AML-Final Project

Image Segmentation of Aerial Forest Pictures

Department of Computer Science
Ruprecht-Karls-Universität Heidelberg

Nico Bruder, Laurin Ernst, Philipp Tepel

Heidelberg, August 31st, 2022

Contents

1 Introduction

Satellite imagery has been around for a long time now and with a rising number of satellites and increasing access to their data, the need for automated analysis has grown significantly over the past years. This analysis can be applied to a lot of areas, one of which is the estimation of forest area through satellites; it can be used by environmental departments to monitor growth of forests over a span of time or cartographers can utilise it to create a map with deatiled forest areas without having to go over the images by hand.

Creating a tool to identify forest on a satellite image can be translated to a semantic segmentation of said image. This raises some interesting questions about what we even want to define as a forest. How many trees does an area need to become a forest? Do we need to differentiate between coniferous, deciduous and mixed forests? Do we want to recognise forests in all seasons, even though they look very different in summer, fall and winter for example? How large does a clearing have to be to be counted as not forest? The problem with these questions is that they are mostly answered by the data set we use to train our model, because creating a dataset on our own is too tedious of a task and so if we find a decent dataset, we have to settle for that.

When attempting semantic segmentation of a given image, the first thing that comes to mind is the U-Net structure, which has been the standard computer vision approach to this kind of problem for a longer time. However, our task is a bit simpler than some many semantic segmentations, because we will just classify pixels as forest or non forest and not start identifying fields etc. The dataset we found on kaggle.com provided us with enough data to train neural networks and gave a RGB picture from which we can calculate a mask of the same dimension to identify the forest parts of the image. After training and evaluation we will see wether the model found a satisfying characterisation of a forest or not.

2 Dataset

2.1 Data Preprocessing

The dataset from Kaggle of 5108 arial forest images and its mask respectively. The where all already formatted to 256 x 256 pixel. Furthermore the images looked clean, hence we did not need to do a lot of cleaning of the data.

Each image was saved in the jpg format meaning our dataset had a size of approximately 185 MB. Our first intuition was that, since jpg is a compressed file format, loading jpgs could take longer than loading from uncompressed files. Thus we tried saving our data first as json and second in pythons pickle dataformat. Both times this lead to a massively inflated dataset of around 5 GB in size.

When timing dataloading, loading jpgs using pythons Python Image Library was actually faster than loading jsons using pandas. Hence we load our data from jpegs.

A big part of our work is our custom dataloader. It arose from our need for a lot of customizability as it not only loads data from our custom source but also formats our masks. Every mask consists of white pixels with value 255 and black pixels with value 0. Trying to make our loss easier to interpret we decided to set each white pixel to 1. In the end our dataloader works similar to a standard dataloader from pytorch for example. On initialization you set the size of the Dataset you want to use, the set of the trainings set, the batch size you want and optionally the batch set for the test set (we used this as we tried our program on different machines thus for more efficiency it was important to be able to control the batch size for the test set).

The batch loading itself is done by the function batchloader. On each new epoch the trainingset is shuffled randomly using numpys shuffle function. The epoch_ finished function is used to check wether the dataset has been completely stepped through and if therefore a new epoch has to be started.

Our image data is returned as pytorch tensors respectively containing three arrays for every image one for red one for green and one for blue. Our mask data is returned as a pytorch tensor consisting of only one array per mask containing of zeros and ones.

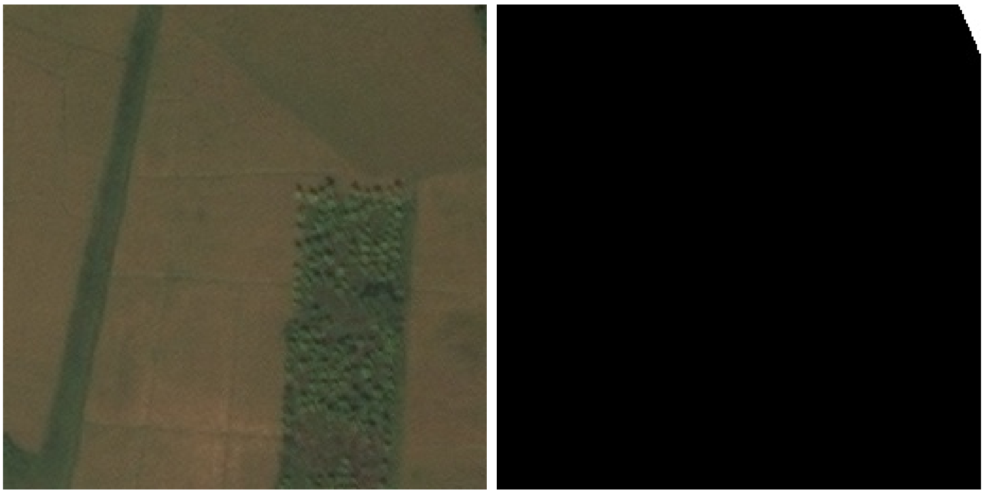

Figure 2.1: Faulty Data: The mask (left, black means no forest) not matching the given satellite image

2.2 Problems with the Dataset

When evaluating the training of our models, we looked at the data on which our model performed the worst. One thing that became noticable was the lacking quality of some of the training data as well as some controversial masking.

For example, in **??** we can see a satellite image on the left and its given mask on the right; remember that black means no forest. In the image there is one big strip of forest in th south, but the whole image is classified as no forest. This is only one example of images where large chunks of forest are not classified as forest and so make it harder for our neural networks to properly train on the data. It is to be expected that a model either overfits to this faulty data, so that it does get a good score on them but fails to classify to forest, or the model cannot accurately segment this image, because the mask does not fit the picutre.

Some data also classifies large chunks of just ground without any trees as forest. Depending on the size of those chunks, one could argue that it is just a clearing in the forest and thus qualifies as part of the forest. But the more often this appears, the harder it becomes for the networks to tell a clearing in the forest and just a plain field apart. An example of this can be seen in **??**, where the reddish part is what is classified as forest and the only non forest pixels are a small part in the upper right part of the image.

Figure 2.2: Controversial Classification: The red part resembles the pixels classified as forest, a lot of brown clearings are classified as forest

3 Naive Segmentator + Jaccard?

3.1 Performance measures

In order to evaluate how good our model performs, we need a metric to measure the precision of the predicted masks. In the following, A is the set of pixels which are actually forest and B is the set of pixels which our segmentator classified as forest. Y is the set of pixels of the true mask, $\hat{Y}$ the set of predicted pixels. The total number of pixels is $n = 256^2$.

An obvious approach for comparing the similarity between A and B is the so-called simple matching coefficient **SMC**. The **SMC** is given by

$$\mathbf{SMC} := \frac{|Y \cap \hat{Y}|}{n}, \tag{3.1}$$

so it is simply computed as the proportion of correctly predicted pixels.

