AML-Final Project

Image segmentation of aerial forest pictures

Department of Computer Science
Ruprecht-Karls-Universität Heidelberg

Laurin Ernst, Philipp Tepel, Nico Bruder

Heidelberg, August 31st, 2022

x

Contents

1 Introduction

Hier kommt die Einleitung rein.

2 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.

3 Unet training

3.1 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.

3.2 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.

3.3 Training hyperparameter's

3.3 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.

3.3 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.

3.3 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 Training evaluation

4.1 Mean squared error loss

4.1 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.

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

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.

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

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.

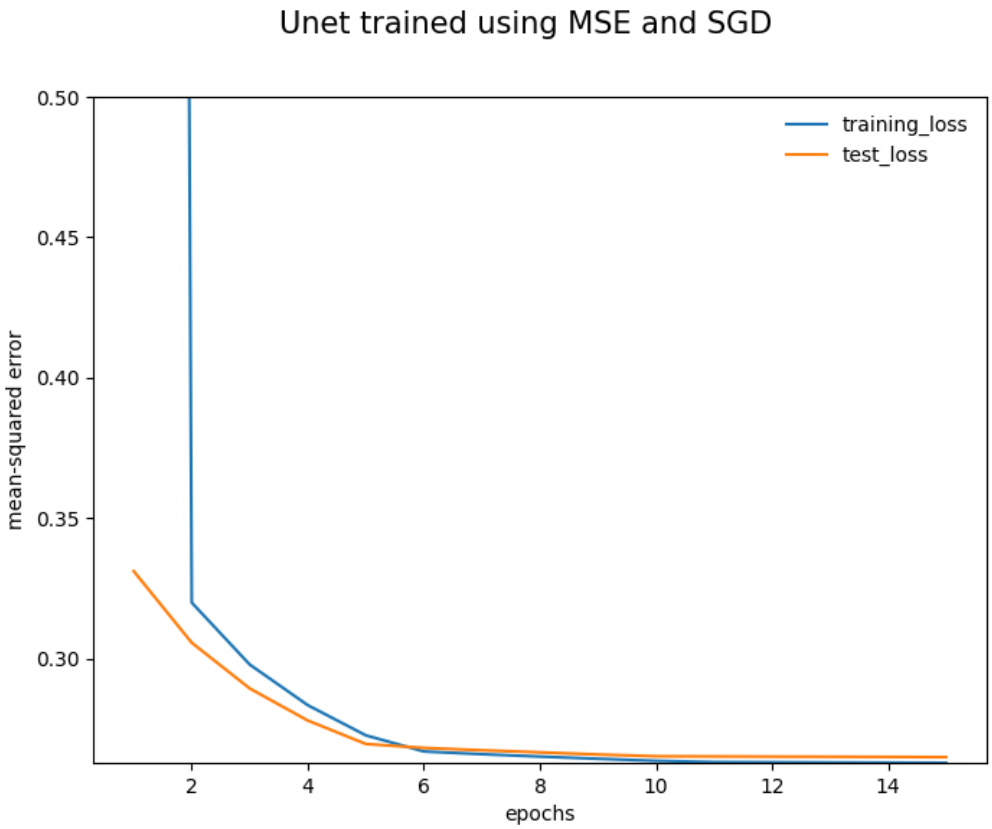

Figure 4.1: Loss over epochs for best fitted model

4.1 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 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

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

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.

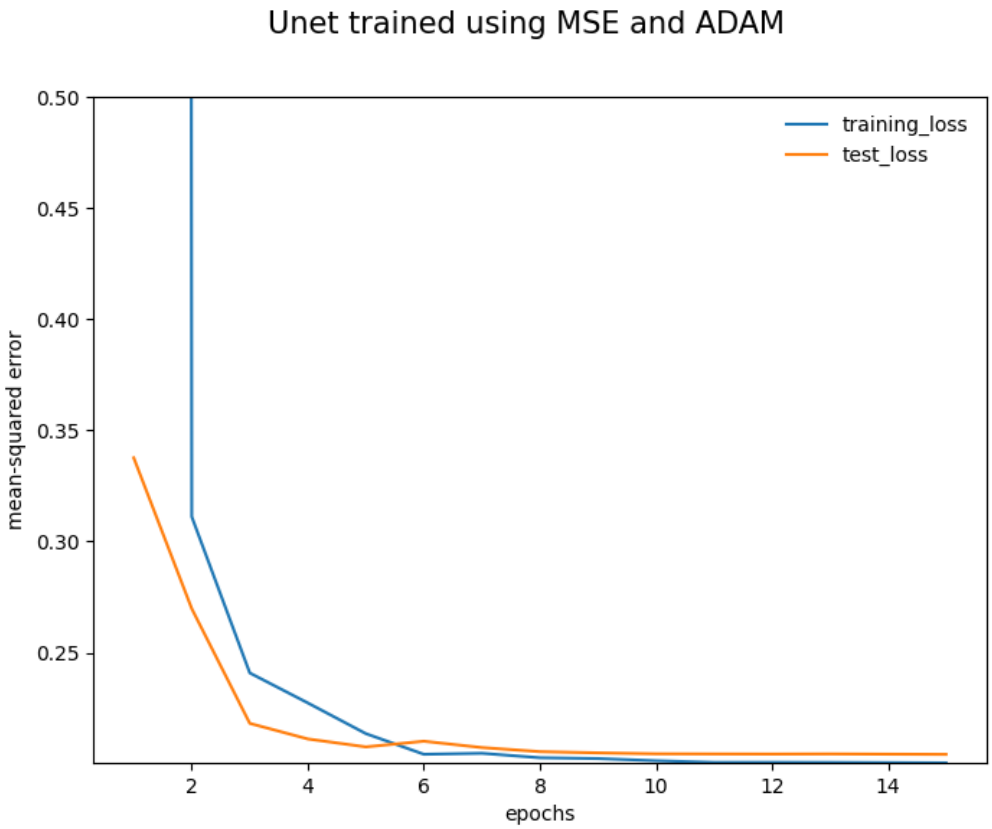

Figure 4.2: Loss over epochs for best fitted model

4.2 Cross entropy loss

4.2 Stochastic gradient descent

Bibliography