Martin Luther University
Halle-Wittenberg
Degree Course in Informatics

Bachelor Thesis:

Deep Learning for Locating Seed Placements in Images

Supervisors:
Prof. Dr.-Ing. Stefan Posch
Dr. rer. nat. Birgit Möller

Autor:
Danik Hollatz
matriculation number:
218102039
Submission date: 01.09.2021

Contents

1. Introduction **3**

2. Fundamentals of Neural Networks **4**
2.1. The need of neural networks 4
2.2. Perceptron - simplest neural network 4
2.3. Multilayer Perceptron 6
2.4. Activation Functions 7
2.4.1. Sigmoid 7
2.4.2. Softmax 8
2.4.3. Rectified Linear Unit 8
2.5. Loss functions 9
2.5.1. Cross Entropy 9
2.5.2. Binary Cross Entropy 10
2.5.3. Dice Loss 10
2.6. Optimization 11
2.6.1. Stochastic gradient descent 11
2.6.2. Extensions of SGD 12
2.7. Convolutional Neural Networks for Image Processing 13
2.7.1. Convolution 13
2.7.2. Pooling layers 16
2.7.3. Batch-Normalization 17
2.8. Networks for semantic segmentation 18
2.8.1. SegNet 18
2.8.2. UNet 19

3. Realisation **20**
3.1. Data Description 20
3.2. Data Prepossessing 21
3.3. Training process 22
3.4. Postprocessing 24

4. Evaluation **25**
4.1. Metrics 25
4.2. Results 26

5. Conclusion **35**

6. Erklärung **36**

Appendices **39**

A. SegNet Implementation **39**

B. UNet implementation **41**

C. Training loop and visualizing **42**

D. Heatmap produced by UNet **44**

E. ”Activated” Heatmap **45**

F. Image with predictions by UNet **46**

1. Introduction

The aim of this work is to investigate how well can popular neural networks perform for the task of semantic segmentation in order to locate objects. Objects locating is an important task in computer vision. Usually, for object locating we are interested in obtaining bounding boxes around the object, but it is not always the best solution. In this work we will be obtaining a single two-dimensional coordinate that will represent the location of each object. This workflow allows us to have a good approach in locating small objects. We also do not know in advance how many objects we have to locate in a certain image.

We are using a dataset consisted of 135 images with seed placements, that was provided by Til Feike and Robin Lieb from Julius Kühn-Institut. Images were taken in three different years: 2018, 2019 and 2020 in October. Images in the dataset vary in two types of sowing and in three types of crop that are used for sowing. Example of one image and corresponding annotation is to see in Figure 1

To locate object we are using Convolution Neural Networks, as they have shown on practice to be the best approach to work with image data.

(a) Sample of seed placements (b) Sample with annotated seed placements

Figure 1: (a) image of seed placement from 2018, (b) same image with annotated seed placements as crosses.

2. Fundamentals of Neural Networks

In this section we describe the basics of neural networks, how they are built, what they can be used for and provide description of the network architectures we are using within our task of object locating. Two main sources were used while writing this section: a book from Mohamed Elgendy called "Deep Learning for Vision Systems" [6] and another book written by François Chollet (creator of Keras [3], also a popular framework for deep learning) called "Deep Learning with Python" [2]. Also a variety another sources are used, that are referenced along the text.

2.1. The need of neural networks

Neural networks are one of the directions in the development of artificial intelligence (AI) systems. The idea is to simulate as closely as possible the functioning of the human nervous system - namely, its ability to learn and correct errors. This is the main feature of any neural network - it is able to independently learn and act on the basis of previous experience, each time making fewer and fewer mistakes. For example, while for classical way of programming it would be almost impossible to classify pictures of dogs and cats, neural network can achieve very high accuracy in this task.

The neural network imitates not only the activity, but also the structure of the human nervous system. Such a network consists of a large number of individual computational elements ("neurons"). In most cases, each "neuron" associated with a specific layer of the network. The input data is sequentially processed by all layers of the network. The network knows the parameters of each "neuron" and the order of operation of the entire system, which it can change depending on the results obtained on previous sets of input data.

2.2. Perceptron - simplest neural network

The simplest neural network is so called 'perceptron'. The most simple perceptron consists of a single neuron. Conceptually, the perceptron functions are similar to a biological neuron (Figure 1). A biological neuron receives electrical signals from its dendrites, modulates electrical signals in varying amounts, and then transmits the output signal through its synapses only if the total strength of the input signals exceeds a certain threshold. Afterwards, the output is fed to another neuron and so on. To simulate the phenomenon of a biological neuron, an artificial neuron performs two successive functions: it calculates the weighted sum of inputs to represent the total strength of the input signals, and applies a some kind of an

activation function to the signal. Not all input functions are equally useful or important. To represent this, each input node is assigned a weight value called its "connection weight" as shown in Figure 2 and in Figure 3.

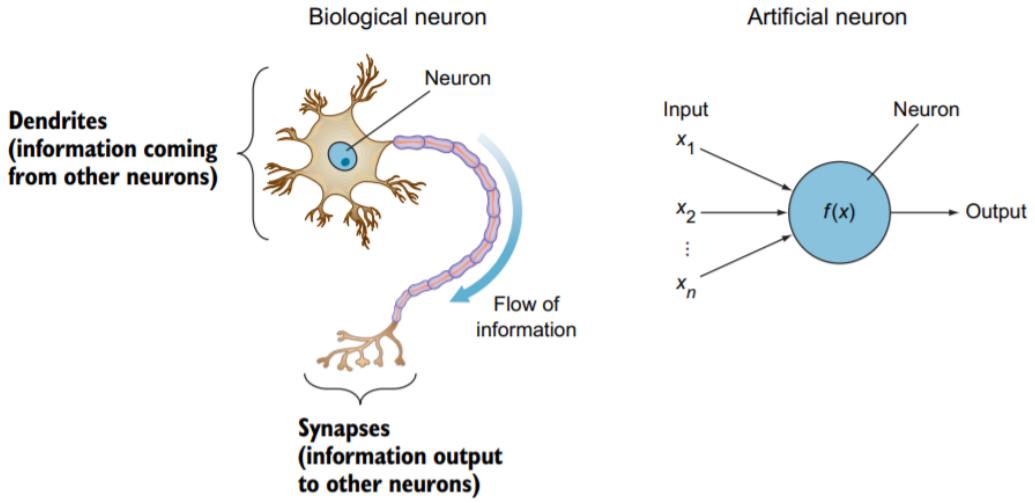


Figure 2: (Left) Representation of an biological neuron, consisted of dendrite, synapses and neuron. (Right) Representation of an artificial neuron, combined of an input vector, a neuron and his function, and an output (Fig 2.3., p. 39 [6]).

In the perceptron diagram in figure 2, we can see the following:

- Input vector — vector that consists of features that are fed to the neuron.
- Weights vector — vector that consists of weights for each feature in input vector. Each x_i is assigned with a weight value w_i that represents weight of a certain feature.
- Calculation of the neuron - calculations performed within a neuron to simulate input signals of a real biological neurons: weighted sum and activation function.
- Output — value that depends on the kind of activation function used in the neural network. Outputs of preceding neurons are considered as inputs for subsequent neurons.

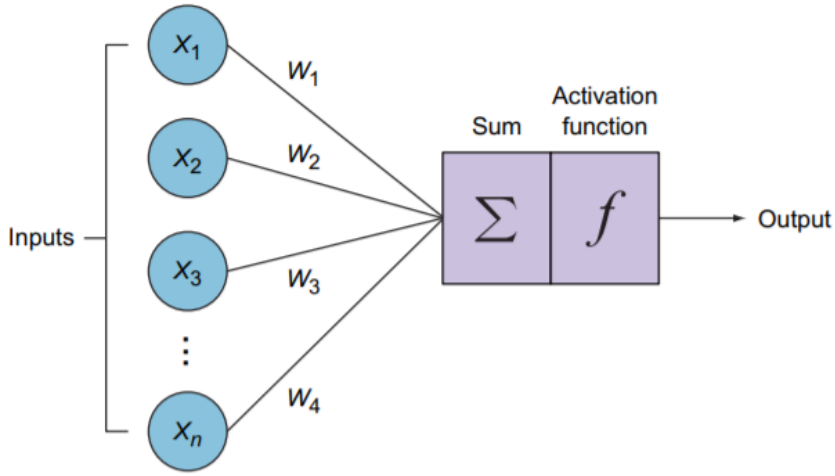


Figure 3: Input vector with assigned weights to each feature, which is fed to the neuron to perform calculations (fig 2.4., p. 40 [6]).

More detailed explanation is to find in chapter 2.1.1 [6].

2.3. Multilayer Perceptron

There is a variety of popular network architectures, one of some is called "Multilayer Perceptron" (MLP). MLP's consist of many neurons that are structured in layers to perform some kind of calculations and predict an output. It is a common neural network architecture to stack the neurons in layers on top of each other, called hidden layers (layers that are lying between first and last layer). Layers are connected to each other by weight connections. The multilayer perceptron architecture is shown in the Figure 4.

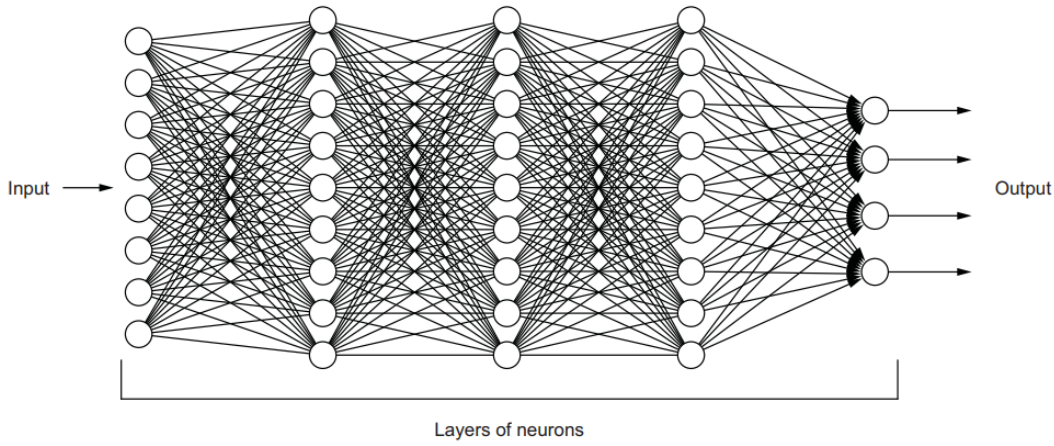


Figure 4: Example of MLP consisted of 4 layers. Input vector on the left and the output on the right (fig 2.4., p. 40 [6]).

In classical MLP network architectures are fully connected to the next hidden layer. That means each node in a layer is connected to all nodes in the previous layer. That kind of networks are called fully-connected networks (they consist of fully-connected or also called ”dense” layers).

More details in chapter 2.1 [6].

2.4. Activation Functions

While creating a neural network, one of important decisions to make is what activation function to use for neuron calculations. Activation function are also called ”nonlinearites” because of their transformation of linear combination of input to a nonlinear model, making it possible for the model to approximate more complex functions. Activation functions can be used for perceptrons in order to come to a decision whether to activate the corresponding neuron or to decide the output value of the neuron depending on the result of the weighted sum of the inputs and the threshold value, defined for the certain activation function. Without activation functions multilayer perceptron will most likely perform as a single perceptron.

2.4.1. Sigmoid

The sigmoid is one of the most common activation functions. It is mainly used for binary classification in order to predict the probability of a class in case of having only two classes. The output of sigmoid lies between 0 and 1. Sigmoid or logistic functions are able to reduce infinite continuous variable to a simplified

probabilities. Curve of sigmoid is also called S-shape curve because of how its look on a on a plot (Figure 5).