Though this method might seem easy and intuitive, it has too serious drawbacks to be used as a meaningful performance measurement. The main issue is that forest and non-forest regions are treated symmetrically, even though the goal of our segmentator is only the correct prediction of the forest regions. This aspect is clarified by the following example:

(a) True mask (b) Prediction

Figure 3.1: The gray regions depict forestial, the black regions non-forestial areas.

We assume the left mask shows the true forestial areas, the right mask is the prediction of our model. Clearly, there is no overlap between the forest regions, so $|A \cap B| = 0$, but there is still overlap between the non-forest regions in the centre of the images. Hence, we get $\mathbf{SMC} = \frac{128 \cdot 256}{256 \cdot 256} = \frac{1}{2}$, even though the forest prediction was completely wrong. This leads to the unpleasant fact that **SMC** is not really useful for measuring the performance of our prediction.

As an alternative, we could change the denominator of the **SMC** such that we only compute the proportion of correctly predicted pixels in the set of the total (predicted and actual) forestial areas. Thus, we get

$$J(A, B) := \frac{|A \cap B|}{|A \cup B|}, \tag{3.2}$$

which can be rewritten as

$$J(A, B) = \frac{|A \cap B|}{|A| + |B \setminus A|}. \tag{3.3}$$

This coefficient is very common and known as Jaccard index $J(A, B)$.

3.2 Expected Jaccard Index of a Random Segmentator

To get a better feeling of the Jaccard index, we wanted to calculate the expected index a simple random segmentator gives to compare it to our models. This random segmentator would just assign a 1 or 0 with probability $\frac{1}{2}$ to each pixel, where 1 means forest and 0 means no forest. Of course, this segmentator does not perform well, but it gives a benchmark for comparison.

Now, let us look at the problem mathematically. Take a arbitrary but fixed satellite image with its mask and define n as the number of pixels it has (for us, that is 256^2 in every image). Define m as the number of pixels which are forest (meaning they have value 1 in the mask). Obviously, this number is not fixed throughout the images we have, but it is a deterministic constant for every single image.

To define the random segmentator, we define n random variables $X_1, \ldots, X_n$ that assign each pixel 0 or 1 with probability $\frac{1}{2}$. Thus,they also have a probability of $\frac{1}{2}$ of assigning the right value for the pixel. Also, each X_i is stochastically independent from the others. This lets us conclude that

$$X_1, \ldots, X_n \overset{\text{iid}}{\sim} \text{Bin}(1, \frac{1}{2}) \tag{3.4}$$

where the binomial distribution stands for X_i assigning **the right value or not** instead of 1 meaning forest and 0 meaning no forest.

If we apply this to the Jaccard index, we first have to look at our sets A and B. A is a deterministic set, because the forest pixels in a single image are fixed. This also means that $|A| = m$ per definition. On the other hand, B is the random set of pixels, our segmentator classified as forest. $A \cap B$ is the subset of pixels classified as forest, which truly are forest. Thus,

$$Y := |A \cap B| = \sum_{i \in A} X_i \tag{3.5}$$

and because the X_i are i.i.d. and $|A| = m$ we conclude that $Y \sim \text{Bin}(m, \frac{1}{2})$. For $A \setminus B$, we can proceed similarly:

$$Z := |A \setminus B| = \sum_{i \notin A} X_i \tag{3.6}$$

and because the X_i are i.i.d. and $|A| = m$ we conclude that $Z \sim \text{Bin}\left(n - m, \frac{1}{2}\right)$. Since Y and Z are independent by definition, we can write

$$\begin{aligned} \mathbb{E}\left(J(A,B)\right) &= \mathbb{E}\left(\frac{Y}{m+Z}\right) \\ &= \mathbb{E}(Y) \cdot \mathbb{E}\left(\frac{1}{m+Z}\right) \end{aligned}$$

With Taylor's theorem, we can approximate

$$\mathbb{E}\left(\frac{1}{m+Z}\right) \approx \frac{1}{m + \mathbb{E}Z} \tag{3.7}$$

and so

$$\mathbb{E}(J(A,B)) = \frac{\frac{m}{2}}{m + \frac{n-m}{2}} = \frac{m}{m+n} \tag{3.8}$$

n is a constant throughout the dataset, but m differs for every image. The average of the dataset is $m = 40900$, so we can say that our models perform better than guessing if their average Jaccard index is greater than $J = 0.39$ (confirmed by random simulations). We could try to improve the random segmentator by adjusting the guess rate according to the proportion of forest in the dataset, but experiments show that this does not make a big difference in performance.

3.3 Segmentation without Neural Networks

As we have seen in the final project of the lecture "Fundamentals of Machine Learning", rather simple machine learning methods could outperform far more complex structures. Thus, we decided to implement a model that does not rely on neural networks.

The model we created works in the following way:

1. Use a common segmentation algorithm to split the satellite image in segments.
2. For each segment, compute features.
3. Use a model like logistic regression or a support vector machine to classify each segment as forest or non-forest.
4. Join all the segments (now consisting of binary data) to obtain a prediction mask.

3.3 Segmentation of images

In order to segment the satellite images in the first step, we use the Felzenszwalb-Huttenlocher algorithm (Zitat!!!!!!!!!!!!!!!!!!!)

4 Unet

4.1 Overall training procedure

For our training we were able to use a server equipped with two AMD EPYC 7313 16-Core Processors, 256 GB RAM and four GPU NVIDIA A40 amp architecture CUDA® processing units running Ubuntu 18.04.6. Our training ran on only one of those GPUs which had 47.85 GB of memory.

All of this included, one training run of 15 epochs took about an hour. The biggest bottleneck being dataloading since the images were only saved on a mounted storage and not locally on the server.

In every run we trained four different models. These were trained with either mean-squared error loss or Cross entropy loss each then paired with either stochastic gradient descent or Adam as optimizer.

Our first approach was to train our models using a lot of epochs. But this method proved inefficient as trainings and test loss always seemed to stop decreasing significantly after around 10 epochs. Since pytorch initializes models with random parameters, we moved to training our model using only 15 epochs but doing so six times with every model, to increase our chance of finding a good optimum, and decreasing the chance of local minima.

One such run took around 24 hours.

For every run we saved the training and test loss of every epoch and the final weights of our trained Unets enabling us to reproduce segmentation's.

4.2 Choice of Loss functions and optimizers

We chose mean squared error as the first loss function as it is easy to understand and interpret. Additionally we have seen many papers on segmentation use the cross-entropy loss. Therefore we decided to use it as well to see if it would improve training. We specifically used cross-entropy loss with a sigmoid layer integrated as this is numerically more stable than using a separate sigmoid layer followed by cross-entropy loss.

For optimizers we went a similar way. We chose stochastic gradient descent for its simplicity and speed. Since we have a training set containing 4000 images, we were concerned about speed and thought stochastic gradient descent could have an advantage in this regard. Additionally we chose the ADAM algorithm as it is a proven optimizer for image segmentation. This algorithm is complexer since it also takes higher moment into consideration. Therefore we thought it could give us more accurate results.

