the baseline model I chose the U-Net architecture. I obtained the Pytorch implementation of U-Net from Github [1].
In the next step I implemented the data_loader() function that creates dataset objects from the previously implemented dataset class, one for the train data set, test dataset and validation dataset each. These objects are then wrapped with the DataLoader class and returned. The desired batch size can be passed as an argument of the DataLoader init() function.
After that, I implemented the training loop, that loops through the train dataset. In the loop the images and labels are moved to GPU, the gradients are set to zero, the forward pass is performed, the loss is calculated, the backwards pass is performed and the model parameters are updated. After every epoch the model is evaluated on the validation and train dataset with the evaluate_model() function and the eval results are saved. After training, some training statistics, like train and validation loss & accuracy are plotted and the plots are saved along with the model in a designated directory. For the loss calculation I

chose the Binary Cross Entropy (BCE) loss. This metric is well suited for the task at hand, as it is commonly used for binary classification problems and works with probabilities, which are the output of the last stage of the model as a result of using the sigmoid function. It also applies an aggressive penalty for confident wrong predictions. As an optimizer I used the Adam optimizer, which is a standard choice for similar tasks.
I further decided to implement the ability to train on GPU as training on my Laptop CPU was way too slow, especially for hyperparameter tuning.
The evaluate_model() function loops through the provided dataset, performs a forwards pass and calculates the loss (again BCE loss) and accuracy. To calculate the accuracy, the probability outputs of the model need to be converted to binary labels. Here I chose a threshold of 0.5. Probabilities over 0.5 are set to 1 and the rest is set to 0. I chose this as this was the most neutral option. Tuning this parameter could yield different performance statistics. For evaluation on the test dataset, the results can also be saved in a .txt file.
Next I also wrote a test function to test the model on the test image which is specified in the task sheet and shows a part of Berlin. The result is saved in the same directory as the other data for a specific training run. Also an image of the model output is saved to compare with the label image. For the accuracy computation and the prediction plotting again a threshold of 50% has been applied to the model output probabilities.
For the hyperparameter tuning I have used Optuna. The tuning is done in the function hyperparam_tuning(). Firstly, a new directory is created to save the results and a logger is set up. Then a new study is defined and set to “maximize”. The objective function is passed as an argument to the optimize() function. I chose to do 20 trials, as more would have taken too long. In the objective function first the parameters to be tuned are set up.
In my experiments, I tuned two training parameters: the learning rate in a range of 0.00001 to 0.1, and the batch size as a choice from 16, 32 and 64, as well as two model parameters: the number of stages as an integer from 1 to 6 and the dropout probability as a float between 0 and 1. Of course many other hyperparameters would have been interesting to tune but due to time constraints I focused on these four.
After setting up the parameters to be tuned, the train_model() function is called and the model is trained with different parameters in each trial. The objective function returns the accuracy of the training run which is maximized by the optimizer.
For the on-the-fly data augmentation, I have defined a get_augmentation() function which returns an Albumentations Compose object that can be passed to the ImageDataset via get_dataloaders(). I chose on-the-fly data augmentation as it is easier to implement and has a smaller memory footprint as classical augmentation where the dataset is expanded.
For my experiments I have defined three different sets of augmentation operations. The first augmentation consists of a horizontal flip, a vertical flip and a random 90 degree rotation all

applied with a probability of 50% as well as a random change of brightness and contrast applied with a probability of 20%. The second augmentation consists of a random affine transformation (prob. of 50%), an random RGB shift (prob. of 50%), again the random change in brightness and contrast (prob. of 50%) and a normalization. The third augmentation consists of an elastic deformation, a grid distortion and an optical distortion also all with an application probability of 50%.

For my experiments, I started with training the baseline PixelClassifier with standard hyperparameters on a dataset containing 10 cities: Zagreb, Denver, Wien, Helsinki, Hamm, Flensburg, Oslo, Stockholm, Marseille and Glasgow. The hyperparameters are a learning rate of 0.005, a batch size of 16, one model stage/layer, a dropout probability of 0%, no augmentations and trained over 20 epochs. On the test dataset this yielded a loss of 0.0116 and an accuracy of 0.8685. On the Berlin test image the model achieved an accuracy of 0.7245.

For the next experiment I tuned the hyperparameters as explained earlier. The result of that process is a learning rate of 0.0012, a batch size of 16, 5 model stages and a dropout probability of 0.1883. Training, again over 20 epochs, yielded a loss of 0.006 and an accuracy of 0.892 on the test dataset as well as an accuracy of 0.8176 on the Berlin test image. We can see that tuning only four hyperparameters has provided a significant increase in performance.

For the next experiment I trained the baseline model with the trained hyperparameters and enabled data augmentation. I tested each of the three described augmentations separately. The first augmentation yielded a loss of 0.0081 and an accuracy of 0.8758 on the test dataset and an accuracy of 0.7758 on the Berlin test image. Training with the second augmentation yielded a loss of 0.0186 and an accuracy of 0.8623 on the test dataset and an accuracy of 0.6987 on the Berlin test image. For the last augmentation the model showed a loss of 0.0063 and an accuracy 0.875 on the test dataset as well as an accuracy of 0.7931 on the Berlin test image. We see that the on-the-fly augmentation has decreased performance of the baseline model. The reason for this is not easily discernible but might be attributed to the technique itself as it is perhaps not well fitted for this specific application. Maybe classical augmentation would have yielded better results. Another reason for the performance decrease could be a poor combination of augmentation operations in the individual compositions.

In the next experiment I trained the U-Net, again over 20 epochs. The model showed a loss of 0.0089 and an accuracy of 0.9018 on the training dataset as well as an accuracy of 0.8306 on the Berlin test image. This means that the model is capable to label over 90% of the pixels in an previously unseen image correctly. This is the overall best performance encountered during my experiments.

In the last experiment, I again trained the U-Net but using the most successful augmentation technique which was the first one. Nevertheless the model performance decreased as observed in the baseline model. The results were a loss of 0.0095 and an accuracy of 0.8829 as well as an accuracy of 0.7843 on the Berlin test image.
The overall best performing model is therefore the vanilla U-Net model with an accuracy of 0.9018. All plots can be seen in the appendix.