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

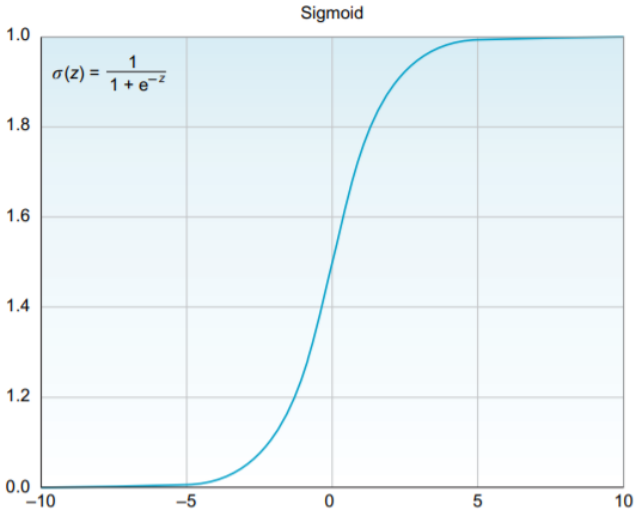


Figure 5: Sigmoid function (Fig 2.15., p. 55 [6]).

2.4.2. Softmax

The softmax function a generalization of the sigmoid functions. Softmax activation function is used for multiclass classification (the case when we have more two classes). Each class than obtains a certain probability of a sample being a given class, those probabilities sums to 1. One of the most common use case is to predict a single class when many options (more than two) are given. The softmax equation is as follows:

$$\sigma\left(x_j\right) = \frac{\mathrm{e}^{x_j}}{\sum_i \mathrm{e}^{x_i}}$$

2.4.3. Rectified Linear Unit

The rectified linear unit (ReLU) activation function activates a neuron only in case if the input is above zero. For outputs, that are below zero, the output is equal to zero. In case of input which value is higher than zero, the output is equal to the

input. ReLU function equation is as follows:

$$f(x) = \max(0, x)$$

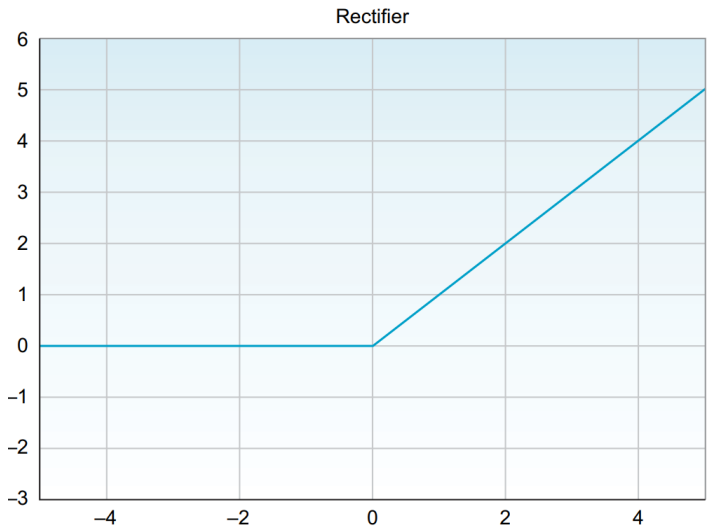


Figure 6: ReLU (Fig 2.18., p. 59 [6]).

More about activation functions can be found in chapter 2.3 [6] and chapter 3.4.3 [2]

2.5. Loss functions

In order to evaluate the prediction of the network a measurement that can calculate the loss (or also called "error") is needed. Loss functions are a measure of how "wrong" the prediction is. It also quantifies how far the prediction is from the correct solution. The smaller the loss is, the better is the performance of the model. The loss function is a very important part of training of a neural network, because the output of the loss function will directly influence how the weights of the model should be updated. Loss function minimization is an optimization problem and by running optimization algorithms we can efficiently update the weights of the neural network in order to minimize the loss functions and, in its turn, get better predictions. More to loss functions in chapter 2.5 [6].

2.5.1. Cross Entropy

One of the most popular loss functions used in neural networks is cross-entropy.

To calculate the cross-entropy loss for all the training samples of size N following formula is used:

$$E(y,p) = -\sum_{i=1}^{N} y_i \log(p_i)$$

with $E(y,p)$ denoted as the loss function, y is the vector consisted of correct outputs, p is the vector of predicted probabilities, y_i is the correct output and p_i is the predicted probability.

2.5.2. Binary Cross Entropy

Binary cross entropy compares every predicted probability of being a class with the actual probability of being class.

$$E(y,p) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p_i)) + (1-y_i) \cdot \log(1-p_i))$$

2.5.3. Dice Loss

Dice Loss achieved its popularity with the paper by Milletari et al. [7] for 3D medical image segmentation but was first introduced in 1945 as Sørensen–Dice coefficient [20] to measure the similarity between two sets.

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

Values of p_i and g_i are representing pairs of corresponding pixel values of prediction and ground truth. The sum $\sum_i^N p_i g_i$ increases only when both p_i and g_i are having high values, otherwise if one of them is equal to 0 or low number, also the multiplication of this two numbers will be zero or just low. The denominator, in opposite, does not care if p_i and g_i are matching and can be seen just as sum of total predicted pixels of both prediction and ground truth.

We can use the Dice coefficient (D) to measure an overlap between two sets. In case when they are same, they will achieve the maximum value of numerator, which is equal to 1. D is used to calculate the loss as simple as:

$$DiceLoss = 1 - D$$

More detailed explanation of Dice loss can be found in [14].

2.6. Optimization

Training a neutral networks is done by showing a lot of examples to the network. The network than makes predictions. When predictions are computed, the loss function will be calculated to update the weights. By repeating this steps the neural network will minimize the loss function and finding the optimum values for the weights. Different optimization algorithms are created to solve this problem. Detailed explanations can be found in chapters 2.4.3 and 2.4.4 [2].

2.6.1. Stochastic gradient descent

We can use the five-step algorithm (see below) called "Stochastic gradient descent" in order to find locally best weights for the neural network: change the weights little by little based on the current value of the losses in the random batch (group of samples) of data. Neural networks can be seen as differentiable functions and in order to get the minimum loss we have to find a combination of weight values, so that gradient from such combination will be equal to zero. Since we are dealing with a differentiable function, we can calculate its gradient, which gives us an effective way to implement step 4. For function φ of n variables $x_1, \ldots, x_n$ gradient is defined as n-dimensional vector equal to $\left(\frac{\partial \varphi}{\partial x_1}, \ldots, \frac{\partial \varphi}{\partial x_n}\right)$. If we update the weights in the opposite direction to the gradient, the losses will be slightly smaller each time:

1. Draw a batch of training samples x and corresponding targets y.
2. Forward pass - run the network on x to obtain predictions $\hat{y}$.
3. Compute the loss of the network on the batch.
4. Backward pass - compute the gradient of the loss with regard to the network's parameters.
5. Move the parameters a little in the opposite direction from the gradient.

Figure 7 illustrates what happens in 1D, when the network has only one parameter and only one training sample.

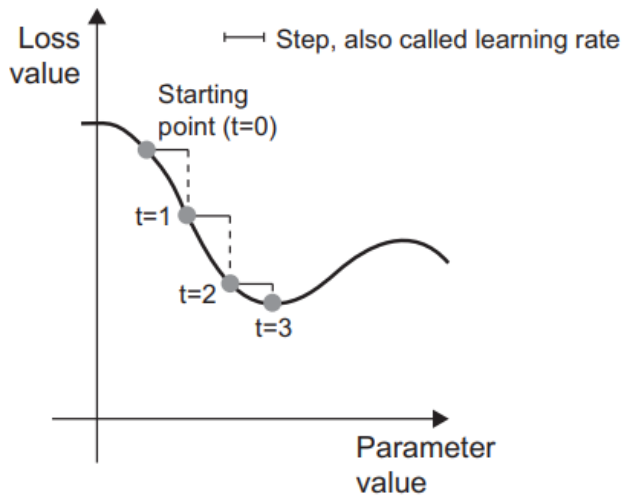


Figure 7: Example of SGD down a loss curve (fig 2.11, p. 49 [2]).

As shown on Figure 7 it is important to pick a plausible value for the step factor (also called "learning rate"). For too small value it will take unnecessary many steps to find the minimum and is likely to stuck in local minimum.

Also, in step 4 we assumed, that we can compute the gradient. Neural network are consisted of a lot of matrix operations and for each such operation a derivative is known. For a given set of operations we can also compute the derivative via "chain rule".

"In calculus, the chain rule is a formula that expresses the derivative of the composition of two differentiable functions f and g in terms of the derivatives f and g. More precisely, if $h = f \circ g$ is the function such that $h(x) = f(g(x))$ for every x, then the chain rule is, in Lagrange's notation, $h'(x) = f'(g(x))g'(x)$." [19].

Running the chain rule for the given set of operations in a neural network and calculating a contribution of weights they had in loss value in a neural network is called "Backpropagation".

2.6.2. Extensions of SGD

Following explanation was inspired by this three sources: [10], [18], [16].

There is a variety of different extensions of SGD, but one the most popular and probably most used one is the Adam optimizer. While SGD uses the same learning rate for all weights, Adam computes separate learning rate for each weight. As many experiments show, due to this extension, Adam usually performs better than SGD.

Also, an extension of Adam is created, called: AdamW. In comparison to Adam, AdamW decouples weight decay from the learning rate, so both of this parameters can be updated separately. Weight decay is technique that adds penalties to the loss function, in such a way, that with this techniques weights will tend to remain small and restricted of getting enormous values, which is bad while training the model.

2.7. Convolutional Neural Networks for Image Processing

Humans can perceive the world though different perceptions: sight, sound smell and many other senses. The idea behind artificial intelligence (AI) systems is to be able such kind of perceptions, so that AI would be able to take actions based in the developed perception. Humans sight can be seen as computer vision as a part of AI system. Computer vision allows AI systems to take actions based on their perception of images or videos. It is also possible to choose another sensitive devices for AI system to operate. Convolutions neural networks is a good mechanism to develop a perception for AI systems.

2.7.1. Convolution

The operation of convolution is based on its ”kernel”. Kernel is a matrix of weights. Convolution is based on ”sliding” this kernel over the two-dimensional inputs (or divide the input to the two-dimensional inputs, in case if the input has 3 or more dimensionalities) and performs an elementwise multiplication with certain areas of the input. Afterwards the multiplication, the output matrices are summed up to a single output. After convolution we get a new matrix, where each pixel is a weighted sum of the input area and weights of the kernel as shown on the Figure 8.

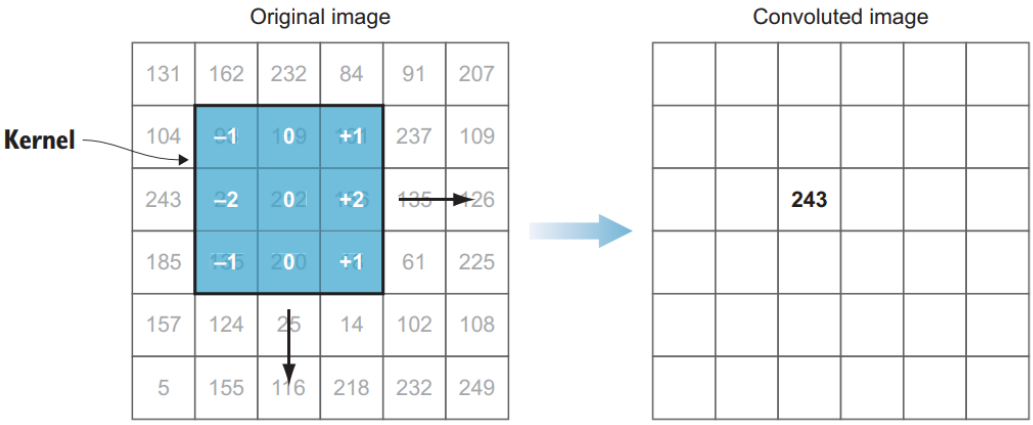


Figure 8: Example of convolution opertaion: highlighted pixels from the original images are multiplied with corresponding values in the convolution filter. Sum is equal to 243 (Fig 3.15., p. 109 [6]).

While dense layers can only learn global patterns, convolutions are able to learn local patterns. That means, once convolution has learned, for example, to find a certain edge or any other pattern on the image, it can be found at every place in that image.

As shown on the example 9 of MNIST [5] digit, image can be divided into different local patterns and once convolution learns the pattern, it can be found everywhere.

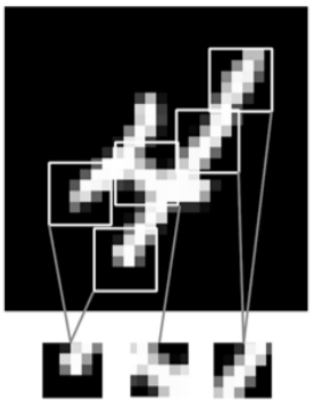


Figure 9: Example of image broken into local patterns (Fig 5.1., p. 122 [2]).