4.3 Training hyperparameters

4.3 Loss functions

For both the mean squared error loss and the cross-entropy loss we used mean reduction meaning the loss function would return the mean loss of the input batch.

4.3 Optimizer parameters

Stochastic gradient descent

For stochastic gradient descent we used an initial learning rate of 10^{-4} which was the biggest learning rate possible. When using a bigger initial learning rate the loss would diverge after three to four batches. Additionally we reduced the learning rate by a factor of ten after every five steps. As we already reduced the learning rate every five epochs, we decided not to use weight decay as lowering the learning rate should reduce overfitting sufficiently. Training confirmed this suspicion as we never had a problem with overfitting.

ADAM

When using ADAM with cross-entropy loss, we used an initial learning rate of 10^{-4}. In combination with mean-squared error loss, we were able to use an initial learning rate of 10^{-3} which had a notable change in the speed of initial convergence. Again we lowered our learning rate by a factor of ten every five epochs and thus did not use weight decay. We left β_1 and β_2 as the standard values of 0.9 and 0.999. All other parameters where left as the pytorch standard values as well.

In summary we mostly changed the learning rate as we had problems with divergence. Since pytorch itself is already very optimized we decided to leave the remaining parameters as default.

4.3 Batch size

In order to get a meaningful gradient we try to maximize the batch size. In this case we would only be limited by the GPUs memory. But we also wanted the gradient of every batch to have comparable significance, hence we tried to find a batch size which can divide 4000. This lead to the batch size being 160. Since for testing all of these considerations were not as important we decided to use a test batch size of 100 as our training ran sufficiently fast.

4.4 Mean squared error loss

Stochastic gradient descent

When training with stochastic gradient descent and mean squared loss the loss always converged to around 0.28 with 0.263 being our best loss.

As you can see in table 4.1, after around 10 epochs the loss does not change much. This lets us conclude, that the model converged to a local minimum. Another explanation could be, that the learning rate was decreased too much thus slowing down convergence. But the same phenomenon occurred when training with a constant learning rate at around the same epoch.

The fact that our final loss varies so much between our run, with the losses converging at the same time, shows that doing a lot of runs with random initialization of parameters is better than doing one long run.

If we now take a look at our training loss, we can see that it is in around the same area. Thus our training has no problem with overfitting. Here we also have the same effect as before in training, after around 10 epochs the loss does not change by much as seen in table 4.2.

In conclusion, the training went well. Test and training loss both converged and are both in the same area. Therefore it seems that our model did not overfit.

Figure 4.1 shows the loss over 15 epochs for our best fitting model. Again this enforces our conclusion, that we had no overfitting and that the model converged. As from epoch 10 onwards no real movement is seen.

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	4.4693975	12.184066	2.0131614	1.5307927	4.0228357	0.50962913
2	0.3200431	0.3480549	0.32500157	0.30779946	0.3079086	0.31367758
3	0.29800013	0.34193915	0.310398	0.2957203	0.300468	0.2935942
4	0.28355855	0.33770123	0.30171904	0.28813368	0.29583827	0.28231227
5	0.27282694	0.3341225	0.29564404	0.28278244	0.29300708	0.2780813
6	0.26709262	0.3317924	0.292984	0.27965817	0.29057825	0.2757975
7	0.26613078	0.331356	0.29230326	0.27914998	0.29010746	0.27554354
8	0.26527855	0.33101973	0.29184073	0.27868813	0.28978932	0.27519086
9	0.26449108	0.330619	0.2914097	0.2782371	0.289464	0.27500996
10	0.26374942	0.33024815	0.29096514	0.27772996	0.289188	0.27473155
11	0.26331055	0.32998294	0.29070583	0.27745888	0.28898048	0.2745347
12	0.26324356	0.32994178	0.2906588	0.27741736	0.2889499	0.27450302
13	0.26317427	0.329906	0.29061228	0.27737102	0.2889224	0.27447924
14	0.26311314	0.32987177	0.29056343	0.27732262	0.2888908	0.2744483
15	0.2630431	0.32983643	0.2905223	0.2772747	0.2888606	0.2744221

Table 4.1: Training loss of all 6 runs using MSE and SGD

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.33122975	0.34711483	0.33074474	0.31236193	0.30903655	0.31827042
2	0.30577785	0.34202188	0.31160694	0.29639706	0.3001178	0.29576242
3	0.28950268	0.33610505	0.30135122	0.2884548	0.2953469	0.27975687
4	0.27809516	0.3319018	0.2946329	0.28317657	0.29173446	0.2751834
5	0.2697628	0.32962403	0.28951994	0.27929276	0.28829837	0.2705783
6	0.26835245	0.32850268	0.28909752	0.27869415	0.28819934	0.27042475
7	0.26751754	0.3282821	0.2888627	0.27789146	0.2880678	0.27025133
8	0.26677686	0.3277812	0.2883609	0.27749544	0.28793824	0.27011687
9	0.26600352	0.32746282	0.28803816	0.27703398	0.28743234	0.2696476
10	0.26539832	0.327034	0.28781492	0.27668744	0.28726363	0.26951388
11	0.26533362	0.3269988	0.28774983	0.27663186	0.28723428	0.26947123
12	0.26526618	0.3269564	0.28769815	0.2765826	0.28720677	0.26944548
13	0.2652073	0.32692036	0.28763285	0.2765277	0.28716826	0.26940757
14	0.26514426	0.32688275	0.2875785	0.27647343	0.28713912	0.2693817
15	0.2650814	0.32684293	0.28752634	0.27641827	0.28711313	0.26935467

Table 4.2: Test loss of all 6 runs using MSE and SGD

ADAM

When training using ADAM as an optimizer, our training loss was always around 0.22. The best run had a final training loss of 0.2.

Looking at table 4.3 again the loss seems to converge after around 10 epochs. Additionally after around 6 epochs we can see that the loss does not change more than 10^{-2}. Like in the