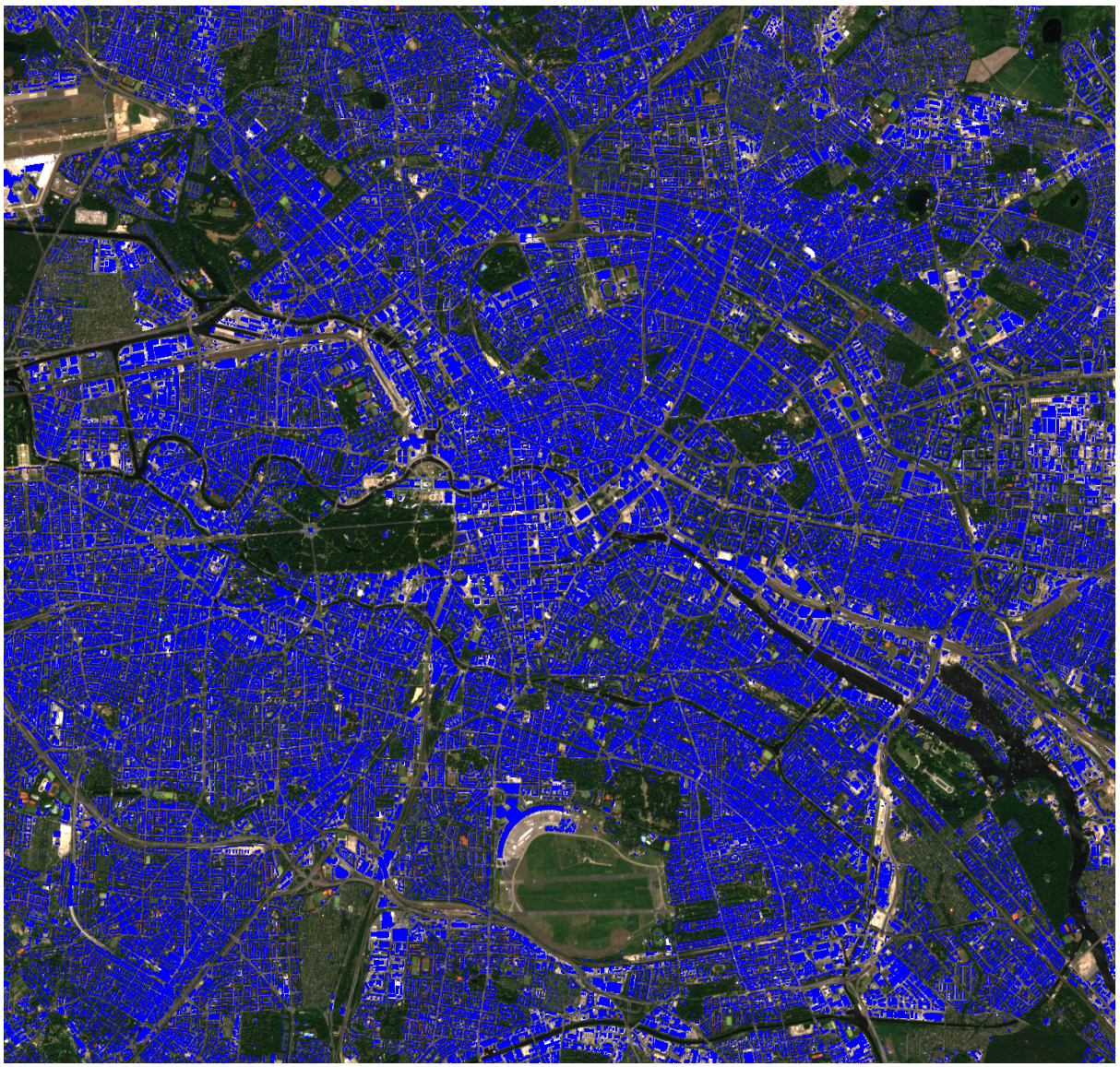
All training was done on an Nvidia A100 I rented over Google Colab. Doing all these experiments on my laptop CPU would have taken way too long. I do not own a CUDA capable GPU so renting such a device online was the only remaining solution that I could employ fast enough.
My laptops OS is Linux Ubuntu (Noble Numbat). I did all coding in a Conda environment that I have exported to a .yml file which can be found in the project repository. With this, installing all necessary dependencies should be straightforward on Linux. On Windows some compatibility issues can arise. For this I have provided a platform independent .yml file which has a higher chance to also work on Windows. Paths are handled mostly by the os library, so no problems should arise here during cross-platform code execution.

[1]
https://github.com/milesial/Pytorch-UNet/tree/67bf11b4db4c5f2891bd7e8e7f58bcde8ee2d2db