Interesting property about convolutions applied to the neural networks is that the first used convolutions in the network are learning more local pattern such as different lines or edges and the deeper the convolution goes, the more complex pattern it shows to have learned. Example is shown in Figure 10

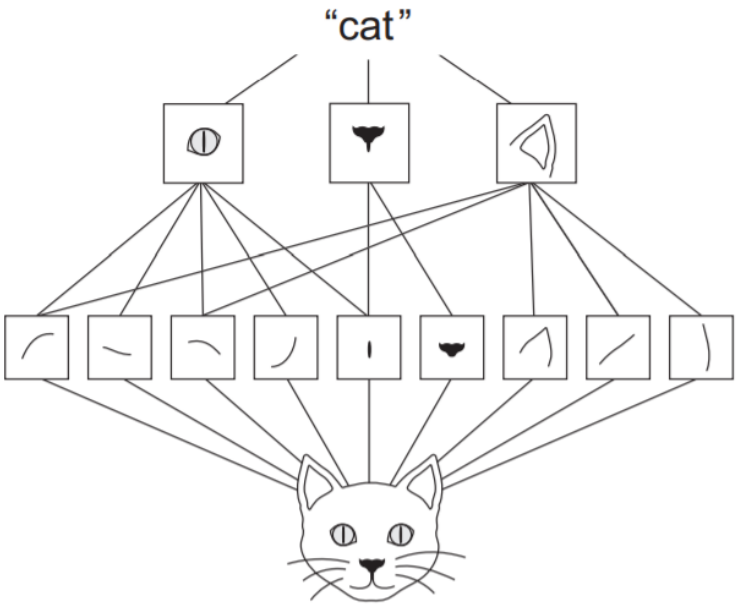


Figure 10: Example of how an image of a cat can be broken first in the local pattern and then in more complex. Based on patterns a decision that there is a cat on the image is taken (Fig 5.2., p. 123 [2]).

Two different parameters that also define how convolution works are stride and padding.

- Strides - the amount by which the filter slides on the image.
- Padding — also called zero-padding because because of adding zeros around the border of an input (figure 11). Padding is used to make the output after convolution operation equal to size of the input. Usually without padding the output shape is smaller than the input shape. Different types of padding exist.

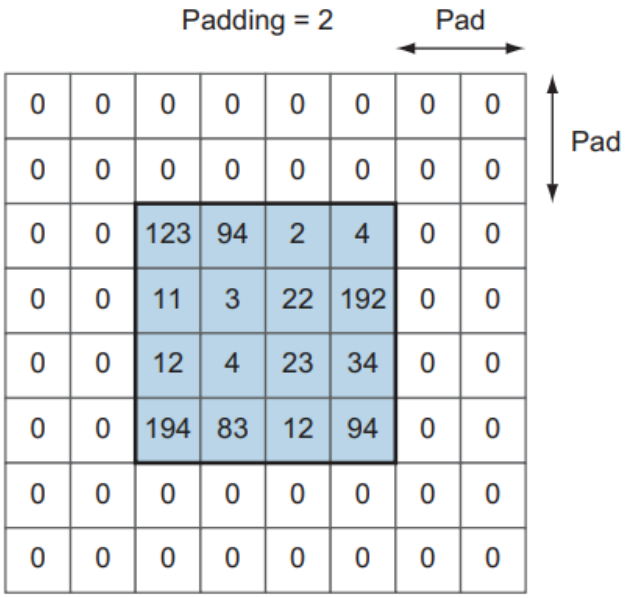


Figure 11: Example of zero-adding with padding equal to 2. (Fig 5.2., p. 123 [2]).

More detailed explanations can be found in chapter 3.3 [6], chapter 5.1.1 [6] and in [9].

2.7.2. Pooling layers

Big number of convolutions operations used in neural networks gives a big number of weights that need to be optimized while training the neural network. So in order to reduce the complexity of the neural network and make the learning faster "pooling" operations are used.

Pooling operation iterates over the input with a certain kernel and saves only one number from the input that is passed to the kernel. As figure 12 shows, pooling operation iterates through the input matrix of size 4×4 with stride equal to 2 and saves only the largest number. Such operation is called "maximum pooling". Another types of pooling, for example "average pooling" exist. With pooling it is possible to reduce the size of inputs passed to the next layers in the neural networks and reduces the number of weights that have to be optimized.

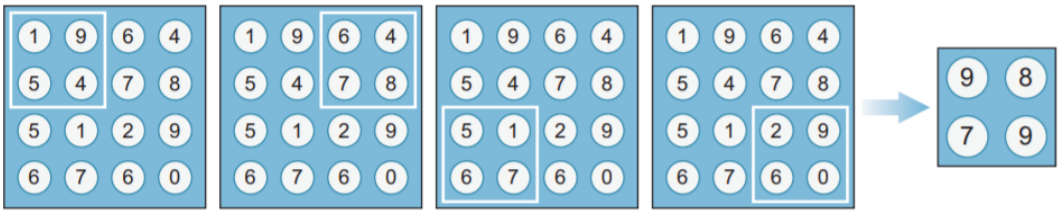


Figure 12: A 2 × 2 pooling filter and strides of 2, reducing the feature map from 4 × 4 to 2 × 2 (Fig 3.21., p. 115 [6]).

In comparison to the convolution operation, pooling operation (as part of the neural network) do not posses any parameters that to be learned, since its only purpose is to reduce the shape of the input.

For more details see chapter 3.3.2 [6] and [9].

2.7.3. Batch-Normalization

Batch-Normalization is frequently used in CNN architectures. The idea behind batch normalization is to normalize an input by subtracting the mean of mini-batch and divide it by its standard deviation.

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

where $\hat{x}$ is denoted as the resulting normalized activation and x is the input vector.

After normalization all inputs will have a mean of 0 and a standard deviation equal to 1. While training the model, different batches are being fed to the network and different batches have different distributions. After applying batch normalization the distribution of the input will be same, so that model does not have to adapt for the changing distribution and thus be able to learn faster.

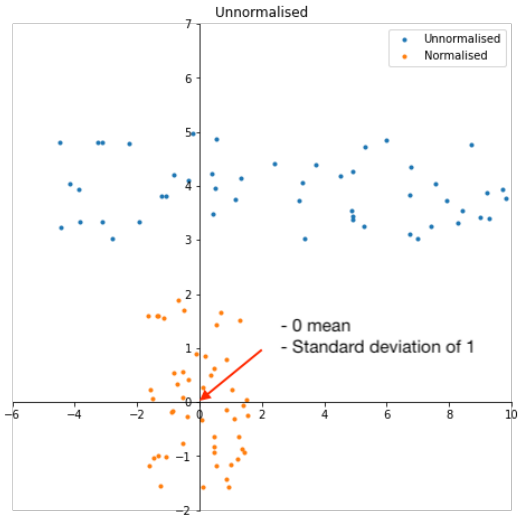


Figure 13: Example of batch normalization [13].

More detailed explanation of batch normalization can be found in [13].

2.8. Networks for semantic segmentation

Goal of semantic image segmentation is to label each pixel of an image with a corresponding class of what is being represented. More about semantic segmentation and used architectures can be found in [8] and [15].

2.8.1. SegNet

SegNet is an architecture that was designed for semantic segmentation. The task it was primarily created for is to segment road scenes into different object, such as pedestrians, cars, road, buildings and road signs.

SegNet is divided to an "encoder" and "decoder" architectures, as shown in Figure 14. Both of them are consisted of convolutions, poolings and batch-normalizations. Encoder creates a set of feature maps and uses max. pooling in order to reduce the shape of the input. Decoder uses created feature maps and recreates the original shape of the input to the network, so that the input and output both will have the same shape, where for each pixel his class (road, car, etc...) is assigned as a number. Both encoder and decoder are having 13 convolutions. To recreate the original shape, while performing max. pooling the chosen indices are memorized and than used while recreating the original shape.

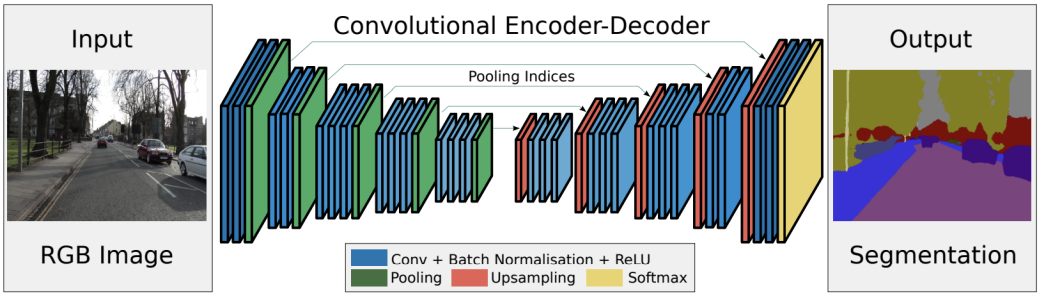


Figure 14: Architecture of SegNet (Fig. 2, p. 4 [17]).

2.8.2. UNet

Compared to SegNet, UNet was developed in order to segmentate medical images. Just like SegNet, UNet is also consisted of encoder and decoder parts. The difference between UNet and SegNet is that UNet does not use batch normalization (batch normalization was introduced in the same year as UNet: in 2015, many of current adaptations and implementations of UNet are also adding batch-normalization to the network) and uses does not use memorized indices like SegNet, but concatenates the corresponding feature maps created by encoder.

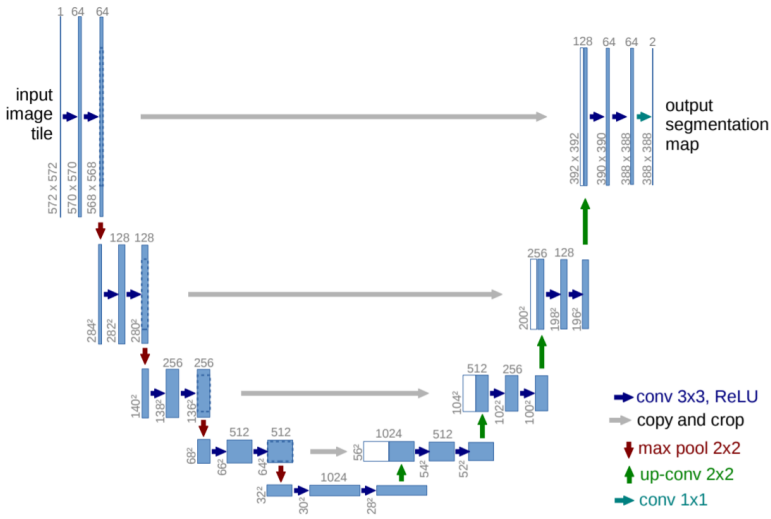


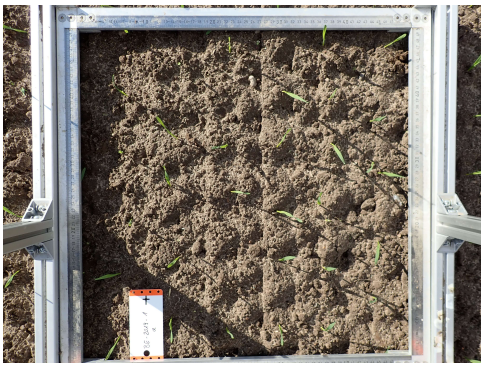
Figure 15: UNet architecture (Fig 1., p.2 [11]).

3. Realisation

3.1. Data Description

Dataset consisted of 135 seedbed images was provided by Til Feike and Robin Lieb from Julius Kühn-Institut. Dataset consists samples from three years: 2018, 2019, 2020, each having 24, 47 and 64 images correspondingly. Every image is of 3000×4000 pixel resolution.

(a) Sample from 2018



(b) Sample from 2019

(c) Sample from 2020

Figure 16: Overview of samples from different years.

Alongside with images, two excel files were provided. One of them had the coordinates of valid area of the image, and the other one had the coordinates of each seed placement and also additional information about characteristics of image, such as year when the image was taken, type of sowing, type of crop used for sowing and identifier of frame, which is crucial in order to extract the coordinates of valid area from excel files. After extracting the needed information from excel

files, images are cut out to their valid areas. Cut out areas will be used for training process.