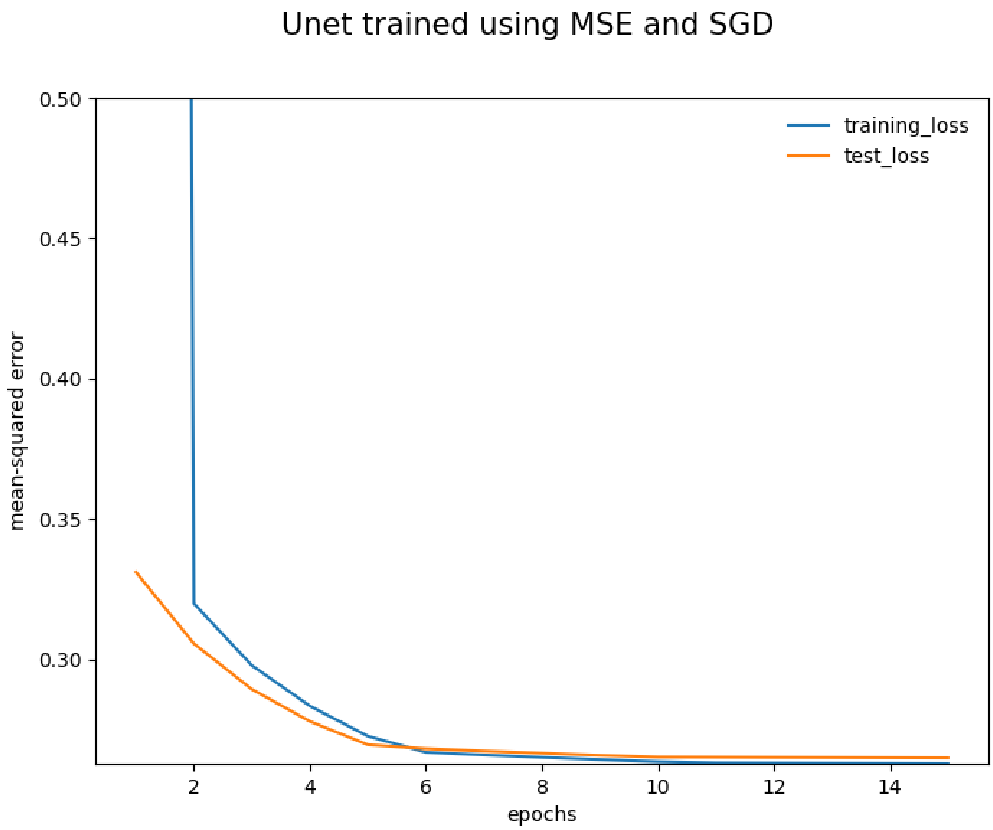

Figure 4.1: Loss over epochs for best fitted model

case of SGD the losses vary by quite some margin. Thus again we can conclude that the strategy of training the network multiple times each with randomized initial parameters is better than one long training run.

Taking a look now at the test loss, we can again see, that it is about the same as the training loss. Therefore overfitting again seems unlikely. We can again see convergence and slowing down of convergence after around 10 epochs.

Finally we take a look at figure 4.2 which displays the test and training loss for the model with the best fit. The model converges nicely with the exception of the sixth epoch, here we get a slight increase of the test loss and the training loss. In the end we can also see the start of what seems like overfitting. The test loss gets bigger than the training loss and the gap starts to widen.

To conclude, training using MSE and Adam works well, after adjusting the learning rate every run converged to a local minimum.

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	16.468616	32.87254	41.616203	12.032867	26.262264	6.64174
2	0.3016394	0.35413027	0.3611106	0.33067077	0.31112534	0.38042334
3	0.2636294	0.27745226	0.30735505	0.314712	0.24085094	0.28627422
4	0.27392736	0.25761652	0.2715208	0.30386814	0.22736278	0.25913927
5	0.23114581	0.2535884	0.24971901	0.2525159	0.21356352	0.23446146
6	0.21954322	0.2502232	0.2319426	0.22899482	0.20426998	0.21969064
7	0.2159541	0.24798293	0.22944567	0.22487594	0.20471057	0.2093104
8	0.21284859	0.24727972	0.22790462	0.22264333	0.20271398	0.20635623
9	0.21017799	0.2465546	0.22756189	0.2198942	0.20233981	0.20524022
10	0.207757	0.24546298	0.22529072	0.21787128	0.20136292	0.204171
11	0.20608918	0.2448993	0.2246989	0.21646391	0.20061451	0.20347168
12	0.20551556	0.24466546	0.2241403	0.2162385	0.20064121	0.20331205
13	0.2055013	0.24453524	0.22393808	0.21613634	0.20059872	0.20313133
14	0.20539114	0.24443771	0.22374646	0.21597092	0.20051412	0.20305358
15	0.20504224	0.2442546	0.22357792	0.21557988	0.20041414	0.20296371

Table 4.3: Training loss of all 6 runs using MSE and ADAM

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.30238324	0.47267693	0.48997444	0.3304595	0.33760715	0.544152
2	0.31420705	0.30845478	0.3268845	0.44472593	0.2697537	0.33348393
3	0.32230446	0.25980768	0.28840277	0.24429911	0.218167	0.27124432
4	0.2573785	0.26047134	0.25550815	0.2575479	0.2111586	0.24399836
5	0.22700256	0.26072893	0.24103026	0.2253271	0.20759247	0.23637411
6	0.22430123	0.25108248	0.23401155	0.22546947	0.21013544	0.22305357
7	0.21912107	0.24997404	0.23316166	0.22113423	0.2072945	0.21716589
8	0.21662062	0.2492017	0.23347506	0.2194097	0.20544857	0.21540354
9	0.21485323	0.24881515	0.23158182	0.21867856	0.20490159	0.2140774
10	0.2123616	0.24691407	0.23091581	0.21738729	0.20447801	0.21366231
11	0.21198182	0.24717368	0.2289134	0.21665032	0.20441326	0.21291277
12	0.21170287	0.24706934	0.2287653	0.2165657	0.20436418	0.21275353
13	0.21137412	0.24683827	0.22865824	0.2161448	0.20442478	0.21277241
14	0.21128596	0.24668425	0.22856991	0.21600612	0.20431736	0.2125489
15	0.21090196	0.24694021	0.2284836	0.21595462	0.20422272	0.21248607

Table 4.4: Test loss of all 6 runs using MSE and ADAM

4.5 Cross entropy loss

Stochastic gradient descent

For training using cross-entropy loss and stochastic gradient descent we got a final training loss of about 0.65 in every run. With our best run converging to 0.6357.