Appendix

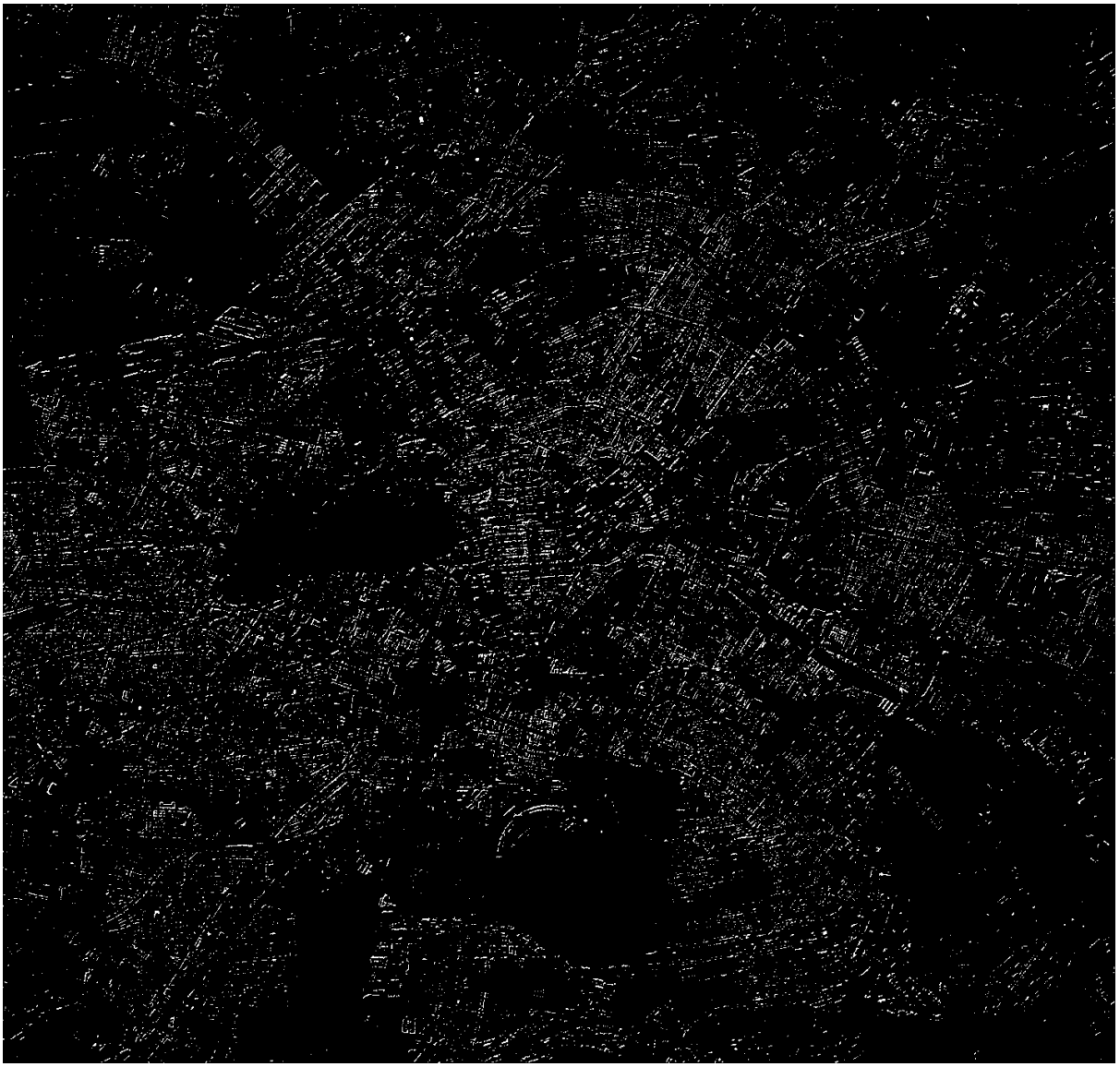
Berlin test image

Berlin test image labels

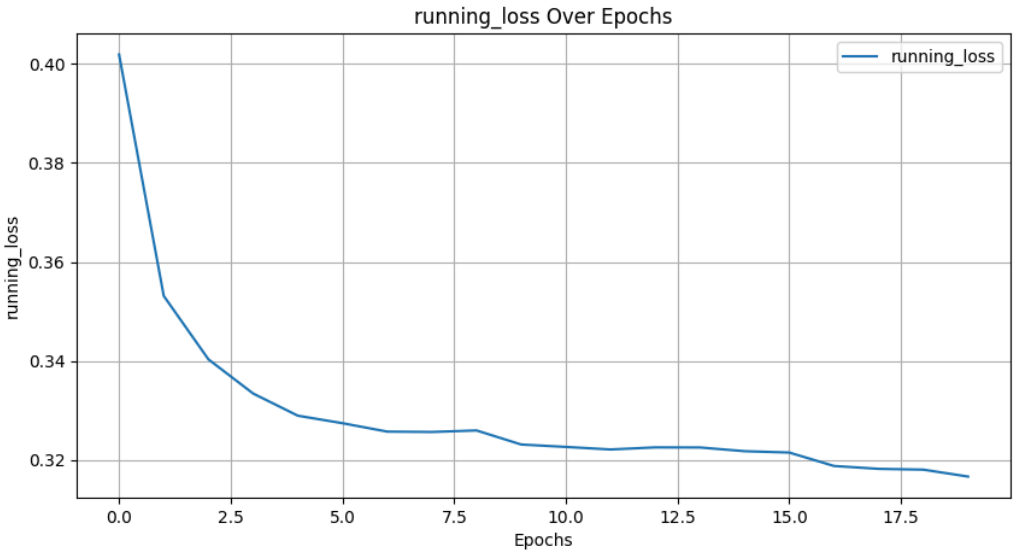


Berlin test image with labels

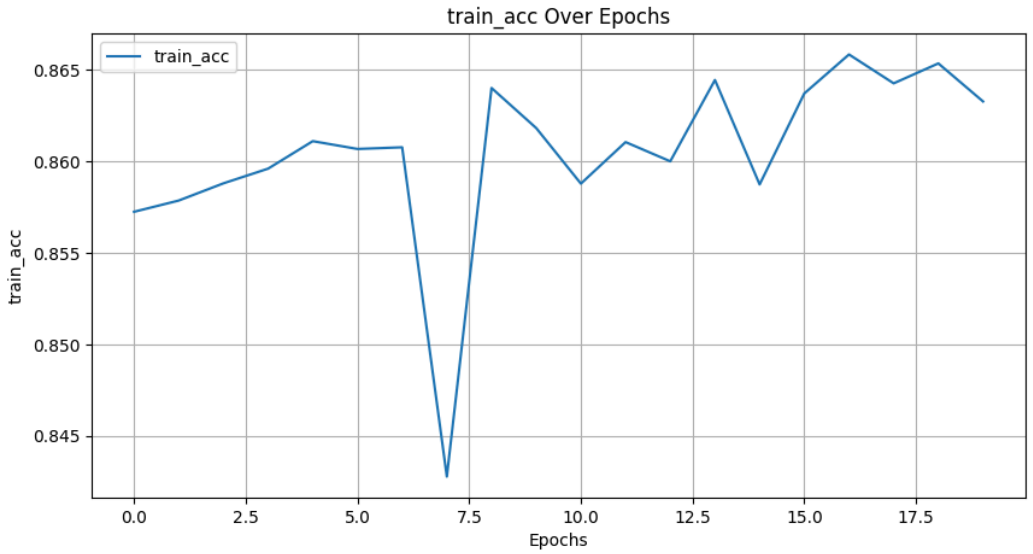
Vanilla baseline model:

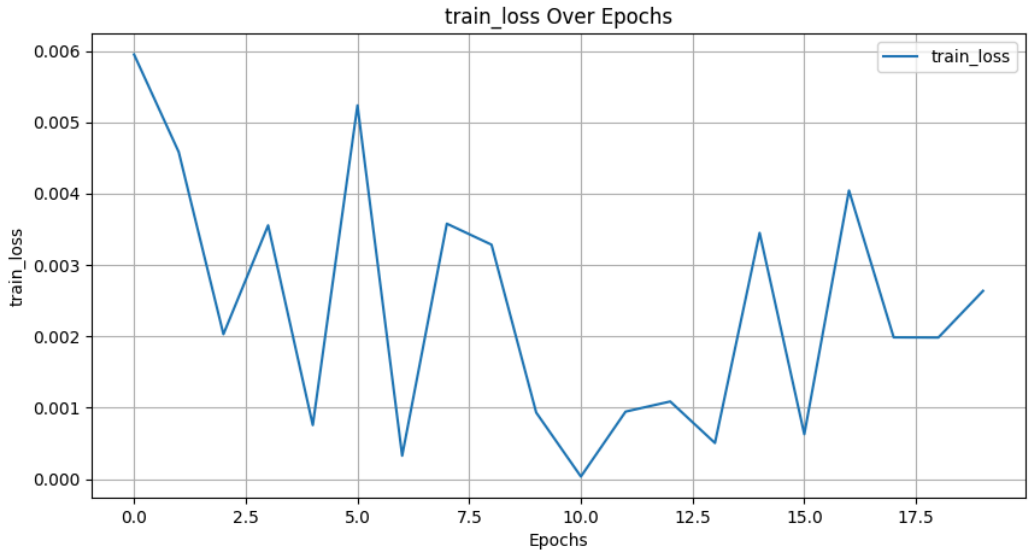


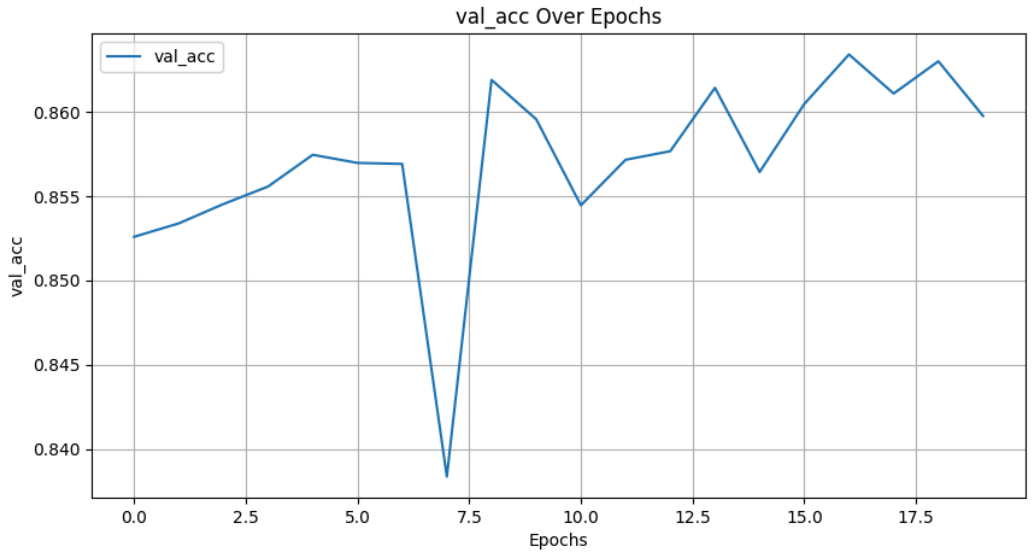
Berlin test image model prediction

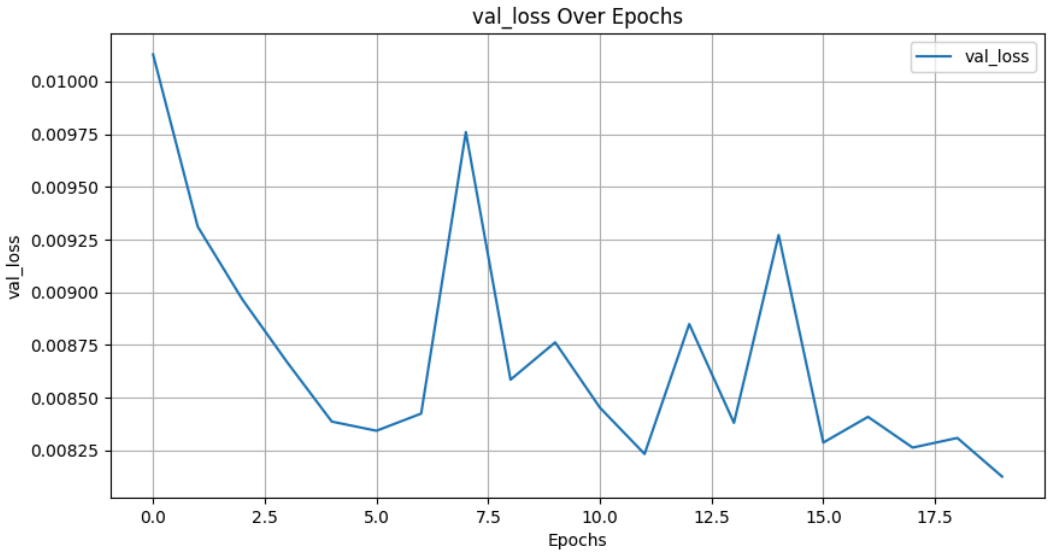

running_loss Over Epochs
running_loss
0.40
0.38
0.36
0.34
0.32
running_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


cont_training_loss Over Training step
cont_training_loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
cont_training_loss
0
1000
2000
3000
4000
5000
Training step


train_acc Over Epochs
train_acc
0.865
0.860
0.855
0.850
0.845
train_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