3.2. Data Prepossessing

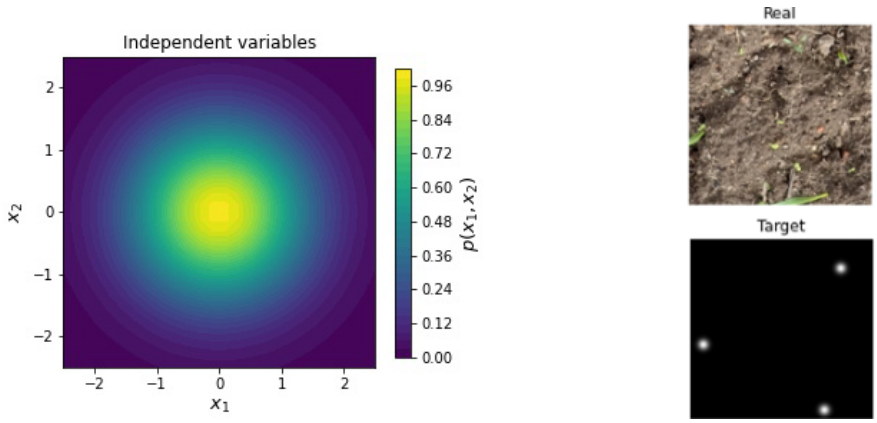
Since the data is annotated in such a way, that every seed placement is only one point on the whole image, that makes very unbalanced dataset. Unbalanced dataset is such a dataset, when the number of one class (in our case: coordinates seed placements) is much larger than another (any other coordinate). So in order to make it easier for neural network to learn, annotated point will have a value of 1 and those around him will be getting lower values. For to create this kind of distribution we can use multivariate normal distribution.

"The multivariate normal distribution is defined by sets of parameters: the mean vector μ, which is the expected value of the distribution; and the covariance matrix Σ, which measures how dependent two random variables are and how they change together. The multivariate normal with dimensionality d has a joint probability density given by:

$$p(\mathbf{x} \mid \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu)\right)$$

Where x is a random vector of size d, μ is the mean vector, Σ is the symmetric, positive definite covariance matrix (of size $d \times d$), and its determinant $|\Sigma|$."[12]

Subfigure (a) in Figure 17 represents the bivariate distribution (distribution of two variables) that are independent from each other. The distribution is normalized and lies between values 0 and 1, as shown on the subfigure and is applied for the grid of size 100×100 pixels. So, the annotated point has value of 1 and those points that lying around him are smoothly getting lower values. This makes possible not to penalize the model that hard, if in the case when model gives a high probability for the point which lies near to the annotated point.



(a) Multivariate normal distribution (b) Training sample and corresponding mask

Figure 17: Example of Multivariate normal distribution and its applying in training samples.

An implementation of multivariate normal distribution was adapted from [12]. A more detailed explanation is also to find at [12].

3.3. Training process

For this work the open deep learning framework PyTorch was used in order to develop SegNet (see appendix A), UNet (see appendix B) and create code for training and visualizing (see appendix C) and validating. Implementation of this networks was a task in online course [4], architectures were implemented and adapted for this work. Visualizing the training process, as shown in Figure 18, gives some understating how well model performs while training. Also Google Colab was used in order to execute Python code. Using a free version of Colab, the network was trained with Nvidia Tesla K80 and 12 GB of GPU (graphics processing unit) memory available.

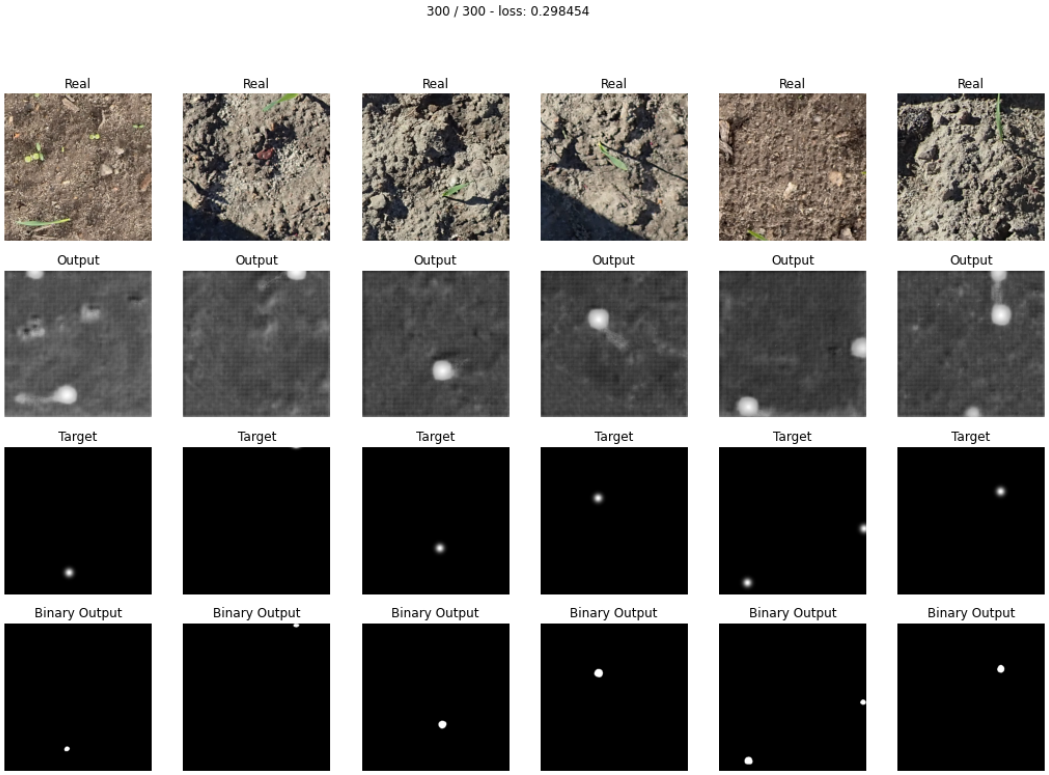
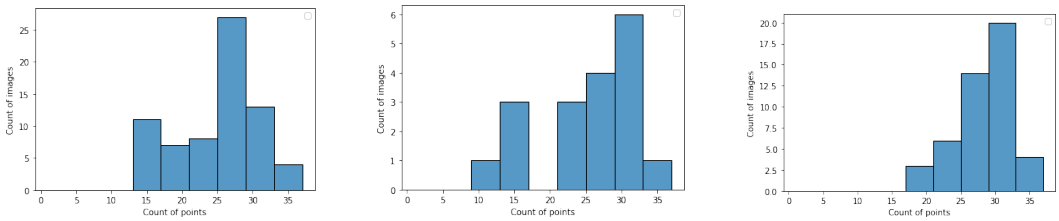


Figure 18: Visualized training process of SegNet at 300 epoch with threshold after sigmoid activation equal to 0.5.

To train the network three sets were created: train, validation and test. For train we are using 80% of images from year 2018 and 80% of images from year 2020, which means there are 19 images from 2018 and 51 images from 2020. Rest of 20% from years 2018 and 2020 are used for validation, meaning there are five images from year 2018 and 13 images from year 2020. Images are randomly divided. For test the whole 2019 year of images was taken.



(a) Distribution of points per image in training set

(b) Distribution of points per image in validating set

(c) Distribution of points per image in test set

Figure 19: Distribution of point per image in training, test and validating sets.

To utilize the maximum of GPU Memory each batch contains six random crops of images with resolution of 512×512 pixels. For every batch the loss is calculated and weights are then updated. Epoch is seen as an iteration over training images, where for each image one random crop is in a certain batch. Models are trained for 300 epochs, taking about 2.5 hours to train, using 11.37 Gb of GPU Memory. Implementation of SegNet uses Dice loss and AdamW as optimizer, UNet uses Binary-cross-entropy (BCE) as loss and also AdamW as optimizer.

3.4. Postprocessing

The output of the neural network is a two-dimensional matrix with the same resolution as an input image, where for each pixel its probability of being an seed placement is saved as its value. After applying the sigmoid activation function to this output and using a threshold of 0.35 (which was found out optimal by trying a lot of different thresholds manually) we are getting an output (Figures 22 and 23, subfigure (c)), where white areas can be interpreted as prediction of the model, so that at that place is probably a seed placement. Since we are only interested in one point as a seed placement, this areas are seen as connected components (group of pixels that are connected to each other). By using an implemented function from OpenCV [1] we are able to find connected components and get the center (averaged sum of coordinates) of each. Centres of each connected component is seen as a seed placement. To reduce the amount of false positive predictions, we are looking for each connected component if there is another one in a certain radius (30, 60 and 100 pixels). If this is the case, the predicted seed placement is a position, calculated as average position of centres of connected components.

Also, the shape of valid areas of images is too big for available GPU memory to predict at once. Because of that, each image is divided into crops and then concatenated back. Since models do not have enough information about sides of the crops, crops are divided in such a way, that every next crop has an overlap with

the last one created. The overlapped area is divided in two equal parts, first part is taken from the first created crop and the second part - from the second crop and so on until the full image is processed.

4. Evaluation

In this section we present performance analysis of the models. We also evaluate out network via 3 common metrics: precision, recall and F1-Score. As best we define a model that achieves highest F1-Score.

4.1. Metrics

To define precision, recall and F1-Score we first need the following definitions:

- True positive - an outcome where the model correctly predicts the positive class.
- True negative - an outcome where the model correctly predicts the negative class.
- False positive - an outcome where the model incorrectly predicts the positive class.
- False negative - an outcome where the model incorrectly predicts the negative class.

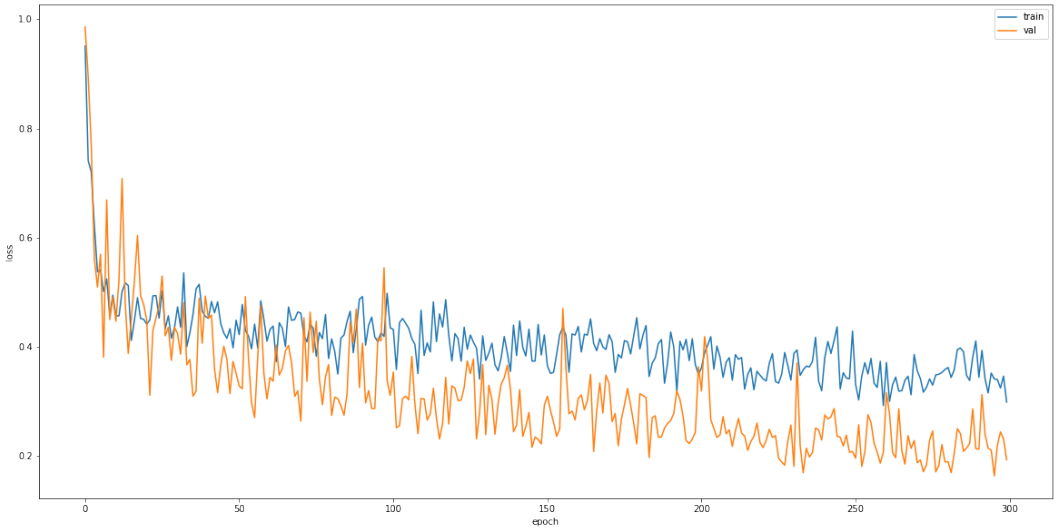
We define our prediction (a coordinate of seed placement) as true positive, if in a given radius of the prediction there is an annotated class (a seed placement). Otherwise, such prediction is seen as false positive. False negative is the case when the seed placement is not found (there is no prediction in a given radius found).
Precision, recall and F1-Score are defined as follows:

- Precision = $\frac{\#\text{True Positive}}{\#\text{True Positive} + \#\text{False Positive}}$
- Recall = $\frac{\#\text{True Positive}}{\#\text{True Positive} + \#\text{False Negative}}$
- F1-Score = $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

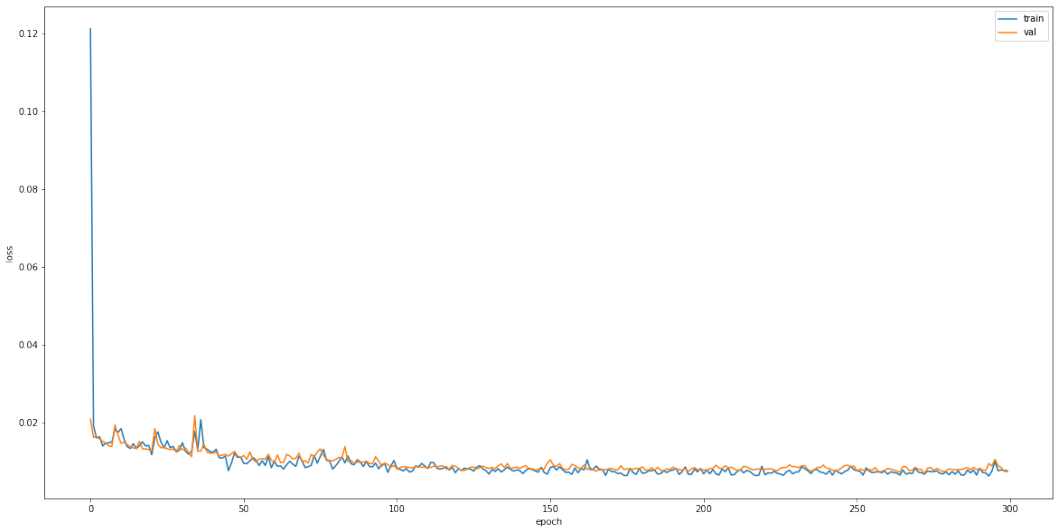
with # denoted as number of specific outcomes.