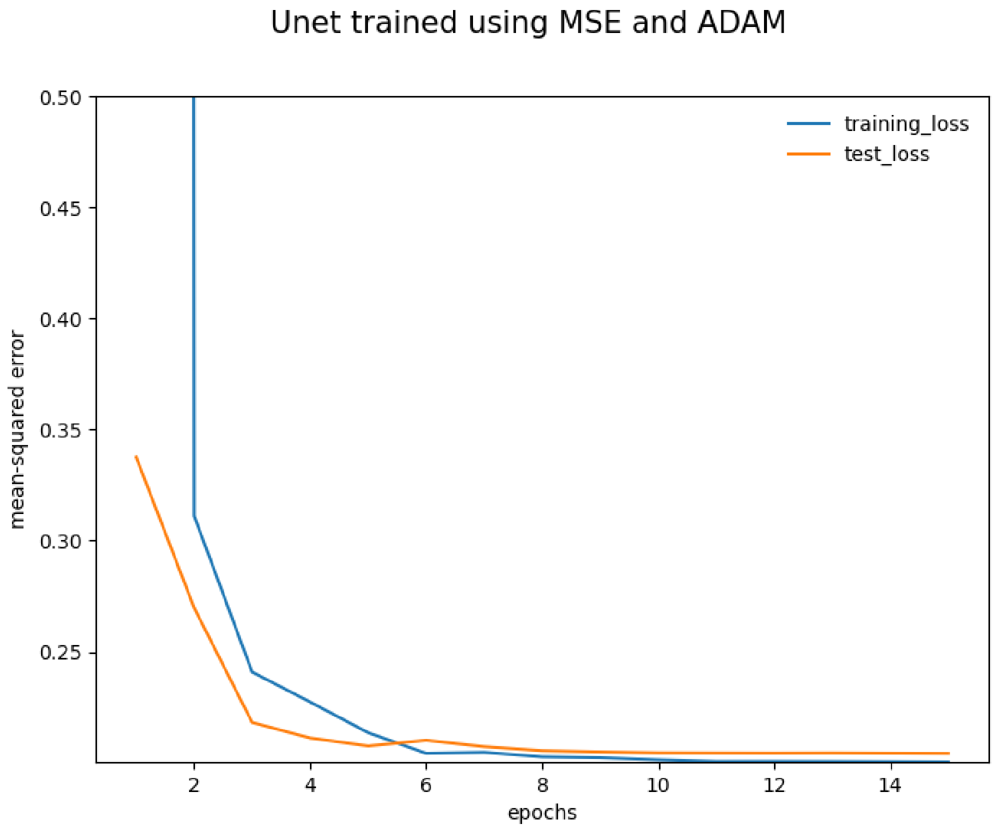

Figure 4.2: Loss over epochs for best fitted model

As we can see in table 4.5 the loss seems to change only slightly after the 10th epoch. The losses stay relatively consistent in between runs. Eventhough they vary in absolute terms about the same, they vary much less in relative terms as they are almost three times as high.

Looking now at the test loss in table 4.6 we see approximately the same result as in the case of the training loss. The loss seems to be in the same area letting us conclude we did not overfit.

Finally we can take a look at the best run for this combination of loss and optimizer in figure 4.3. This again visualizes that after an initial steep drop, the loss seems to change only slightly after 10 epochs and the test loss gets bigger than the training loss.

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.8173364	0.76068455	0.77815217	0.72728735	0.6711792	0.6598618
2	0.69916785	0.6946063	0.7287909	0.6892015	0.6576305	0.6485882
3	0.6742714	0.68608665	0.7061338	0.6786511	0.6544444	0.6442214
4	0.6648476	0.67854613	0.6887577	0.6702527	0.6513985	0.6403597
5	0.6606141	0.673209	0.6761121	0.66377324	0.6491724	0.63790745
6	0.65787476	0.6691471	0.6713925	0.6601952	0.64789677	0.6364391
7	0.6574388	0.66859066	0.6705435	0.65960544	0.64755905	0.63631696
8	0.65712637	0.66813964	0.66985464	0.65910095	0.64732987	0.6361612
9	0.65684086	0.6675795	0.669216	0.65860045	0.64706326	0.63597316
10	0.656529	0.66708374	0.668607	0.6581397	0.6468631	0.6358646
11	0.65631473	0.6667787	0.66823846	0.657823	0.6466894	0.635735
12	0.6562815	0.6667253	0.66818315	0.6577744	0.64666617	0.6357188
13	0.656252	0.6666797	0.66812503	0.65771925	0.64664227	0.63570505
14	0.65622205	0.6666302	0.6680641	0.6576743	0.6466213	0.6356925
15	0.6561918	0.6665789	0.66800225	0.65762556	0.64659595	0.6356761

Table 4.5: Training loss of all 6 runs using BCE and SGD

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.71672803	0.69796896	0.73800945	0.69404376	0.6671659	0.6569546
2	0.68283176	0.68978554	0.7144342	0.68395793	0.6645384	0.6528363
3	0.66726273	0.6829562	0.6955856	0.67420524	0.66188	0.64957654
4	0.66106933	0.677026	0.68176156	0.6681182	0.6601306	0.647834
5	0.65685767	0.67209834	0.6730943	0.6627164	0.6584907	0.64603174
6	0.6560621	0.67167234	0.6722516	0.66223365	0.65839386	0.64564204
7	0.6555674	0.6712711	0.6716326	0.66199446	0.65831107	0.6456054
8	0.65524787	0.6708662	0.67105925	0.6613193	0.6581983	0.64545655
9	0.6548931	0.67045116	0.67050034	0.66097677	0.65802324	0.6454915
10	0.65461195	0.67000425	0.6699379	0.6606827	0.65787286	0.6453148
11	0.6545693	0.66996336	0.669888	0.6606309	0.6578581	0.6452977
12	0.65452975	0.6699253	0.669835	0.66057676	0.6578409	0.64528203
13	0.65449786	0.66988266	0.66978145	0.6605275	0.65782756	0.64526963
14	0.65445566	0.6698414	0.6697256	0.6604745	0.65781116	0.6452546
15	0.654421	0.6698	0.6696722	0.66043127	0.65779793	0.6452409

Table 4.6: Test loss of all 6 runs using BCE and SGD

ADAM

As with MSE, the training loss is significantly when training with ADAM. We got an average training loss off around 0.51 with our lowest run converging to 0.507.

This time the rate of change seems to be still quite high after 15 epochs. Maybe we should have used more epochs in this case. Still the results seem to be consistent over all 6 runs

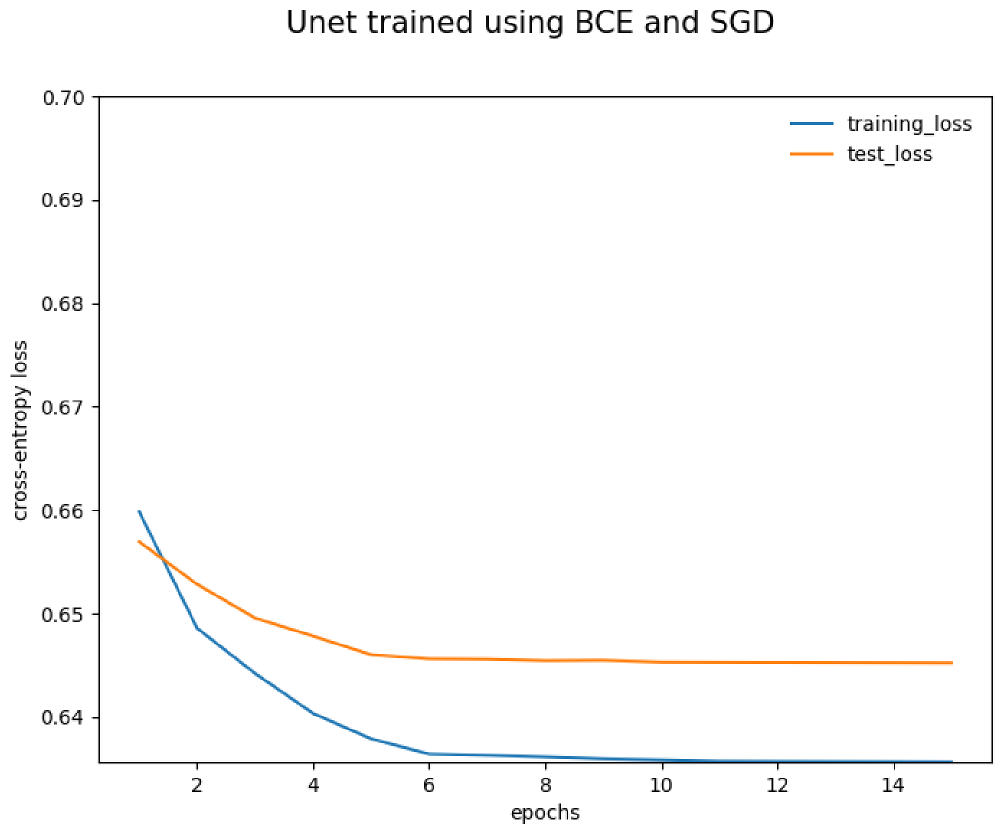

Figure 4.3: Loss over epochs for best fitted model

we had no big outlier and rate of change has already slowed down significantly after 12 epochs.