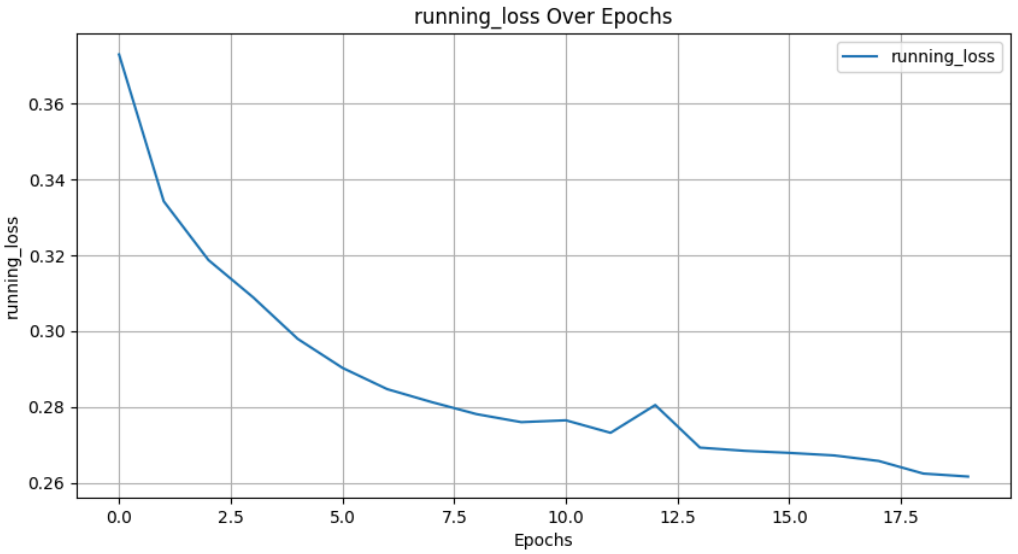

train_loss Over Epochs
train_loss
0.006
0.005
0.004
0.003
0.002
0.001
0.000
train_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


val_acc Over Epochs
val_acc
0.860
0.855
0.850
0.845
0.840
val_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

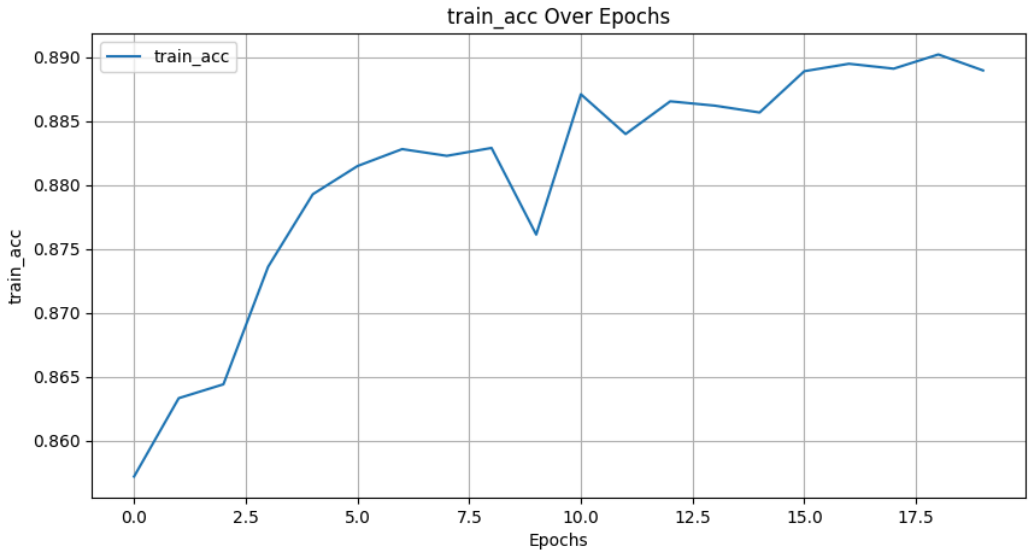

val_loss Over Epochs
val_loss
0.01000
0.00975
0.00950
0.00925
0.00900
0.00875
0.00850
0.00825
val_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

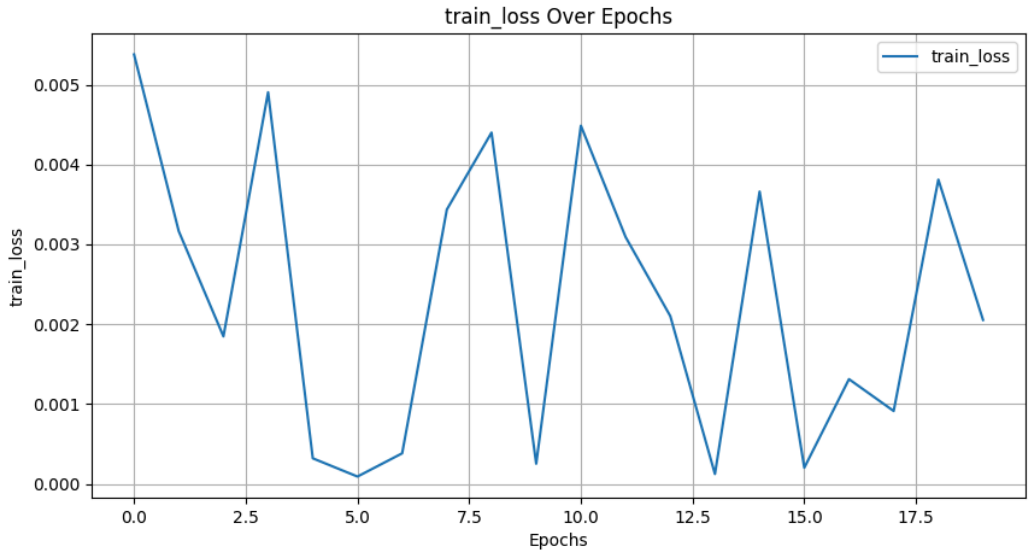
Tuned baseline model:

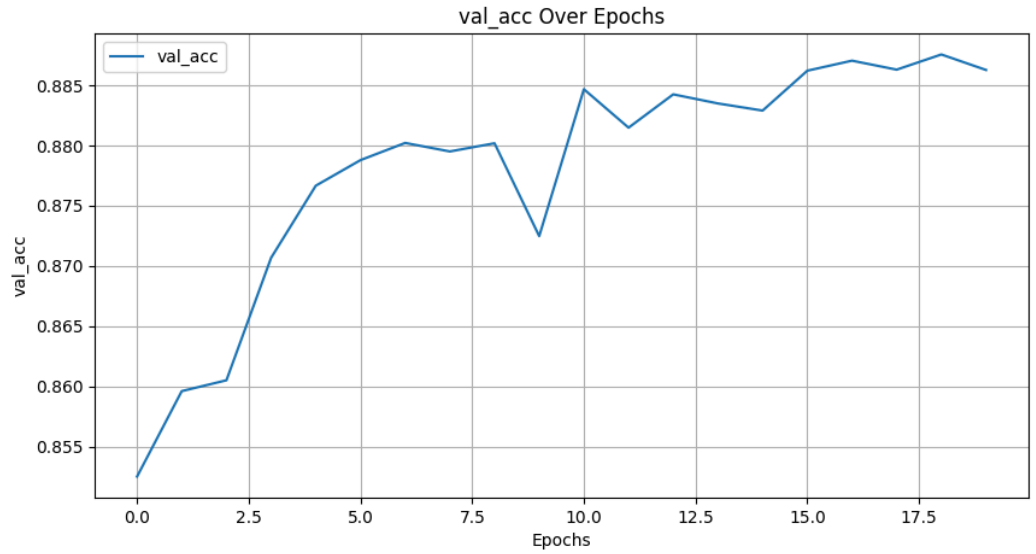
Berlin test image model prediction

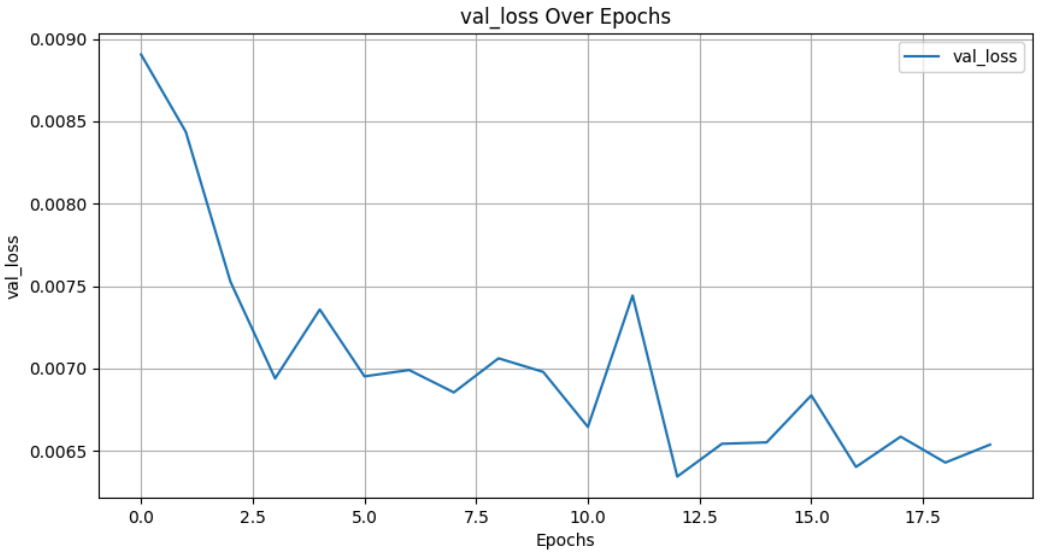

running_loss Over Epochs
running_loss
0.36
0.34
0.32
0.30
0.28
0.26
running_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


cont_training_loss Over Training step
cont_training_loss
0.8
0.6
0.4
0.2
0.0
cont_training_loss
0
1000
2000
3000
4000
5000
Training step


train_acc Over Epochs
train_acc
0.890
0.885
0.880
0.875
0.870
0.865
0.860
train_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