4.2. Results

After the training is completed, we obtain the curve of the loss function (Figure 20). Since our primary task is not the get the best possible segmentation but to achive the highest F1-Score, the loss curve can provide us an understanding that our network is learning and is not overfitting but does not provide information how well it performs on predicting seed placements. Overfitting is a situation when model performs good on the training data but is unable to perform nearly as well on the unseen data.



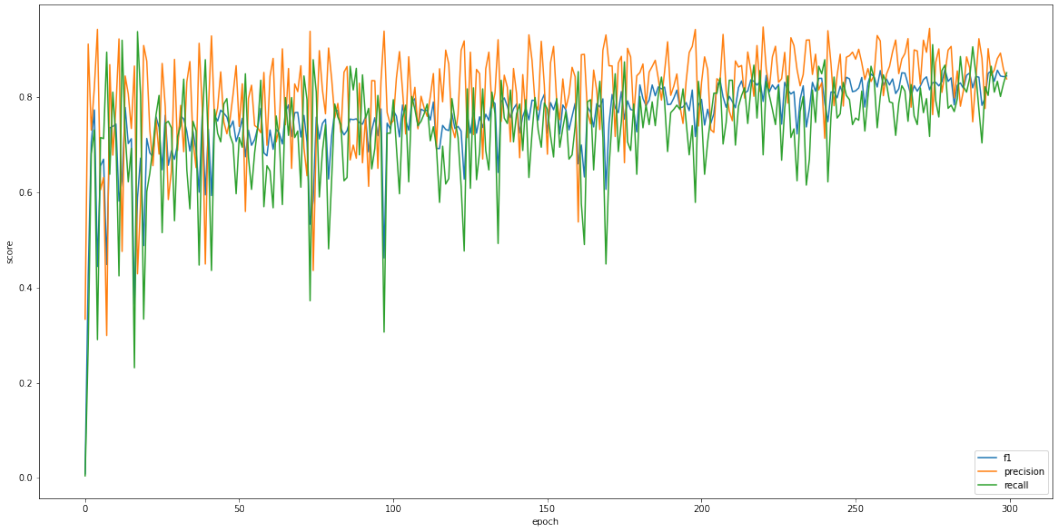
(a) Curve of Dice loss over epoches while training for SegNet



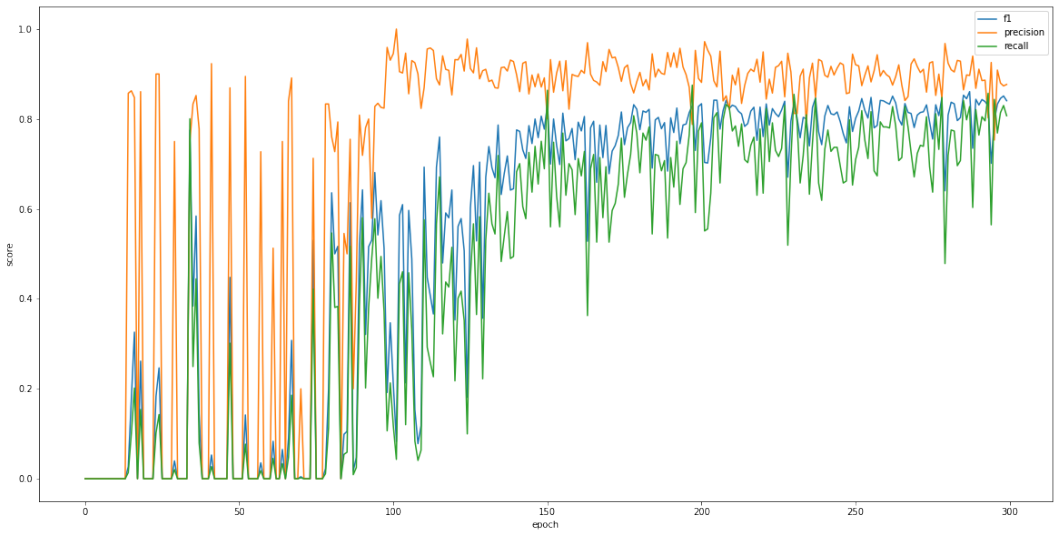
(b) Curve of BCE loss over epoches while training for UNet

Figure 20: Curves of losses for SegNet and UNet.

To measure how well out network performs on prediction task we are calculating precision, recall and F1-score for every epoch (Figure 21)



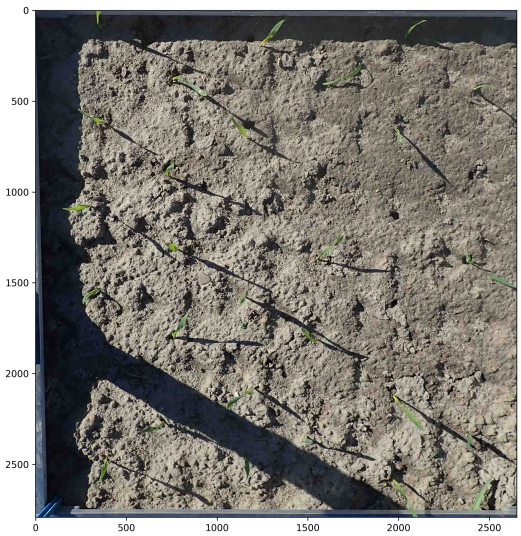
(a) Curve of F1-score, precision and recall over epoches for SegNet



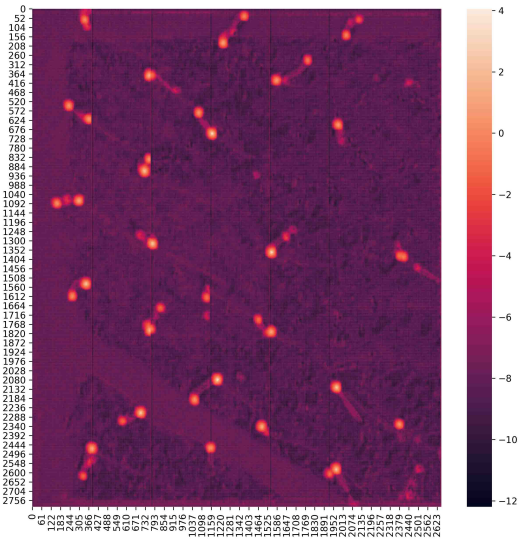
(b) Curve of F1-score, precision and recall over epoches for UNet

Figure 21: Curves of precision, recall and F1-Score for SegNet and UNet.

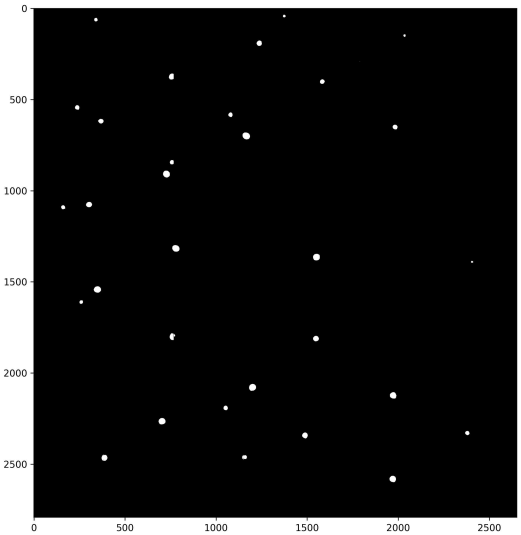
Using a threshold of 0.35 and radius of finding equal to 100 pixels, we are obtaining the best epoch which numbered as 280th for SegNet architecture and 288 for UNet architecture. Example of how this epochs perform are in Figures 22 and 23 to see.



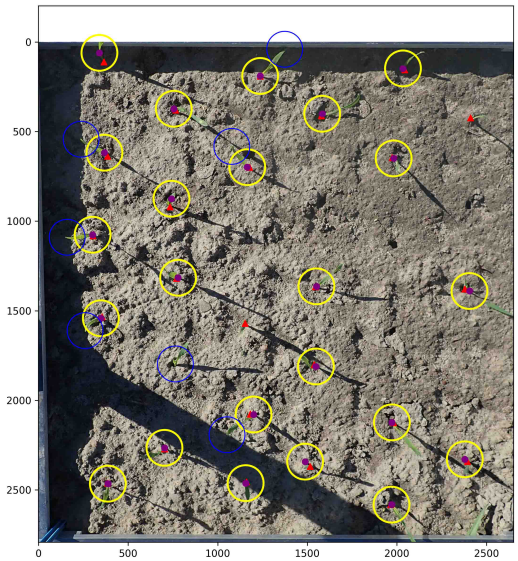
(a) Original image



(b) Heatmap

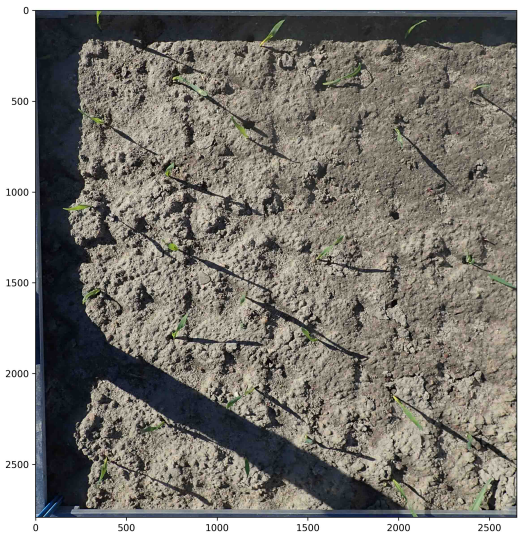


(c) Heatmap after applying sigmoid activation function and threshold of 0.35

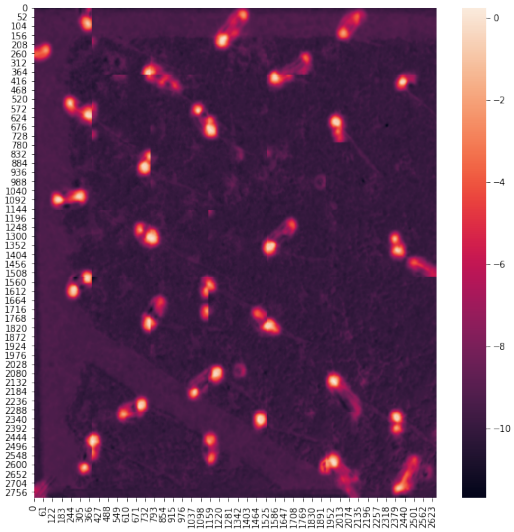


(d) Image with predictions

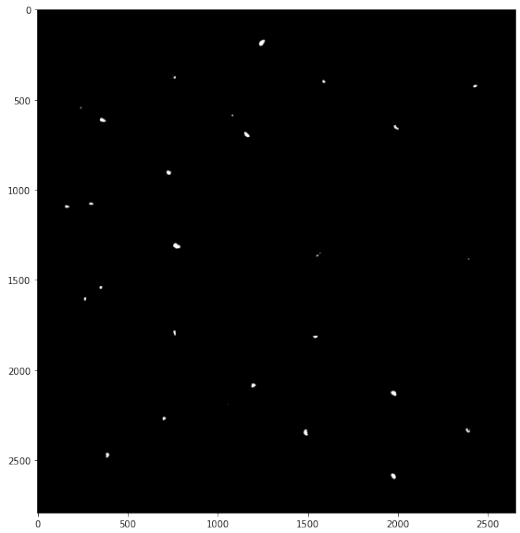
Figure 22: Overview of steps to obtain image with predictions for SegNet.



