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. 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 customizabilty 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 $\beta_1$ and $\beta_2$ 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.