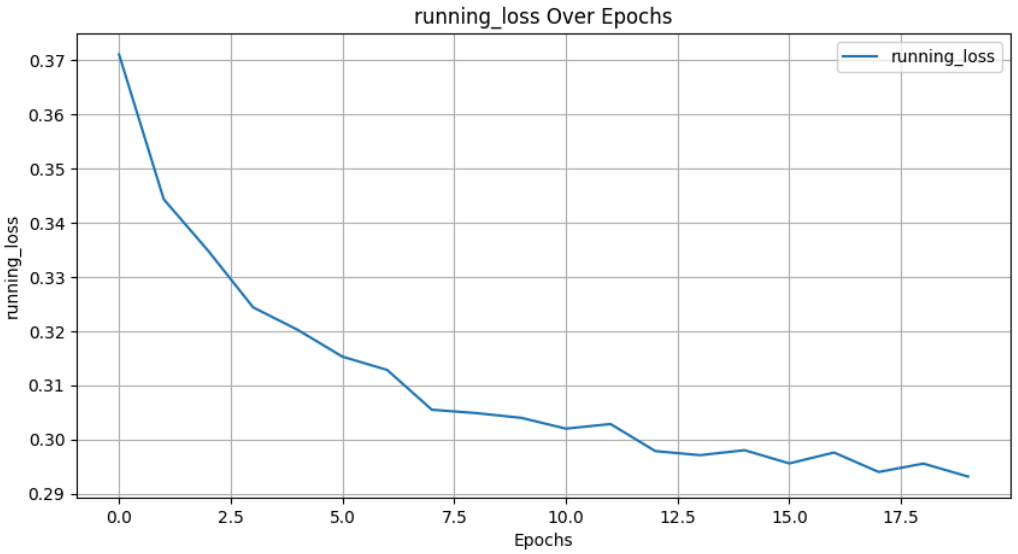

train_loss Over Epochs
train_loss
0.005
0.004
0.003
0.002
0.001
0.000
train_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


val_acc Over Epochs
val_acc
0.885
0.880
0.875
0.870
0.865
0.860
0.855
val_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

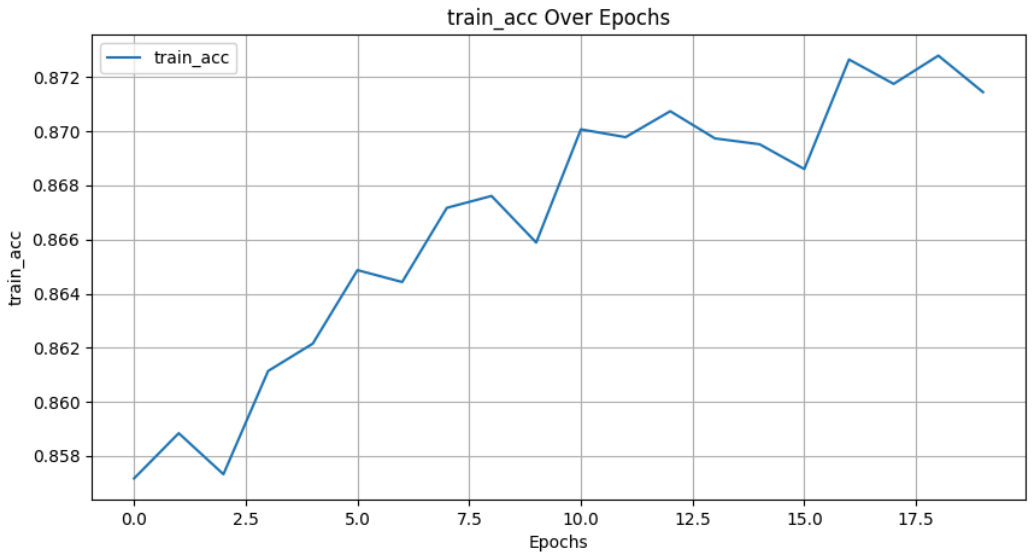

val_loss Over Epochs
val_loss
0.0090
0.0085
0.0080
0.0075
0.0070
0.0065
val_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

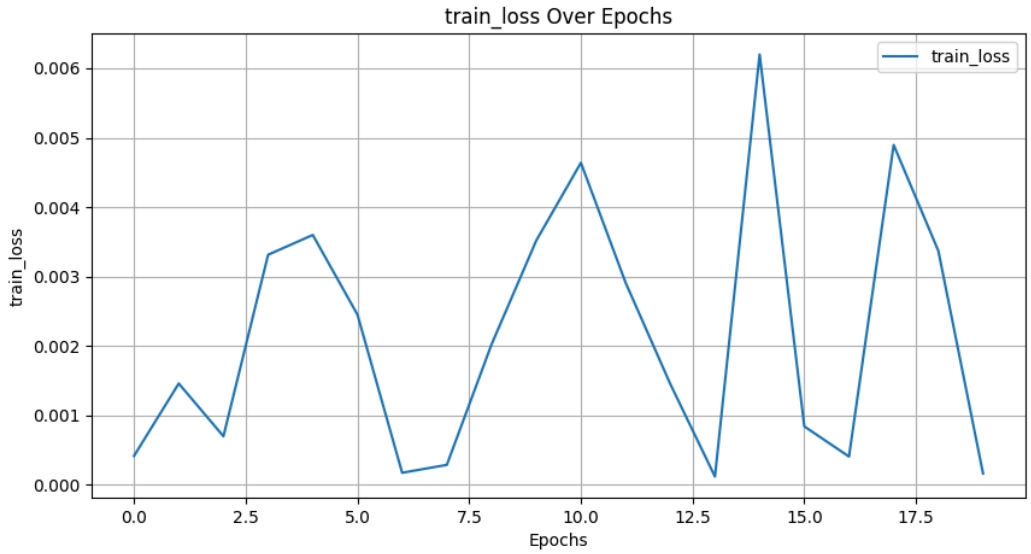
Augmented baseline model (Augmentation 1):