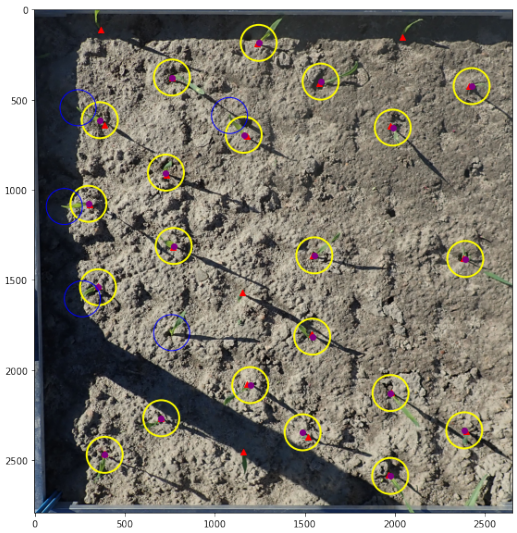
(a) Original image



(b) Heatmap



(c) Heatmap after applying sigmoid activation function and threshold of 0.35



(d) Image with predictions

Figure 23: Overview of steps to obtain image with predictions for UNet.

Subfigures (d) have following annotation:

- Red triangle - annotated seed placement
- Purple circle - true positive prediction

- Yellow circle - zone of 100 pixels around true positive prediction
- Blue circle - zone of 100 pixels around false positive prediction

Images of higher resolution for subfigures (b), (c) and (d) for figure 23 are to find in appendices D, E, F.

With this epochs we have also calculated same metrics for different radius of finding seed placements, obtaining following results for test set (Table 1) and validation set (Table 2).

Table 1: Evaluation of networks on test set

Network	radius = 30			radius = 60			radius = 100		
	F1	precision	recall	F1	precision	recall	F1	precision	recall
SegNet+Dice	0.74	0.65	0.84	0.79	0.72	0.86	0.83	0.80	**0.87**
UNet+BCE	0.78	0.76	0.81	0.82	0.80	0.83	**0.84**	**0.85**	0.84

Table 2: Evaluation of networks on validation set

Network	radius = 30			radius = 60			radius = 100		
	F1	precision	recall	F1	precision	recall	F1	precision	recall
SegNet+Dice	0.77	0.73	0.82	0.82	0.79	0.85	0.85	0.83	**0.86**
UNet+BCE	0.79	0.81	0.77	0.84	0.87	0.81	**0.85**	**0.89**	0.82

As we see from the Table 1 and Table 2, our networks performs with an insignificant decrease in metrics for the test data, which means the trained network can also perform quite well on the unseen data. As of interesting behaviour, it is to see that for different radiuses, recall stays about the same and the precision is drastically changing. The intuition behind that is that model can easy define a seed placement but has a hard time to separate it from the leaf of the seed as shown in the figure 24. That is the main source of false positive cases.

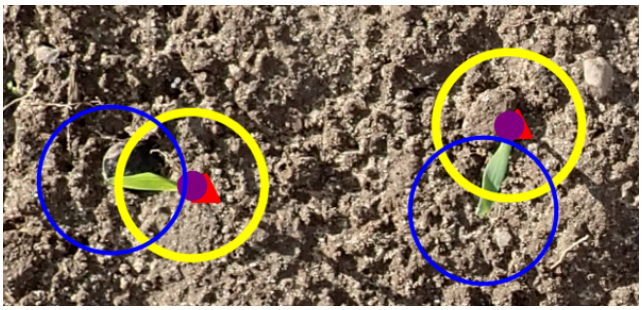


Figure 24: Example of false positive prediction by SegNet on the leaf.

Another interesting source of false positive predictions, that is not that significant, is that model sometimes triggers on the white description panel (figure 25), which is not a part of the images from years 2018 and 2020.

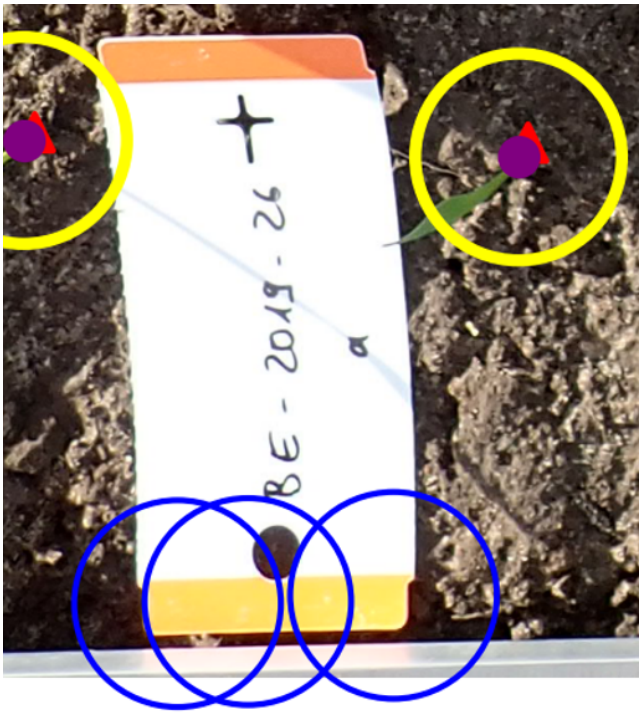
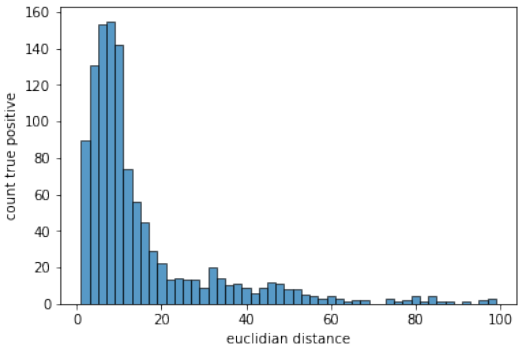


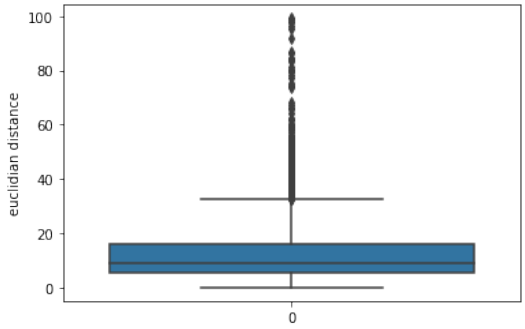
Figure 25: Example of false positive predictions by SegNet on the description panel.

There are also a very small amount of false positives coming from the false annotation in the images. As for example, in subdifures (d) of figures 22 and 23 in the second row, second from the bottom point as predicted, but was not in the annotation.

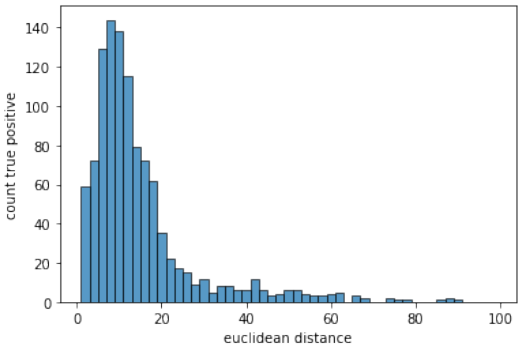
Figure 26 provides information in a form of histogram about the euclidean distances from the predicted and annotated points. We can see the that the major part of predictions are lying under 20 pixels distance.



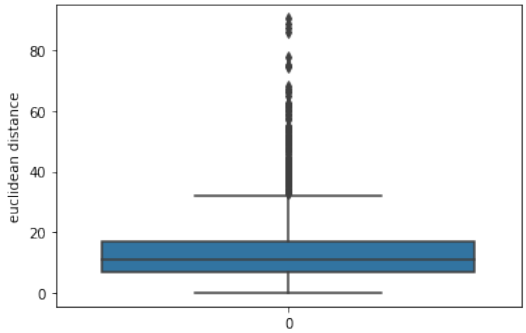
(a) Distribution of euclidean distances of predictions as histogram for SegNet



(b) Subfigure (a) as box plot



(c) Distribution of euclidean distances of predictions as histogram for UNet



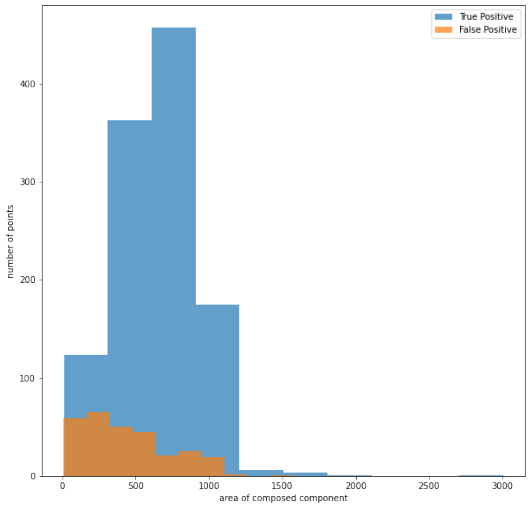
(d) Subfigure (c) as box plot

Figure 26: Histograms and box plots of euclidean distances between prediction and annotated point of test set

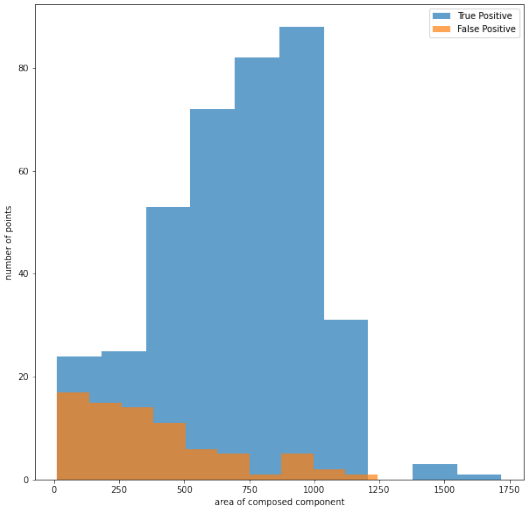
.

One another important mark is if for one seed there are two or more predicted placements. As for the test set, that consists 1294 annotated seed placements, SegNet with radius of finding equal 100 pixels has 15 cases when around an annotated seed placement 2 seed placements were predicted, UNet with the same radius has only 2 such cases. For validation set, consisted of 441 annotated seed placement, SegNet has 0 cases, as well as UNet, also having 0 such cases.

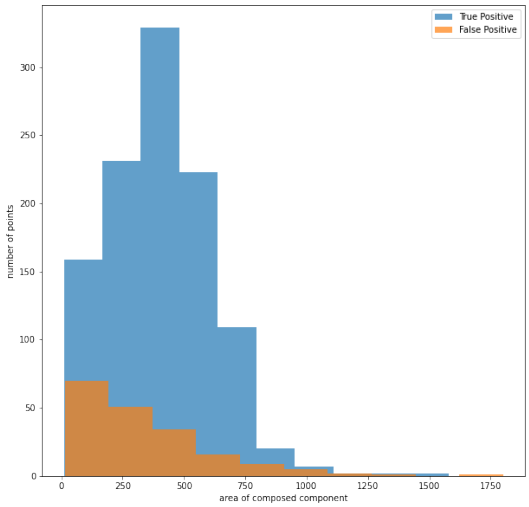
As shown on the subfigures (d) in figures 22 and 23 the areas of connected components differ. This gave us an assumption, that components with smaller area could be neglected in order to reduce the number of false positive. As shown on the figure 27, this assumption was wrong using a threshold for the area of component would only cut more true positive than false positive predictions.