Again the test loss seems to be approximately the same as the training loss. Interestingly in our best training run,the test loss seems to be significantly worse than on average. This could mean, that we should have used more epochs to train.

Finally taking a look at our best trainings run once more in figure 4.4 we can see that there is a big difference between training and test loss after the 5th epoch. This phenomenon only occurred in this run. In all the other runs the test loss is much closer to the training loss.

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.65830934	0.70530957	0.73098403	0.7553478	0.6963218	0.6989336
2	0.6067861	0.6118383	0.622002	0.6177935	0.6163941	0.6030678
3	0.57389337	0.57276547	0.5896112	0.5814634	0.5848018	0.5820509
4	0.572949	0.5601315	0.5770633	0.5574847	0.56839913	0.5516847
5	0.5561481	0.5424935	0.55888504	0.5507084	0.5757851	0.5312709
6	0.5364449	0.5338383	0.5432497	0.5312137	0.5513115	0.51587135
7	0.532941	0.5256305	0.5389623	0.52633727	0.5476161	0.5137691
8	0.53010124	0.5225718	0.5353402	0.5228115	0.54461324	0.51234573
9	0.5266955	0.5205638	0.5325129	0.51956135	0.5421652	0.5114221
10	0.5252609	0.51839405	0.52926993	0.51767886	0.53950185	0.5087806
11	0.52121717	0.5170028	0.52693367	0.51388	0.53678423	0.507745
12	0.5208412	0.5163002	0.52617997	0.5136164	0.53651273	0.5067676
13	0.52059245	0.51618713	0.5258741	0.5132377	0.5361978	0.5066837
14	0.5201951	0.51604027	0.52547926	0.51307505	0.535855	0.5064898
15	0.5198292	0.5158567	0.52528584	0.5124003	0.5355725	0.50654024

Table 4.7: Training loss of all 6 runs using BCE and ADAM

epoch	1st Run	2nd Run	3rn Run	4th Run	5th Run	6th Run
1	0.62222517	0.6304792	0.650266	0.64774585	0.6450115	0.6502997
2	0.6127994	0.5911052	0.61116695	0.6112345	0.6006177	0.59954464
3	0.56413865	0.5533587	0.5838492	0.57314724	0.5676928	0.6095978
4	0.60099006	0.54774123	0.5802889	0.57163227	0.5549136	0.5582189
5	0.54441684	0.56050503	0.5649236	0.5489276	0.5546435	0.550628
6	0.5395218	0.5285362	0.55067337	0.54289705	0.5512127	0.53947175
7	0.5361257	0.5262004	0.5480118	0.54036915	0.54933226	0.5381737
8	0.53230333	0.5228678	0.54828644	0.53706986	0.5442867	0.53867507
9	0.53142047	0.5205792	0.5409243	0.5374959	0.5430919	0.53440034
10	0.5264803	0.5184883	0.53914946	0.52918005	0.5380025	0.5344262
11	0.52641296	0.5179452	0.53895676	0.5291239	0.53749406	0.53313345
12	0.5257408	0.51773334	0.5374101	0.5288156	0.5371681	0.53354484
13	0.52537686	0.51755416	0.5376477	0.52927	0.53689975	0.5333335
14	0.5251515	0.5186142	0.5370112	0.52886564	0.5365701	0.53299874
15	0.5245699	0.51723737	0.5370193	0.5278556	0.5362334	0.5325686

Table 4.8: Test loss of all 6 runs using BCE and ADAM

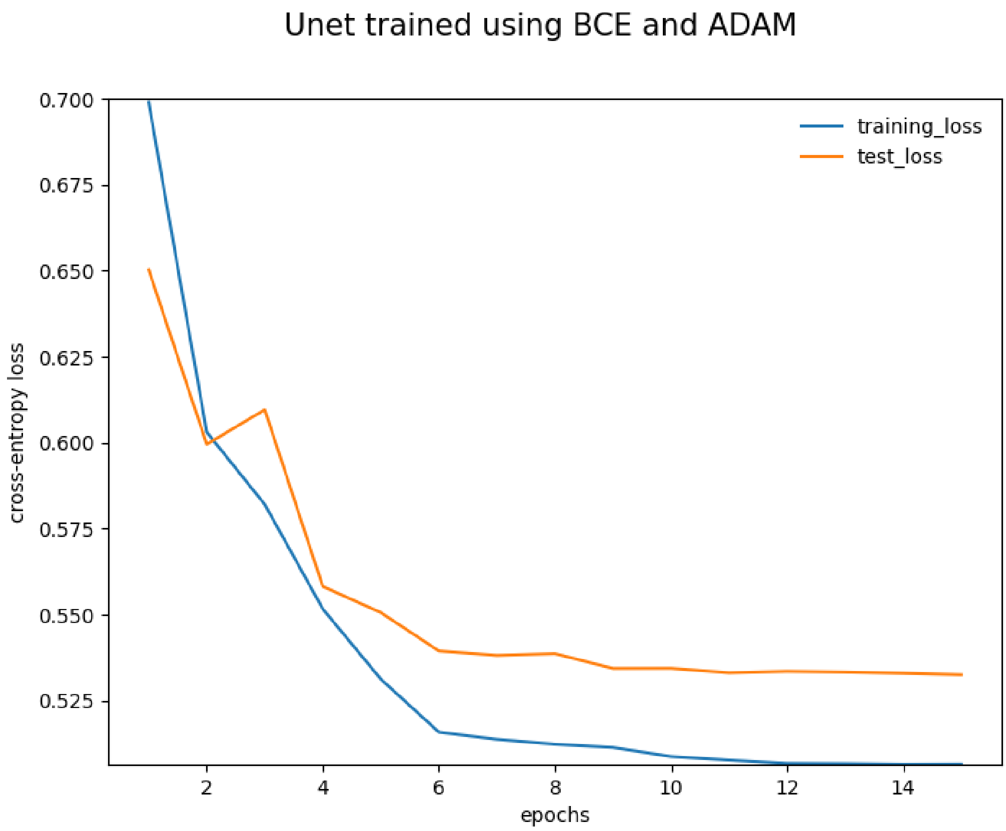

Figure 4.4: Loss over epochs for best fitted model

4.6 SGD compared to ADAM

Taking a look at figure 4.5 we can compare the effectiveness of SGD and ADAM. In all cases the models trained using ADAM have a steeper drop in the beginning compared to the models trained using SGD. After 5 epochs the change of loss gets smaller and both models seem to converge. But SGD remains higher than ADAM. This could have different explanations. Since we decreased the learning rate every five epochs, it could be that SGD just converges slower than ADAM thus it does not have enough time under a big learning rate to go down enough. At the same time it could also be possible, that ADAM is just better at minimizing and that SGD only circles around a local minimum and thus we should have decreased the learning rate more such that SGD start taking small steps towards the local minimum and does not overshoot it every time.

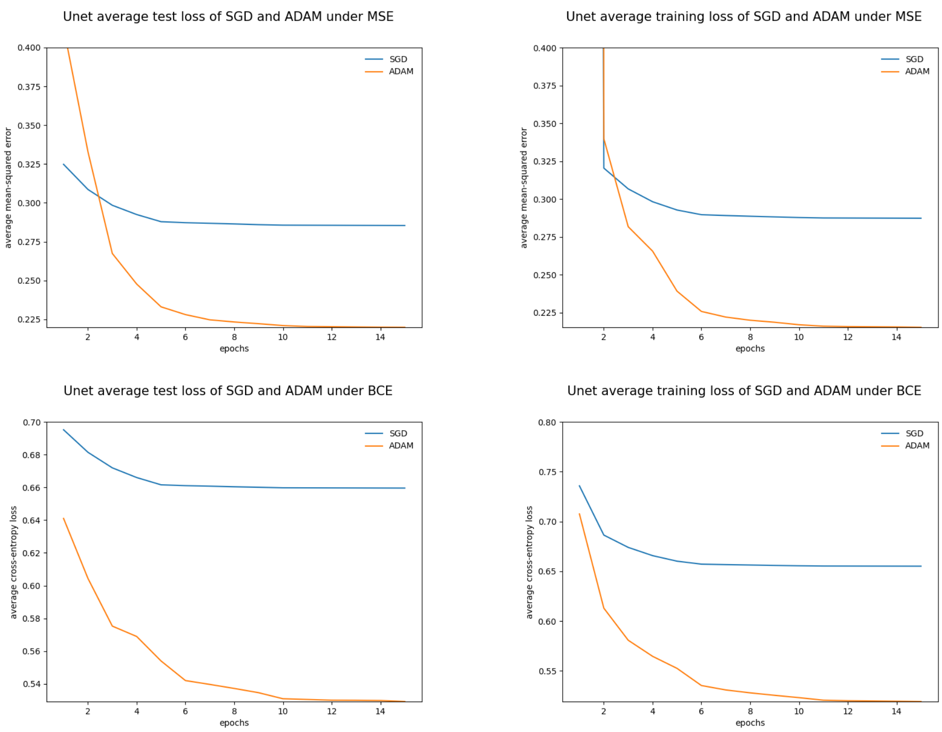

Figure 4.5: Loss of model trained with SGD vs ADAM

5 SatNet

After implementing the U-Net, we wanted to try a slightly different approach of using convolutional networks, so we researched other methods used in satellite image analysis and found a paper (see **zitieren**) segmentating satellite images using a so called "SatNet". They used the SatNet to identify roads and buildings on satellite images, which is not too far from our task, so we used the structure they provided in the paper (**??**).

5.1 Introduction to SatNet

As explained in **zitieren**, the SatNet is heavily inspired by the ResNet, because in the SatNet there are few connections which do not add the input to the convoluted output. First of all, this reduces the risk of vanishing gradients, because no matter the learned function, the identity still provides a large gradient for the top layers. Second, the skip connections allow the SatNet to be much deeper and have much more layers than other ConvNets, because it is very easy for the layers to learn the identity function and thus, these layers can extract information only if necessary.

From an overall structure, the SatNet still encodes and then decodes the images, similarly to the U-Net. The big difference here is that the SatNet downsamples the images by striding instead of pooling. In our structure, we have two convolution layers with a stride parameter of two, the others all keep the dimension constant. After the encoding, the SatNet quickly upsamples the codes with two transposed convolutions also using a stride parameter of two. The advantage of downsampling with stride is its efficiency in computation, because such a layer downsamples and convolutes the data at the same time and with less parameters.

The SatNet is designed such that after every downsampling there are a lot of convolution layers without reducing the dimension. This allows the network to "adjust" to the loss of information and gives it time to finely extract all information from the new downsampled data before downsampling it again. Overall, SatNet does not reduce the data as far as the U-Net does, because the lowest dimension SatNet reaches is a 63x63 grid as opposed to the 28x28 grid of the U-Net. However, the SatNet keeps the number of channels used low, increasing the number only when downsampling, contrary to what the U-Net does. So the

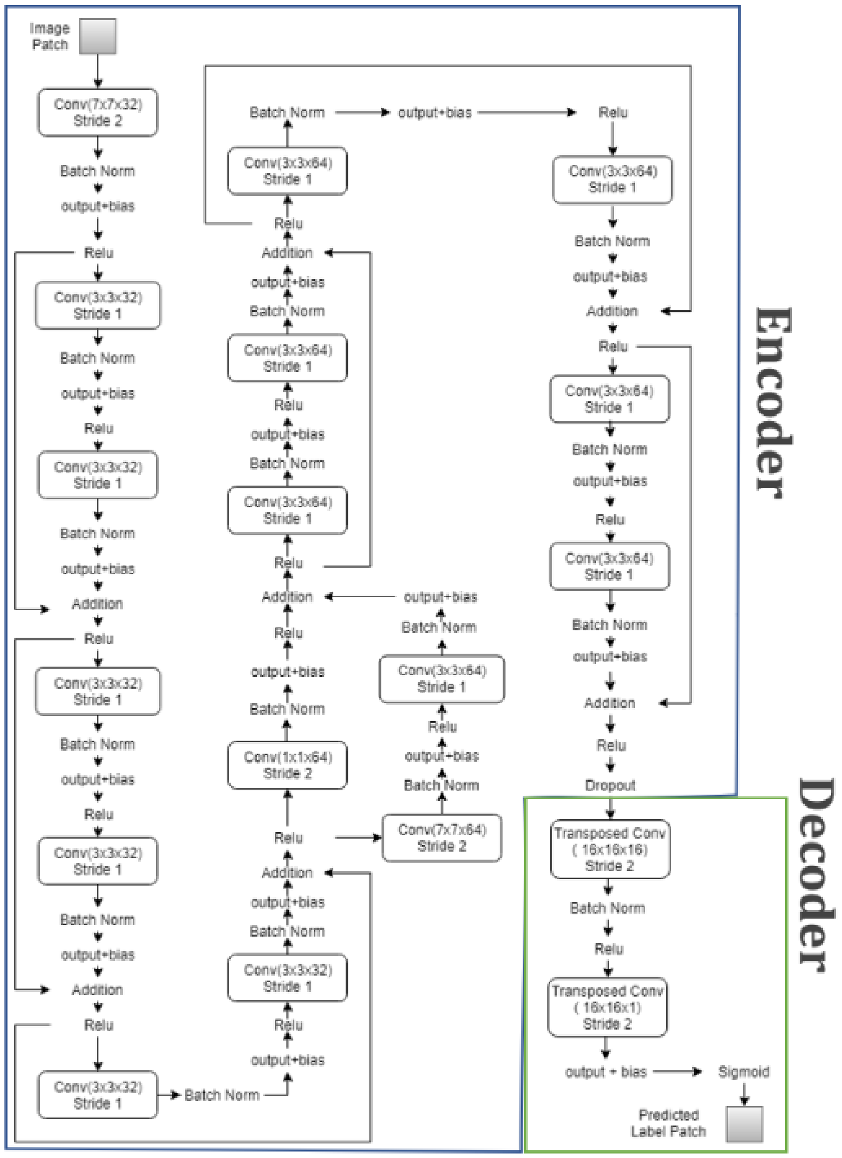

Figure 5.1: Structure of the SatNet, taken from **Satnetzitieren**

best way to describe the approach of the SatNet is that it takes few features (channels) and refines those in an optimal way to keep all important information, whereas the U-Net uses a lot of channels to keep as much information as possible.

Looking at the details, ReLU is applied to the output of every convolution layer to keep the inputs positive. The exception is the last layer where a sigmoid function is applied to give a result between 0 and 1, providing a probability of a pixel being forest. Obviously, the last layer only needs one channel for segmentation.

To improve training, a batch normalisation is performed after every convolution to keep the values centered and numerically stable. Furthermore, there is one dropout layer at the end of the encoder, which is enough to prevent overfitting, because this is where the information loss by dropping a neuron is the highest due to the information being most dense in the encoded data.

To summarise, we expect the SatNet to be faster in training, because it has significantly less channels and thus trainable parameters than the U-Net: The U-Net we trained has approximately 1.8 million parameters whereas the SatNet only has around 700k. Thus, the performance ceiling of the SatNet is expected to be lower than the U-Net's but maybe it can achieve better results in less time.

5.2 Training

The training algorith for the SatNet does not differ from the one we used for the U-Net; we had to different optimisers, namely the ADAM algorithm and the classical SGD and tried the BCE loss as well as the MSE loss. The same procedure was applied, going through multiple runs to avoid getting stuck in a local minimum. However, because of time troubles, we reduced the amount of runs. Also, the runs are expected to be faster than the U-Net runs, because we have less parameters und thus the gradients become easier to compute. As mentioned before, this of course means that the SatNet could perform worse than the U-Net, but a faster training is ideal when facing a lack of time.

One training parameter which we do not have in the U-Net training is the dropout probability of the dropout layer in the SatNet, however, we did not experiment with it and left it at the standard $p_{drop} = 0.5$.

5.3 Results

bad on img with little to no forest

5.4 Interesting Data

After training the SatNet, we let it run over the whole dataset and picked out some of the images where the model performed best, worst and most average on (w.r.t. the Jaccard index), to get a deeper understanding of the SatNet's strengths and weaknesses.

Strengths

Starting with the strengths, the SatNet performes best on images which are just plain forest, see **??**. This is to be expected, because the picture is basically only trees, which makes it easy for the model to assign all pixels to forest.

Figure 5.2: Image where SatNet performed best on; left is the image with the real mask and the calculated mask on the right. It fits perfectly

Weaknesses

For the SatNet performances, there are two examples which represent most of what went wrong when the SatNet misclassified almost all pixels in an image. In the first example, as mentioned in the Dataset chapter, in some of the images in the dataset it is not very clear if there is forest or just bushes and our model performed quite bad on those. In **??**, one could make an argument that the green parts are bushes and not really forest, but the segmentation of our model seems more appropriate than claiming that there is no

Figure 5.3: Image where SatNet performed worst on; left is the image with the real mask (no pixels are classified as forest) and the calculated mask on the right

Figure 5.4: Image where SatNet performed worst on; left is the image with the real mask (no pixels are classified as forest) and the calculated mask in the middle with its outline on the right

forest at all. In addition to that, the SatNet actually covered the forest really accurately, which makes it unfortunate that this is an image with one of the worst scores.

Then again, bad performances also occur on images where the given masks are completely reasonable. In these cases, it often is fields the SatNet struggles with, maybe because they form a coherent shape, which forests also do, and often have a brownish color, which is not too far from a forest either. In **??**, we can see that a lot of the fields, which the true mask correctly does not identify as forest, are claimed to be forest by the SatNet. Again, the argument could be made that our model did a better job than the true mask in the top right corner but that is besides the point.

Average Performance

We will just give three quick examples of an average performance to give an insight to how the SatNet performed. In ?? we can see three images, where the SatNet did not miss any forest parts, but seems to struggle with fields again. The forest was identified very accurately, sometimes better than the true mask, but the SatNet seems to identify every field quite confidently as forest.

Figure 5.5: Images where SatNet had a decent Jaccard index on. The left one has a field in the bottom left corner. The middle one has wine on the top side of the image. The right one has a field in the top left corner.

6 U-Net vs. SatNet

evtl in Conclusion wenn nicht so lang?

7 Conclusion and Outlook

7.1 Conclusion

In summary, all models we trained performed quite decently on our dataset and definetly succeeded in finding forest on satellite images. More often, the problem is overclassifying pixels as forest, which in reality are fields or something else.

Comparing the results of our different models, sadly there is no real contrast between the individual approaches, because they all had a similar score and more or less the same strengths and weaknesses. This could very well be a consequence of the U-Net and SatNet having the exact same training algorithm, so that their difference in structure mattered less than the fact that they followed the same learning procedure.

hier noch was zu svm dass die ca genauso gut wie nns funktionieren

7.2 Outlook

As always in machine learning projects, more time and resources could have improved results. Especially with such a deep network as the U-Net, experimenting with different structures, more training algorithms than just ADAM and SGD and perhaps a grid search for the optimal hyperparameters would have given us the optimal setup to train a model, which can identify forest in a satellite image. Naturally, the same goes for the SatNet, where on can always try out different filter sizes, channel sizes and block lengths, but in the end the structures we settled for gave good results.

In addition, another thing to experiment with is to workout a different training procedure for the SatNet. We just applied the same steps we used for the U-Net, but maybe different optimisers or algorithms can improve and accelerate the training process, especially given that the net structure of SatNet differs a lot from the U-Net.

Lastly, going through the dataset and looking for problematic masks could give more accurate results, because with refined masks the amount of brown ground classified as forest goes down a lot, which would probably immensely help with our models classifying fields as forest. It would also result in more precise loss and score functions.

Hier noch part ob man SVM etc. improven könnte