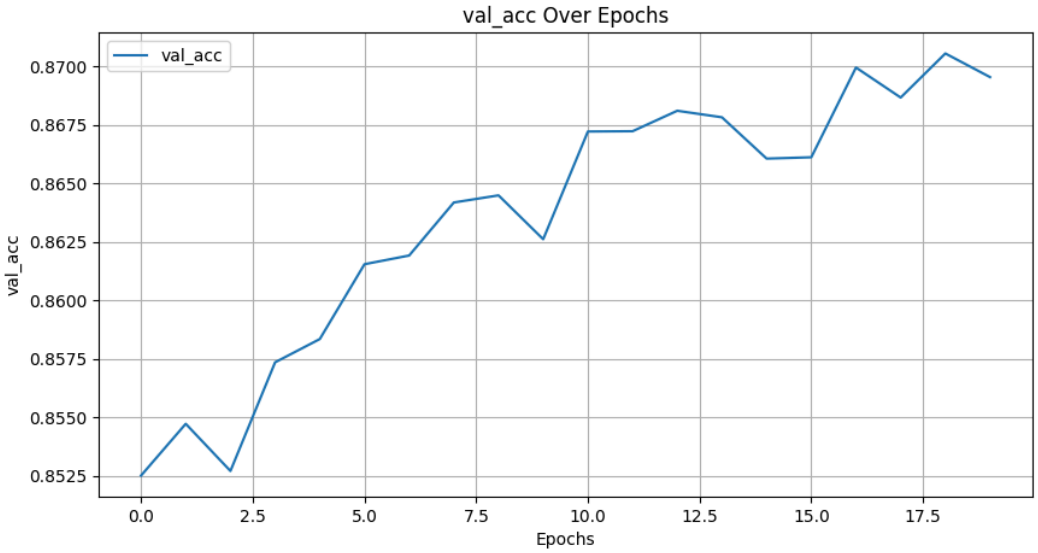
Berlin test image model prediction

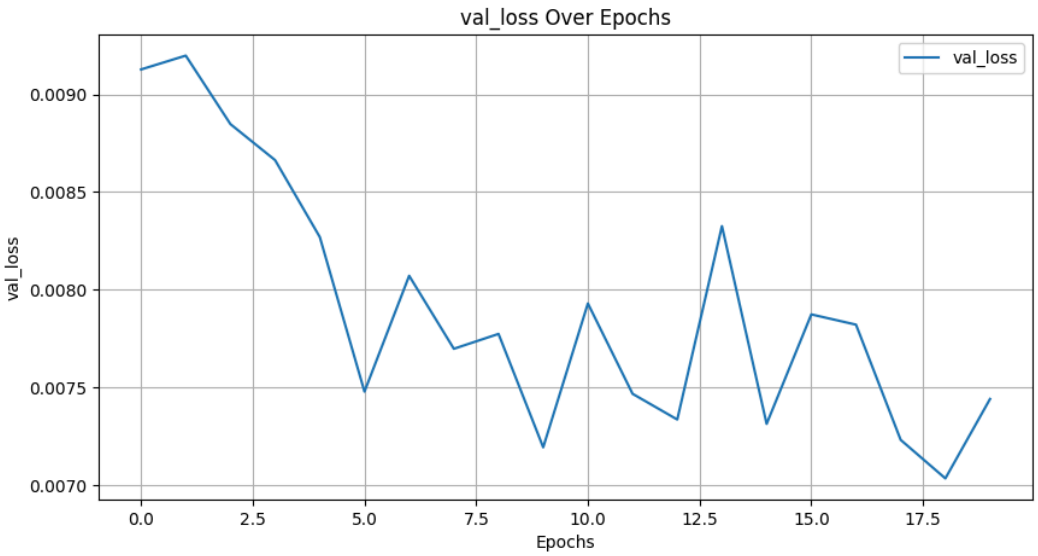

running_loss Over Epochs
running_loss
0.37
0.36
0.35
0.34
0.33
0.32
0.31
0.30
0.29
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


cont_training_loss Over Training step
cont_training_loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
1000
2000
3000
4000
5000
Training step


train_acc Over Epochs
train_acc
0.872
0.870
0.868
0.866
0.864
0.862
0.860
0.858
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

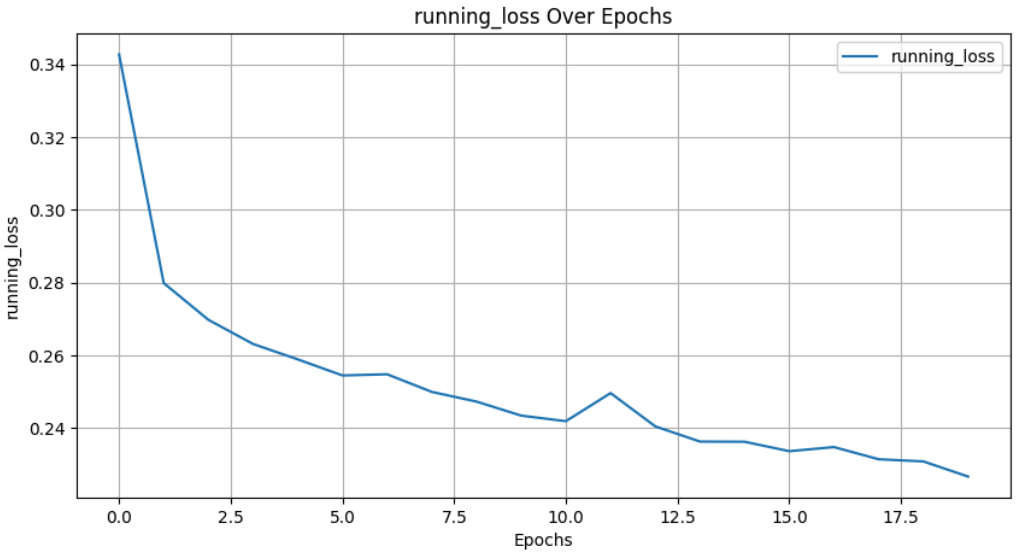

train_loss Over Epochs
train_loss
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


val_acc Over Epochs
val_acc
0.8700
0.8675
0.8650
0.8625
0.8600
0.8575
0.8550
0.8525
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

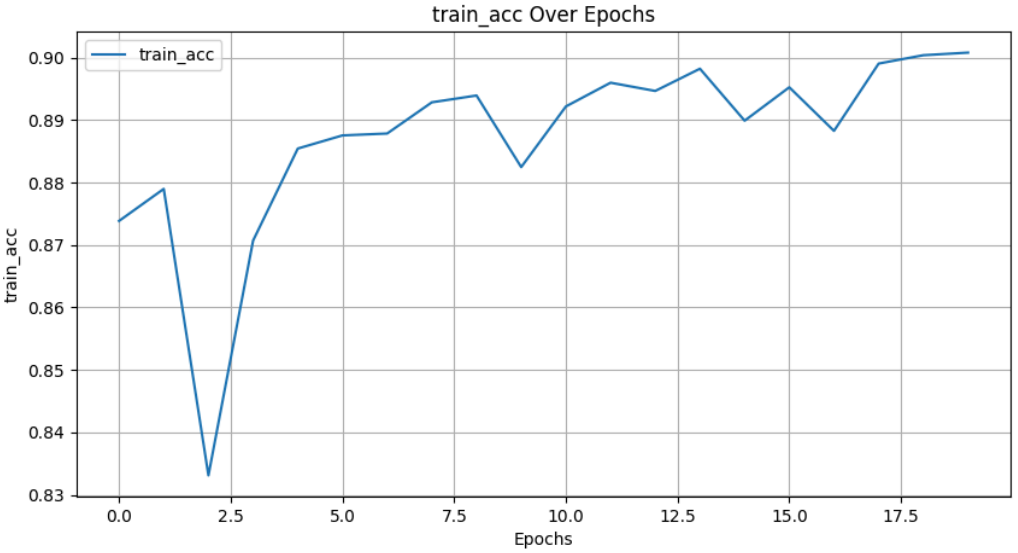

val_loss Over Epochs
val_loss
0.0090
0.0085
0.0080
0.0075
0.0070
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

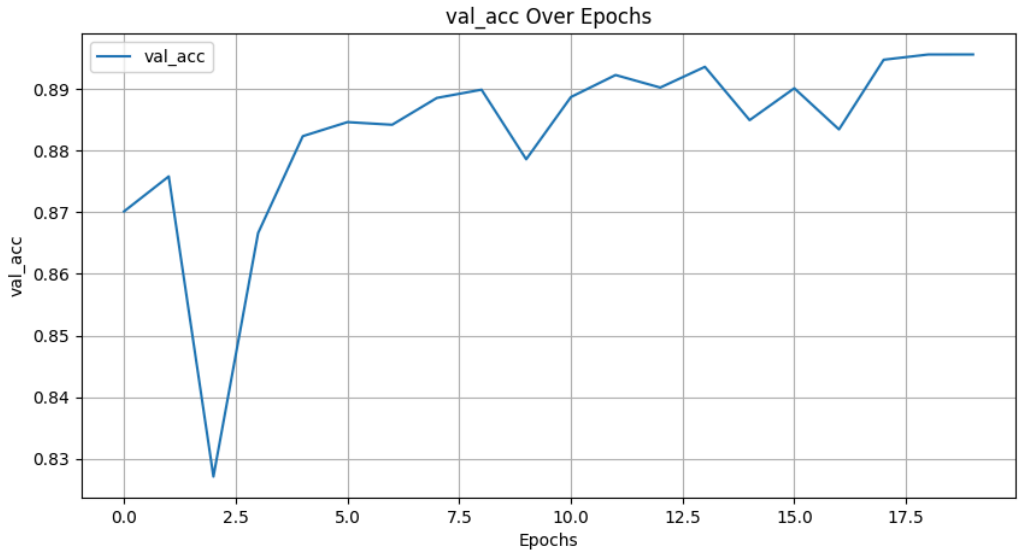
Vanilla U-Net model:

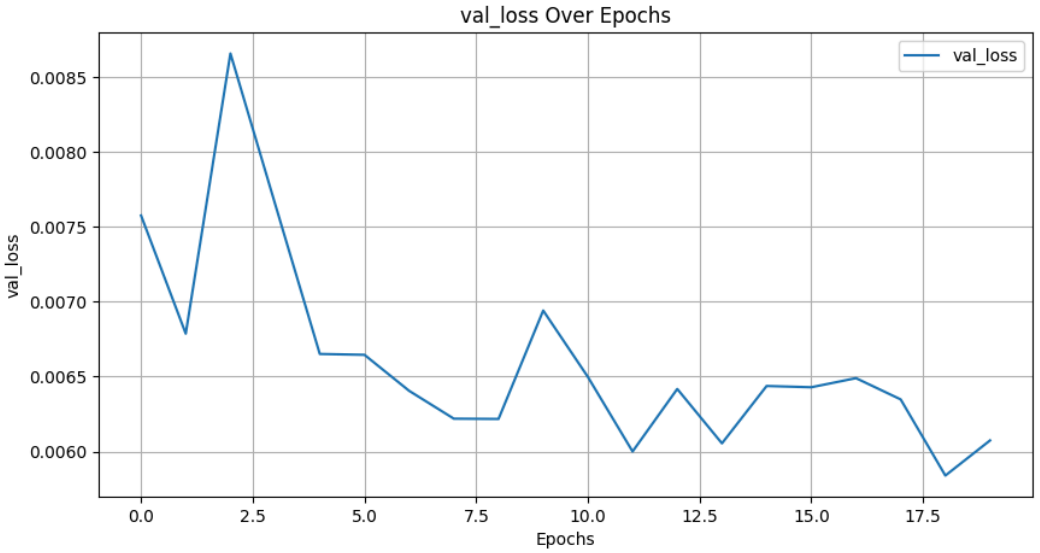
Berlin test image model prediction

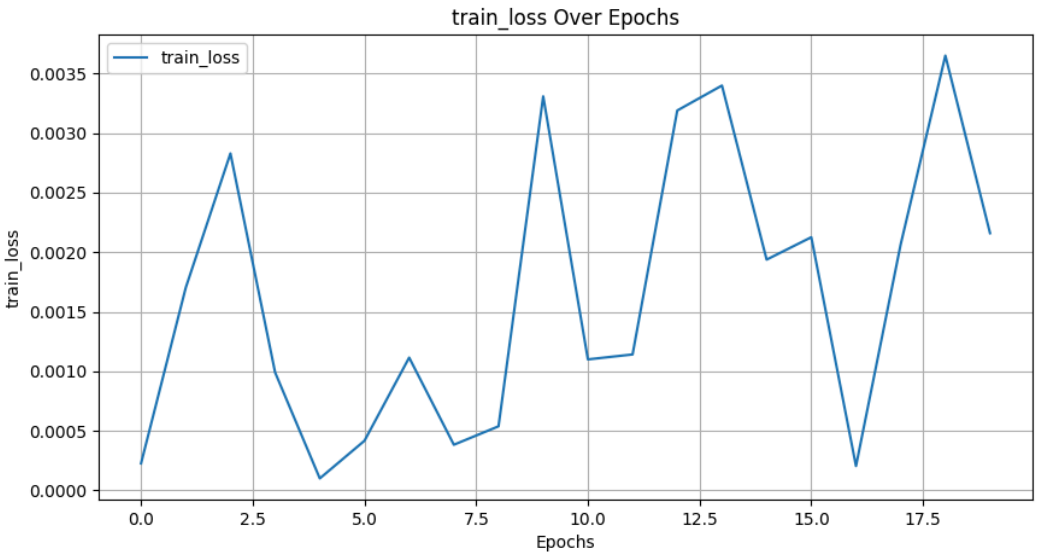

running_loss Over Epochs
running_loss
0.34
0.32
0.30
0.28
0.26
0.24
running_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


cont_training_loss Over Training step
cont_training_loss
1.0
0.8
0.6
0.4
0.2
cont_training_loss
0
1000
2000
3000
4000
5000
Training step


train_acc Over Epochs
train_acc
0.90
0.89
0.88
0.87
0.86
0.85
0.84
0.83
train_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

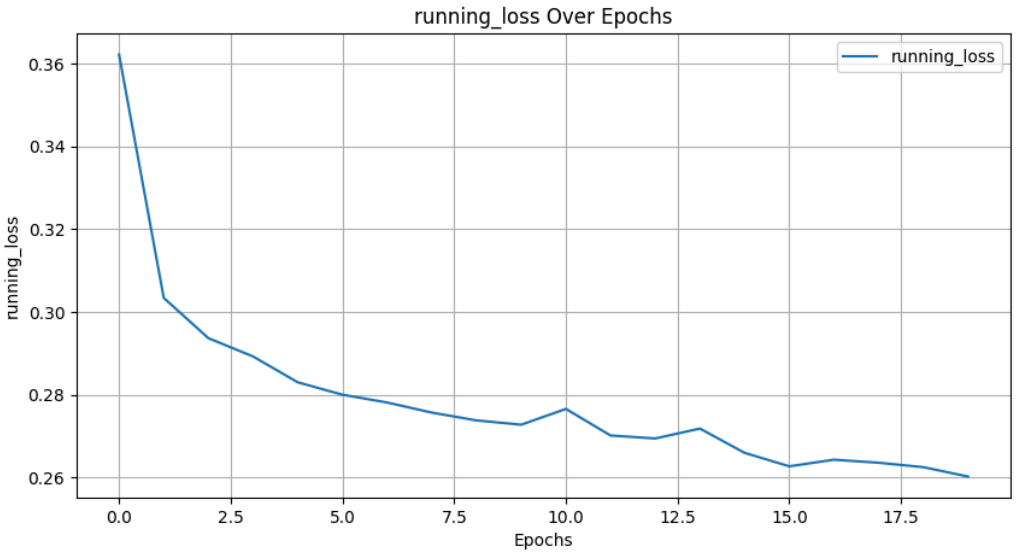

val_acc Over Epochs
val_acc
0.89
0.88
0.87
0.86
0.85
0.84
0.83
val_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

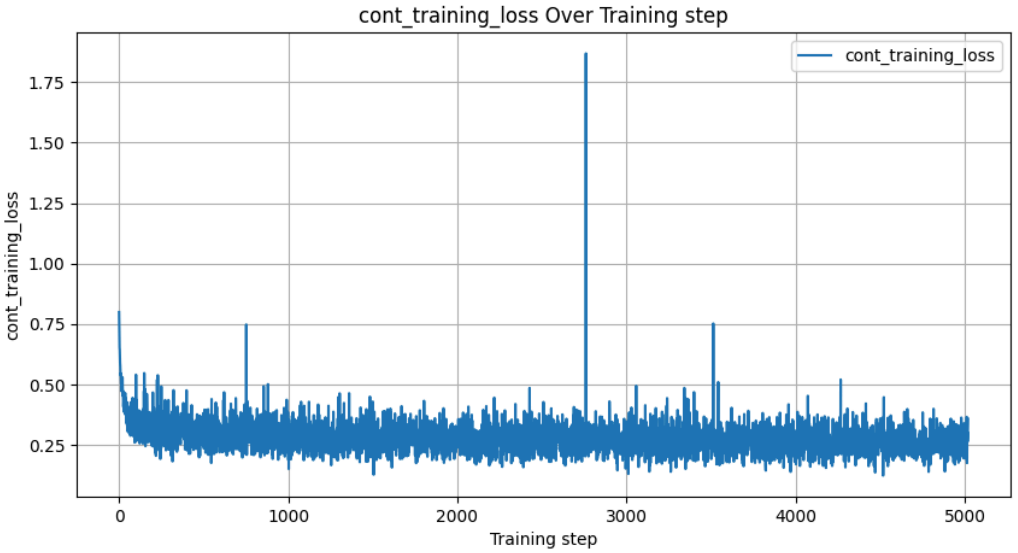

val_loss Over Epochs
val_loss
0.0085
0.0080
0.0075
0.0070
0.0065
0.0060
val_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