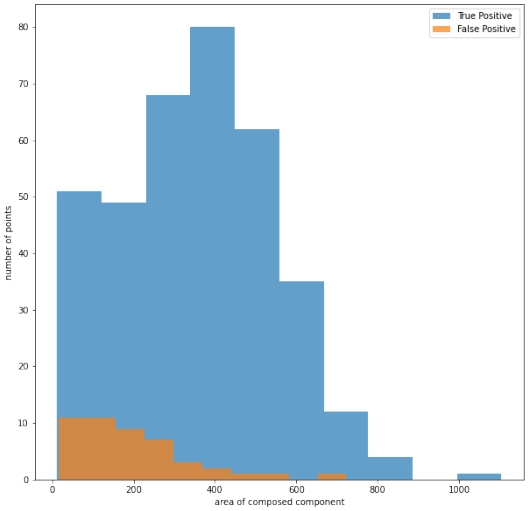
(a) Distribution of areas of the predictions of SegNet for test set



(b) Subfigure (a) as box plot



(c) Distribution of areas of the predictions of UNet for test set



(d) Subfigure (c) as box plot

Figure 27: Areas of connected components for SegNet and UNet in test and validation sets

5. Conclusion

In this work we have investigated how well can SegNet and UNet architectures with different losses perform within a segmentation task and be used for object locating. During the work a skill working with PyTorch has been severely increased, developed a skill to work with real-life datasets and to manipulate with data within different sources. Using PyTorch we have created this architectures and tested them on an unbalanced dataset. As the results show, we have achieved a satisfactory results represented by precision, recall and F1-Score, leaving a lot of room to investigate and try other combinations of networks and losses. As the curves of loss and other metrices show, it is also possible to continue the training already implemented networks. As for future work it is possible to try pre-trained models such as ResNet, EfficientNet or VGG and finetune them on the given dataset; try more complex losses such as boundary loss or hausdorff loss.

6. Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und die allgemeinen Grundsätze guter wissenschaftlicher Praxis beachtet habe.

Halle (Saale), 1. September 2021

.....
Danik Hollatz

References

[1] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] François Chollet. *Deep Learning with Python*. Manning Publications Co., 2018.

[3] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[4] *Deep Learning School*. [Online; Accessed on August 31, 2021]. URL: https://stepik.org/course/91156/promo.

[5] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[6] Mohamed Elgendy. *Deep Learning for Vision Systems*. Manning Publications Co., 2020.

[7] Fausto Milletari, Nassir Navab, Seyed-Ahmad Ahmadi. *V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation*. 2016. URL: https://arxiv.org/abs/1606.04797.

[8] Fezan. *Understanding of Semantic Segmentation & How Segnet Model work to perform Semantic Segmentation*. [Online; Accessed on August 20, 2021]. URL: https://medium.com/@fezancs/understanding-of-semantic-segmentation-how-segnet-model-work-to-perform-semantic-segmentation-5c426112e499.

[9] Irhum Shafkat. *Intuitively Understanding Convolutions for Deep Learning*. [Online; Accessed on August 20, 2021]. URL: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1.

[10] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. [Online; Accessed on August 20, 2021]. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[11] Olaf Ronneberger, Philipp Fischer, Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. URL: https://arxiv.org/abs/1505.04597.

[12] Peter Roelants. *Multivariate normal distribution*. [Online; Accessed on August 20, 2021]. URL: https://peterroelants.github.io/posts/multivariate-normal-primer/.

[13] Robin Vinod. *Batch Normalisation Explained.* [Online; Accessed on August 20, 2021]. URL: https://towardsdatascience.com/batch-normalisation-explained-5f4bd9de5feb.

[14] Shuchen Du. *Understanding Dice Loss for Crisp Boundary Detection.* [Online; Accessed on August 20, 2021]. URL: https://medium.com/ai-salon/understanding-dice-loss-for-crisp-boundary-detection-bb30c2e5f62b.

[15] Sik-Ho Tsang. *Review: SegNet (Semantic Segmentation).* [Online; Accessed on August 20, 2021]. URL: https://towardsdatascience.com/review-segnet-semantic-segmentation-e66f2e30fb96.

[16] Sophia Yang. *Deep learning basics — weight decay.* [Online; Accessed on August 20, 2021]. URL: https://medium.com/analytics-vidhya/deep-learning-basics-weight-decay-3c68eb4344e9.

[17] Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla. *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation.* 2015. URL: https://arxiv.org/abs/1511.00561.

[18] Vitaly Bushaev. *Adam — latest trends in deep learning optimization.* [Online; Accessed on August 20, 2021]. URL: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

[19] Wikipedia, the free encyclopedia. *Chain rule.* [Online; Accessed on August 20, 2021]. URL: https://en.wikipedia.org/wiki/Chain%5Crule.

[20] Wikipedia, the free encyclopedia. *Sørensen–Dice coefficient.* [Online; Accessed on August 20, 2021]. URL: https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dicecoefficient.

Appendices

A. SegNet Implementation

```
class SegNet(nn.Module):
    def __init__(self):
        super().__init__()

        # encoder (downsampling)
        self.enc_conv0 = nn.Sequential(
                            nn.Conv2d(in_channels=3, out_channels=64,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.BatchNorm2d(64),
                            nn.ReLU(),
                            nn.Conv2d(in_channels=64, out_channels=64,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.ReLU(),
                        )
        self.pool0 = nn.MaxPool2d(2, 2, return_indices=True)

        self.enc_conv1 = nn.Sequential(
                            nn.Conv2d(in_channels=64, out_channels=128,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.BatchNorm2d(128),
                            nn.ReLU(),
                            nn.Conv2d(in_channels=128, out_channels=128,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.ReLU(),
                        )
        self.pool1 = nn.MaxPool2d(2, 2, return_indices=True)

        self.enc_conv2 = nn.Sequential(
                            nn.Conv2d(in_channels=128, out_channels=256,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.BatchNorm2d(256),
                            nn.ReLU(),
                            nn.Conv2d(in_channels=256, out_channels=256,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.ReLU(),
                        )
        self.pool2 = nn.MaxPool2d(2, 2, return_indices=True)

        self.enc_conv3 = nn.Sequential(
                            nn.Conv2d(in_channels=256, out_channels=512,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.BatchNorm2d(512),
                            nn.ReLU(),
                            nn.Conv2d(in_channels=512, out_channels=512,
                            kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
                            nn.ReLU(),
                        )
        self.pool3 = nn.MaxPool2d(2, 2, return_indices=True)

        # bottleneck
        self.bottleneck_conv = nn.Sequential(
                                    nn.Conv2d(in_channels=512, out_channels=512,
                                    kernel_size=1, stride=1, bias=False),
```

```
                                    nn.BatchNorm2d(512),
                                    nn.ReLU(),
                                    nn.Conv2d(in_channels=512, out_channels=512,
                                    kernel_size=1, stride=1, bias=False),
                                    nn.BatchNorm2d(512),
                                    nn.ReLU(inplace=True),
                                )

        # decoder (upsampling)
        self.upsample0 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv0 = nn.Sequential(
                                nn.Conv2d(in_channels=512, out_channels=512,
                                kernel_size=3, stride=1, padding=1),
                                nn.BatchNorm2d(512),
                                nn.ReLU(),
                                nn.Conv2d(in_channels=512, out_channels=256,
                                kernel_size=3, stride=1, padding=1),
                                nn.ReLU(),
                            )
        self.upsample1 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv1 = nn.Sequential(
                                nn.Conv2d(in_channels=256, out_channels=256,
                                kernel_size=3, stride=1, padding=1),
                                nn.BatchNorm2d(256),
                                nn.ReLU(),
                                nn.Conv2d(in_channels=256, out_channels=128,
                                kernel_size=3, stride=1, padding=1),
                                nn.ReLU(),
                            )
        self.upsample2 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv2 = nn.Sequential(
                                nn.Conv2d(in_channels=128, out_channels=128,
                                kernel_size=3, stride=1, padding=1),
                                nn.BatchNorm2d(128),
                                nn.ReLU(),
                                nn.Conv2d(in_channels=128, out_channels=64,
                                kernel_size=3, stride=1, padding=1),
                                nn.ReLU(),
                            )
        self.upsample3 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv3 = nn.Sequential(
                                nn.Conv2d(in_channels=64, out_channels=64,
                                kernel_size=3, stride=1, padding=1),
                                nn.BatchNorm2d(64),
                                nn.ReLU(),
                                nn.Conv2d(in_channels=64, out_channels=1,
                                kernel_size=3, stride=1, padding=1),
                            )

    def forward(self, x):
        x = x.to(device)

        # encoder
        e0, ind0 = self.pool0(self.enc_conv0(x))
        e1, ind1 = self.pool1(self.enc_conv1(e0))
        e2, ind2 = self.pool2(self.enc_conv2(e1))
        e3, ind3 = self.pool3(self.enc_conv3(e2))

        # bottleneck
        b = self.bottleneck_conv(e3)
```

```
        # decoder
        d0 = self.dec_conv0(self.upsample0(b, ind3))
        d1 = self.dec_conv1(self.upsample1(d0, ind2))
        d2 = self.dec_conv2(self.upsample2(d1, ind1))
        d3 = self.dec_conv3(self.upsample3(d2, ind0))

        return d3
```

B. UNet implementation

```
class UNet(nn.Module):
    def __init__(self):
        super().__init__()

        self.enc_conv0 = nn.Conv2d(in_channels=3, out_channels=64,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.pool0 = nn.MaxPool2d(kernel_size=2)
        self.enc_conv1 = nn.Conv2d(in_channels=64, out_channels=128,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.enc_conv2 = nn.Conv2d(in_channels=128, out_channels=256,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.enc_conv3 = nn.Conv2d(in_channels=256, out_channels=256,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.pool3 = nn.MaxPool2d(kernel_size=2)

        self.bottleneck_conv = nn.Conv2d(in_channels=512, out_channels=512,
        kernel_size=3, padding=1, padding_mode='replicate')

        # decoder (upsampling)
        self.upsample0 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv0 = nn.Conv2d(in_channels=512, out_channels=256,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.upsample1 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv1 = nn.Conv2d(in_channels=256, out_channels=128,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.upsample2 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv2 = nn.Conv2d(in_channels=128, out_channels=64,
        kernel_size=3, padding=1, padding_mode='replicate')
        self.upsample3 = nn.MaxUnpool2d(2, stride=2)
        self.dec_conv3 = nn.Conv2d(in_channels=64, out_channels=1,
        kernel_size=3, padding=1, padding_mode='replicate')

    def forward(self, x):
        # encoder
        e0 = self.pool0(F.relu(self.enc_conv0(x)))
        e1 = self.pool1(F.relu(self.enc_conv1(e0)))
        e2 = self.pool2(F.relu(self.enc_conv2(e1)))
        e3 = self.pool3(F.relu(self.enc_conv3(e2)))

        # bottleneck
        b = self.bottleneck_conv(F.relu(e3))

        # decoder
```

```
        d0 = self.dec_conv0(F.relu(self.upsample0(b)))
        d1 = self.dec_conv1(F.relu(self.upsample1(
        self.dec_conv0(torch.cat((e2, d0), dim=1)))))
        d2 = self.dec_conv2(F.relu(self.upsample2(
        self.dec_conv1(torch.cat((e1, d1), dim=1)))))
        d3 = self.dec_conv3(self.upsample3(
        self.dec_conv2(torch.cat((e0, d2), dim=1))))
        return d3
```

C. Training loop and visualizing

```
def train(model, opt, loss_fn, epochs, data_tr, data_val):
    val_batch = next(iter(data_val))
    train_losses = []
    test_losses = []
    for epoch in range(epochs):
        print('* Epoch \%d/\%d' \% (epoch+1, epochs))
        avg_loss = 0
        model.train()  # train mode
        for batch in data_tr:
            data = model(batch['image'].to(device))
            opt.zero_grad()
            loss =  loss_fn(batch['mask'].to(device), data) # forward-pass
            loss.backward()  # backward-pass
            opt.step()  # update weights
            avg_loss += loss / len(data_tr)
        print('loss: \%f' \% avg_loss)
        train_losses.append(avg_loss)

        model.eval()  # testing mode
        Y_hat = model(val_batch['image']).detach().cpu()
        Y_target = val_batch['mask'].detach().cpu()

        test_loss = loss_fn(Y_target, Y_hat)
        test_losses.append(test_loss)

        #Visualizing tools
        plt.figure(figsize=(18, 12))
        clear_output(wait=True)
        for k in range(6):
            plt.subplot(4, 6, k+1)
            plt.imshow(np.rollaxis(val_batch['image'][k].numpy(), 0, 3),
            cmap='gray')
            plt.title('Real')
            plt.axis('off')

            plt.subplot(4, 6, k+7)
            plt.imshow(Y_hat[k, 0], cmap='gray')
            plt.title('Output')
            plt.axis('off')

            plt.subplot(4, 6, k+13)
            plt.imshow(Y_target[k, 0], cmap='gray')
            plt.title('Target')
            plt.axis('off')
```

```
            plt.subplot(4, 6, k+19)
            plt.imshow(torch.sigmoid(Y_hat[k, 0])>0.5, cmap='gray')
            plt.title('Binary Output')
            plt.axis('off')

        plt.suptitle('\%d / \%d - loss: \%f' \% (epoch+1, epochs, avg_loss))
        plt.show()
    return train_losses, test_losses
```

D. Heatmap produced by UNet

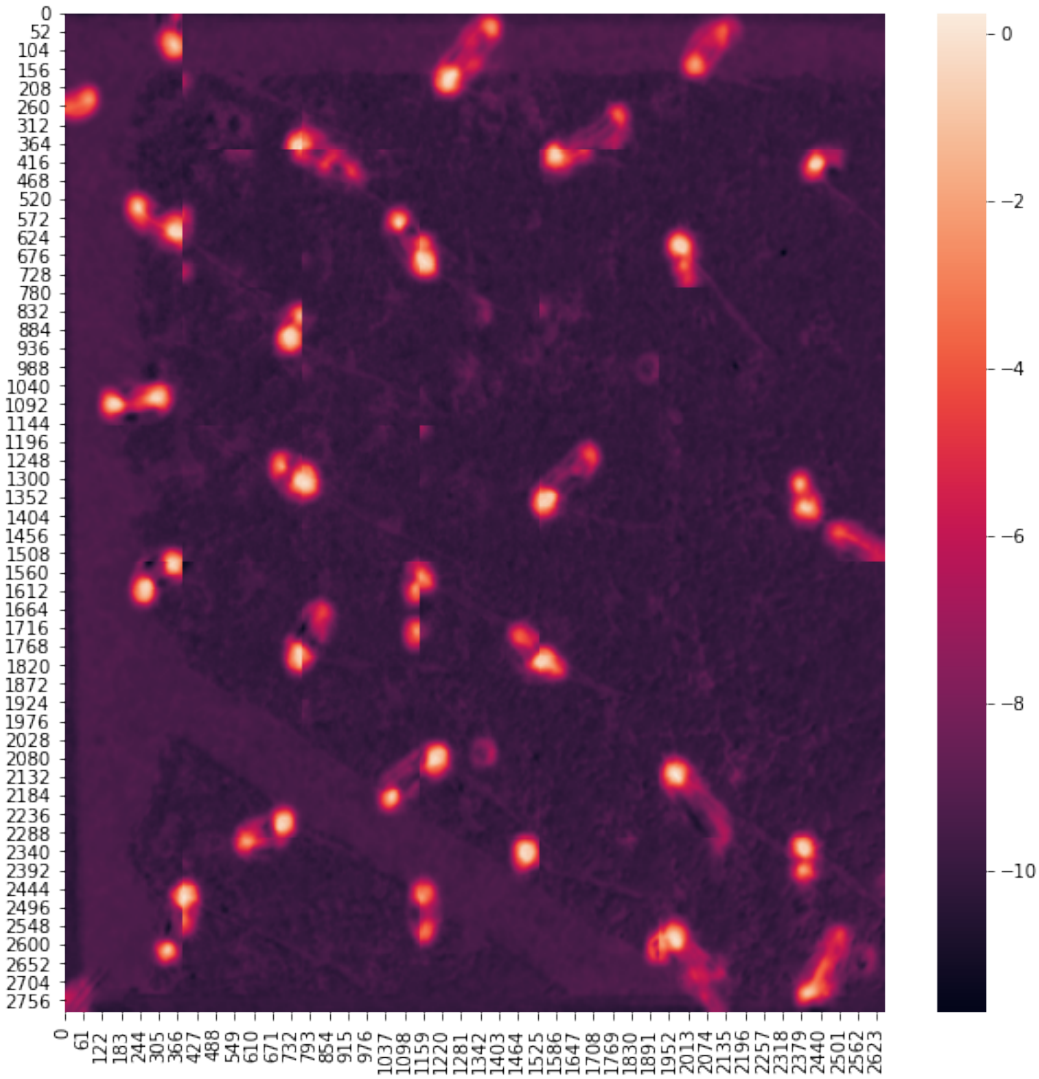


Figure 28

E. ”Activated” Heatmap

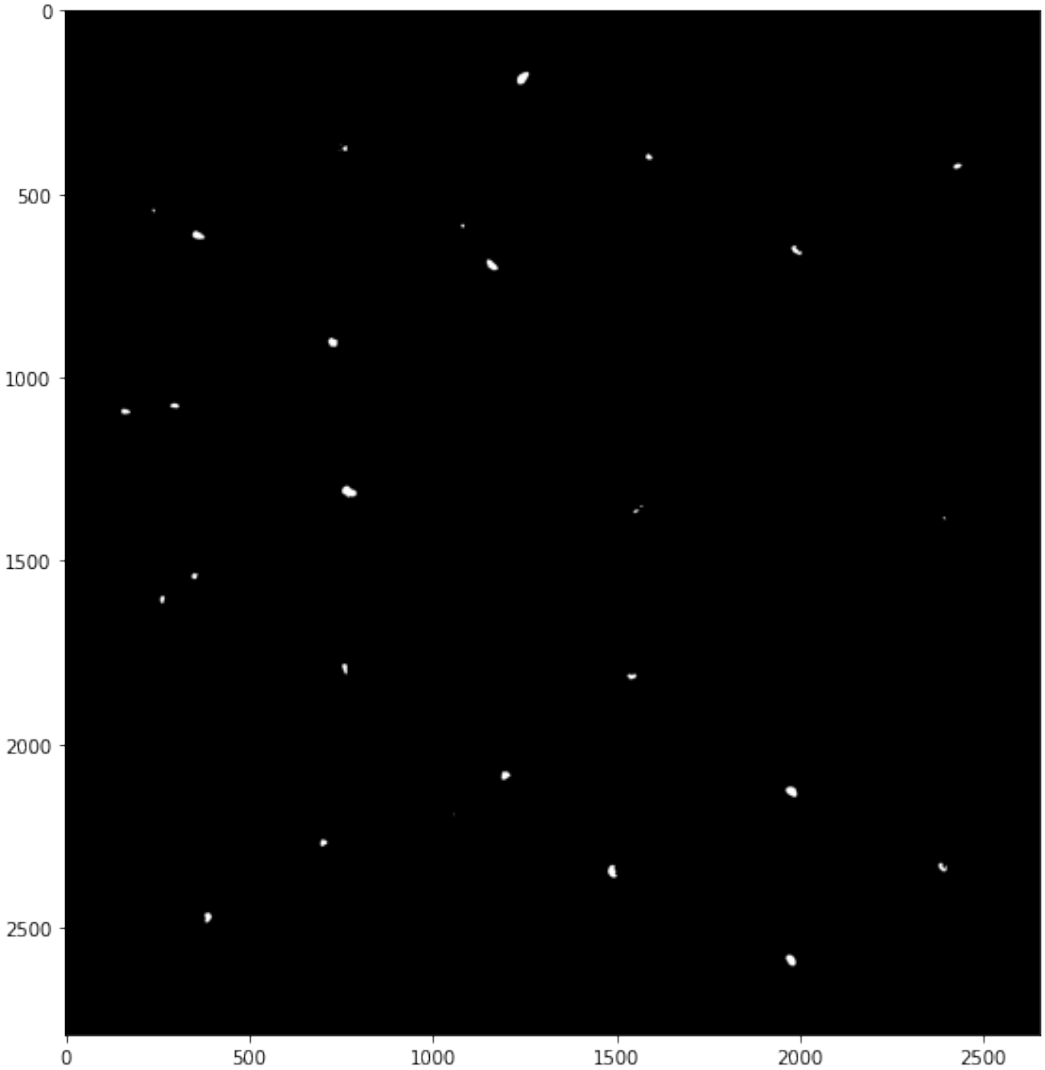


Figure 29

F. Image with predictions by UNet

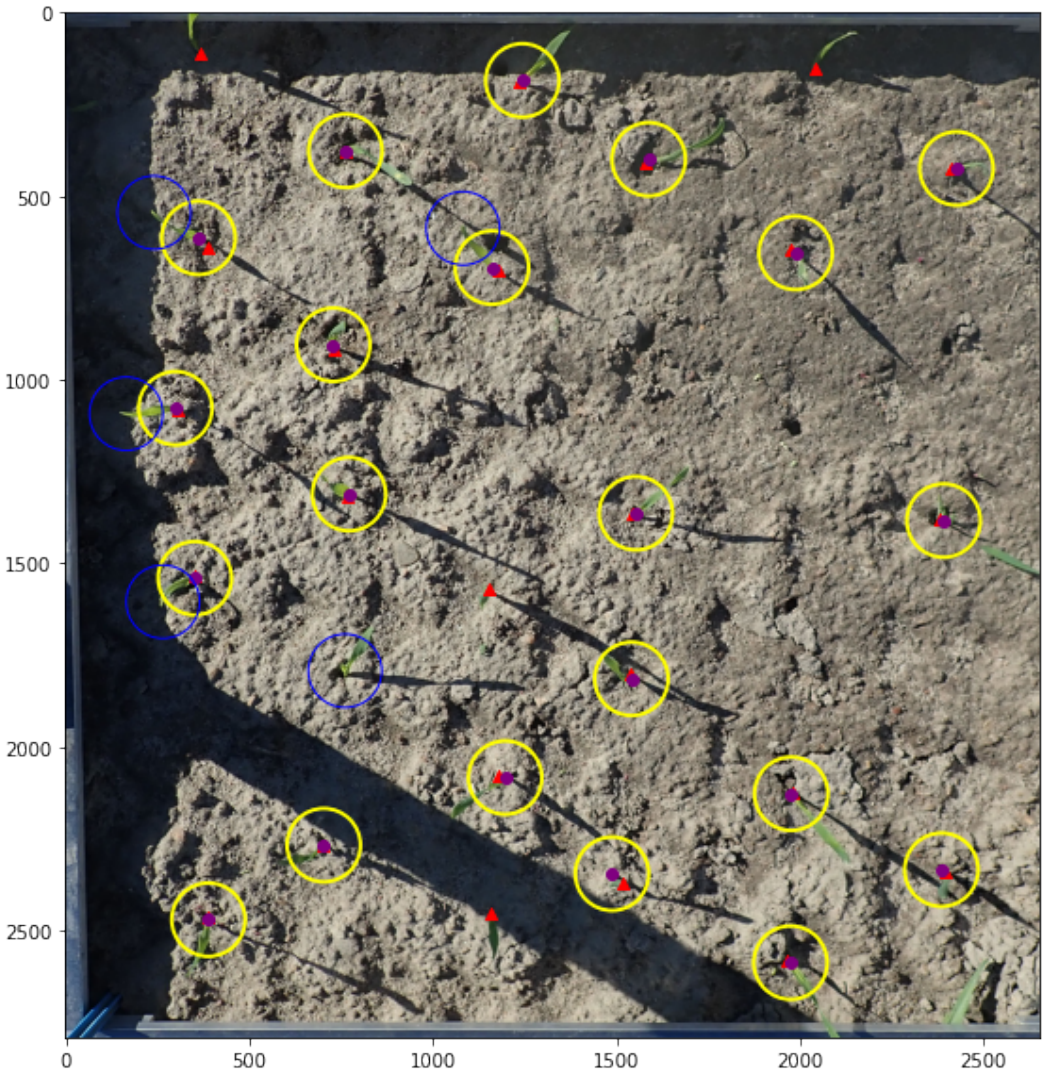


Figure 30