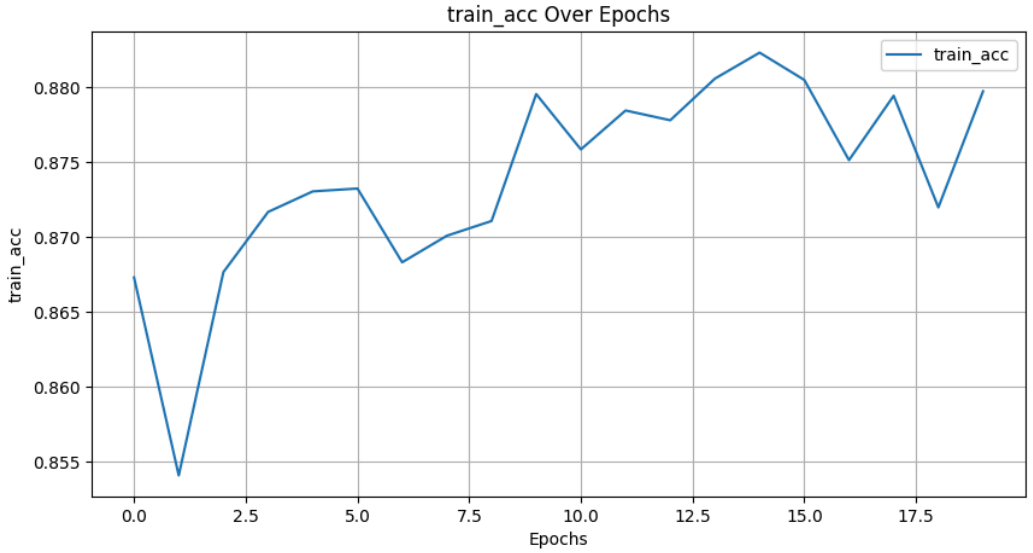

train_loss Over Epochs
train_loss
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
train_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

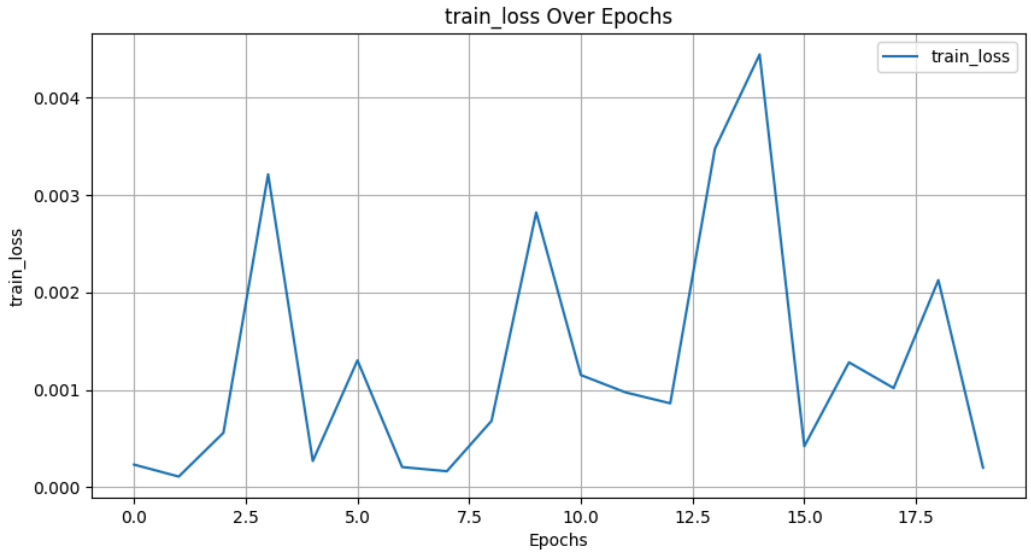
Augmented U-Net model (Augmentation 1):

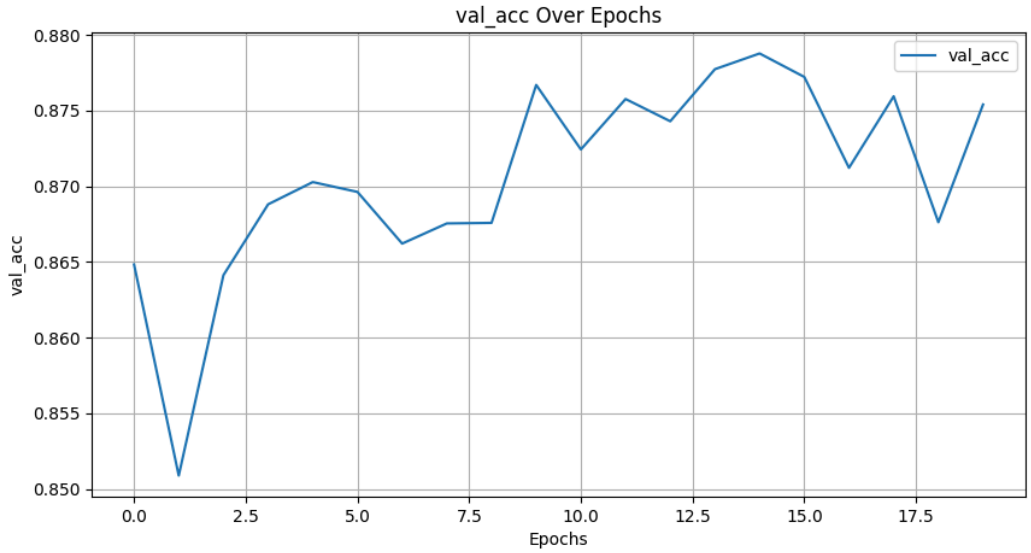
Berlin test image model prediction

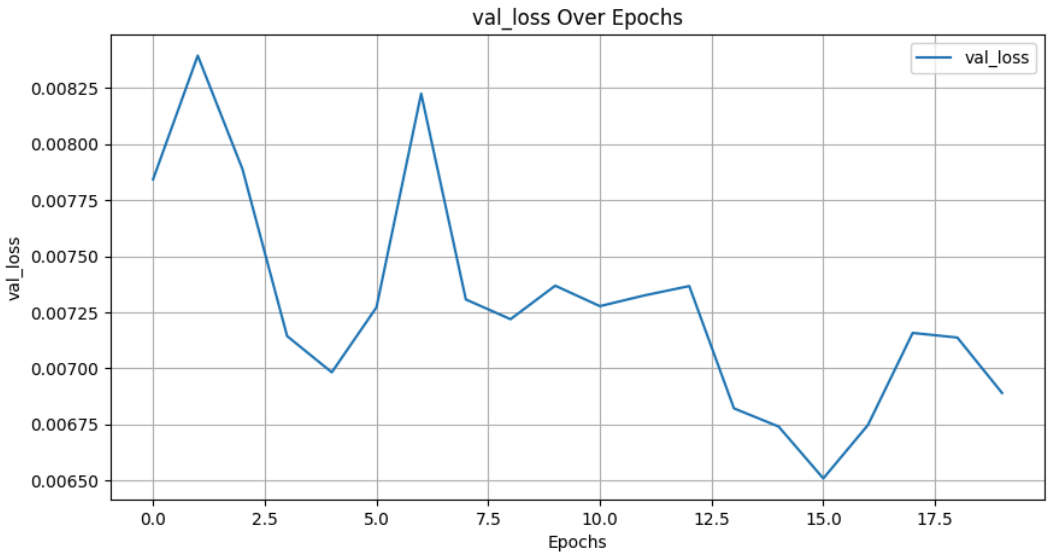

running_loss Over Epochs
running_loss
0.36
0.34
0.32
0.30
0.28
0.26
running_loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


cont_training_loss Over Training step
cont_training_loss
1.75
1.50
1.25
1.00
0.75
0.50
0.25
cont_training_loss
0
1000
2000
3000
4000
5000
Training step


train_acc Over Epochs
train_acc
0.880
0.875
0.870
0.865
0.860
0.855
train_acc
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


train_loss Over Epochs
train_loss
0.004
0.003
0.002
0.001
0.000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


val_acc Over Epochs
val_acc
0.880
0.875
0.870
0.865
0.860
0.855
0.850
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


val_loss Over Epochs
val_loss
0.00825
0.00800
0.00775
0.00750
0.00725
0.00700
0.00675
0.00650
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs