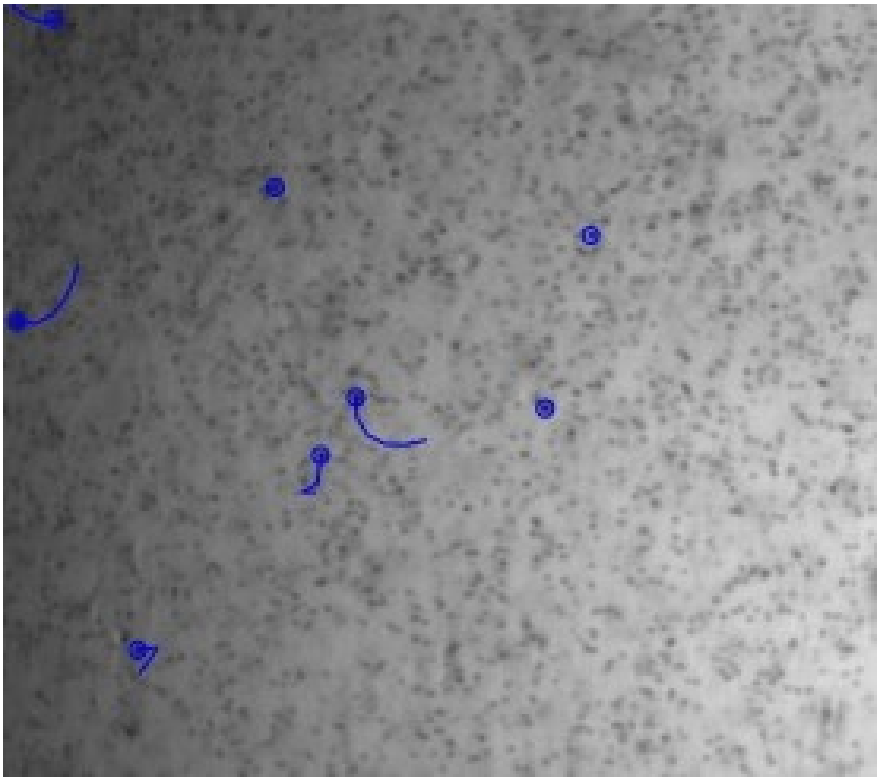


Tracking plankton using neural networks trained on simulated images

Master's thesis in Complex and Adaptive Systems

Agaton Fransson

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Tracking plankton using neural networks trained on simulated images

Agaton Fransson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Tracking plankton using neural networks trained on simulated images
Agaton Fransson

Supervisor: Daniel Midtvedt, Doctor, Physics Department, University of Gothenburg
Examiner and Supervisor: Giovanni Volpe, Professor, Physics Department, University of Gothenburg

Master's Thesis 2021
Department of Physics
Soft matter lab
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A sample containing *Strombidium arenicola* (big) and *Rhodomonas baltica* (small). *Strombidium arenicola* are tracked.

Typeset in LaTeX
Gothenburg, Sweden 2021

Tracking plankton using neural networks trained on simulated images
Agaton Fransson
Department of Physics
Chalmers University of Technology and University of Gothenburg

Abstract

Softwares to track particles often use algorithmic approaches to detect particles and to create tracks using the found positions, requiring human fine-tuning of parameters to achieve sought-for results. This can be time consuming and difficult, while also creating opportunities for human error and bias. With the developments of computational power and machine learning techniques such as deep learning, data driven approaches have made their way into many fields of science. Barriers preventing advances of such methods are the lack of available training data within a field and the level of proficiency required to create custom machine learning solutions. DeepTrack 2.0 is a software providing us with means to simulate digital microscopy images, build and train neural networks such as U-nets. In this paper DeepTrack 2.0 is utilized and built on to fit the needs of marine biologists when tracking plankton. Here I show that DeepTrack 2.0 provides us with the tools necessary to detect and track different types of plankton filmed in a variety of conditions with performance on par with and with the potential to outperform conventional tracking softwares. I also show that for plankton in a messy environment moving uniformly a network trained to detect motion rather than a shape proves more successful. These results demonstrate the versatility of deep learning methods and the potential of training networks on simulations for applications on real data, as is the case for marine biologists studying plankton. They also show the impact the structure of the training data has on the nature of the network.

Keywords: deep learning, U-net, digital microscopy, deeptrack, fiji trackmate.

Acknowledgements

I want to thank Giovanni Volpe, Daniel Midtvedt, Harshith Bachimanchi and Erik Selander for giving me feedback on my progress continually throughout the project. I want to thank Jesus Pineda and Benjamin Midtvedt for helping me with debugging and explaining DeepTrack 2.0 in the initial stages of the project. And finally I want to thank Kristie Rigby for analyzing the videos using TrackMate.

Agaton Fransson, Gothenburg, June 2021

Contents

List of Figures

1
Background

Plankton are organisms defined by their motility; they can't swim against the current of the fluid that they reside in [2]. Their size vary from less than a micron [3] (eg. protists) to metres (eg. jellyfish) [4]. Many species of fish are also plankton during their early larvae stage [5]. Plankton contribute to about 70% of the world's total production of oxygen [6] and is the food source for almost all fish at some stage in their life [7]. Marine biologists that study plankton sometimes want to track them in a microscopy sample, to do this they can use softwares that use algorithms to find the plankton frame by frame in the video. However, there are a set of assumptions made when running such a program, as well as hyper parameters to adjust to get the best result [8]. This makes the methods subject to human bias and mistakes, both with regards to how the algorithms themselves are designed but also how skilled the person using the software is at optimizing the hyper parameters. Thanks to recent developments of powerful computers and machine learning techniques, such as deep learning, data driven approaches has become more prevalent for both industrial purposes and various scientific fields [9]. These approaches has also made their way into the field of particle tracking [10, 11], but for the purpose of tracking plankton the lack of available training data has hindered their advance [12, 13, 14, 15, 16].

DeepTrack 2.0 is a deep learning framework for digital microscopy, providing functions to generate different types of artificial neural networks and to simulate microscopy images to use as training data. In this paper DeepTrack 2.0 is used and built on to train networks on simulated training data to detect and track plankton in real microscopy samples. The functions and methods that has been added and used through this project are described in chapter Methods. [17]

To solve the task of segmenting biomedical images where localization of cells is of interest the U-Net can be used. The output then is an image of the same width and height as the input where a class label is assigned to each pixel. [1] In this paper we expand on the approach of using the image to be segmented as input to also include using a sequence of images and also the computed difference between the images as input. The particles are then assigned helical motion to update their positions [18]. The idea behind this choice is that there is valuable information in comparing frames since it is easier for humans to notice something that is moving compared to finding something that is stationary. [19]

1.1 UNet

The UNet is a type of convolutional neural network (CNN) [20] that outputs a segmentation of the image instead of a classification. This is done through symmetrically adding an expansive up-convolution path at the end of a typical CNN as seen in figure 1.1.[1] The segmentations in figures 1.2b, 1.2c and 1.2d will be annotated as segmentation layers in this paper.

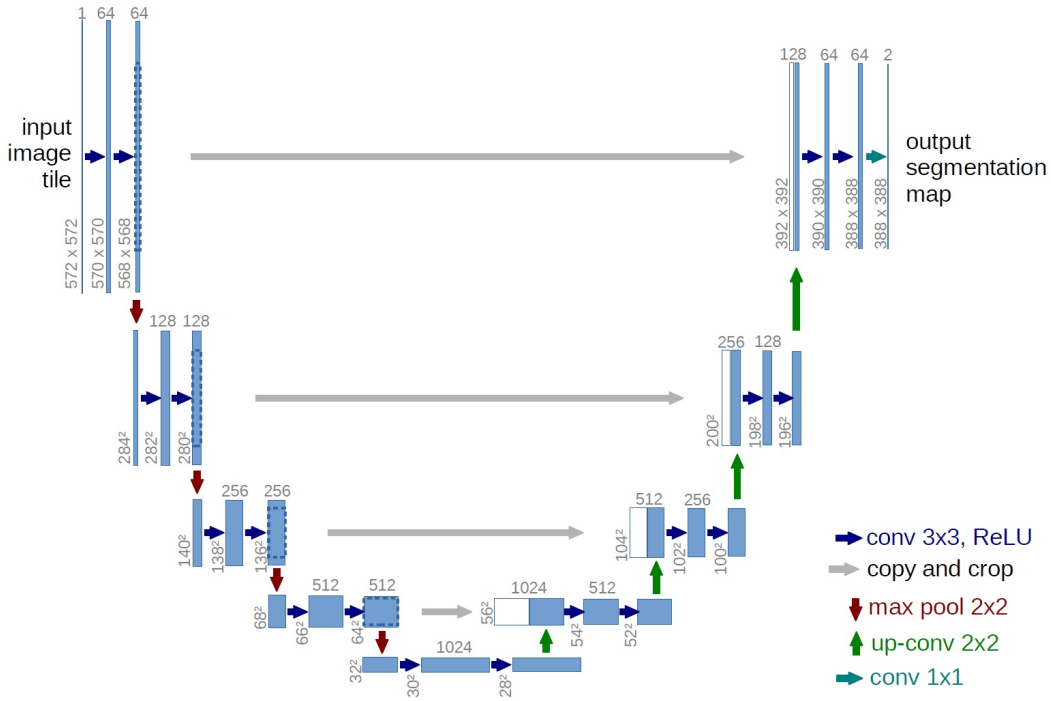
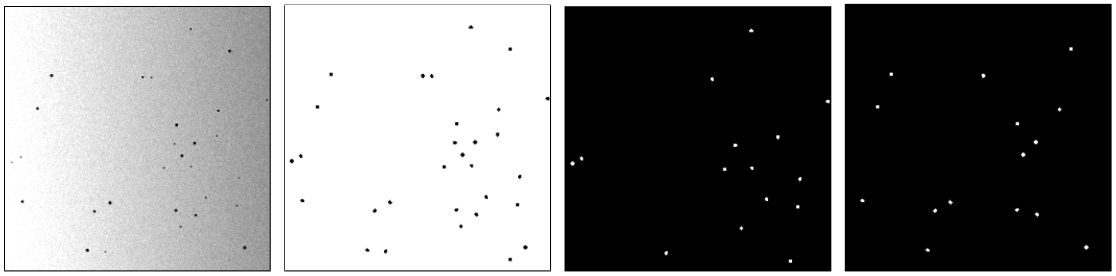


Figure 1.1: Architecture of a UNet. The left side follows the structure of a typical convolutional neural network contracting the image while the right side expands the image. The output of such a network can be seen in figure 1.2. Figure from [1].



(a) Simulation **(b)** Background **(c)** Particle 1 **(d)** Particle 2

Figure 1.2: One simulated image as input with corresponding target outputs. Each particle gets segmented to the layer corresponding to its label.

1.2 ImageJ TrackMate

The software the network will be compared to is ImageJ TrackMate [21]. The main points they will be compared on are

- Detection of plankton.
- Linking the positions of found plankton.

For detection of plankton TrackMate uses three algorithms based on Laplacian of Gaussian segmentation, these are

- Laplacian of Gaussan detector.
  - It applies a Gaussian filter to the image after which the Laplacian is calculated, which results in strong positive responses for dark blobs and strong negative responses for bright blobs. This method is strongly sensetive to the relationship between the size of the blob structures and the scale of the Gaussian filter. Recommended for spots of sizes between ≈ 5 and ≈ 20 pixels in diameter.
- Differences of Gaussians detector
  - Can be seen as an approximation of the Laplacian of Gaussian detector. Recommended for spots of sizes smaller than ≈ 5 pixels in diameter.
- Downsample Laplacian of Gaussan detector.
  - Downsizes the image before applying Laplacian of Gaussian detector. Recommended for spots of sizes bigger than ≈ 20 pixels in diameter.

[22], [23]
For linking of positions the software also has three algorithms,

- Linear Assignment Problem (LAP) framework by Jaqaman et al [24]
  - Links positions based on a linking cost calculated based on square distance. The calculation of cost can be modified to add penalties if the particles have different intensities, shapes etc.
- Kalman filter [25]
  - Traces particles and predicts their next most probable position based on their previous positions and an assumption of constant velocity.
- Nearest-neighbour
  - Links the positions closest to each other between frames.

[26]
Of these algorithms nearest-neighbour is the one most similar to what is used by our software.

2
Methods

This chapter can be seen as a user manual of how to use DeepTrack 2.0 to track plankton. The functions will be explained in a suggested order of usage. A couple of notebooks will also be provided showing how to implement these functions. The order followed here is

- Generating training data
  - Simulating particles
  - Simulating objective
  - Creating a sample, image, sequence
  - Plotting image
  - Creating labels
  - Plotting label
- Training network
  - Create batch function
  - Create training data generator
  - Create network
  - Train network
  - Evaluating network
  - Saving/loading model
- Analyze footage
  - View network's classification
  - Segmenting images and extracting positions
  - Create list of plankton with assigned positions, processing of positions
  - Plot and save track, create video
  - Plot net- vs gross distance, export positions

2.1 Generating training data

This section contains function used to generate and visualize training data.

2.1.1 load_and_plot_folder_image

Plots and returns an image in a folder.
Inputs:

- folder_path
  - The path to the folder with the images.
- frame

  - The number of the frame one wants to plot (assuming the images are sorted alphabetically).

Outputs:

- Plots the image
- image
  - The image is returned as a numpy array in RGB-format.

2.1.2 stationary_spherical_plankton

This function uses the function Sphere in DeepTrack 2.0 with some preset values to simplify generation of plankton-like particles. The particles will look like small, black dots. In this function im_size_height and -width define the area within which the particles will be simulated. The sphere will have a radius in the interval [input radius, input radius + input radius $\cdot$ 0.5].

Inputs:

- im_size_height
  - Image height with regard to number of pixels.
- im_size_width
  - Image width with regard to number of pixels.
- radius
  - Radius of one plankton measured in meters. However, the microscope function used uses a preset magnification such that values in the order of 10^{-7} should be used for usable results.
- label
  - Label of generated particle, decides segmentation layer. If set to -1 plankton will be classified to background.

Output:

- plankton
  - A spherical scatterer with the specified properties.

2.1.3 stationary_ellipsoid_plankton

This function uses the function Ellipsoid in DeepTrack 2.0 with some preset values to simplify generation of plankton-like particles. The particles will look like small, black ellipses. In this function im_size_height and -width define the area within which the particles will be simulated. Each of the radiuses of the ellipsoid will be in the interval [input radius, input radius + input radius $\cdot$ 0.5].

Inputs:

- im_size_height
  - Image height with regard to number of pixels.
- im_size_width
  - Image width with regard to number of pixels.
- radius
  - Radius of one plankton measured in meters in x-, y- and z-direction. However, the microscope function uses a preset magnification such that values in the order of magnitude 10^{-7} should be used for viable results.

- label
  - Label of generated particle, decides segmentation layer. If set to -1 plankton will be classified to background.

Outputs:

- plankton
  - An ellipsoidal scatterer with the specified properties.

2.1.4 moving_spherical_plankton

This function uses the function Sphere in DeepTrack 2.0 with some preset values to simplify generation of plankton-like particles. The particles will look like small, black dots. In this function im_size_height and -width define the area within which the particles will be simulated. To make the particles move between images in a sequence a function that defines their motion is necessary. The sphere will have a radius in the interval [input radius, input radius + input radius $\cdot$ 0.5].

Inputs:

- im_size_height
  - Image height with regard to number of pixels.
- im_size_width
  - Image width with regard to number of pixels.
- radius
  - Radius of one plankton measured in meters. However, the microscope function used uses a preset magnification such that values in the order of 10^{-7} should be used for usable results.
- label
  - Label of generated particle, decides segmentation layer. If set to -1 plankton will be classified to background.
- diffusion_constant_coeff
  - Constant multiplied to preset diffusion constant, can be seen as velocity.

Outputs:

- plankton
  - A spherical scatterer with the specified properties.

2.1.5 moving_ellipsoid_plankton

This function uses the function Ellipsoid in DeepTrack 2.0 with some preset values to simplify generation of plankton-like particles. The particles will look like small, black dots. In this function im_size_height and -width define the area within which the particles will be simulated. To make the particles move between images in a sequence a function that defines their motion is necessary. Each of the radiuses of the ellipsoid will be in the interval [input radius, input radius + input radius $\cdot$ 0.5].

Inputs:

- im_size_height
  - Image height with regard to number of pixels.
- im_size_width
  - Image width with regard to number of pixels.

- radius
  - Radius of one plankton measured in meters in x-, y- and z-direction. However, the microscope function used uses a preset magnification such that values in the order of 10^{-7} should be used for usable results.
- label
  - Label of generated particle, decides segmentation layer. If set to -1 plankton will be classified to background.
- diffusion_constant_coeff
  - Constant multiplied to preset diffusion constant, can be seen as velocity.

Outputs:

- plankton
  - An ellipsoidal scatterer with the specified properties.

2.1.6 Generating a sequence of moving plankton

To make the plankton move between frames they need to be turned into sequential particles. This is done by feeding the plankton into the function Sequential together with a function that updates the feature (eg. position) that one wants to change between the frames. The function Sequential is in the original library of DeepTrack 2.0 so it wont be described here, but the functions used to update the positions will.

2.1.7 get_position_moving_plankton

Many plankton move in a helical trajectory [18], this function updates the positions of the plankton in such a manner. What I found most effective though was to only generate a sequence of three images so the type of motion isn't very important. The function receives its inputs from the already generated plankton.

2.1.8 get_position_stationary_plankton

If one wants to add stationary particles to the sequence this update function simply returns the previous position of the particle as its next position.

2.1.9 plankton_brightfield

This function creates a brightfield microscope with an illumination gradient of white light using functions available in DeepTrack 2.0.

Inputs:

- im_size_height
  - Image height with regard to number of pixels, must be divisible with 16 since a U-net is used.
- im_size_width
  - Image width with regard to number of pixels, must be divisible with 16 since a U-net is used.
- gradient_amp
  - Sets the amplitude of the illumination gradient.

Outputs:

- brightfield_microscope
  - The microscope object that will be used to create an image with the sample.

2.1.10 create_image

This function creates an image with the plankton sample and microscope. It also adds a poisson noise to the image, normalizes it and flips the image so that one simulated image creates four training images. To create strongly shaded areas and strongly lit areas one can set the normalization values to values outside the range 0 and 1, then the values above and below will be clipped to 0 and 1.
Inputs:

- noise_amp
  - Amplitude of noise.
- sample
  - The plankton sample.
- microscope
  - The microscope.
- norm_min
  - The lower bound for the normalization.
- norm_max
  - The upper bound for the normalization.

Outputs:

- image
  - The image object, like a recipe for how the image shall be generated. Must use *image.resolve()* generate an image.

2.1.11 create_sequence

This function creates a sequence with the plankton sample and microscope. It also adds a poisson noise to the images and normalizes them. To create strongly shaded areas and strongly lit areas one can set the normalization values to values outside the range 0 and 1, then the values above and below will be clipped to 0 and 1.
Inputs:

- noise_amp
  - Amplitude of noise.
- sample
  - The sequential plankton sample.
- microscope
  - The microscope.
- norm_min
  - The lower bound for the normalization.
- norm_max
  - The upper bound for the normalization.

Outputs:

- sequence

  - The sequence object, like a recipe for how the sequence shall be generated. Must use *sequence.resolve()* generate a sequence.

2.1.12 plot_image

Updates and plots the image.
Input:

- image
  - The simulated image you want to plot.

Outputs:

- Plots the image.

2.1.13 plot

To plot a sequence write *sequence.plot()* where sequence is the sequence of images to plot. This is a function from DeepTrack 2.0 and will create an animation of the sequence. Write *cmap='gray'* to plot the image in gray scale, more arguments are found in the DeepTrack 2.0 documentation.

2.1.14 get_target_image

Generates a target image from the simulated image of plankton. The output is a stack (depends on number of labels) of segmented images of the input, one with a white background and black dots and the rest with black background and white dots. The white pixels signify what in the image belongs to the specified label, the goal is to turn plankton pixels white.
Input:

- image_of_particles
  - The simulated image of plankton.

Outputs:

- label
  - A three dimensional numpy.array.

2.1.15 get_target_sequence

Generates a target sequence from the simulated sequence of plankton. The output is a stack of segmented images of the input where each image corresponds to one frame. The image with a white background and black dots corresponds to the background label, the rest with black background and white dots corresponds to each of the images in the sequence. If the sequence contains more than one type of plankton the label will be of image $\frac{\text{sequence\_length}}{2}$ rounded to first integer up. The white pixels signify what in the image belongs to the specified label, the goal is to turn plankton pixels white.
Input:

- sequence_of_particles
  - The simulated sequence of plankton.

Outputs:

- label
  - A three dimensional numpy.array.

2.1.16 plot_label

Plots the labels generated from the input image.
Inputs:

- label_function
  - The function used to generate labels, either *get_target_image* or *get_target_sequence*.
- image
  - Image or sequence to get labels from.

Outputs:

- Plots the label.

2.2 Training network

This section contains functions to generate and train network.

2.2.1 create_custom_batch_function

Before the network is trained with the simulated images one might want to apply some functions to them, remove the mean image from them or subtract subsequent images from each other. This function allows us to customize how the training images are treated and presented to the network during training. The output is the batch function which takes the resolved image/sequence as input and outputs the treated image/sequence.
Inputs:

- imaged_particle_sequence
  - The simulated image or sequence.
- outputs
  - A list of how the output should be organized. The input [[0,1], [1,2], 0, 1, 2] means that the first two outputs is the differences image1 − image0 and image2 − image1 and the last three outputs are the images image0, image1 and image2 in that order. Image 0 is the first image of the sequence. If one simulates one image this input should be [0].
- function_img
  - A list of functions to be applied to the images. Should usually be ended with some sort of normalization. Example: [*lambda img: -img*, *normalize_image*]. Keyword arguments to the functions can be added to the input. Functions that aren't supported by the function signature (eg. *numpy.exp*, *numpy.log*) must be written as *lambda x: numpy.exp(x)*.
- function_diff
  - A list of functions to be applied to the differences between the images. Should usually be ended with some sort of normalization.

Outputs:

- custom_batch_function
  - Function that applies all adjustments to the simulated image/sequence.

2.2.2 plot_batch

Plots the output of the created batch function.
Inputs:

- images
  - The simulated image/sequence you use.
- batch_function
  - The custom made batch function.

Outputs:

- Plots what the training images will look like.

2.2.3 normalize_image

Normalizes the image between the specified values, default to 0 and 1.
Inputs:

- image
  - Image to normalize.
- min_value
  - Lower normalization value.
- max_value
  - Upper normalization value.

Outputs:

- image
  - The normalized image as a numpy.array.

2.2.4 remove_running_mean

The running mean of an image is the mean of its local, surrounding images. This function removes the running mean from an image, the mean is calculated from a specified number of images before and after the specified image.
Inputs:

- image
  - The image the running mean will be removed from.
- folder_path
  - The path to the folder with the images.
- tot_no_of_frames
  - Total number of frames to be used in averaging.
- center_frame
  - Frame the local mean is calculated around.
- im_width
  - Image width with regard to number of pixels.
- im_height
  - Image height with regard to number of pixels.

Outputs:
- image
  - The normalized image with running mean removed as a numpy.array.

2.2.5 get_mean_image

Calculates and returns the resized mean image of the images in the folder.
Inputs:
- folder_path
  - The path to the folder with the images.
- im_size_width
  - Image width with regard to number of pixels.
- im_size_height
  - Image height with regard to number of pixels.

Outputs:
- image
  - The mean image of the images in the folder as a numpy.array.

2.2.6 ContinuousGenerator

This function is from the original DeepTrack 2.0 library. It allows us to generate all the training data before starting training and reuse it during training so that the training can be run entirely on the GPU. A more extensive documentation is available on DeepTrack 2.0's github.
Inputs:
- imaged_particle_sequence
  - The simulated image/sequence.
- get_target_sequence
  - The label function.
- batch_function
  - The batch function.
- batch_size
  - The size of one batch, ex. 8.
- min_data_size
  - Number of generated data samples before starting training, must be bigger than or equal to the batch size times the number of steps per epoch in training.
- max_data_size
  - The maximum number of data samples generated before new data starts to replace old data.

Outputs:
- generator
  - The generator used during training to generate the training set.

2.2.7 generate_unet

This function uses the function unet from DeepTrack 2.0 with some preset parameters to create a UNet, the output is the keras model. If the width and height are set to None then network will be able to segment any image size (divisible by 16).
Inputs:

- im_size_height
  - Image height with regard to number of pixels.
- im_size_width
  - Image width with regard to number of pixels.
- no_of_inputs
  - Number of images per training sample.
- no_of_outputs
  - Number of output images.

Outputs:

- model
  - The UNet.

2.2.8 train_model_early_stopping

Uses model.train to train the model using early stopping to stop the training when the loss hasn't improved for a number of epochs.
Inputs:

- model
  - The keras model to be trained
- generator
  - The generator used to create the training data.
- patience
  - Stops training after this number of epochs since the last improvement
- epochs
  - Maximum number of epochs to train network.
- steps_per_epoch
  - Number of steps per epoch.

Outputs:

- model
  - The trained network.

2.2.9 model.save

To save the model you write *model.save(save_path)* (if the network is called model).
Input:

- save_path
  - String of the path to where the network will be saved.

Outputs:

- The network will be save in the specified location.

2.2.10 keras.models.load_model

To load a model you have saved you write *keras.models.load_model(load_path_model, custom_objects={'softmax_categorical':softmax_categorical})* where the second argument is used to load the custom defined loss function used in the U-Net.
Input:

- load_path_model
  - String of the path to where the network to be loaded is saved.

Outputs:

- model
  - The model saved at the location of the path.

2.2.11 softmax_categorical

Loss function used when training a U-Net.
Inputs:

- T
  - Truth, label of the image.
- P
  - The image.

Outputs:

- error
  - The error of the prediction.

2.3 Analyze footage

This section contains function used to analyze the plankton video.

2.3.1 get_image_stack

Similar to the create custom batch function. This function is used to structure the input images to the correct format for the network to predict them. The structure must be the same as the output from the batch function. The output is an array with the images stacked on each other.
Inputs:

- outputs
  - A list of how the output should be organized. The input [[0,1], [1,2], 0, 1, 2] means that the first two outputs is the differences image1 − image0 and image2 − image1 and the last three outputs are the images image0, image1 and image2 in that order. Image 0 is the first image of the sequence. If one simulates one image this input should be [0].
- folder_path
  - The path to the folder with images to analyze. Only images can be in the folder.
- frame_im0

  - The frame number of the first image to be analyzed, this will be image0 in the "outputs" argument.
- im_size_width
  - Image width of the output image with regard to number of pixels.
- im_size_height
  - Image height of the output image with regard to number of pixels.
- im_resize_width
  - If the image is to be up-/down sized this is the width with regard to number of pixels. For instance the images might be 1280x1024 pixels, but the training has been done with the dimensions 640x512, this is where the resizing is done.
- im_resize_height
  - If the image is to be up-/down sized this is the height with regard to number of pixels. For instance the images might be 1280x1024 pixels, but the training has been done with the dimensions 640x512, this is where the resizing is done.
- function_img
  - A list of functions to be applied to the images. Should usually be ended with some sort of normalization. Example: [*lambda img: -img*, *Normalize_image*]. Keyword arguments to the functions can be added to the input of im_stack. Functions that aren't supported by the function signature (eg. *numpy.exp*, *numpy.log*) must be written as *lambda x: numpy.exp(x)*.
- function_diff
  - A list of functions to be applied to the differences between the images. Should usually be ended with some sort of normalization.

Outputs:
- im_stack
  - The stack of images to be used as input to the network, as numpy.array.

2.3.2 plot_image_stack

Plots the images generated by the image stack function.
Input:
- im_stack
  - The output of the get_image_stack function.

Outputs:
- Plots the image stack.

2.3.3 plot_prediction

Plots the prediction of the network on the input image stack.
Inputs:
- model
  - The trained model.
- im_stack

  - The output of the get_image_stack function.

Outputs:

- Plots the prediction of the model on the image stack.

2.3.4 get_blob_centers

Finds clusters of ones in an array and assigns labels to them by changing each cluster of ones to clusters of another integer. The filter looking for clusters is a 3x3 array of ones. Uses the function label from scipy.ndimage.

Inputs:

- prediction
  - A prediction from the model where all values are set to ones or zeros.
- value_threshold
  - Values in the prediction above the threshold are set to 1 and values below are set to 0.
- prediction_size
  - Filters away clusters with fewer than or equal to the number of pixels assigned.

Outputs:

- centers
  - A numpy.array of the row-/column-coordinates of the blobs.

2.3.5 get_blob_center

Takes the labeled array and finds the center coordinates of each cluster.

Inputs:

- label
  - Integer of which cluster it looks for.
- array
  - Clustered prediction array.

Outputs:

- row_center
  - The row (y-coordinate) of the center of the blob.
- col_center
  - The column (x-coordinate) of the center of the blob.

2.3.6 extract_positions_from_predictions

Uses the model to make a prediction on the image stack and finds the positions of all the white dots in the specified layer of the output of the prediction. The output is a list of x-/y-coordinates.

Inputs:

- im_stack
  - The output of the *get_image_stack function.*
- model
  - The trained model.

- layer
  - The layer of the prediction of which to extract positions from.

Outputs:

- positions
  - The found positions in one prediction of the network.

2.3.7 extract_positions

Loops over *get_image_stack* and *extract_positions_from_predictions* to produce a list of lists where each nested list contains all found positions in one frame of the video to be analyzed. All arguments used in aforementioned functions should also be provided as keyword arguments.

Inputs:

- no_of_frames
  - The number of frames to be analyzed.
- frame_im0
  - The frame the predictions starts on, useful if one doesn't want to start on the first frame.
- value_threshold
  - Values in the prediction above the threshold are set to 1 and values below are set to 0. Isn't used by this function but gets sent to get_blob_centers, is mentioned here because of its importance when using this function.
- prediction_size
  - Filters away clusters with fewer than or equal to the number of pixels assigned. Isn't used by this function but gets sent to get_blob_centers, is mentioned here because of its importance when using this function.

Outputs:

- positions
  - A list of the found positions in all of the predicted images.

2.3.8 plot_found_positions

Takes the list of positions and plots the positions of the first frame.

Inputs:

- positions
  - The positions found by the network and clustering.
- width
  - The number of pixels on the width.
- height
  - The number of pixels on the height.

Outputs:

- A plot with white dots at the found positions on a black background.

2.3.9 crop_and_append

Can be used to remove parts of an image, useful to save processing time and reduce risk for misclassifications. It will remove the pixels between the specified x-values

and y-values, and remove rows and columns on the edges of the resulting image to make the dimensions be divisible by "mult_of" (16 as default).
Inputs:

- image
  - The image to crop
- col_delete_list
  - A list of an even number of values between which the pixels will be removed.
- row_delete_list
  - A list of an even number of values between which the pixels will be removed.
- mult_of
  - The value the dimensions of the image will be divisible with.
- print_shape
  - If True the shape of the resulting image will be printed as well.

Outputs:

- image
  - The numpy.array with the specified rows and columns removed.

2.3.10 fix_positions_from_cropping

Takes the positions received from *extract_positions* and maps them back to the positions they would have on the image before cropping.
Inputs:

- positions
  - The positions found by the network and clustering.
- col_delete_list
  - A list of an even number of values between which the pixels will be removed, same list as used in *crop_and_append*.
- row_delete_list
  - A list of an even number of values between which the pixels will be removed, same list as used in *crop_and_append*.

Outputs:

- positions
  - The list of positions where they would have been on the uncropped image.

2.3.11 class Plankton

A class that creates instances called planktonx where x is a positive integer (0 for the first plankton) decided by the order of creation. Each plankton is initialized by assigning it a position at the row of the specified time step, all other rows are numpy.nan. The planktons will then be filled with new positions as the list of positions is processed.
Inputs:

- position
  - The position the plankton is initialized with.

- number_of_timesteps
  - The number of frames to be analyzed.
- current_timestep
  - The time step of the position to be assigned.

Outputs:

- plankton
  - The plankton.

The methods are:

self.add_position

Adds a position to the plankton at the specified time step.
Inputs:

- position
  - The position to be added to the plankton.
- timestep
  - The time step of the position to be assigned.

Outputs:

- Adds the positions to the plankton.

self.get_latest_position

Extracts the latest position a plankton had in relation to a specified time step.
Inputs:

- timestep
  - The time step the position will first look for a position at.
- time_threshold
  - Number of time steps backwards in time to search if no position is found at the specified time step.

Outputs:

- latest_position
  - The latest found position in relation to the time step and time_threshold.

self.get_mean_velocity

Calculates the mean velocity of the plankton. Outputs:

- mean_velocity
  - The mean velocity of the specified plankton.

2.3.12 initialize_plankton

Creates a dictionary where every position found gets assigned to a plankton, the plankton are named plankton0, plankton1, etc.
Inputs:

- positions
  - The list of positions to be assigned.
- number_of_timesteps

  - The number of frames analyzed.
- current_timestep
  - If no positions were found in the first time step this argument lets us initialize the plankton from another time step.

Outputs:

- list_of_plankton
  - The list of plankton produced by the first prediction.

2.3.13 update_list_of_plankton

Assigns the positions found at one time step to the plankton in the list according to which plankton is closest from the previous time step(s). If two or more plankton are found the position is assigned trying to maintain each plankton's mean velocity. If no plankton is found close enough to a position a new plankton is initialized with that position at the given time step. The option to search for plankton close to the plankton's extrapolated positions is also possible. The extrapolated position is calculated through linear extrapolation.

Inputs:

- list_of_plankton
  - The list of plankton.
- positions
  - The positions at the given time step.
- max_dist
  - The maximum distance from a position a plankton will be assigned a position at.
- timestep
  - The time step the positions were extracted from.
- threshold
  - The number of time steps back plankton are searched for in the vicinity of the position.
- extrapolate
  - Boolean, True if extrapolation should be used when looking for plankton. Useful if the plankton have a uniform motion.

Outputs:

- list_of_plankton
  - The list of plankton updated with the positions found in the next prediction.

2.3.14 assign_positions_to_planktons

Loops over *initialize_plankton* and *update_list_of_plankton* until all found positions have been assigned.

Input:

- positions
  - The list of positions to be assigned.

Outputs:

- list_of_plankton
  - List of plankton with all the found positions assigned to plankton.

2.3.15 interpolate_gaps_in_plankton_positions

If a plankton position wasn't found one time step but found again in the previous and next this function interpolates to fill the gap.
Input:

- list_of_plankton
  - The list of plankton.

Outputs:

- list_of_plankton
  - The list of plankton where missing values, with neighbouring values, has been interpolated.

2.3.16 extrapolate_positions

If a plankton position wasn't found one time step but found in the previous two this function linearly extrapolates the position.
Inputs:

- list_of_plankton
  - The list of plankton.
- timestep
  - The time step of the position to be extrapolated.

Outputs:

- list_of_plankton
  - The list of plankton where missing values, two previous values, has been extrapolated.

2.3.17 trim_list_from_stationary_planktons

This function is used to remove plankton that doesn't move further than a specified distance in its observed positions.
Inputs:

- list_of_plankton
  - The list of plankton.
- min_distance
  - The minimum distance a plankton is allowed to travel to not get removed from the list.

Outputs:

- list_of_plankton
  - The list of plankton where plankton moving too short distances have been removed.

2.3.18 split_plankton

Splits the list of plankton into two lists depending on the percentage of observed positions each plankton has. The output is two lists, the first one being the list of plankton passing the threshold.
Inputs:

- list_of_plankton
  - The list of plankton.
- percentage_threshold
  - A number between 0 and 1 that decides which list the plankton get assigned to.

Outputs:

- plankton_track
  - The list of plankton with positions found on more of the images than the specified fraction.
- plankton_dont_track
  - The list of plankton with positions found on fewer of the images than the specified fraction.

2.3.19 plot_and_save_track

Plots the positions as circles on the images analyzed. Possible to: add a trace, add numbers to the plankton, track one (or more) specific plankton, change x- and y-axis units from pixel to a unit of length. With *save_images* True the function wont output images but will instead save them to a specified path with the number of order added (0, 1, 2, ...) to the selected filename.
Inputs:

- no_of_frames
  - The number of frames to be plotted.
- plankton_track
  - A list of plankton.
- plankton_dont_track
  - A list of plankton.
- folder_path
  - Path to the folder with the images that was analyzed.
- frame_im0
  - The number of the first image of the analyzed frames.
- save_images
  - True if the images should be saved, False if one wants to see the plots.
- show_plankton_track
  - True if one wants to plot the plankton used as input to plankton_track.
- show_plankton_dont_track
  - True if one wants to plot the plankton used as input to plankton_dont_track.
- show_specific_plankton
  - True if one wants to plot a specific plankton.
- show_numbers_track

  - True if one wants to plot the number of each plankton in plankton_track next to them.
- show_numbers_dont_track
  - True if one wants to plot the number of each plankton in plankton_dont_track next to them.
- show_numbers_specific_plankton
  - True if one wants to plot the number of the plankton in specific_plankton next to it.
- specific_plankton
  - List of numbers where number [14] would mean plankton14 would get plotted.
- color_plankton_track
  - The color of the circle and trace plotting the plankton entered in plankton_track.
- color_plankton_dont_track
  - The color of the circle and trace plotting the plankton entered in plankton_dont_track.
- color_specific_plankton
  - The color of the circle and trace plotting the plankton entered in specific_plankton.
- im_size_width
  - Image width with regard to number of pixels of the images the network is fed with. This re-scales the positions to fit the in the full size image they are plotted on.
- im_size_height
  - Image height with regard to number of pixels of the images the network is fed with. This re-scales the positions to fit the in the full size image they are plotted on.
- x_axis_label
  - Text under x-axis.
- y_axis_label
  - Text next to y-axis.
- pixel_length_ratio
  - Scaling factor for conversion from pixels to the wanted unit of length.
- save_path
  - String of the path to where the images will be saved.
- frame_name
  - String of the wanted name of each image, a number ranging from 0 to the number of images will be added to the name.
- file_type
  - A string of the file type, eg. '.jpg'.

Outputs:

- Either plots or saves the images.

2.3.20 get_mean_net_and_gross_distance

Calculates the mean gross distance and the mean net distance of the plankton. The output is first the mean net distance and then the mean gross distance.
Inputs:

- list_of_plankton
  - The list of plankton.
- use_3D_dist
  - Multiplying with scaling factor to adjust for that the calculated distances don't take into account movement in the z-direction.

Outputs:

- mean_net_distances
  - An array of the mean net distances of all the plankton.
- mean_gross_distances
  - An array of the mean gross distances of all the plankton.

2.3.21 save_positions

Saves the positions of the plankton in the provided list as either .xlsx or .csv. A scaling factor needs to be specified or the positions will be in pixels.
Inputs:

- list_of_plankton
  - The list of plankton.
- save_path
  - Path to where the file will be saved.
- file_format
  - Format of the file, '.csv' or '.xlsx'.
- pixel_length_ratio
  - The scaling factor between pixels and the unit of length wanted.

Outputs:

- Saves the positions as an .xlsx- or .csv-file at the specified location.

2.3.22 make_video

Takes the saved images and puts them together to a .avi-video.
Inputs:

- frame_im0
  - The first frame of the video in the folder.
- folder_path
  - Path to the folder where the images are saved.
- save_path
  - Path to where the video will be saved.
- fps
  - The frame rate of the video in frames per second.
- no_of_frames
  - The number of frames to be used in the video.

Outputs:

- video
  - Saves a video at the specified location.

2.3.23 get_track_durations

Creates an array where each row corresponds to a track length. Each row will be filled with a number corresponding to the number of plankton having that track length. Takes the difference between the first frame and the last frame so missing values wont change the length.
Inputs:

- plankton_track
  - The list of plankton whose track lengths will be summed.

Outputs:

- track_durations
  - An array of the number of plankton for different track lengths.

2.3.24 get_found_plankton_at_timestep

Creates an array where each row corresponds to a frame. Each row will be filled with a number corresponding to the number of plankton found in that frame.
Input:

- plankton_track
  - The list of plankton whose track lengths will be summed.

Outputs:

- found_plankton_at_timestep
  - An array of the number of plankton found each frame.

2.3.25 extract_positions_from_list

Creates an array of the positions of the plankton where each row corresponds to a time step.
Input:

- plankton_track
  - The list of plankton.

Outputs:

- positions_array
  - An array with the positions of the plankton in all frames.

2.4 Notebook Segmentation frame by frame

Segmenting images frame by frame

June 16, 2021

1 Example: Segmenting images frame by frame

1.0.1 Explanations for all used functions can be found in the paper

First we need to load all the necessary functions.

[1]:
```
from loader import *
from models import *
from utils import *
from plotting import *
```

The next step is to create a sample of plankton. First we need to decide what type of plankton we want, here we want two different types of spherical plankton. Next we decide the size of the plankton and the size of the image they should be simulated on (in actuality we define the borders of the area the plankton can be initialized in), this is done through the parameters im_size_width, im_size_height, radius1 and radius2. To make the network treat them as two different types of plankton and segment them to different layers of the output later on we also need to assign them labels. One can also make them part of the background if the label is set to -1.

[2]:
```
im_size_width, im_size_height, radius1, radius2 = 256, 256, 0.17e-6, 0.3e-6
plankton1 = stationary_spherical_plankton(im_size_height, im_size_width,
                                          radius1, label=0)
plankton2 = stationary_spherical_plankton(im_size_height, im_size_width,
                                          radius2, label=1)
```

Then we create the microscope through which the plankton will be sampled. For this we use a brightfield microscope. To create it we need to define the size of the image to be simulated, this is done with im_size_height and im_size_width from the previous step. It is not necessary for the simulated image to be of the same size as the images to be predicted on as will be explained when the network is built. We can also apply a lighting gradient with the parameter gradient_amp, if it is set to 0 there will be no gradient.

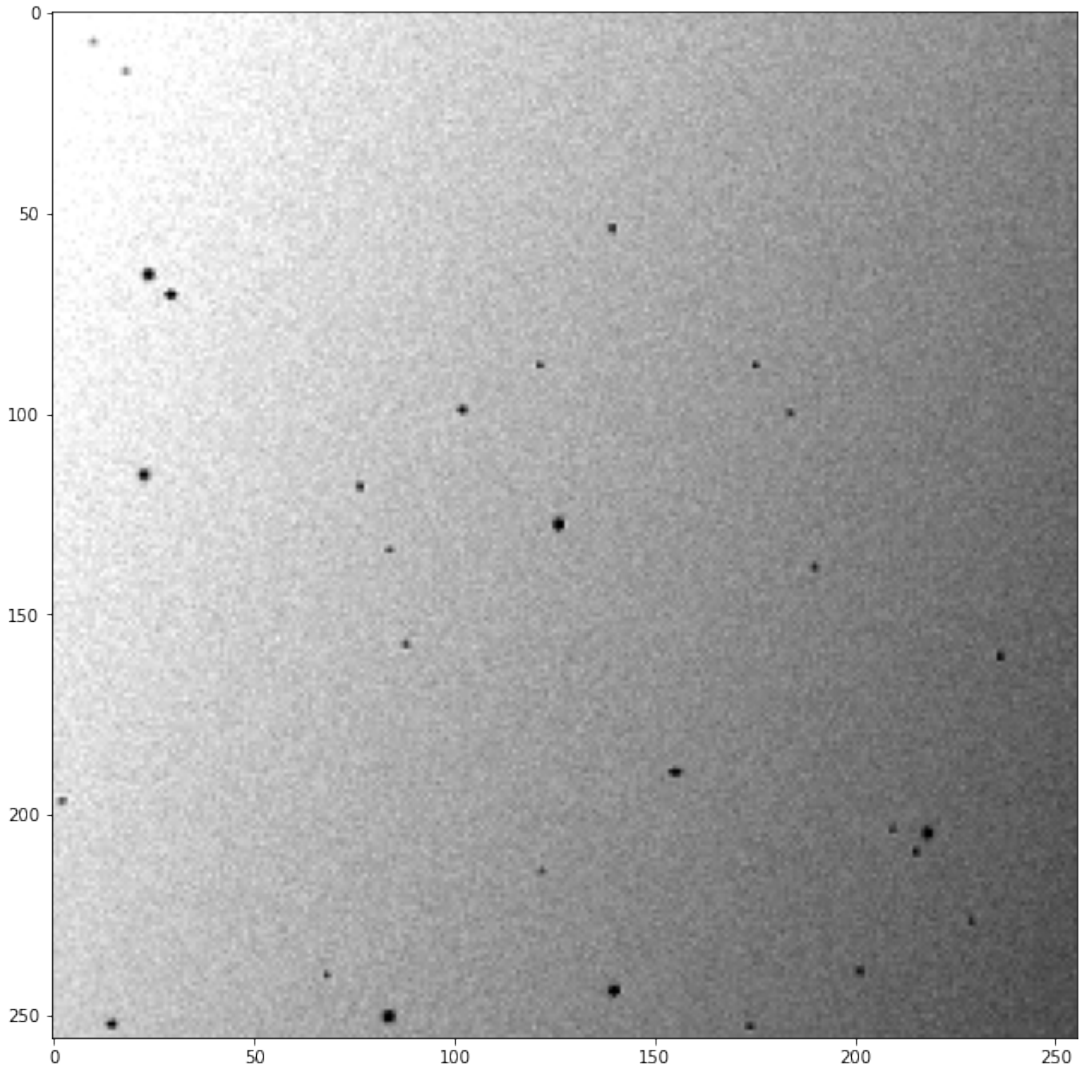
[3]:
```
gradient_amp = 1
microscope = plankton_brightfield(im_size_height, im_size_width, gradient_amp)
```

Then we can create a sample. If we want the number of plankton to vary on each simulated image we need to create a function that when called generates a random number, we can do this through the use of lambda functions as seen below. The plankton are then raised to the power of these functions to generate many plankton, plankton**4 would create 4 instances of plankton. To create the sample we simply add them together.

[4]:
```
no_of_plankton1 = lambda: np.random.randint(10, 20)
no_of_plankton2 = lambda: np.random.randint(10, 20)

sample = plankton1**no_of_plankton1 + plankton2**no_of_plankton2
```

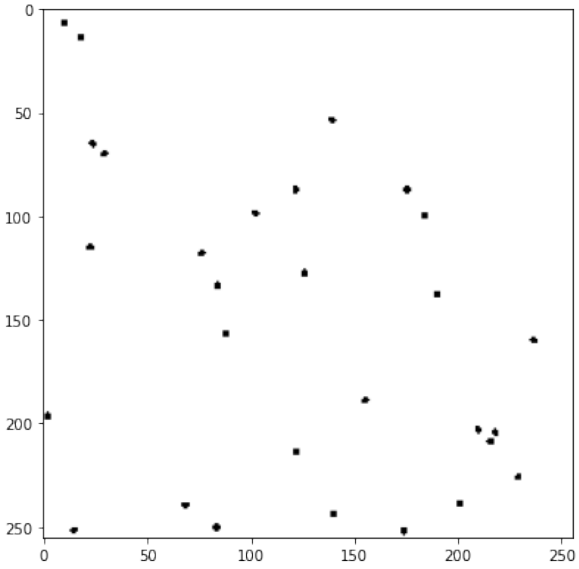
Finally we create the image. We can add noise to the image through the parameter noise_amp where a value of 0 means no noise. With the next two parameters, norm_min and norm_max, we can saturate the image to contain both completely white and completely black pixels. To make the image contain very bright pixels we set norm_max to a value larger than 1, and norm_min to less than 0 for very dark pixels. After that we plot the image. For this cell to generate a new image each time we also use image.update().

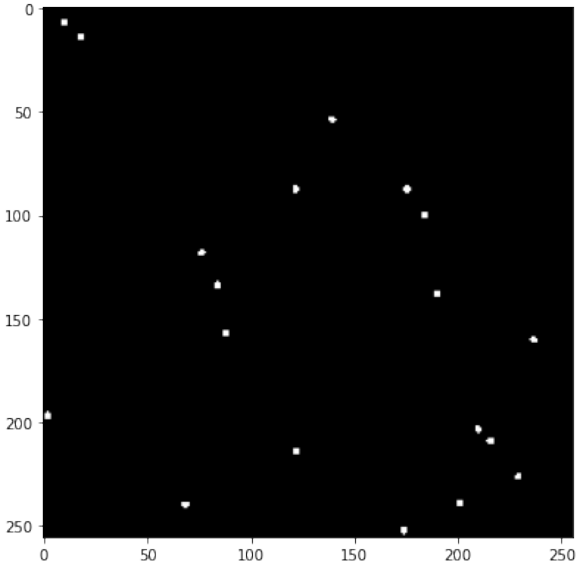
[5]:
```
noise_amp, norm_min, norm_max = 2, -0.2, 1.2
image = create_image(noise_amp, sample, microscope, norm_min, norm_max)

image.update()
plot_image(image)
```

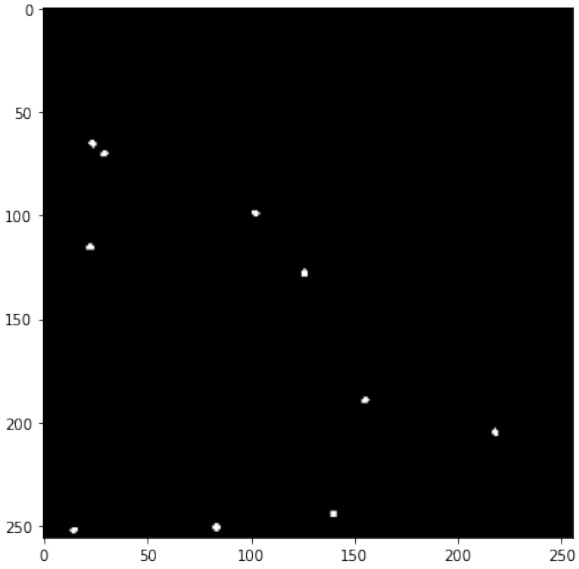


To create a training set, we also need labels for our images. For a simulated image (not a sequence) we use the function get_target_image() to create these. If the image has different types of plankton, these will be segmented to different layers as seen below where we plot the output of the label_function with plot_label. This label_function will later be used in the training.

[6]:
```
label_function = get_target_image
plot_label(label_function, image)
```

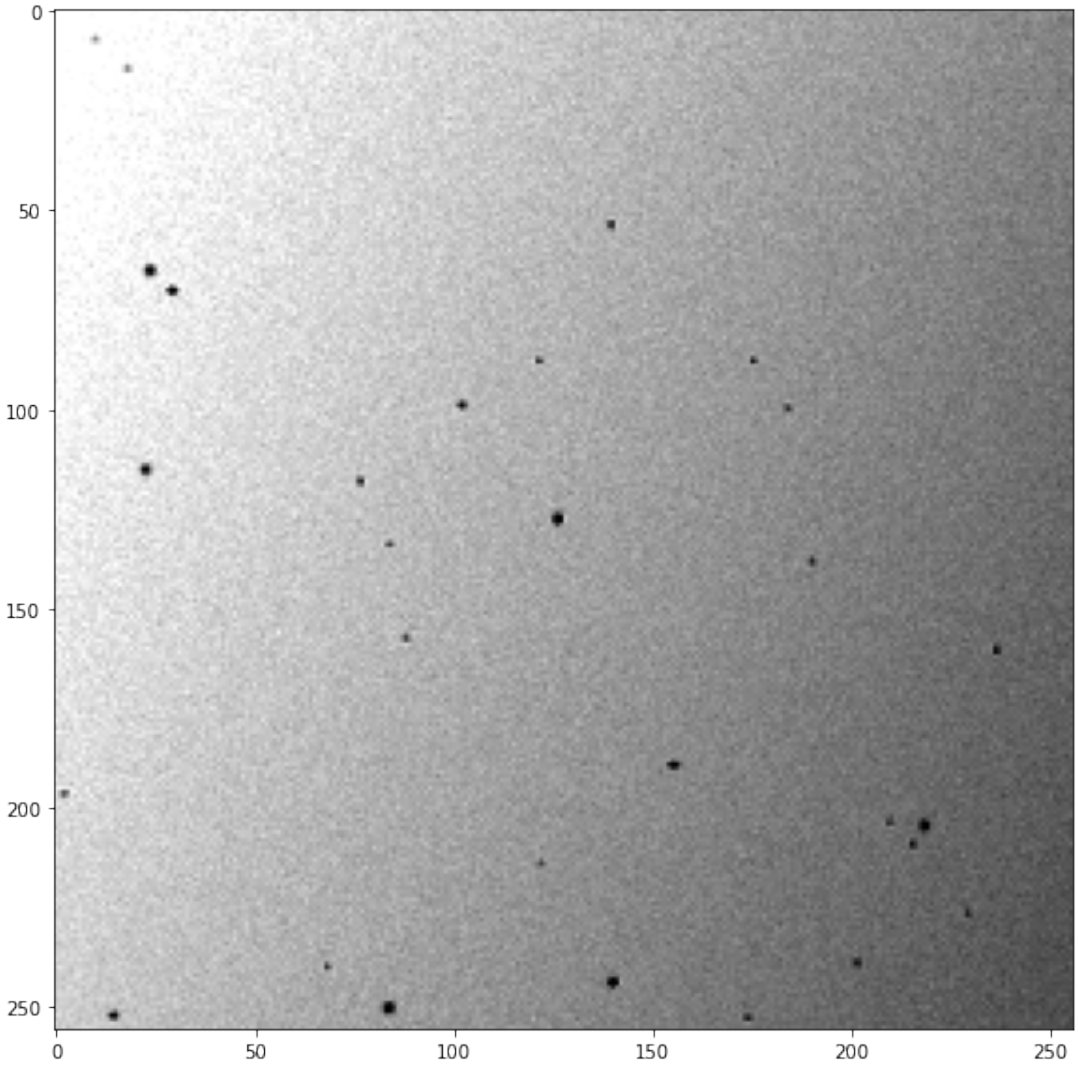

0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250

The next step is to create a batch function. The function takes the simulated image(s) and processes them before sending them to the network for training. The first input is the image, the second one is what the output of the batch function should looke like, in our case just image 0. In the third input, function_img, we can send a list of functions that will be applied to the image. To view the output of our batch function one can use plot_batch.

[7]:
```
batch_function = create_custom_batch_function(image,
                                              outputs=[0],
                                              function_img=[normalize_image])

plot_batch(image, batch_function)
```



Now we can create a generator that creates the training data from our simulated image, our label function and our batch function. The continuous generator generates a data set of the defined minimum size before the training starts. It continues to add data to the data set during training until the data size exceeds defined maximum size, then the new data replaces the old data. We also need to define the batch size.

Note that min_data_size needs to be bigger than or equal to the batch_size multiplied with the number of steps_per_epoch defined in the next step.

[8]:
```
from deeptrack.generators import ContinuousGenerator
generator = ContinuousGenerator(
    image,
    get_target_image,
    batch_function,
    batch_size=8,
    min_data_size=128,
    max_data_size=512)
```

The last step before starting the training is to define the model. The two Nones are width and height of the images, they don't need to be defined and are better left as Nones if one intends to use the model on differently sized images or images with a different size than the images in the training data. However no_of_inputs and no_of_outputs need to defined, they correspond to the number of images in the input to the network (same as the number of images in the output of the batch_function) and the output of the network (same as the number of images in the output of the label_function).

The training starts by running the function train_model_early_stopping which stops the training early if the performance of the network doesn't improve for a number of epochs. The inputs are the model, the generator and parameters of the training. The patience parameter defines the number of epochs after which the training will stop if no improvements are made. The parameter epochs sets the maximum number of epochs the network will be trained for and steps_per_epoch sets how many batches one epoch will consist of.

[9]:
```
no_of_inputs, no_of_outputs = 1, 3
model = generate_unet(None, None, no_of_inputs, no_of_outputs)
model = train_model_early_stopping(model, generator, patience=10,
                                   epochs=100, steps_per_epoch=10)
```

```
Generating 131 / 128 samples before starting training
Epoch 1/100
10/10 [==============================] - 1s 64ms/step - loss: 0.0028
Epoch 2/100
10/10 [==============================] - 1s 64ms/step - loss: 0.0027
Epoch 3/100
10/10 [==============================] - 1s 69ms/step - loss: 0.0025
Epoch 4/100
10/10 [==============================] - 1s 67ms/step - loss: 0.0021
Epoch 5/100
10/10 [==============================] - 1s 63ms/step - loss: 0.0015
Epoch 6/100
```

```
10/10 [==============================] - 1s 63ms/step - loss: 1.5169e-04
Epoch 55/100
10/10 [==============================] - 1s 64ms/step - loss: 1.5667e-04
Epoch 56/100
10/10 [==============================] - 1s 64ms/step - loss: 1.6638e-04
Epoch 57/100
10/10 [==============================] - 1s 64ms/step - loss: 1.4138e-04
Epoch 58/100
10/10 [==============================] - 1s 63ms/step - loss: 1.4243e-04
Epoch 59/100
10/10 [==============================] - 1s 63ms/step - loss: 1.7213e-04
Epoch 60/100
10/10 [==============================] - 1s 66ms/step - loss: 1.4958e-04
```

When the model is trained one might want to save it to avoid going through this process unnecessarily, this is done with model.save(save_path). The save_path defines the path to where the network will be saved and with what name.

To load it one uses keras.models.load_model(load_path) with the entire path to the model as input. In our case we use a custom defined loss function called softmax_categorical which needs to be added as an input to the function as seen.

[10]:
```
# save_path = 'E:\\models\\frame-by-frame.keras'
# model.save(save_path)
```

[11]:
```
# load_path = 'E:\\models\\frame-by-frame.keras'
# model = keras.models.load_model(load_path,␣
 ↪custom_objects={'softmax_categorical':softmax_categorical})
```

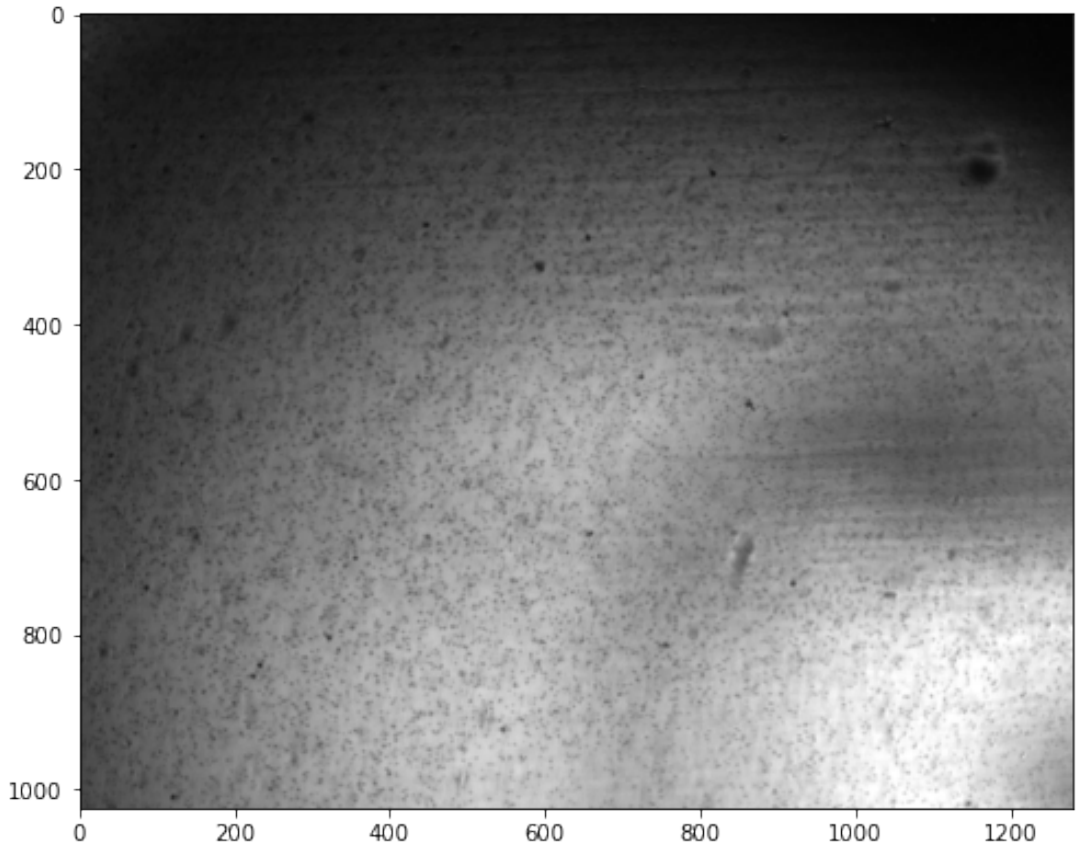
Now to evaluate our model on a frame from our plankton video. To do this we need to feed the images to the network in the way they were fed to it during the training, for this we use the function im_stack. The outputs parameter is set to be identical to the outputs of the batch_function, the path to the folder of the frames is also necessary. The image to be analyzed is assigned through the parameter frame_im0, the image shown will in our case be the 16th image in the folder in alphabetical order.The size of the images when fed to the network is defined in im_size_width and im_size_height. If the images to be analyzed need to be rescaled to a certain size these values are specified in im_resize_width and im_resize_height, if not these values are the dimensions of the image. Usually these two size pairs are equal to eachother. They differ from eachother if one wants to crop the full size image, then im_resize_width and im_resize_height will be the same as the dimensions of the full sized image.

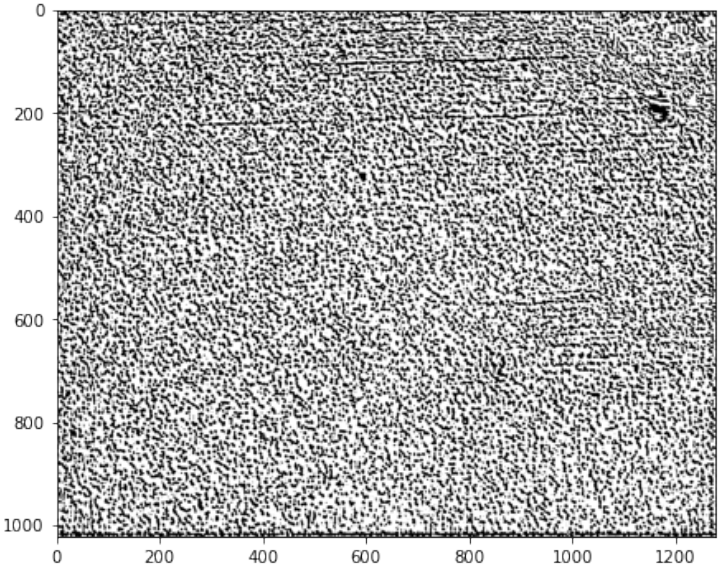
We also have the option to treat the image with a set of functions, these are specified in the function_img the same way as in create_custom_batch_function.

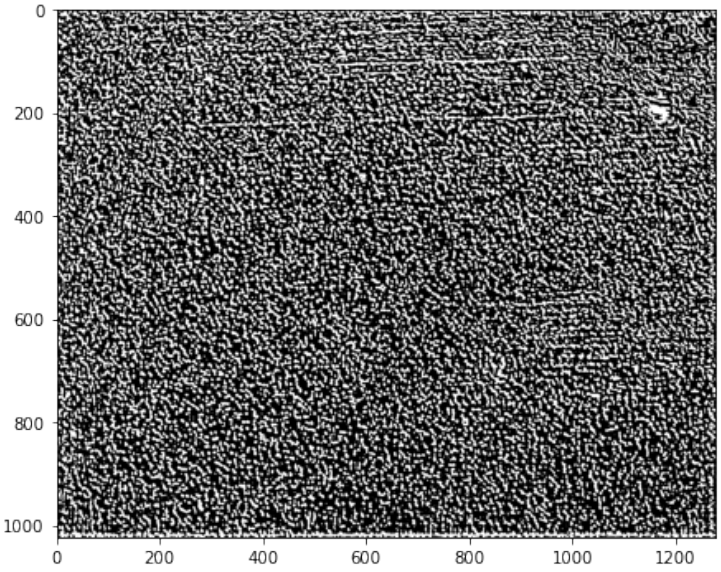
When the image is properly prepared we can plot it using plot_image_stack() and see the model's prediction on it using plot_prediction().

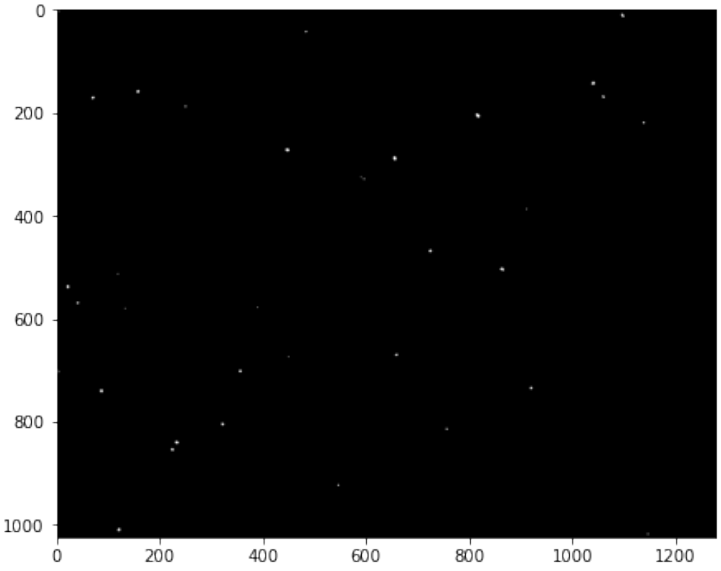
[12]:
```
folder_path = 'E:\\Documents\\Anaconda\\Jupyterkod\\Exjobb\\Egen␣
 ↪kod\\Exjobb\\From erik\\raw output'
```

```
im_stack = get_image_stack(
    outputs=[0],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1280,
    im_size_height=1024,
    im_resize_width=1280,
    im_resize_height=1024,
    function_img=[]
)

plot_image_stack(im_stack)
plot_prediction(model=model, im_stack=im_stack)
```




0
200
400
600
800
1000
0
200
400
600
800
1000
1200

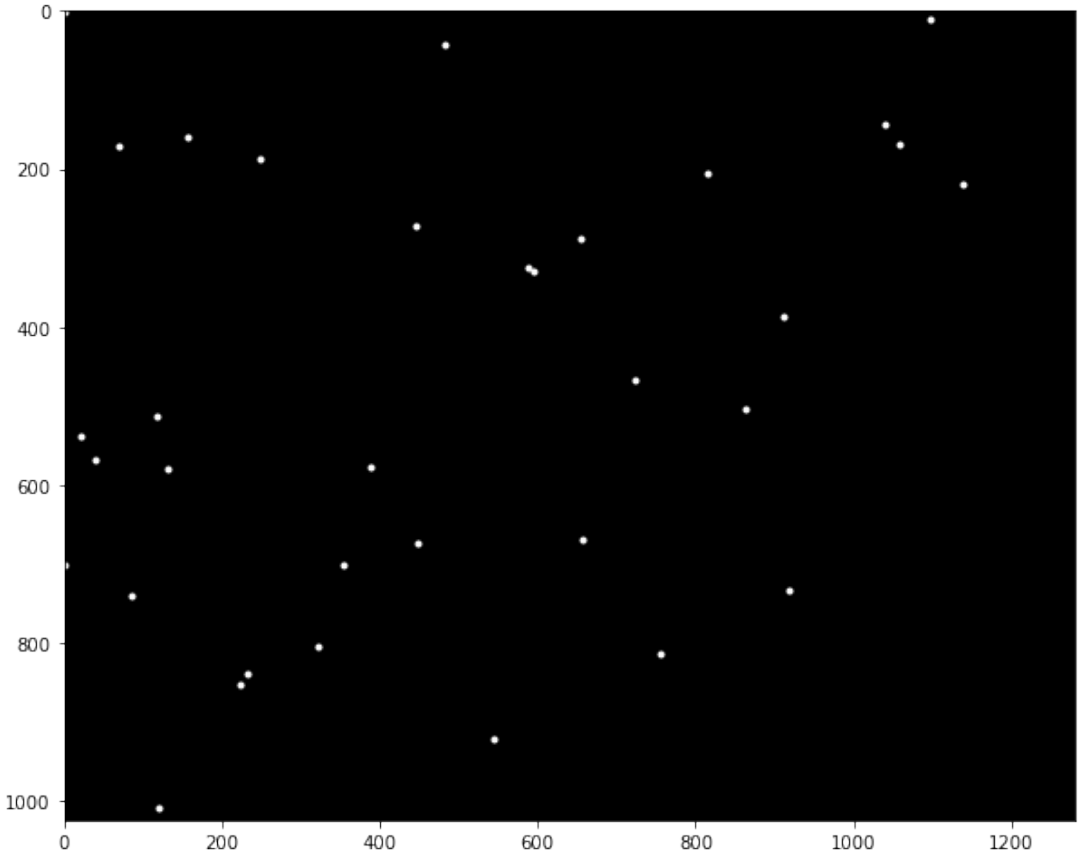

0
200
400
600
800
1000
0
200
400
600
800
1000
1200


0
200
400
600
800
1000
0
200
400
600
800
1000
1200

When the prediction of the network is satisfactory we can extract the positions from the predictions using extract_positions. The function shares its inputs with im_stack with the addition of no_of_frames, model, layer, value_threshold and prediction_size. no_of_frames decides the number of frames to be analyzed and model is the trained network. layer is the segmentation layer from which we will extract the positions, in this example the large particles are on layer 2. value_threshold is the minimum value a pixel can have for it to be considered a plankton pixel. prediction_size is the maximum number of pixels a blob can have to be considered background.

To see which predictions were extracted we can use plot_found_positions. Using this function we can optimize the parameters value_threshold and predictions_size to filter out noise.

[13]:
```
positions = extract_positions(
    no_of_frames=5,
    outputs=[0],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1280,
    im_size_height=1024,
    im_resize_width=1280,
    im_resize_height=1024,
    model=model,
    layer=2,
    value_threshold=0.9,
    prediction_size=0,
    function_img=[])

plot_found_positions(positions, width=1280, height=1024)
```



When we have our list of positions it's time to create traces with them, to do this we use the function assign_positions_to_planktons. The inputs are the found positions, max_dist, threshold and a bolean for extrapolation. max_dist is a maximum search radius from its latest position within which the algorithm will assign a found position to a plankton. If more than one is found the algorithm will chose the position that maintains the plankton's mean velocity. The parameter time_threshold sets the maximum number of time steps back in time the algorithm will search for a position if there are recent time steps where no positions were found for it. If extrapolate is set to True the algorithm will extrapolate a new position based on the previous two and look for new positions around that one. The output is a list of plankton where each plankton contains an array of positions.

If a plankton has gaps in its list of positions the function interpolate_gaps_in_plankton_positions will fill these if they only last for one timestep.The input and output is the list of plankton

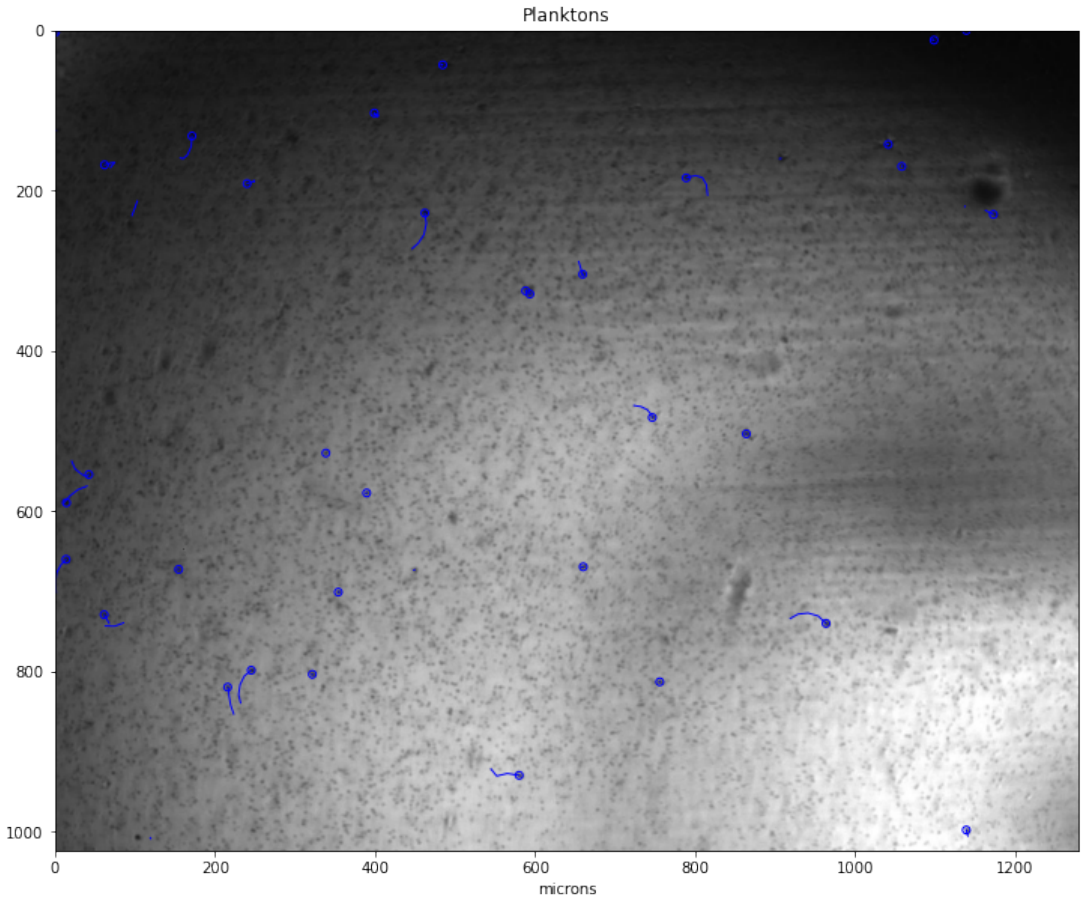
If there are plankton that are stationary or move too slow they can be filtered using the function trim_list_from_stationary_planktons. The inputs are the list of plankton and a minimum total distance a plankton can travel throughout it's found positions.

To further trim the list of plankton we can divide it into two lists of plankton depending on what fraction of the frames they are present in with the function split_plankton. The inputs are percentage_threshold which is this fraction, and the list of plankton.

[14]:
```
list_of_plankton = assign_positions_to_planktons(
    positions, max_dist=15, time_threshold=10, extrapolate=True)

list_of_plankton = interpolate_gaps_in_plankton_positions(
    list_of_plankton=list_of_plankton)

list_of_plankton = trim_list_from_stationary_planktons(
    list_of_plankton=list_of_plankton, min_distance=0)

plankton_track, plankton_dont_track = split_plankton(
    percentage_threshold=0.1, list_of_plankton=list_of_plankton)
```

Finally we can plot the found positions onto the images that was analyzed, this is done using the function plot_and_save_track. The inputs are can be found in the paper. im_size_width and im_size_height here are used to rescale the found positions to the dimensions of the video. In this case we reshape the video to the dimensions 1024x1024.

[15]:
```
plot_and_save_track(no_of_frames=5,
                    plankton_track=list_of_plankton,
                    plankton_dont_track=plankton_dont_track,
                    folder_path=folder_path,
                    frame_im0=16,
                    save_images=0,
                    show_plankton_track=True,
                    show_plankton_dont_track=False,
                    show_numbers_track=0,
                    show_numbers_dont_track=0,
                    show_numbers_specific_plankton=False,
                    show_specific_plankton=False,
                    specific_plankton=None,
                    color_plankton_track='b',
                    color_plankton_dont_track='r',
                    color_specific_plankton='w',
                    im_size_width=1280,
                    im_size_height=1024,
                    x_axis_label='microns',
                    y_axis_label='microns',
                    save_path='E:\\Raw_output\\frame-by-frame',
                    frame_name='track',
                    file_type='.jpg')
```



From these positions we can extract the mean net/gross distance and plot them using the function get_mean_net_and_gross_distance and plot_net_vs_gross_distance. Both takes a list of plankton as inputs and we can compensate for the lack of a 3rd direction by setting use_3D_dist=True.

[16]:
```
# mean_net_distance, mean_gross_distances = get_mean_net_and_gross_distance(
#     list_of_plankton, use_3D_dist=False)

# plot_net_vs_gross_distance(list_of_plankton=plankton_track, use_3D_dist=True)
```

We can save the positions contained in a list of plankton using save_positions as either .csv or .xlsx.

[17]:
```
# save_positions(list_of_plankton,
#                save_path='E:\\track plankton',
#                file_format='.csv',
#                pixel_length_ratio=1)
```

We can also create a video with the saved images using Make_video.

[18]:
```
# make_video(frame_im0=0,
#            folder_path='E:\\Raw_output\\frame-by-frame',
#            save_path='E:\\Raw_output\\frame-by-frame.avi',
#            fps=7,
#            no_of_frames=50)
```

2.5 Notebook Cropping and removing running mean

Cropping and removing running mean

June 16, 2021

1 Example: Cropping and removing running mean

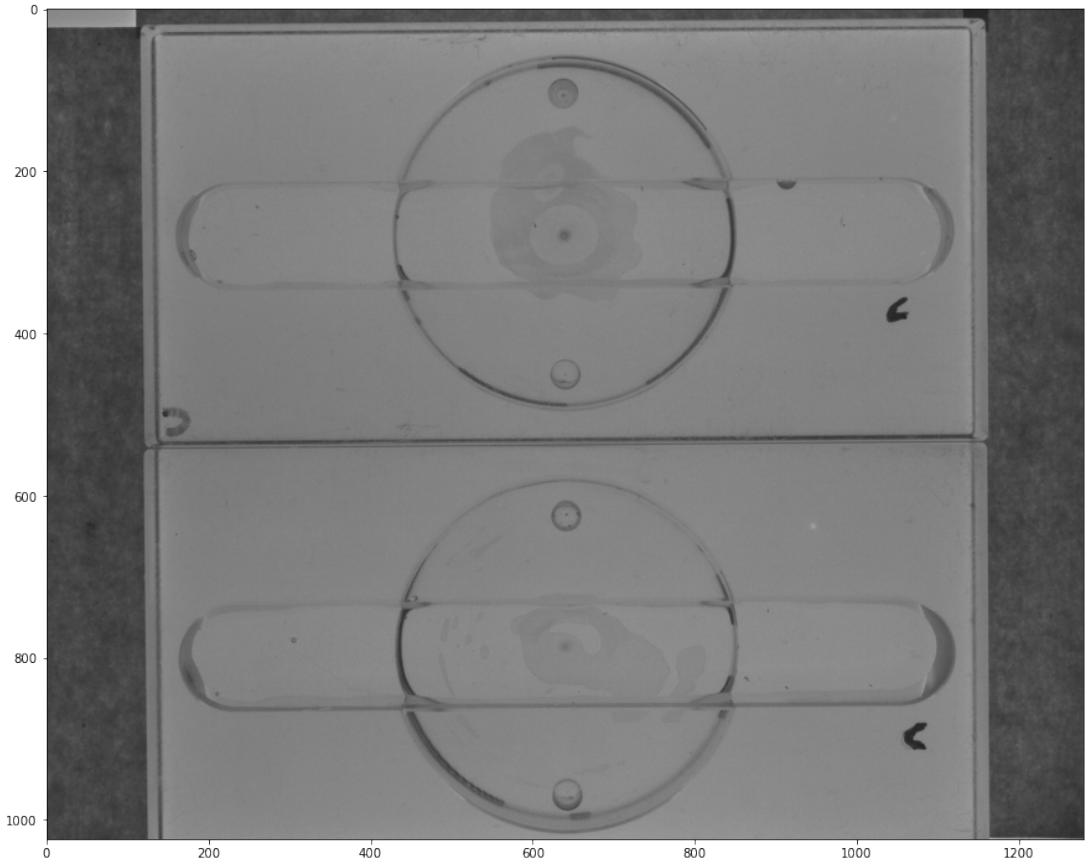
1.0.1 Explanations for all used functions can be found in the paper

First we need to load all the necessary functions.

[1]:
```
from loader import *
from models import *
from utils import *
from plotting import *
```

One can use load_and_plot_folder_image() to load and view an image from a folder. In this case we use image 10 in the folder sorted alphabetically.

[2]:
```
folder_path = 'E:\\Documents\\Anaconda\\Jupyterkod\\Exjobb\\Egen␣
 ↪kod\\Exjobb\\From erik\\Transfer'
image = load_and_plot_folder_image(folder_path, 10)
```

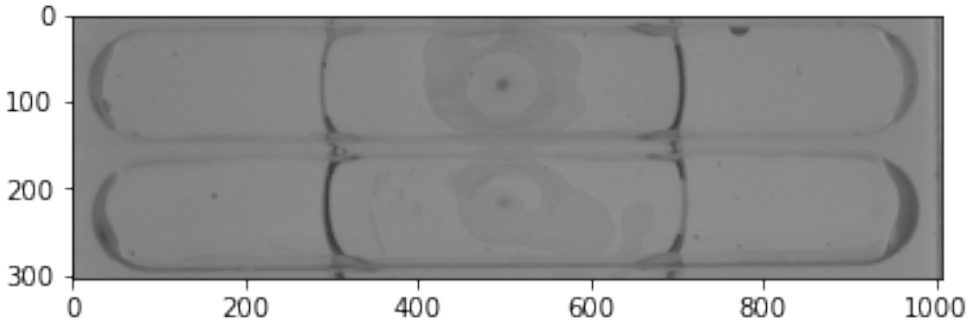


In some cases the area of interest is just a small part of an image. In the image above the plankton are contained in the horisontal tubes thus making them the areas of interest. In those cases we can crop away the unnecessary parts using crop_and_append_image. The function can also print the number of rows (height) and number of columns (width) of the image for later use. Since we will use a U-Net, the function crops to dimensions closest to a multiple of 16. In the x-direction the pixels deleted are those in the intervals [0, 140] and [1150, 1280].

[3]:
```
col_delete_list=[0, 140, 1150, 1280]
row_delete_list=[0, 200, 350, 720, 880, 1024]

img = crop_and_append_image(image=image,
                            col_delete_list=col_delete_list,
                            row_delete_list=row_delete_list,
                            print_shape=True)
plt.imshow(img, cmap='gray')
```

```
(304, 1008)
```

[3]:
```
<matplotlib.image.AxesImage at 0x1ed79bff208>
```



The next step is to create a sample of plankton. First we need to decide what type of plankton we want, here we want one typ of elliptical plankton. Next we decide the size of the plankton and the size of the image they should be simulated on (in actuality we define the borders of the area the plankton can be initialized in), this is done through the parameters im_size_width, im_size_height and radius. The label is set to 0 to make the network look for them. The label -1 makes a particle part of the background. The radius here is the dimensions of the principal axes of the ellipsoid.

[4]:
```
im_size_width, im_size_height, radius = 256, 256, (1.5e-7, 9e-7, 1.5e-7)
plankton = stationary_ellipsoid_plankton(
    im_size_height, im_size_width, radius, label=0)
```

Then we create the microscope through which the plankton will be sampled. For this we use a brightfield microscope. To create it we need to define the size of the image to be simulated, this is done with im_size_height and im_size_width from the previous step. It is not necessary for the simulated image to be of the same size as the images to be predicted on as will be explained when the network is built. We can also apply a lighting gradient with the parameter gradient_amp, if it is set to 0 there will be no gradient.

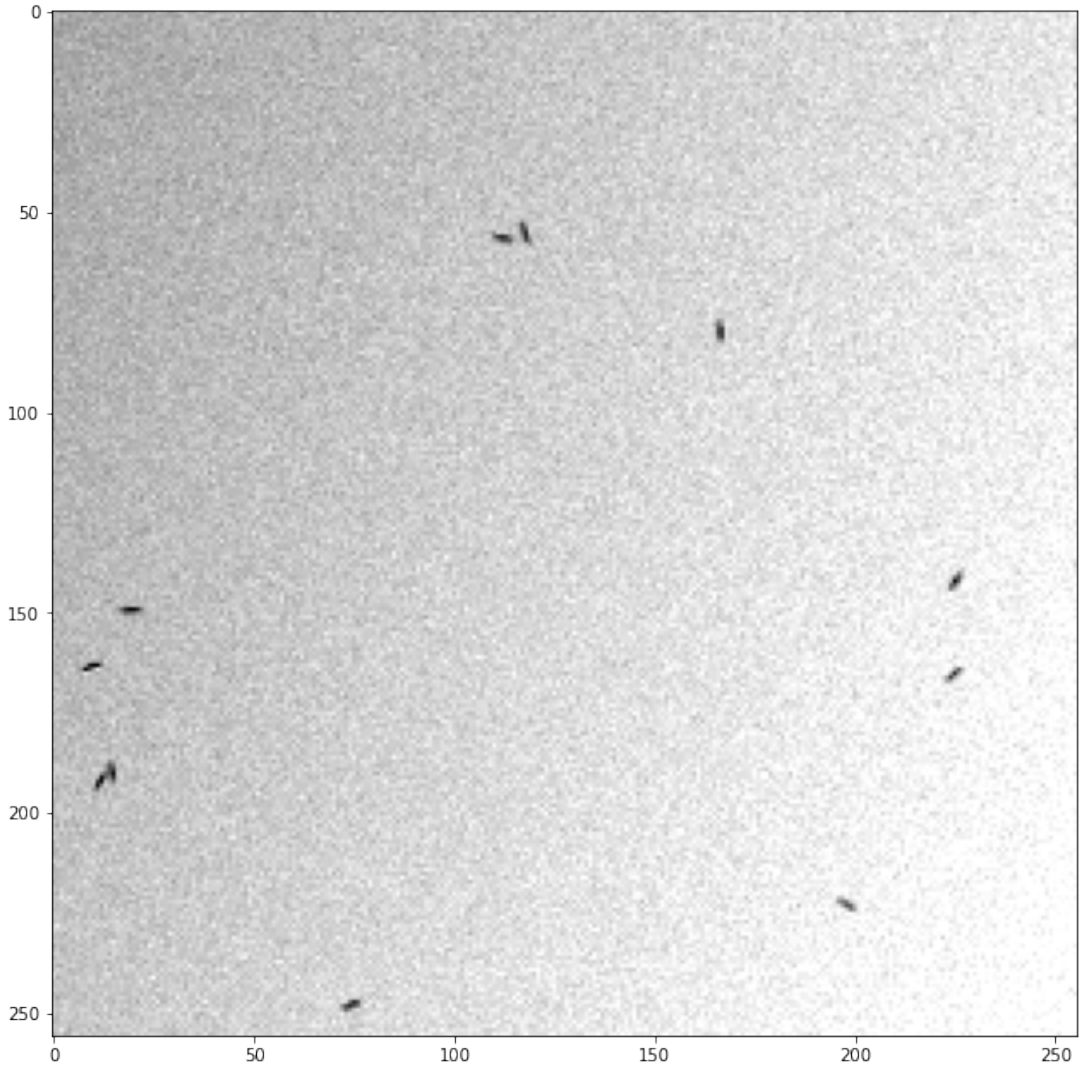
[5]:
```
gradient_amp = 0.5
microscope = plankton_brightfield(im_size_height, im_size_width, gradient_amp)
```

Then we can create a sample. If we want the number of plankton to vary on each simulated image we need to create a function that when called generates a random number, we can do this through the use of lambda functions as seen below. The plankton are then raised to the power of these functions to generate many plankton, plankton**4 would create 4 instances of plankton.

[6]:
```
no_of_planktons = lambda: np.random.randint(10, 20)
sample = plankton**no_of_planktons
```

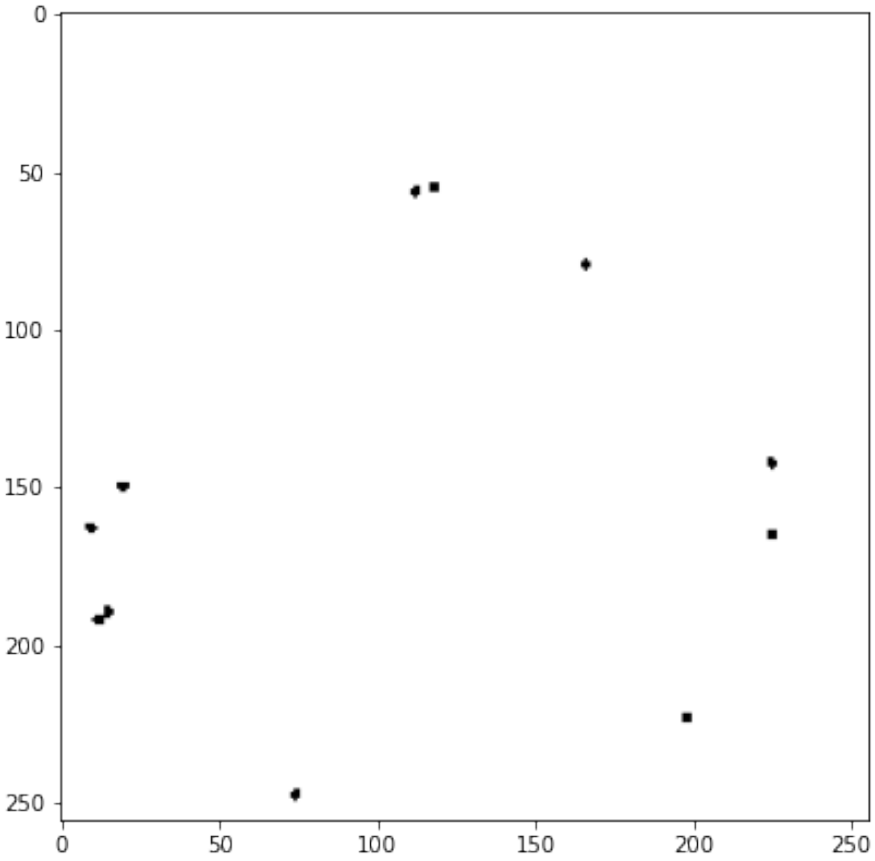
Finally we create the image. We can add noise to the image through the parameter noise_amp where a value of 0 means no noise. With the next two parameters, norm_min and norm_max, we can saturate the image to contain both completely white and completely black pixels. To make the image contain very bright pixels we set norm_max to a value larger than 1, and norm_min to less than 0 for very dark pixels. After that we plot the image. For this cell to generate a new image each time we also use image.update().

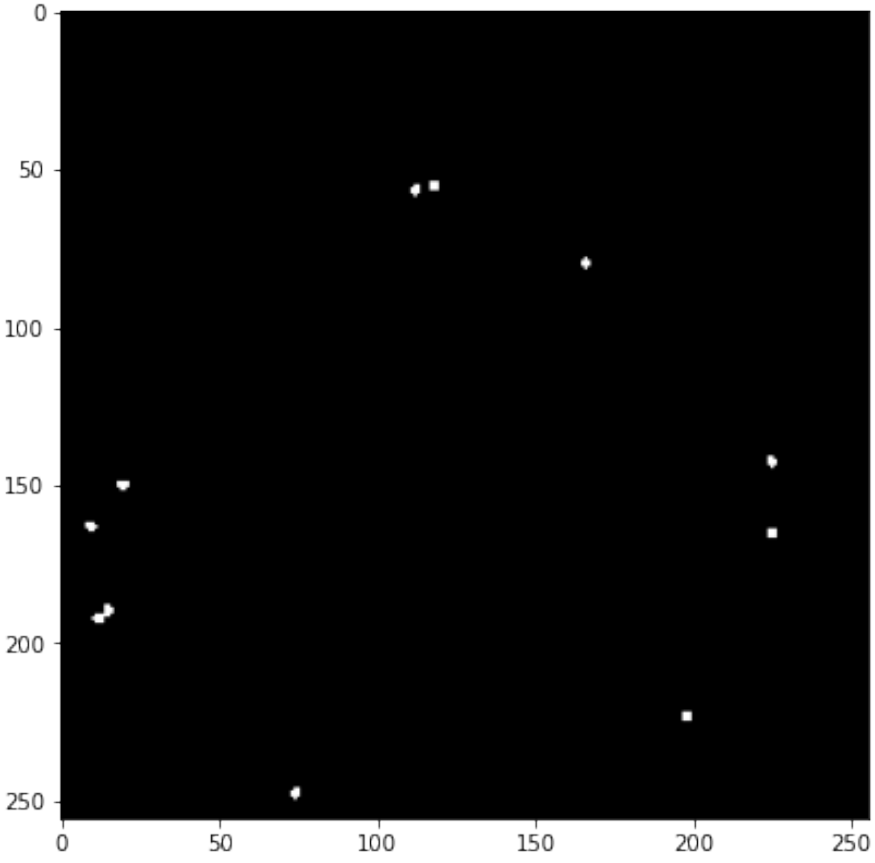
[7]:
```
noise_amp = 2
norm_min, norm_max = -0.2, 1.2
image = create_image(noise_amp, sample, microscope, norm_min, norm_max)
image.update()
plot_image(image)
```



To create a training set, we also need labels for our images. For a simulated image (not a sequence) we use the function get_target_image() to create these. We can plot the output of the label_function with plot_label. This label_function will later be used in the training.

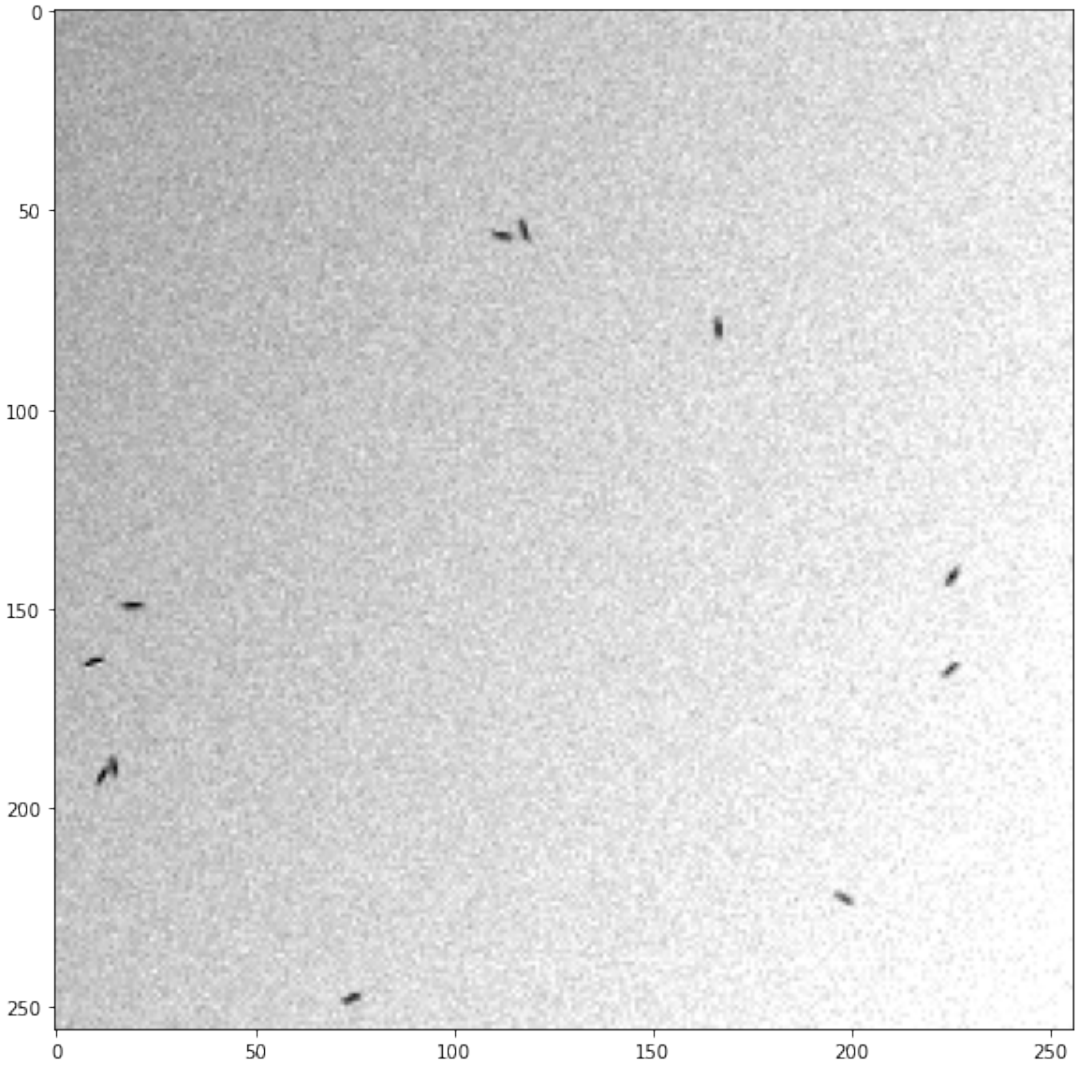
[8]:
```
label_function = get_target_image
plot_label(label_function, image)
```


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250

The next step is to create a batch function. The function takes the simulated image(s) and processes them before sending them to the network for training. The first input is the image, the second one is what the output of the batch function should looke like, in our case just image 0. In the function_img we can send a list of functions that will be applied to the image. To view the output of our batch function one can use plot_batch.

[9]:
```
batch_function = create_custom_batch_function(image,
                                              outputs=[0],
                                              function_img=[normalize_image])

plot_batch(image, batch_function)
```



Now we can create a generator that creates the training data from our simulated image, our label function and our batch function. The continuous generator generates a data set of the defined minimum size before the training starts. It continues to add data to the data set during training until the data size exceeds defined maximum size, then the new data replaces the old data. We also need to define the batch size.

Note that min_data_size needs to be bigger than or equal to the batch_size multiplied with the number of steps_per_epoch defined in the next step.

[10]:
```
from deeptrack.generators import ContinuousGenerator
generator = ContinuousGenerator(
    image,
    get_target_image,
    batch_function,
    batch_size=8,
    min_data_size=128,
    max_data_size=512
    )
```

The last step before starting the training is to define the model. The two Nones are width and height of the images, they don't need to be defined and are better left as Nones if one intends to use the model on differently sized images or images with a different size than the images in the training data. However no_of_inputs and no_of_outputs need to defined, they correspond to the number of images in the input to the network (same as the number of images in the output of the batch_function) and the output of the network (same as the number of images in the output of the label_function).

The training starts by running the function train_model_early_stopping which stops the training early if the performance of the network doesn't improve for a number of epochs. The inputs are the model, the generator and parameters of the training. The patience parameter defines the number of epochs after which the training will stop if no improvements are made. The parameter epochs sets the maximum number of epochs the network will be trained for and steps_per_epoch sets how many batches one epoch will consist of.

[11]:
```
no_of_inputs, no_of_outputs = 1, 2
model = generate_unet(None, None, no_of_inputs, no_of_outputs)
model = train_model_early_stopping(model, generator, patience=10, epochs=100,␣
 ↪steps_per_epoch=10)
```

```
Generating 135 / 128 samples before starting training
Epoch 1/100
10/10 [==============================] - 1s 65ms/step - loss: 0.0013
Epoch 2/100
10/10 [==============================] - 1s 64ms/step - loss: 0.0011
Epoch 3/100
10/10 [==============================] - 1s 64ms/step - loss: 6.6750e-04
Epoch 4/100
10/10 [==============================] - 1s 65ms/step - loss: 2.1200e-04
Epoch 5/100
10/10 [==============================] - 1s 65ms/step - loss: 7.4073e-05
```

```
Epoch 30/100
10/10 [==============================] - 1s 65ms/step - loss: 2.0512e-05
Epoch 31/100
10/10 [==============================] - 1s 62ms/step - loss: 2.2334e-05
Epoch 32/100
10/10 [==============================] - 1s 64ms/step - loss: 2.0708e-05
Epoch 33/100
10/10 [==============================] - 1s 65ms/step - loss: 2.0266e-05
Epoch 34/100
10/10 [==============================] - 1s 63ms/step - loss: 2.3057e-05
Epoch 35/100
10/10 [==============================] - 1s 63ms/step - loss: 2.2174e-05
Epoch 36/100
10/10 [==============================] - 1s 64ms/step - loss: 2.0682e-05
Epoch 37/100
10/10 [==============================] - 1s 64ms/step - loss: 2.0653e-05
Epoch 38/100
10/10 [==============================] - 1s 63ms/step - loss: 2.2847e-05
Epoch 39/100
10/10 [==============================] - 1s 63ms/step - loss: 2.0056e-05
Epoch 40/100
10/10 [==============================] - 1s 65ms/step - loss: 1.9533e-05
Epoch 41/100
10/10 [==============================] - 1s 64ms/step - loss: 1.9932e-05
Epoch 42/100
10/10 [==============================] - 1s 66ms/step - loss: 1.8520e-05
Epoch 43/100
10/10 [==============================] - 1s 64ms/step - loss: 2.2399e-05
Epoch 44/100
10/10 [==============================] - 1s 65ms/step - loss: 2.2892e-05
Epoch 45/100
10/10 [==============================] - 1s 66ms/step - loss: 2.1756e-05
Epoch 46/100
10/10 [==============================] - 1s 62ms/step - loss: 2.4047e-05
Epoch 47/100
10/10 [==============================] - 1s 64ms/step - loss: 1.9753e-05
Epoch 48/100
10/10 [==============================] - 1s 65ms/step - loss: 1.9657e-05
Epoch 49/100
10/10 [==============================] - 1s 63ms/step - loss: 2.0559e-05
Epoch 50/100
10/10 [==============================] - 1s 64ms/step - loss: 2.1009e-05
Epoch 51/100
10/10 [==============================] - 1s 63ms/step - loss: 2.4708e-05
Epoch 52/100
10/10 [==============================] - 1s 64ms/step - loss: 2.1062e-05
```

When the model is trained one might want to save it to avoid going through this process unneces-

sarily, this is done with model.save(save path). The save_path_model defines the path to where the network will be saved and with what name.

To load it one uses keras.models.load_model(load path) with the entire path to the model as input. In our case we use a custom defined loss function called softmax_categorical which needs to be added as an input to the function as seen.

[12]:
```
# save_path_model = 'E:\\models\\one_frame_all_sizes.keras'
# model.save(save_path_model)
```

[13]:
```
# load_path_model = 'E:\\one_frame_all_sizes.keras'
# model = keras.models.load_model(load_path_model,␣
 ↪custom_objects={'softmax_categorical':softmax_categorical})
```

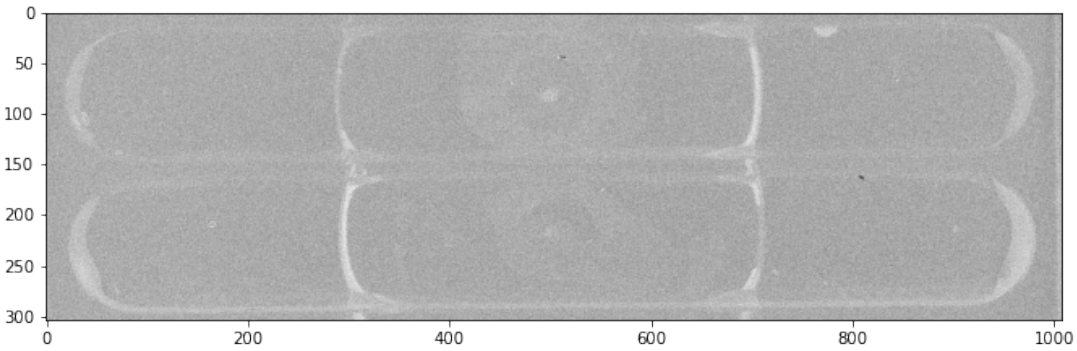
Now to evaluate our model on a frame from our plankton video. To do this we need to feed the images to the network in the way they were fed to it during the training, for this we use the function im_stack. The outputs parameter is set to be identical to the outputs of the batch_function, the path to the folder of the frames is also necessary. The image to be analyzed is assigned through the parameter frame_im0, the image shown will in our case be the 16th image in the folder in alphabetical order. The size of the images when fed to the network is defined in im_size_width and im_size_height. If the images to be analyzed need to be rescaled to a certain size these values are specified in im_resize_width and im_resize_height, if not these values are the dimensions of the image. Usually these two size pairs are equal to eachother. They differ from eachother if one wants to crop the full size image, then im_resize_width and im_resize_height will be the same as the dimensions of the full sized image.

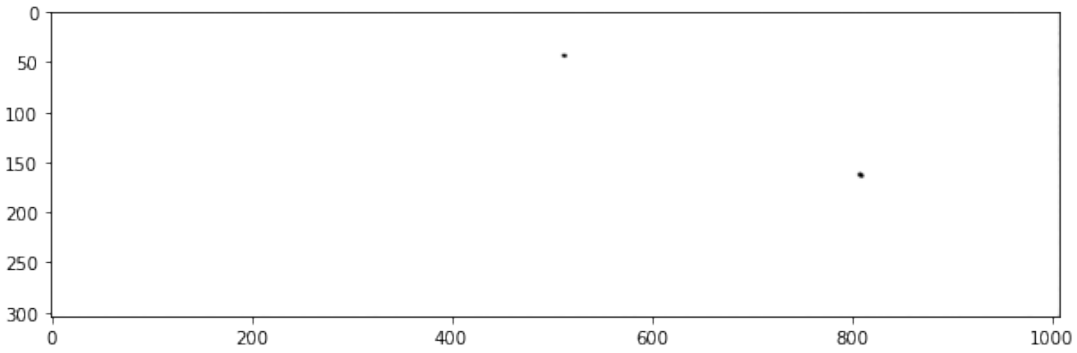
We also have the option to treat the image with a set of functions, these are specified in the function_img the same way as in create_custom_batch_function. Since we know which part of the image we are interested in we can crop it to only analyze the part on interest. We also have a very messy background that can cause misclassification, to remove it we can add the function remove_running_mean before we crop the image. The parameter tot_no_of_frames sets the number of images to be used when calculating the running mean.

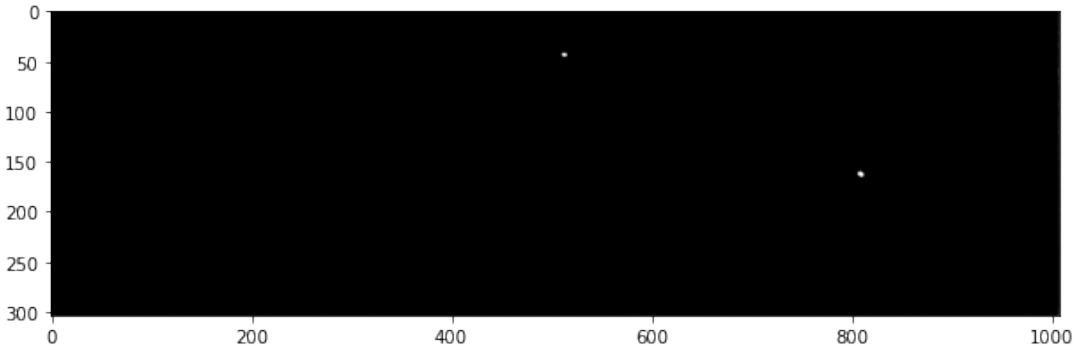
When the image is properly prepared we can plot it using plot_image_stack() and see the model's prediction on it using plot_prediction().

[14]:
```
folder_path = 'E:\\Documents\\Anaconda\\Jupyterkod\\Exjobb\\Egen␣
 ↪kod\\Exjobb\\From erik\\Transfer'

im_stack = get_image_stack(
    outputs=[0],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1008,
    im_size_height=304,
    im_resize_width=1280,
    im_resize_height=1024,
    function_img=[remove_running_mean, crop_and_append_image, normalize_image],
    row_delete_list=row_delete_list,
```

```
    col_delete_list=col_delete_list,
    path_folder=folder_path,
    tot_no_of_frames=10,
    im_height=1024,
    im_width=1280
)
plot_image_stack(im_stack)
plot_prediction(model=model, im_stack=im_stack)
```





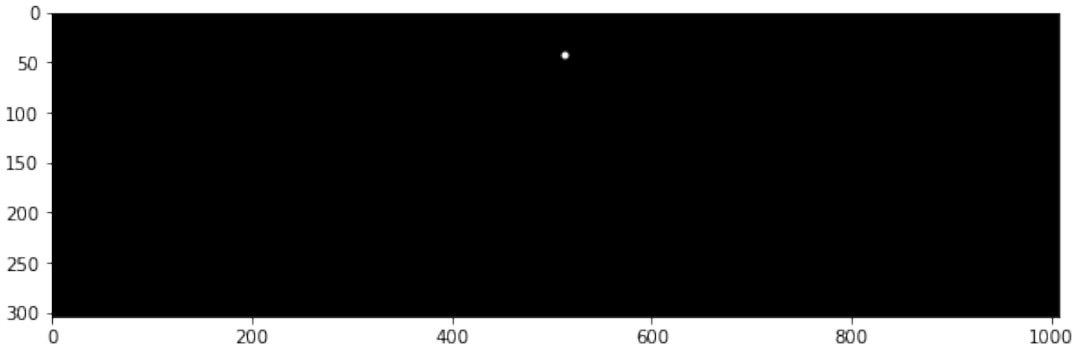


When the prediction of the network is satisfactory we can extract the positions from the predic-

tions using extract_positions. The function shares its inputs with im_stack with the addition of no_of_frames, model, layer, value_threshold and prediction_size. no_of_frames decides the number of frames to be analyzed and model is the trained network. layer is the segmentation layer from which we will extract the positions, in this example all plankton are on layer 1. value_threshold is the minimum value a pixel can have for it to be considered a plankton pixel. prediction_size is the maximum number of pixels a blob can have to be considered background.

To see which predictions were extracted we can use plot_found_positions. Using this function we can optimize the parameters value_threshold and predictions_size to filter out noise.

[15]:
```
positions = extract_positions(
    no_of_frames=10,
    outputs=[0],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1008,
    im_size_height=304,
    im_resize_width=1280,
    im_resize_height=1024,
    model=model,
    layer=1,
    value_threshold=0.6,
    prediction_size=0,
    function_img=[remove_running_mean, crop_and_append_image, normalize_image],
    row_delete_list=row_delete_list,
    col_delete_list=col_delete_list,
    path_folder=folder_path,
    tot_no_of_frames=10,
    im_height=1024,
    im_width=1280)

plot_found_positions(positions, width=1008, height=304)
```



We cropped the frames in the video to deal only with the areas of interest, to make the found positions overlap with the positions of the plankton in the uncropped video we run them through

fix_positions_from_cropping.

[16]:
```
new_positions = fix_positions_from_cropping(
    positions, col_delete_list=col_delete_list, row_delete_list=row_delete_list)
```

When we have our list of positions it's time to create traces with them, to do this we use the function assign_positions_to_planktons. The inputs are the found positions, max_dist, threshold and a bolean for extrapolation. max_dist is a maximum search radius from its latest position within which the algorithm will assign a found position to a plankton. If more than one is found the algorithm will chose the position that maintains the plankton's mean velocity. The parameter time_threshold sets the maximum number of time steps back in time the algorithm will search for a position if there are recent time steps where no positions were found for it. If extrapolate is set to True the algorithm will extrapolate a new position based on the previous two and look for new positions around that one. The output is a list of plankton where each plankton contains an array of positions.

If a plankton has gaps in its list of positions the function interpolate_gaps_in_plankton_positions will fill these if they only last for one timestep.The input and output is the list of plankton

If there are plankton that are stationary or move too slow they can be filtered using the function trim_list_from_stationary_planktons. The inputs are the list of plankton and a minimum total distance a plankton can travel throughout it's found positions.

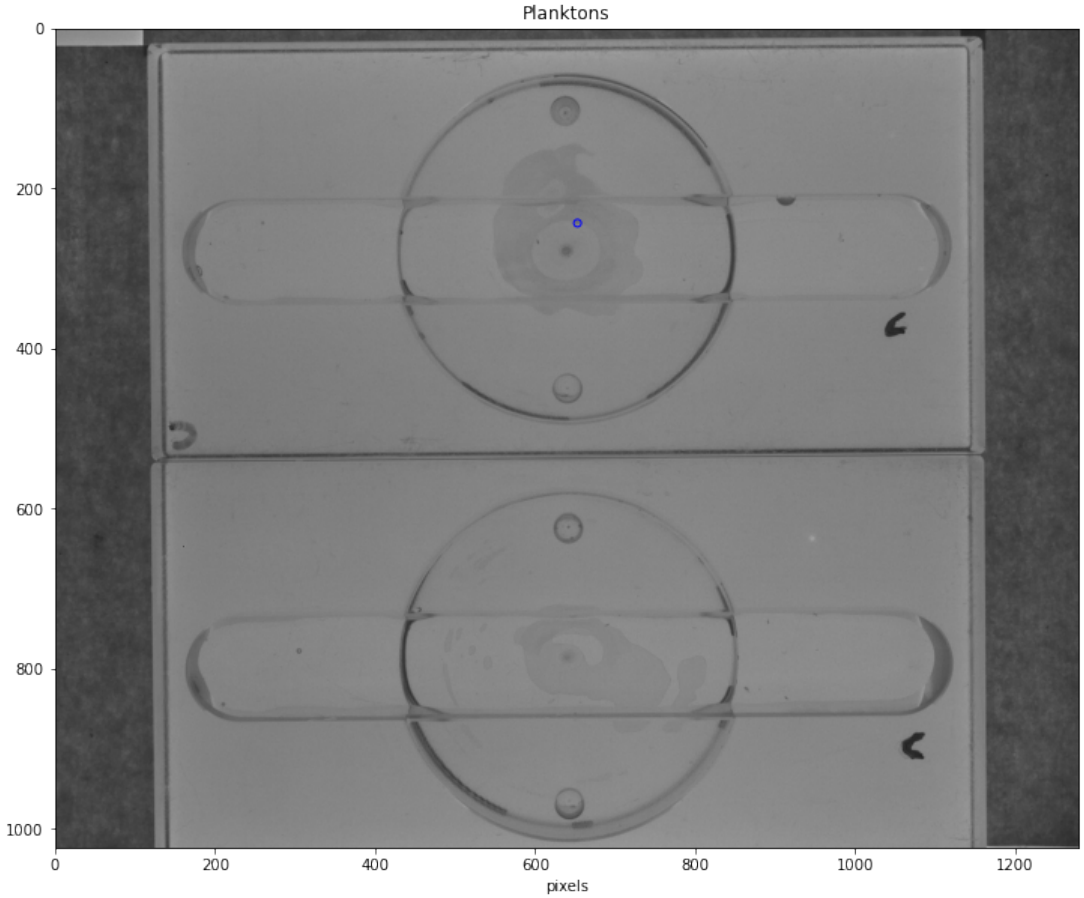
To further trim the list of plankton we can divide it into two lists of plankton depending on what fraction of the frames they are present in with the function split_plankton. The inputs are percentage_threshold which is this fraction, and the list of plankton.

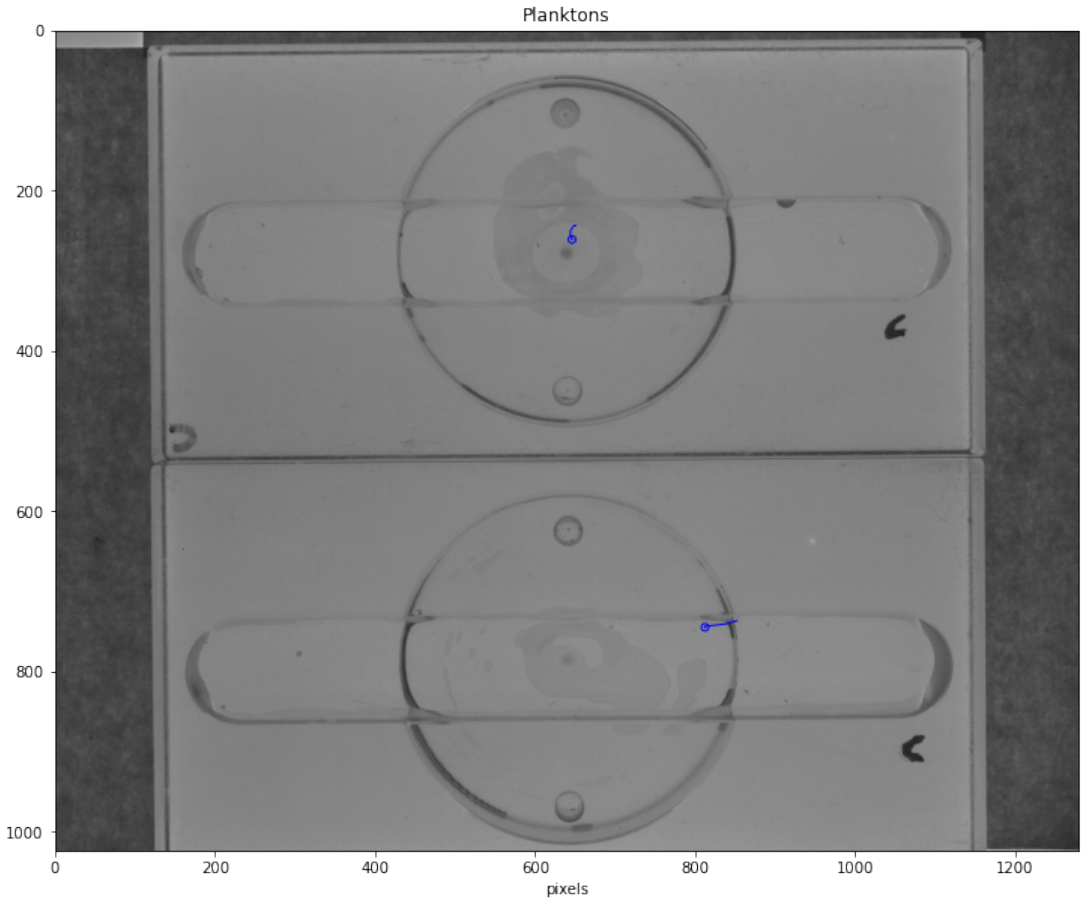
[17]:
```
list_of_plankton = assign_positions_to_planktons(
    new_positions, max_dist=60, time_threshold=10, extrapolate=True)

list_of_plankton = interpolate_gaps_in_plankton_positions(
    list_of_plankton=list_of_plankton)

list_of_plankton = trim_list_from_stationary_planktons(
    list_of_plankton=list_of_plankton, min_distance=5)

plankton_track, plankton_dont_track = split_plankton(
    percentage_threshold=0.0, list_of_plankton=list_of_plankton)
```

Finally we can plot the found positions onto the images that was analyzed, this is done using the function plot_and_save_track. The inputs are can be found in the paper. im_size_width and im_size_height here are used to rescale the found positions to the dimensions of the video. In this case we reshape the video to the dimensions 1024x1024.

[19]:
```
plot_and_save_track(no_of_frames=10,
                    plankton_track=plankton_track,
                    plankton_dont_track=plankton_dont_track,
                    folder_path=folder_path,
                    frame_im0=16,
                    save_images=False,
```

```
                    show_plankton_track=True,
                    show_plankton_dont_track=0,
                    show_numbers_track=0,
                    show_numbers_dont_track=0,
                    show_numbers_specific_plankton=False,
                    show_specific_plankton=False,
                    specific_plankton=None,
                    color_plankton_track='b',
                    color_plankton_dont_track='r',
                    color_specific_plankton='w',
                    im_size_width=1280,
                    im_size_height=1024,
                    save_path='E:\\transfer\\track',
                    frame_name='track',
                    file_type='.jpg')
```





We can also create a video with the saved images using Make_video.

[20]:
```
# make_video(frame_im0=0,
#            folder_path='E:\\transfer\\track',
#            save_path='E:\\transfer\\Test_video.avi',
#            fps=7,
#            no_of_frames=100)
```

2.6 Notebook Segmenting moving plankton

Sequential simulations differences between frames

June 16, 2021

1 Example: Sequential simulations, differences between frames

1.0.1 Explanations for all used functions can be found in the paper

First we need to load all the necessary functions.

[1]:
```
from loader import *
from models import *
from utils import *
from plotting import *
```

The next step is to create a sample of plankton. First we need to decide what type of plankton we want, here we want two different types of spherical plankton. Next we decide the size of the plankton and the size of the image they should be simulated on (in actuality we define the borders of the area the plankton can be initialized in), this is done through the parameters im_size_width, im_size_height, radius1 and radius2. To make the network treat them as two different types of plankton and segment them to different layers of the output later on we also need to assign them labels. One can also make them part of the background if the label is set to -1. In this case we want to track the moving plankton so we create a few stationary ones and assign them the label -1, the slow moving ones (drifting ones) areassigned label 0 and the faster ones are given the label 1.

[2]:
```
im_size_width, im_size_height, radius1, radius2 = 256, 256, 0.15e-6, 0.3e-6



moving_plankton1 = moving_spherical_plankton(
    im_size_height=im_size_height, im_size_width=im_size_width,
    radius=radius1, label=0, diffusion_constant_coeff=1)

stationary_plankton1 = stationary_spherical_plankton(
    im_size_height=im_size_height, im_size_width=im_size_width,
    radius=radius1, label=-1)

moving_plankton2 = moving_spherical_plankton(
    im_size_height=im_size_height, im_size_width=im_size_width,
    radius=radius2, label=1, diffusion_constant_coeff=13)

stationary_plankton2 = stationary_spherical_plankton(
    im_size_height=im_size_height, im_size_width=im_size_width,
```

```
    radius=radius2, label=-1)
```

When we have created our plankton we need to make them sequential by assigning them a function that updates one of their properties through the images of the sequence. The moving plankton are given helical motion and the stationary plankton are given an update function that returns their previous position.

[3]:
```
sequential_moving_plankton1 = Sequential(
    moving_plankton1, position=get_position_moving_plankton)

sequential_stationary_plankton1 = Sequential(
    stationary_plankton1, position=get_position_stationary_plankton)

sequential_moving_plankton2 = Sequential(
    moving_plankton2, position=get_position_moving_plankton)

sequential_stationary_plankton2 = Sequential(
    stationary_plankton2, position=get_position_stationary_plankton)
```

Then we create the microscope through which the plankton will be sampled. For this we use a brightfield microscope. To create it we need to define the size of the image to be simulated, this is done with im_size_height and im_size_width from the previous step. It is not necessary for the simulated image to be of the same size as the images to be predicted on as will be explained when the network is built. We can also apply a lighting gradient with the parameter gradient_amp, if it is set to 0 there will be no gradient.

[4]:
```
gradient_amp=1
microscope = plankton_brightfield(im_size_height, im_size_width, gradient_amp)
```

Then we can create a sample. If we want the number of plankton to vary on each simulated image we need to create a function that when called generates a random number, we can do this through the use of lambda functions as seen below. The plankton are then raised to the power of these functions to generate many plankton, plankton**4 would create 4 instances of plankton. To create the sample we simply add them together.

[5]:
```
no_of_moving_plankton1 = lambda: np.random.randint(50, 100)
no_of_stationary_plankton1 = lambda: np.random.randint(35, 70)
no_of_moving_plankton2 = lambda: np.random.randint(50, 100)
no_of_stationary_plankton2 = lambda: np.random.randint(15, 30)

sample = sequential_moving_plankton1**no_of_moving_plankton1 + \
sequential_stationary_plankton1**no_of_stationary_plankton1 + \
sequential_moving_plankton2**no_of_moving_plankton2 + \
sequential_stationary_plankton2**no_of_stationary_plankton2
```

Then we add the final properties to the sequence. We can add noise to the images through the parameter noise_amp where a value of 0 means no noise. With the next two parameters, norm_min and norm_max, we can saturate the images to contain both completely white and completely black pixels. To make the images contain very bright pixels we set norm_max to a value larger than 1,

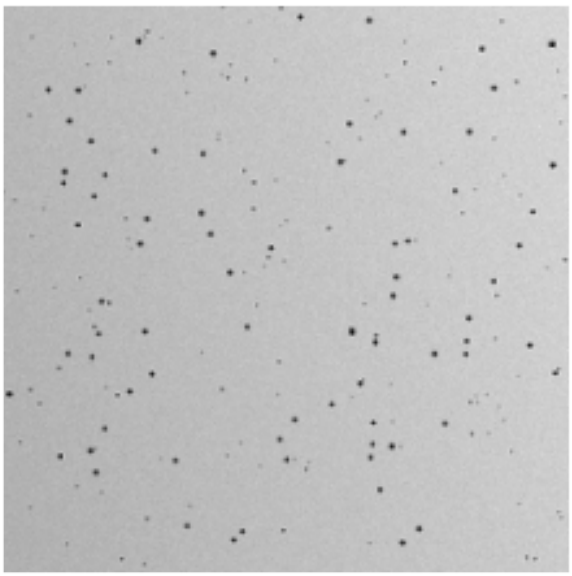
and norm_min to less than 0 for very dark pixels.

To create a sequence of images we use the function Sequence() with the abstract sequence as input together with the number of images we want.

After that we plot the sequence. For this cell to generate a new sequence each time we also use imaged_particle_sequence.update().

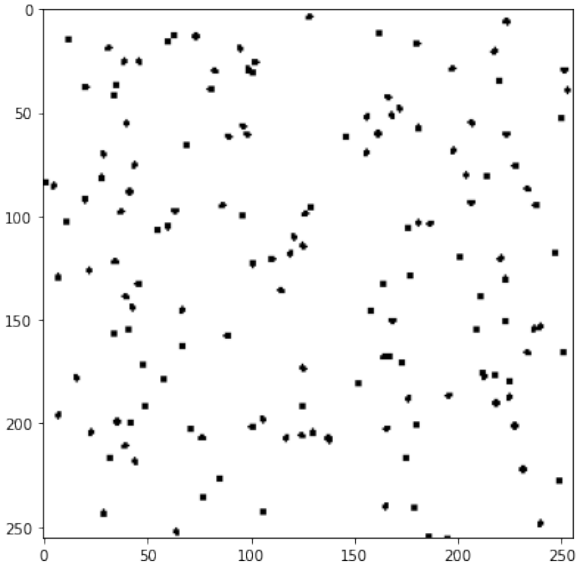
[6]:
```
noise_amp, norm_min, norm_max = 1, 0, 1
sequence = create_sequence(noise_amp, sample, microscope, norm_min, norm_max)
sequence_length = 3
imaged_particle_sequence = Sequence(sequence, sequence_length=sequence_length)
imaged_particle_sequence.update()
imaged_particle_sequence.plot(cmap='gray');
```

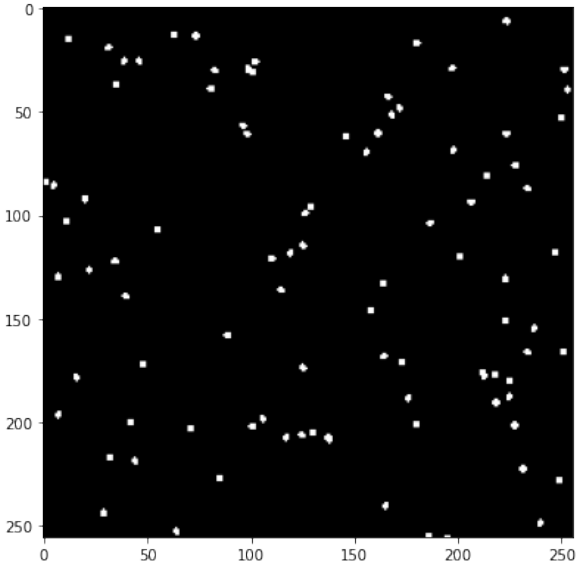
```
<IPython.core.display.HTML object>
```

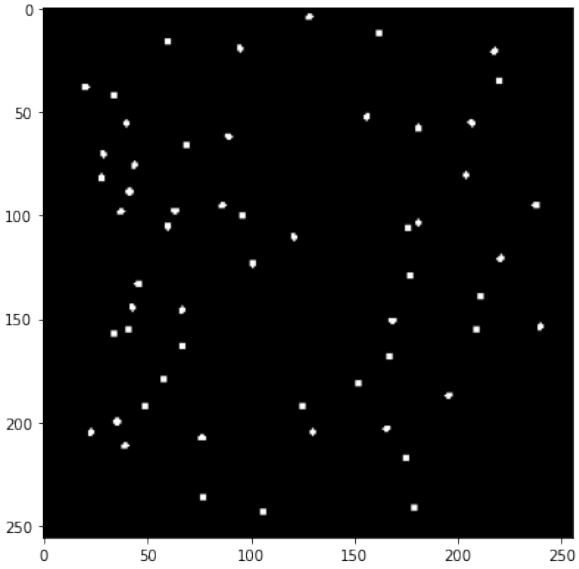


To create a training set, we also need labels for our images. For a simulated sequence we use the function get_target_sequence() to create these. If the sequence has different types of plankton, these will be segmented to different layers as seen when we plot the output of the label_function with plot_label. If the sequence contains only one type of plankton the output of the label_function will be three labels the positions of the plankton one each frame. This label_function will later be used in the training.

[7]:
```
label_function = get_target_sequence
plot_label(label_function, imaged_particle_sequence)
```


0
50
100
150
200
250
0
50
100
150
200
250

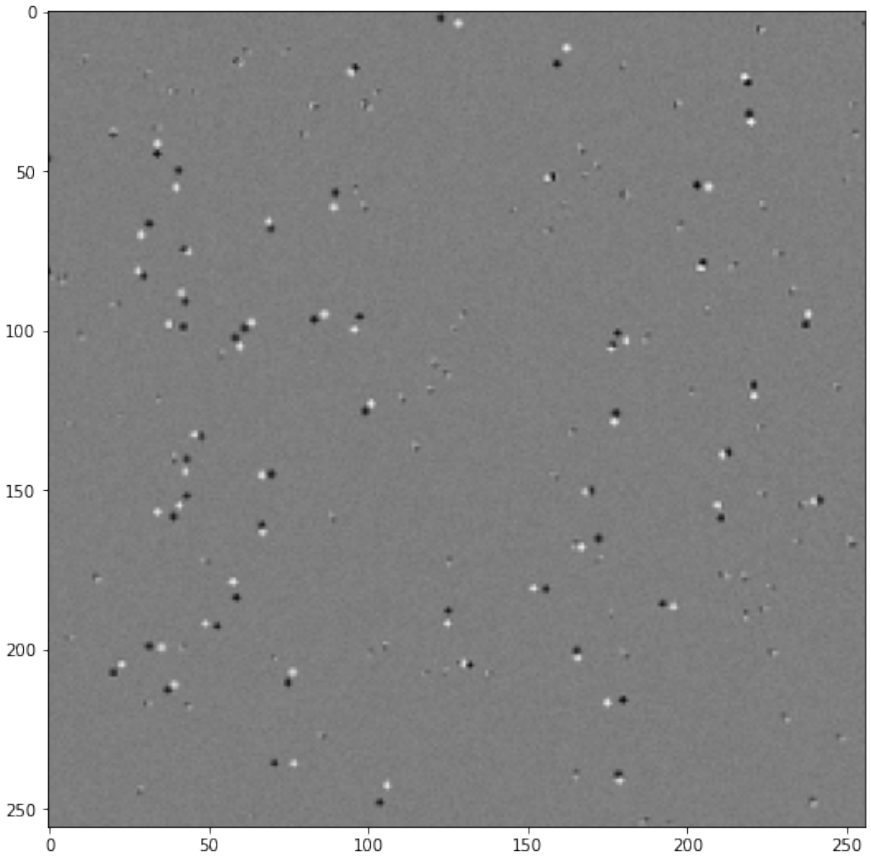

0
50
100
150
200
250
0
50
100
150
200
250

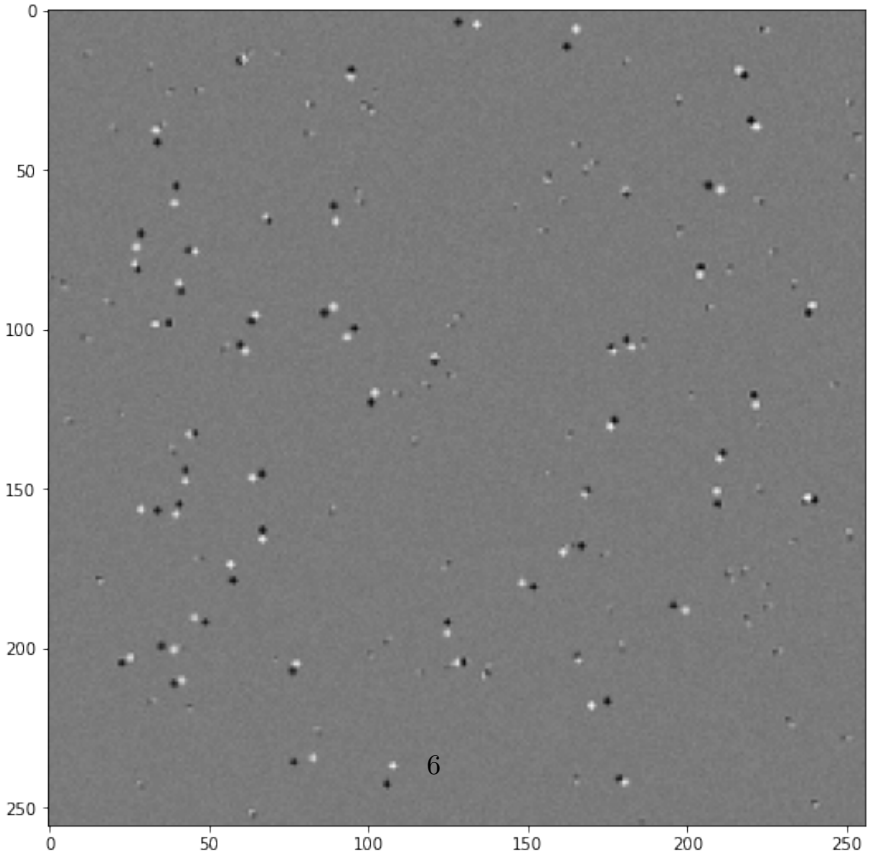

0
50
100
150
200
250
0
50
100
150
200
250

The next step is to create a batch function. The function takes the simulated image(s) and processes them before sending them to the network for training. The first input is the image, the second one is what the output of the batch function should looke like. In this case we use ywo differences between the images as following - image0 - image1 - image1 - image2

In the arguemnt function_img/function_diff we can send a list of functions that will be applied to the images/differences. To view the output of our batch function one can use plot_batch.

[8]:
```
batch_function = create_custom_batch_function(imaged_particle_sequence,
                                              outputs=[[1,0],[2,1]],
                                              function_diff=[normalize_image])

plot_batch(imaged_particle_sequence, batch_function)
```


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250

Now we can create a generator that creates the training data from our simulated image, our label function and our batch function. The continuous generator generates a data set of the defined minimum size before the training starts. It continues to add data to the data set during training until the data size exceeds defined maximum size, then the new data replaces the old data. We also need to define the batch size.

Note that min_data_size needs to be bigger than or equal to the batch_size multiplied with the number of steps_per_epoch defined in the next step.

[9]:
```
from deeptrack.generators import ContinuousGenerator
generator = ContinuousGenerator(
    imaged_particle_sequence,
    get_target_sequence,
    batch_function,
    batch_size=8,
    min_data_size=128,
    max_data_size=512
    )
```

The last step before starting the training is to define the model. The two Nones are width and height of the images, they don't need to be defined and are better left as Nones if one intends to use the model on differently sized images or images with a different size than the images in the training data. However no_of_inputs and no_of_outputs need to defined, they correspond to the number of images in the input to the network (same as the number of images in the output of the batch_function) and the output of the network (same as the number of images in the output of the label_function).

The training starts by running the function train_model_early_stopping which stops the training early if the performance of the network doesn't improve for a number of epochs. The inputs are the model, the generator and parameters of the training. The patience parameter defines the number of epochs after which the training will stop if no improvements are made. The parameter epochs sets the maximum number of epochs the network will be trained for and steps_per_epoch sets how many batches one epoch will consist of.

[10]:
```
no_of_inputs, no_of_outputs = 2, 3
model = generate_unet(None, None, no_of_inputs, no_of_outputs)
model = train_model_early_stopping(model, generator, patience=15,
                                   epochs=1000, steps_per_epoch=10)
```

```
Generating 128 / 128 samples before starting training
Epoch 1/1000
10/10 [==============================] - 1s 80ms/step - loss: 0.0142
Epoch 2/1000
10/10 [==============================] - 1s 93ms/step - loss: 0.0141
Epoch 3/1000
10/10 [==============================] - 1s 88ms/step - loss: 0.0139
Epoch 4/1000
```

```
Epoch 97/1000
10/10 [==============================] - 1s 83ms/step - loss: 4.7144e-04
Epoch 98/1000
10/10 [==============================] - 1s 82ms/step - loss: 4.0659e-04
Epoch 99/1000
10/10 [==============================] - 1s 92ms/step - loss: 3.3161e-04
Epoch 100/1000
10/10 [==============================] - 1s 88ms/step - loss: 3.1443e-04
Epoch 101/1000
10/10 [==============================] - 1s 91ms/step - loss: 3.1006e-04
Epoch 102/1000
10/10 [==============================] - 1s 88ms/step - loss: 2.8570e-04
Epoch 103/1000
10/10 [==============================] - 1s 89ms/step - loss: 3.7877e-04
Epoch 104/1000
10/10 [==============================] - 1s 93ms/step - loss: 3.3936e-04
Epoch 105/1000
10/10 [==============================] - 1s 82ms/step - loss: 3.1728e-04
Epoch 106/1000
10/10 [==============================] - 1s 80ms/step - loss: 4.2311e-04
Epoch 107/1000
10/10 [==============================] - 1s 79ms/step - loss: 3.6756e-04
Epoch 108/1000
10/10 [==============================] - 1s 76ms/step - loss: 3.4025e-04
Epoch 109/1000
10/10 [==============================] - 1s 87ms/step - loss: 3.5810e-04
Epoch 110/1000
10/10 [==============================] - 1s 85ms/step - loss: 4.3984e-04
Epoch 111/1000
10/10 [==============================] - 1s 81ms/step - loss: 4.0356e-04
Epoch 112/1000
10/10 [==============================] - 1s 78ms/step - loss: 3.6430e-04
Epoch 113/1000
10/10 [==============================] - 1s 77ms/step - loss: 3.2153e-04
Epoch 114/1000
10/10 [==============================] - 1s 81ms/step - loss: 3.0856e-04
Epoch 115/1000
10/10 [==============================] - 1s 78ms/step - loss: 3.4763e-04
Epoch 116/1000
10/10 [==============================] - 1s 75ms/step - loss: 3.2666e-04
Epoch 117/1000
10/10 [==============================] - 1s 83ms/step - loss: 3.0965e-04
```

When the model is trained one might want to save it to avoid going through this process unnecessarily, this is done with model.save(save path). The save_path_model defines the path to where the network will be saved and with what name.

To load it one uses keras.models.load_model(load path) with the entire path to the model as input. In our case we use a custom defined loss function called softmax_categorical which needs to be

added as an input to the function as seen.

[11]:
```
# save_path_model = 'E:\\Documents\\models\\diffs.keras'
# model.save(save_path_model)
```

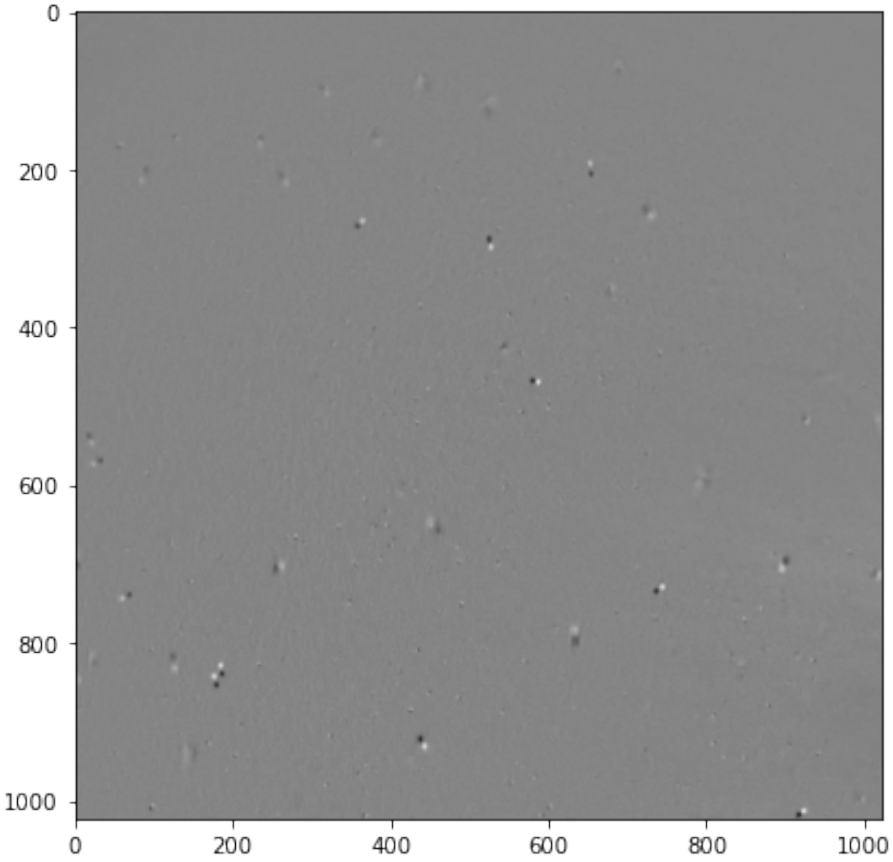
[12]:
```
# load_path_model = 'E:\\Documents\\models\\diffs.keras'
# model = keras.models.load_model(
#     load_path_model, custom_objects={'softmax_categorical':
 ↪softmax_categorical})
```

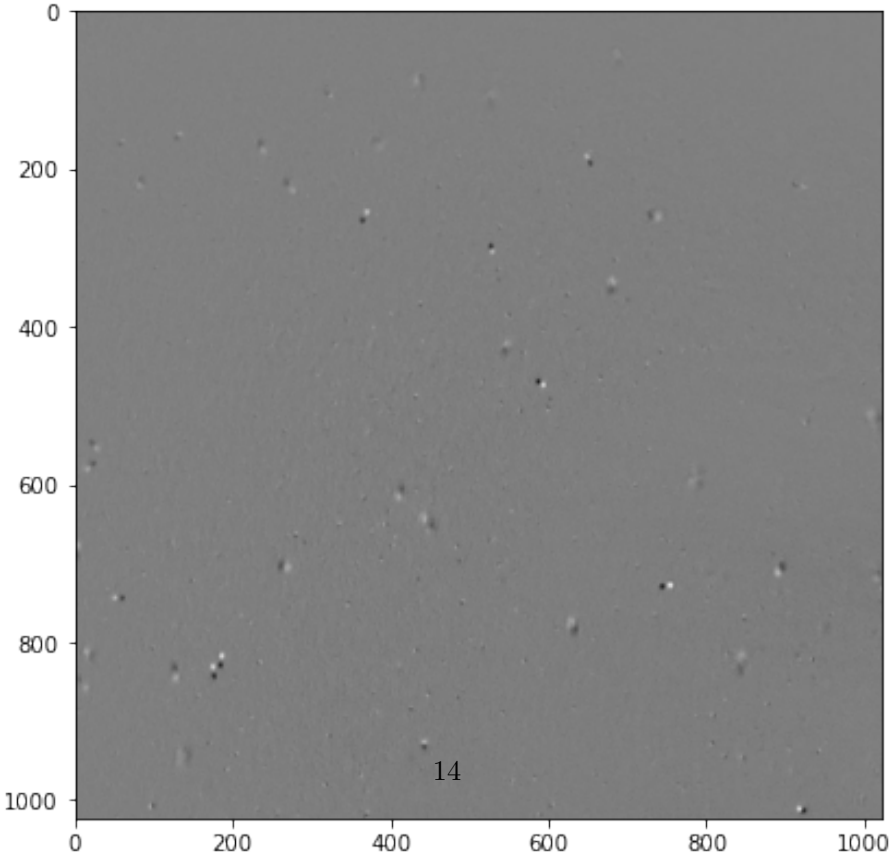
Now to evaluate our model on a frame from our plankton video. To do this we need to feed the images to the network in the way they were fed to it during the training, for this we use the function im_stack. The outputs parameter is set to be identical to the outputs of the batch_function, the path to the folder of the frames is also necessary. The image to be analyzed is assigned through the parameter frame_im0, the image shown will in our case be the 16th image in the folder in alphabetical order. The size of the images when fed to the network is defined in im_size_width and im_size_height. If the images to be analyzed need to be rescaled to a certain size these values are specified in im_resize_width and im_resize_height, if not these values are the dimensions of the image. Usually these two size pairs are equal to eachother. They differ from eachother if one wants to crop the full size image, then im_resize_width and im_resize_height will be the same as the dimensions of the full sized image.

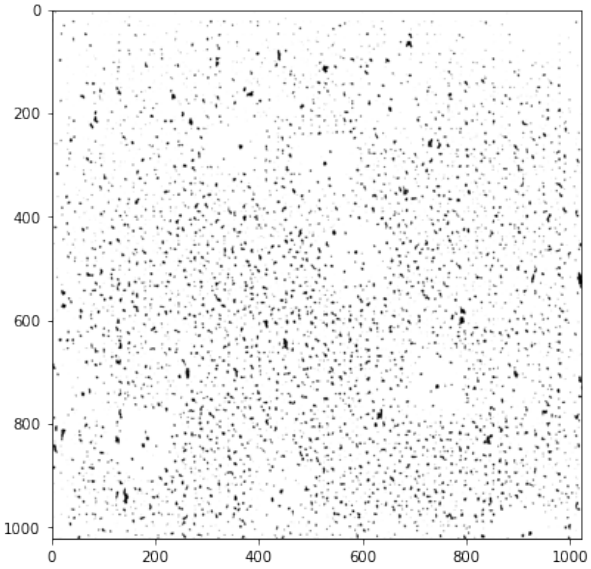
We also have the option to treat the image with a set of functions, these are specified in the function_img/function_diff the same way as in create_custom_batch_function.

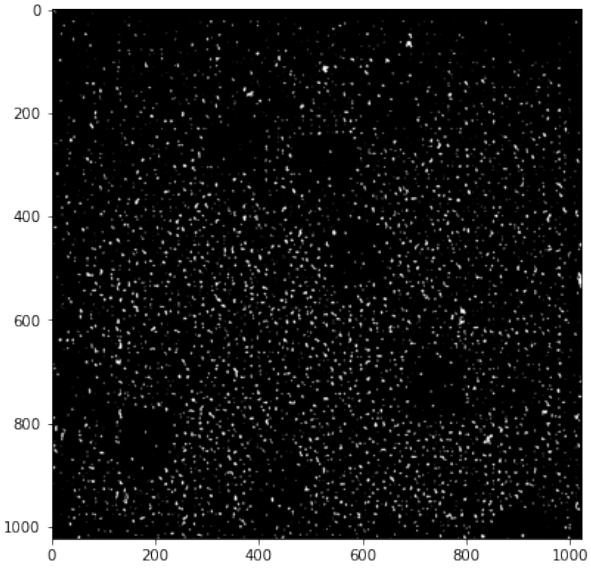
When the image is properly prepared we can plot it using plot_image_stack() and see the model's prediction on it using plot_prediction().

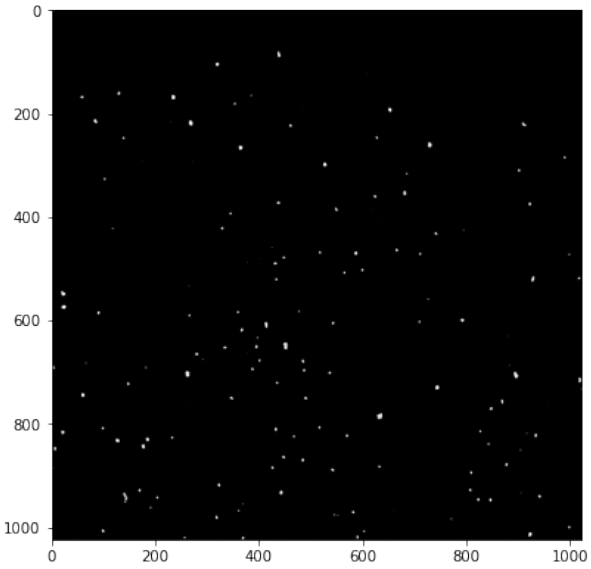
[13]:
```
folder_path = 'E:\\raw output'
im_stack = get_image_stack(
    outputs=[[1,0],[2,1]],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1024,
    im_size_height=1024,
    im_resize_width=1024,
    im_resize_height=1024,
    function_img=[],
    function_diff=[normalize_image])
plot_image_stack(im_stack)
plot_prediction(model=model, im_stack=im_stack)
```


0
200
400
600
800
1000
0
200
400
600
800
1000


0
200
400
600
800
1000
0
200
400
600
800
1000


0
200
400
600
800
1000
0
200
400
600
800
1000

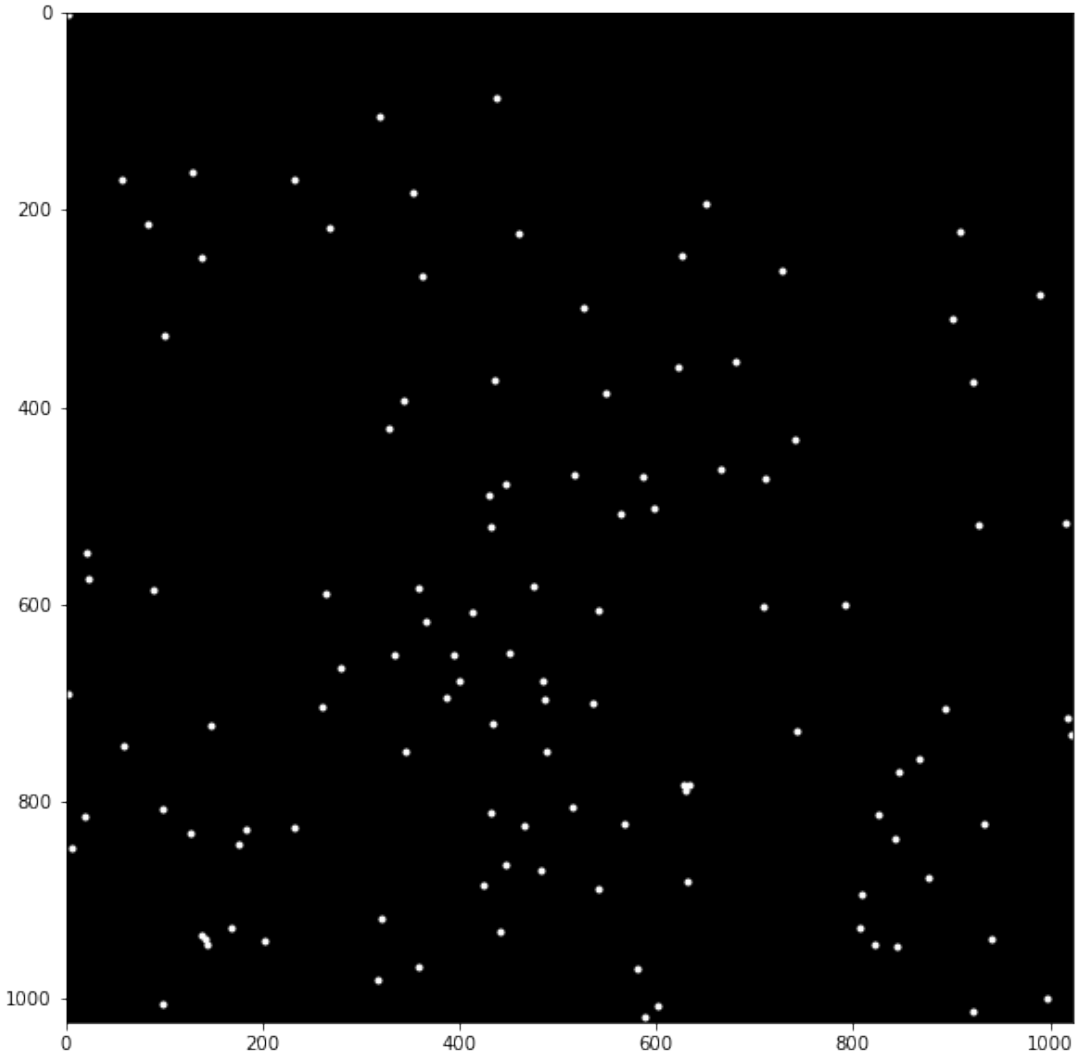

0
200
400
600
800
1000
0
200
400
600
800
1000


0
200
400
600
800
1000
0
200
400
600
800
1000

When the prediction of the network is satisfactory we can extract the positions from the predictions using extract_positions. The function shares its inputs with im_stack with the addition of no_of_frames, model, layer, value_threshold and prediction_size. no_of_frames decides the number of frames to be analyzed and model is the trained network. layer is the segmentation layer from which we will extract the positions, in this example the large particles are on layer 2. value_threshold is the minimum value a pixel can have for it to be considered a plankton pixel. prediction_size is the maximum number of pixels a blob can have to be considered background.

To see which predictions were extracted we can use plot_found_positions. Using this function we can optimize the parameters value_threshold and predictions_size to filter out noise.

[14]:
```
positions = extract_positions(
    no_of_frames=10,
    outputs=[[1,0],[2,1]],
    folder_path=folder_path,
    frame_im0=16,
    im_size_width=1024,
    im_size_height=1024,
    im_resize_width=1024,
    im_resize_height=1024,
    model=model,
    layer=2,
    value_threshold=0.95,
    predictions_size=0,
    function_diff=[normalize_image])

plot_found_positions(positions, width=1024, height=1024)
```



When we have our list of positions it's time to create traces with them, to do this we use the function assign_positions_to_planktons. The inputs are the found positions, max_dist, threshold and a bolean for extrapolation. max_dist is a maximum search radius from its latest position within which the algorithm will assign a found position to a plankton. If more than one is found the algorithm will chose the position that maintains the plankton's mean velocity. The parameter time_threshold sets the maximum number of time steps back in time the algorithm will search for a position if there are recent time steps where no positions were found for it. If extrapolate is set to True the algorithm will extrapolate a new position based on the previous two and look for new positions around that one. The output is a list of plankton where each plankton contains an array of positions.

If a plankton has gaps in its list of positions the function interpolate_gaps_in_plankton_positions will fill these if they only last for one timestep.The input and output is the list of plankton

If there are plankton that are stationary or move too slow they can be filtered using the function

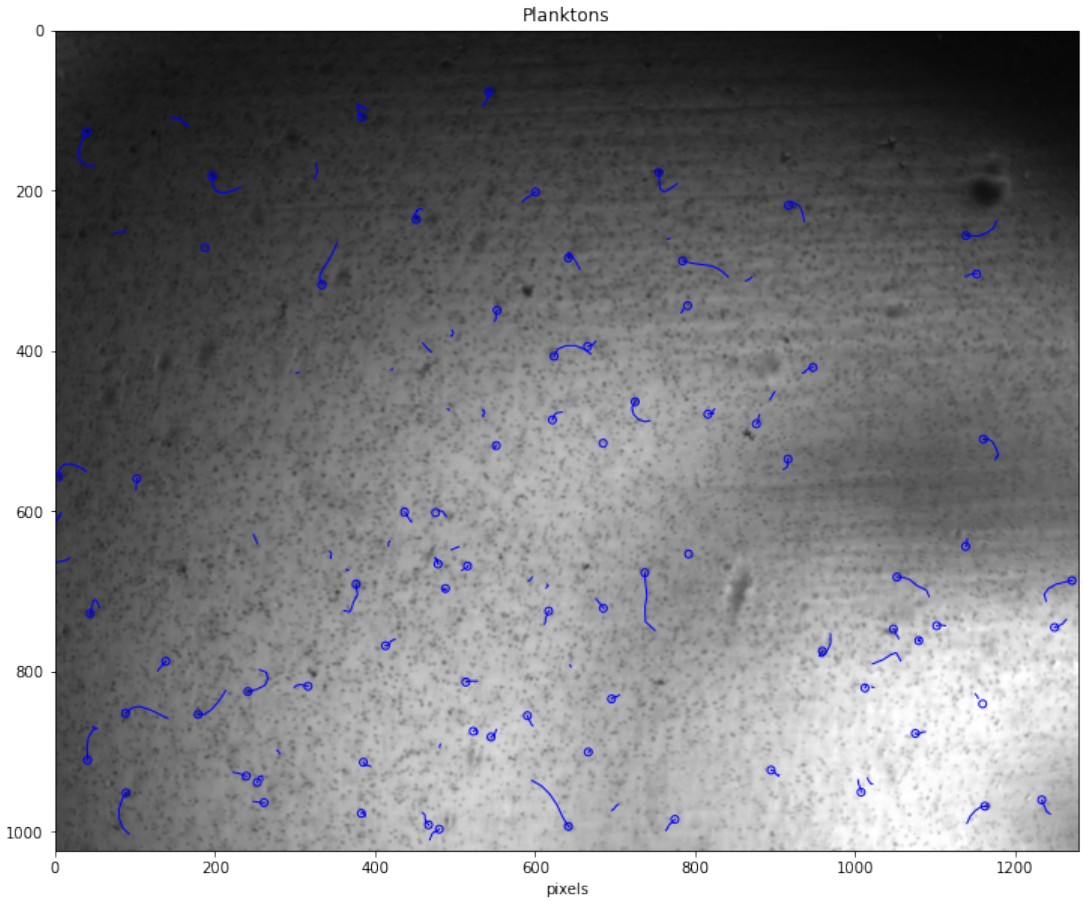
trim_list_from_stationary_planktons. The inputs are the list of plankton and a minimum total distance a plankton can travel throughout it's found positions.

To further trim the list of plankton we can divide it into two lists of plankton depending on what fraction of the frames they are present in with the function split_plankton. The inputs are percentage_threshold which is this fraction, and the list of plankton.

[15]:
```
list_of_plankton = assign_positions_to_planktons(
    positions, max_dist=15, time_threshold=5, extrapolate=True)

list_of_plankton = interpolate_gaps_in_plankton_positions(
    list_of_plankton=list_of_plankton)

list_of_plankton = trim_list_from_stationary_planktons(
    list_of_plankton=list_of_plankton, min_distance=2)

plankton_track, plankton_dont_track = split_plankton(
    percentage_threshold=0.5, list_of_plankton=list_of_plankton)
```

Finally we can plot the found positions onto the images that was analyzed, this is done using the function plot_and_save_track. The inputs are can be found in the paper. im_size_width and im_size_height here are used to rescale the found positions to the dimensions of the video. In this case we reshape the video to the dimensions 1024x1024.

[16]:
```
plot_and_save_track(no_of_frames=10,
                    plankton_track=plankton_track,
                    plankton_dont_track=plankton_dont_track,
                    folder_path=folder_path,
                    frame_im0=17,
                    save_images=0,
                    show_plankton_track=True,
                    show_plankton_dont_track=False,
                    show_numbers_track=0,
                    show_numbers_dont_track=0,
                    show_numbers_specific_plankton=False,
                    show_specific_plankton=False,
                    specific_plankton=None,
                    im_size_width=1024,
                    im_size_height=1024,
                    color_plankton_track='b',
                    color_plankton_dont_track='r',
                    color_specific_plankton='w',
                    save_path='E:\\Documents\\Raw_output',
                    frame_name='track',
                    file_type='.jpg')
```



We can also create a video with the saved images using Make_video.

[17]:
```
# make_video(frame_im0=0,
#            folder_path='E:\\Documents\\Raw_output',
#            save_path='E:\\Documents\\Differences.avi',
#            fps=7,
#            no_of_frames=10)
```

3
Results

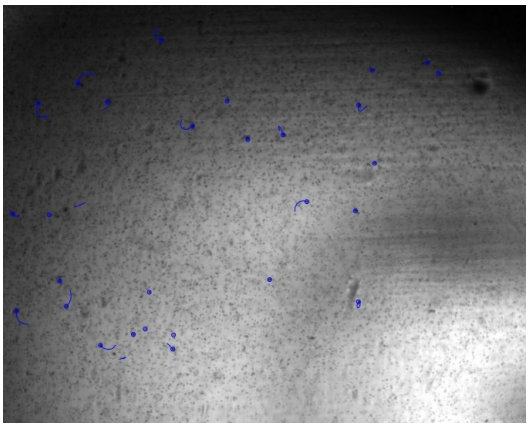
In this section the software TrackMate is compared to the networks trained using the introduced techniques. The networks and simulations were built with the samples of the salmon lice and *Strombidium arenicola* and *Rhodomonas baltica* in mind and then applied on five other samples to test generality. The comparison consists of using the networks and TrackMate to track plankton in different videos. For the network a blue circle is put on the position of a found plankton and for TrackMate a purple one. The line following the circle shows the previous five positions of the plankton indicating that the positions have been used to make a trace for that specific plankton. For our method only the segmentation of the images is performed by a network. Extracting positions from the segmentation and linking them is done using simple algorithms as described in the documentation. With this in mind the results generated by this method may be referred to as some variation of "the result of the network/network-based approach", even though the network only performed the segmentation. TrackMate may be refered to as "TrackMate" or some variation of "the algorithm/algorithmic approach". The plankton videos and tracks of TrackMate were provided by the department of marine sciences at the University of Gothenburg. A complete playlist of all videos can be found here.

3.1 *Strombidium arenicola* and *Rhodomonas baltica*

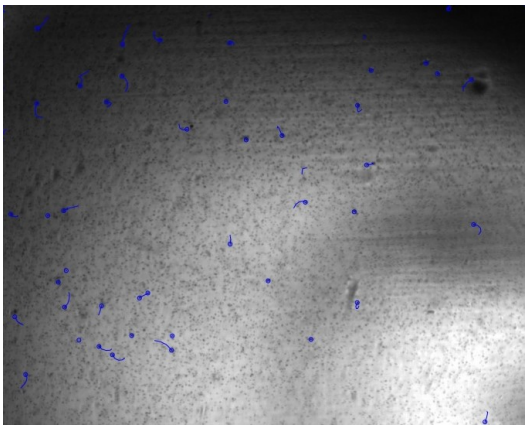
This is the first plankton sample of two that the simulations and training was optimized for. In this sample two species of plankton are present: *Strombidium arenicola* and *Rhodomonas baltica*. *Strombidium arenicola* are plankton of the group heterotrophic ciliate, heterotrophic meaning that they consume organic matter for nutrients and ciliate denoting the numerous hair-like cilia used for locomotion and feeding in this group [27, p. 31, 89]. They also exhibit a diel feeding rhythm meaning their feeding activity is different between night and day [28]. A typical size is 30µm in length and 25µm wide [29]. The other species, *Rhodomonas baltica*, is an autotrphic cryptophyte micro algae [30]. Autotrophy means that it uses photosynthesis as its source of energy [27, p. 18] and Cryptophyceans is a group of small biflagellate plankton common in both marine and freshwater habitats [31]. Their size is usually around 5µm to 10µm [30].

Applying the different combinations of techniques when training the networks on this plankton sample generated the results seen in figures 3.1, 3.2 and 3.3. In figure

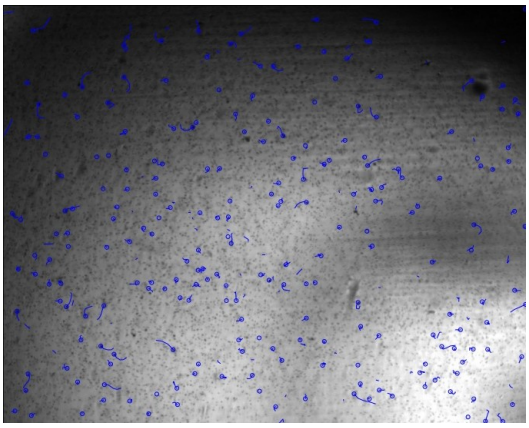
3.1a the segmentation of a network that takes one frame at a time as input is shown. It finds most of the big visible plankton and makes no distinction between moving and stationary ones, it also differentiates between between big ones and small ones (as seen in notebook 2.4). In figure 3.1b the result of a network that takes a sequence of three consecutive images as input and segments the image trying to differentiate between different types of plankton is shown. It finds more of the big, more blurry plankton. In figure 3.1c the result of a network trained on the differences (as seen in notebook 2.6) between the images in a sequence of length 3 is shown, it finds specifically the plankton that are moving between the frames. In figure 3.1d the result of a network trained on three images in a sequence combined with the differences between these images is shown, it finds fewer moving plankton than the network trained on the differences between the images but is less susceptible to noise as seen in the videos. In figure 3.1e the result of TrackMate is seen, tracking some big ones while being susceptible to noise and missing plankton in the brighter regions of the sample.



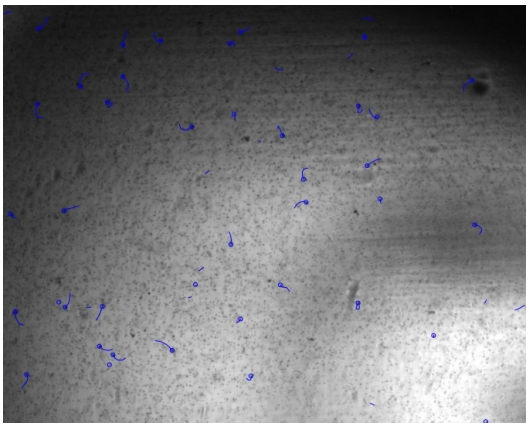
(a) Tracking of network trained on one image at a time. Video.



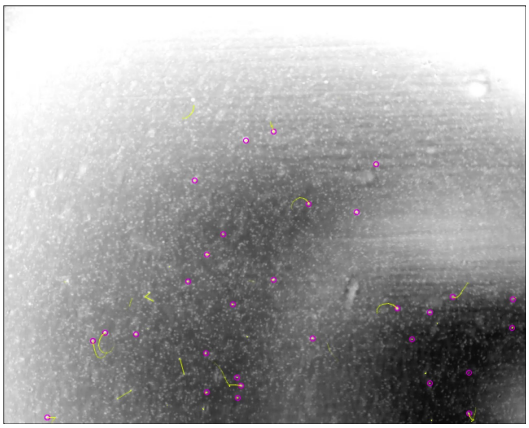
(b) Tracking of network trained on a sequence of three images. Video.



(c) Tracking of network trained on the differences between three images in a sequence to detect moving plankton. Video.

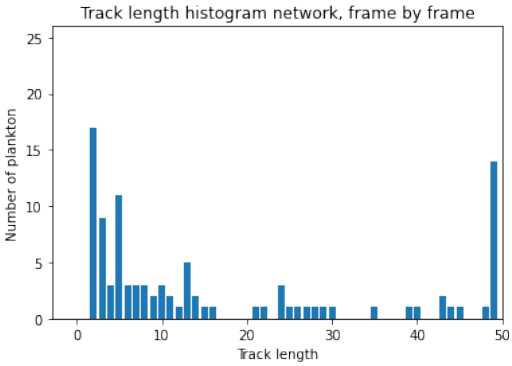


(d) Tracking of network trained on two differences combined with the three images in a sequence. Video.

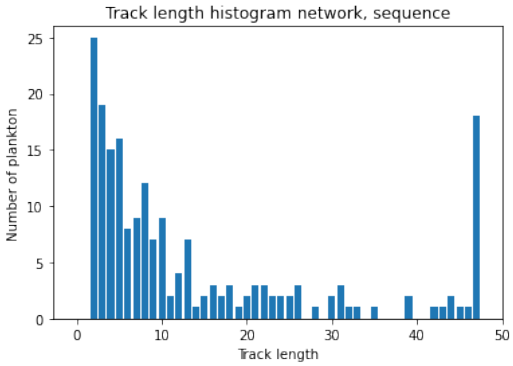


(e) Tracking of TrackMate on the inverted image. Video.

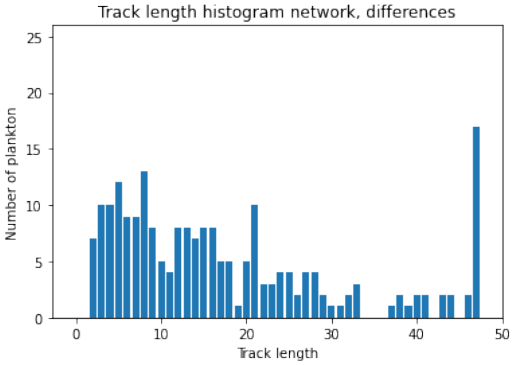
Figure 3.1: Comparison of differently trained networks on the same frame.



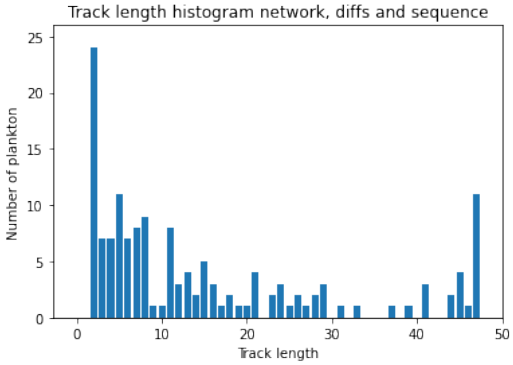
(a) Track length distribution of a network trained on one image at a time.



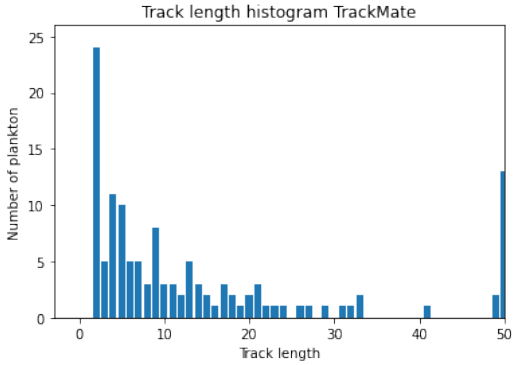
(b) Track length distribution of a network trained on a sequence of three images.



(c) Track length distribution of a network trained on the differences between three images in a sequence to detect moving plankton.



(d) Track length distribution of a network trained on two differences combined with the three images in a sequence.



(e) Track length distribution for TrackMate.

Figure 3.2: Track length distributions of differently trained networks. If errors in linking is ignored longer track lengths are better.

In figure 3.2 the distribution of the track lengths for the five trackings in figure 3.1 is shown. When emphasis is put on identifying moving dots instead of plankton-like

dots the distribution moves toward longer tracks. The number of plankton found per frame and on average for the four networks and TrackMate is shown in figure 3.3 and again more plankton are found per frame as emphasis is put in finding moving plankton. This is likely because when provided with two frames where the plankton move the network is provided with more usable information to localize the plankton. Another reason why the differences between the frames works so well could be that the sharpness of the edges of a plankton vary as it moves over the background during its trajectory, while if two adjacent frames are subtracted from each other the background will be removed and what remains will be two dots, one black and one white, at the position of the plankton in the two frames.

We compare the network trained on a sequence of three images (figure 3.1b) with TrackMate (figure 3.1e) and see that the network-based method catches the same moving plankton as the algorithm does and additionally some in the darker areas of the sample (brighter areas for the in the video produced by TrackMate). In this frame our network finds 30 moving plankton while TrackMate finds 9. The other trackless circles found by the network/TrackMate either represent a stationary plankton or a plankton just found. The network found 20 of these and TrackMate found 21 of which 10 are either the wrong species of plankton or dirt. 12 of the 51 found plankton were either dirt or the wrong species for the network. In figure 3.2b and 3.2e the distributions of the track lengths for the two tracking methods is shown. The distributions are of the same shape with the difference being that more and longer tracks in general was found by the network. In figure 3.3 two plots over the number of plankton found in each frame combined with the mean for each method is shown. We note that the shape of the orange and purple curves are similar and that the network finds 63% more plankton each frame on average. These results indicate that the network is able to find more plankton than TrackMate in each frame with higher accuracy but the tracing of the methods are similar.

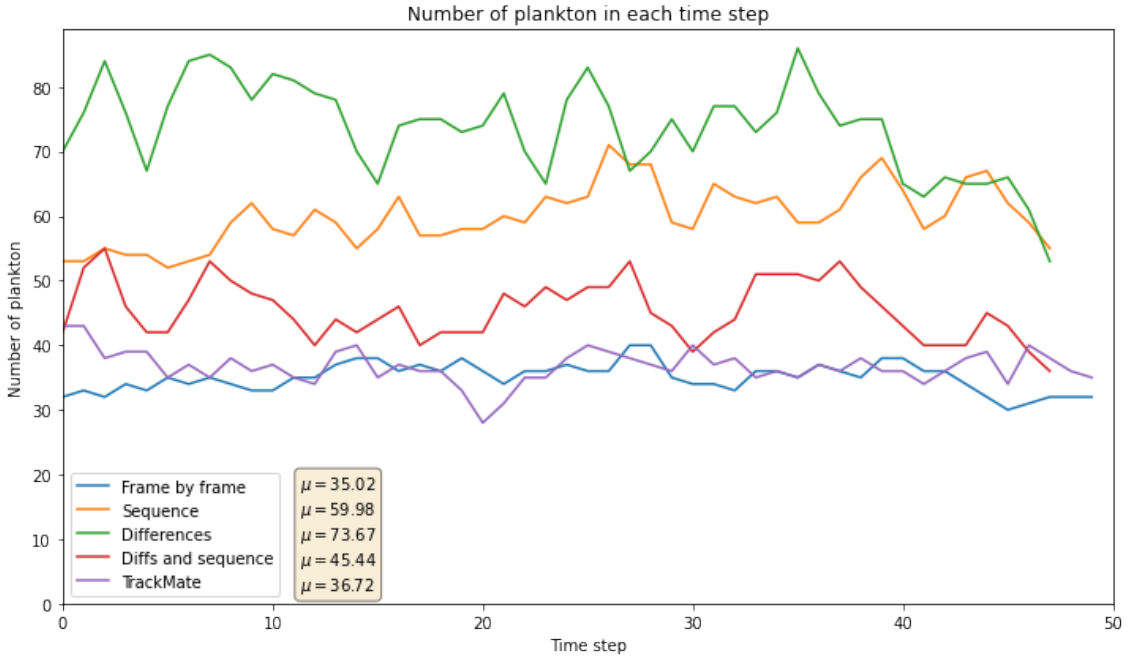
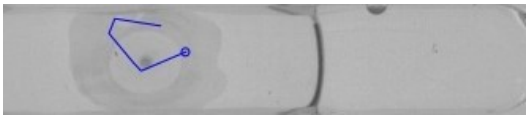


Figure 3.3: The number of found positions in each frame by the different networks and TrackMate. **Blue line**: Network trained frame by frame. **Orange line**: Network trained on sequences of 3 images. **Green line**: Network trained on the differences between the images in a sequence of length 3. **Red line**: Network trained on the differences combined with the images in a sequence of length 3. **Purple line**: Trackmate. The mean number of positions found by each method is displayed next to the legend.

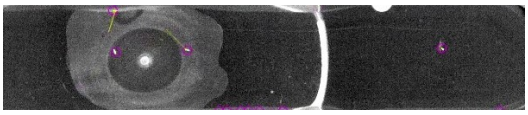
3.2 Salmon lice (*Lepeophtheirus salmonis*)

This is the second plankton sample of the two that the simulations and training was optimized for. The sea lice, *Lepeophtheirus salmonis*, is a type of parasitic copepod (a group of crustaceans)exclusively living on Salmonidae fish such as the Atlantic salmon (*Salmo salar*). In farmed salmonids these parasites can often result in heavy losses both in terms of treatment costs and in terms of reduced growth and increased mortality making it an economically important parasite. [32] Males usually grow to a size of 5-6mm in length while females grow to a size of 8-18mm [33].

For this sample a network trained on one frame at a time was used in combination with removing the running mean from the frames (see notebook 2.5), the sample has also been divided into the two different regions of the video containing the plankton for comparison purposes. In figures 3.4a and 3.4c the network's tracking is shown together with TrackMate's tracking in figures 3.4b and 3.4d. Note the number of misclassifications made by trackMate along the edges and on debris. Note also that the network misses one plankton in figure 3.4c. The true number of plankton in the sample is one plankton in the top image and two plankton in the bottom one.

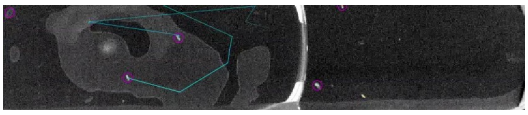


(a) Tracking of network trained on one frame at a time with running mean removed from each image. Video.



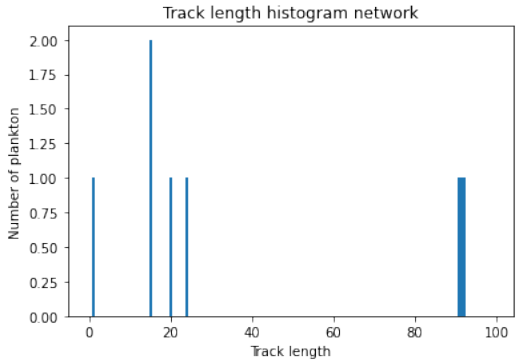
(b) Tracking of TrackMate. Video.

(c) Tracking of network trained on one frame at a time with running mean removed from each image. Video.

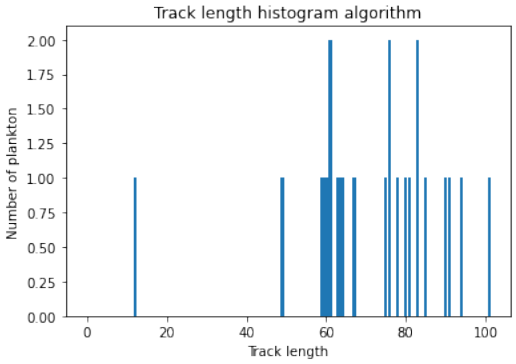


(d) Tracking of TrackMate. Video.

Figure 3.4: Comparison between one of the trained networks and TrackMate on the same frame. In the top image there is only one plankton and in the bottom one there are two plankton.



(a) Track length distribution for the network.



(b) Track length distribution for TrackMate.

Figure 3.5: Track length distributions of the network and TrackMate for the methods used in figure 3.4.

In figure 3.5 the distribution of the track lengths for the network and the algorithm is shown. For the algorithm all tracks lasted through the majority of the video while for the network all but three tracks last for less than 20 frames. The reason TrackMate has long tracks is likely due to the number of misclassifications being tracked throughout the video. In figure 3.6 the number of plankton found in each time step is shown. The algorithm on average finds 16 plankton on each frame while the network finds 2. These results indicate that the network is less susceptible to misclassifications than TrackMate.

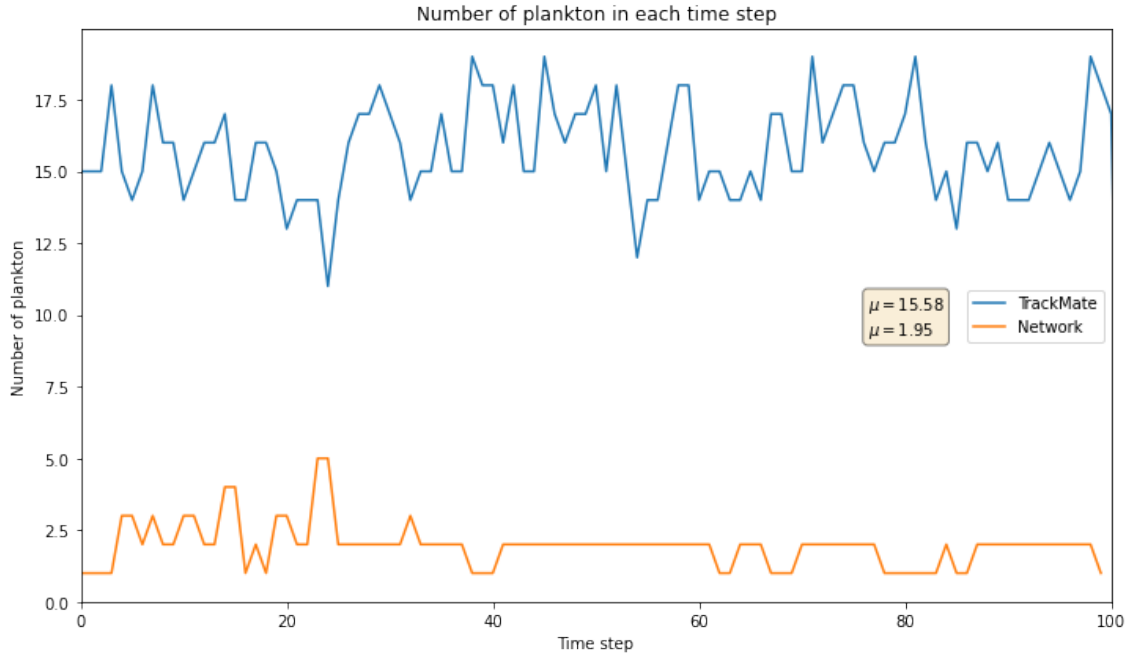
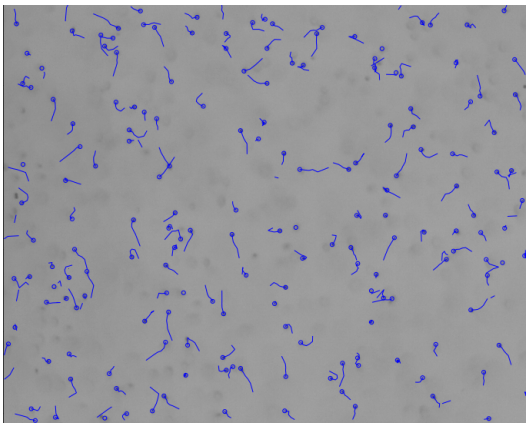


Figure 3.6: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained frame by frame with the running mean removed from each image. The mean number of positions found by each method is displayed next to the legend.

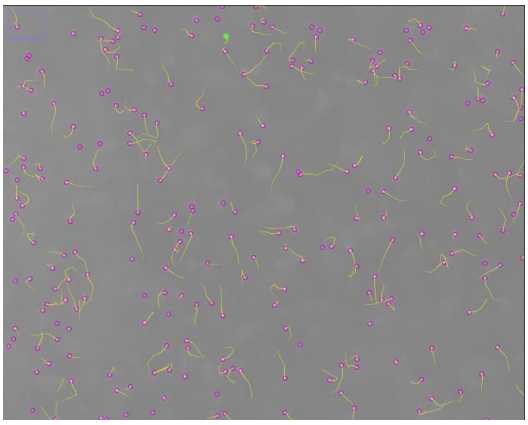
3.3 *Alexandrium sp.*

Alexandrium is a genus of dinoflagellates containing most species responsible for toxic algal blooms, specifically producing saxitoxin and its analogs (STX). These toxins can accumulate in eg. shellfish which can lead to Paralytic Shellfish Poisoning if ingested thus potentially costing aquaculture large amounts of money when discarding exposed shellfish. Alexandrium are generally not strong competitors within their habitat with regards to nutrient uptake efficiency and growth rate compared to other phytoplankton. Their success can instead be attributed to their adaptability being occasionally mixotrophic, performing diel vertical migration and producing toxins targeting predators (eg. copepods [34]) and competitors of other species. Their size is of the magnitude 60µm. [35]

This sample of plankton was analyzed to test how well the networks/techniques generalize to images it hasn't been optimized for. For this sample a network trained on the differences between the images has been used as shown in notebook 2.6, the same network and weights as in figure 3.1c. In figure 3.7 the comparison of TrackMate and our network is shown. Note that the blurriest plankton aren't caught by the network but caught by TrackMate.

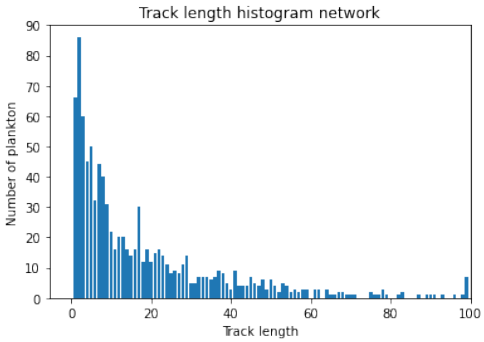


(a) Tracking of network trained on two differences. Video.

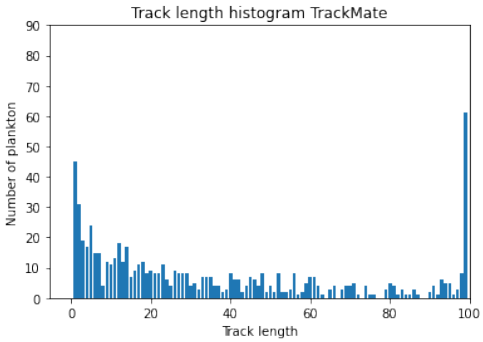


(b) Tracking of TrackMate. Video.

Figure 3.7: Comparison between one of the trained networks and TrackMate on the same frame.



(a) Track length distribution for the network.



(b) Track length distribution for TrackMate.

Figure 3.8: Track length distributions of the network and TrackMate for the methods used in figure 3.7.

In figure 3.8 the track length distribution of the two methods is shown. Both methods display a similar shape with an initial high number of plankton for short track lengths with a decreasing number of plankton for increasing lengths until 100 where both have a peak. TrackMate however kept track of 61 plankton while the network kept track of 7 plankton throughout the video. In figure 3.9 the number of found plankton in each frame is shown. The shape of the curves are similar but TrackMate finds about 40% more plankton. This difference in number of found plankton is likely due to the size and blurriness of the plankton since the particles simulated for the training data weren't blurrier than seen in the notebooks. In fact, we have gotten better results after some recent tests with blurrier and bigger particles in the training data that will be presented in an upcoming paper. Nonetheless these results indicate that the network performs fairly well on samples it hasn't specifically been trained for.

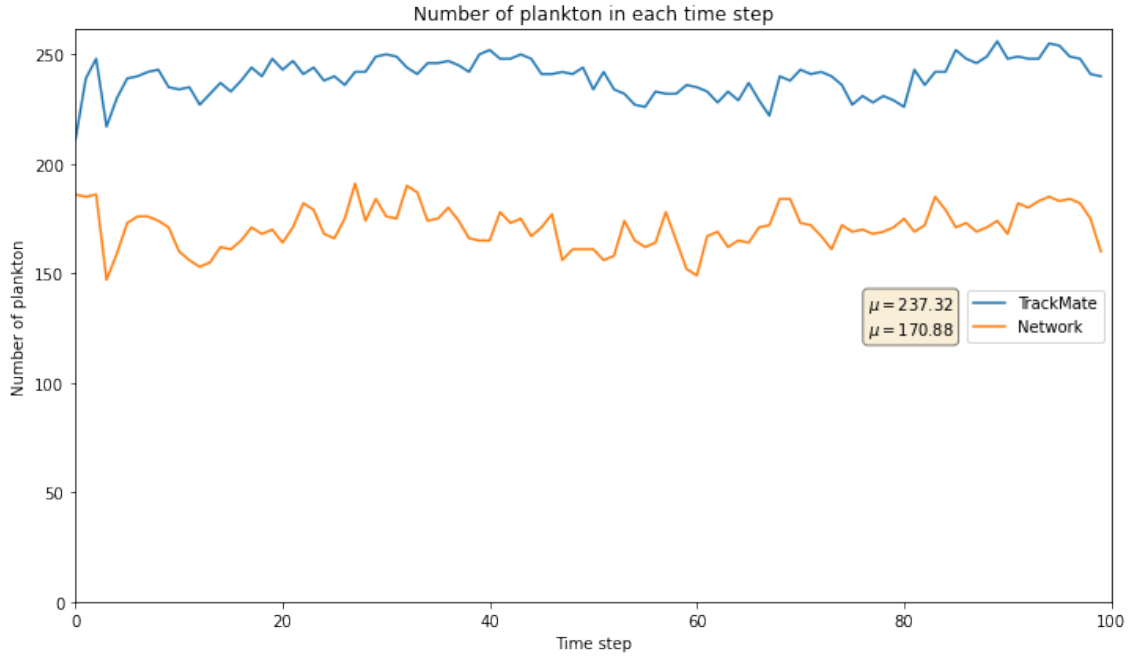
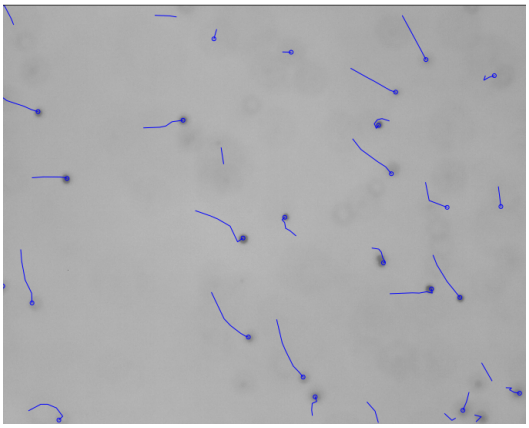


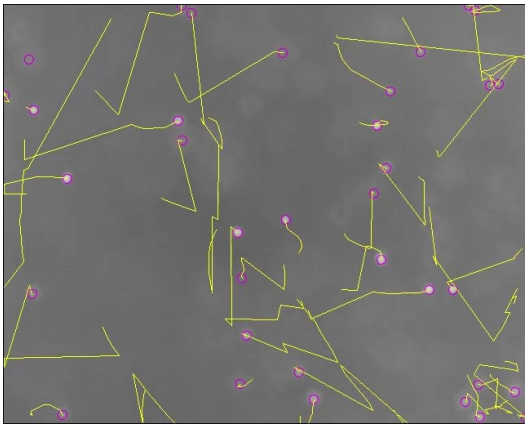
Figure 3.9: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained on the differences between the images in a sequence of length 3. The mean number of positions found by each method is displayed next to the legend.

3.4 *Alexandrium sp.* higher magnification

This plankton sample was also analyzed using a network trained on the differences between the images in a sequence of length 3, as in notebook 2.6, with the purpose to test the generality of the networks. It is the same sample as in figure 3.7 but filmed with a higher magnification. In figure 3.10 the tracking of the network and TrackMate side by side is shown. TrackMate finds more of the blurry ones compared to the network but it also links positions from different plankton into the same traces as can be seen by the many sharp angles and straight lines. In this frame the network finds 24 plankton and the algorithm finds 35.

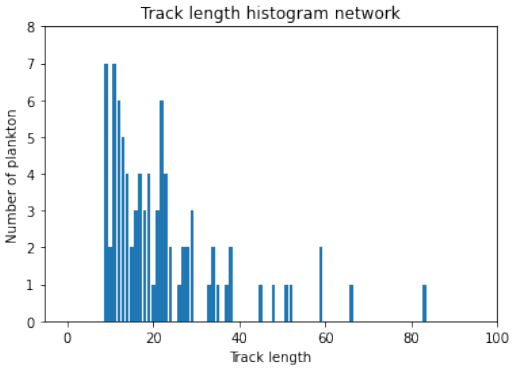


(a) Tracking of network trained on two differences. Video.

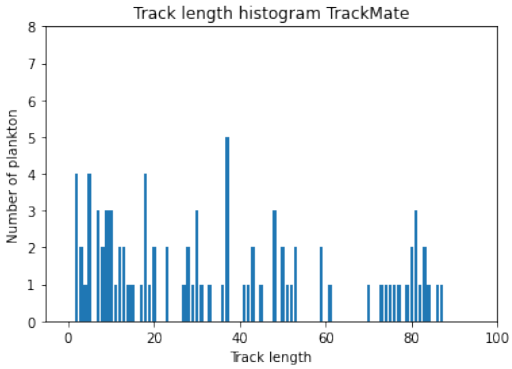


(b) Tracking of TrackMate. Video.

Figure 3.10: Comparison between one of our trained networks and TrackMate on the same frame.



(a) Track length distribution for the network.



(b) Track length distribution for TrackMate.

Figure 3.11: Track length distributions of the network and TrackMate for the methods used in figure 3.10.

In figure 3.11 the distribution of track lengths for this plankton sample is shown. 91% of the track lengths of the network are shorter than 40 frames with most being centered around a track length of 10-20 while for TrackMate this number is 61% and we have a more flat distribution. The high number of mislinking by TrackMate could explain why the track length distribution is flatter compared to the one seen in figure 3.8b. In figure 3.12 the number of recognized plankton in each frame is shown. The algorithm finds 35 while the network finds 19 on average each frame. Again the network fails to detect blurrier plankton.

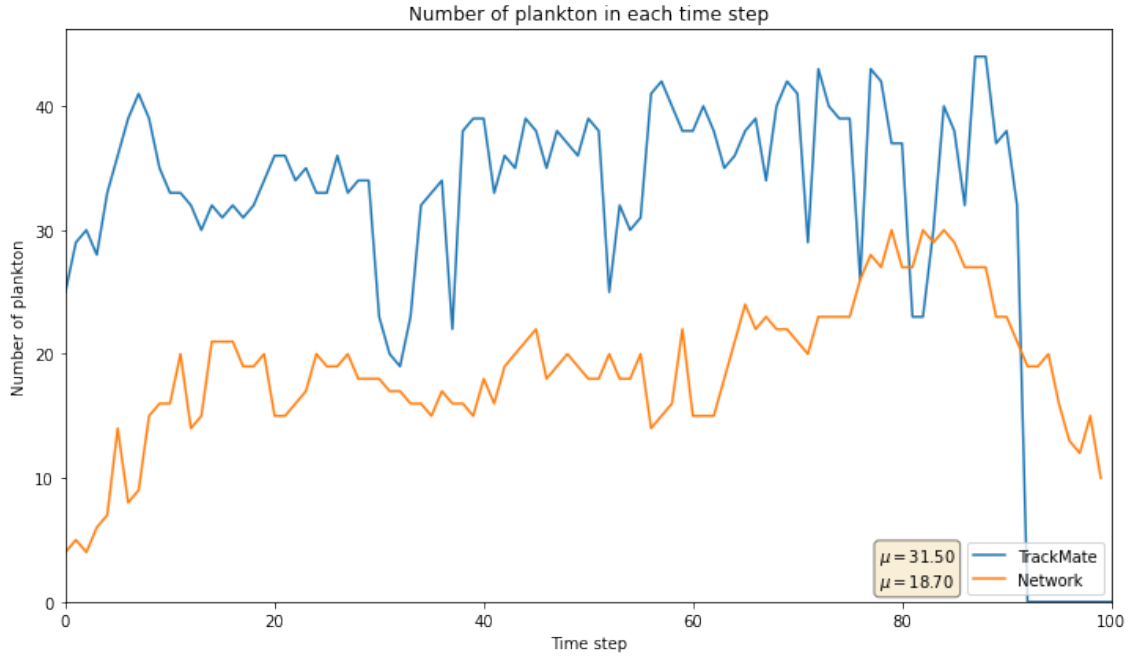
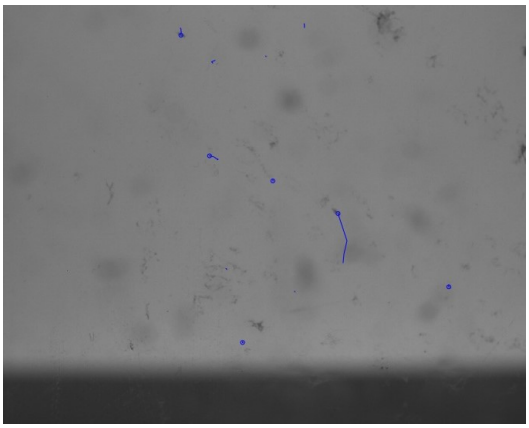


Figure 3.12: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained on the differences between the images in a sequence of length 3. The mean number of positions found by each method is displayed next to the legend.

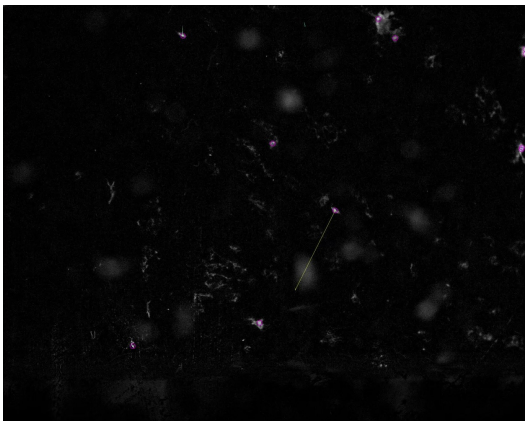
3.5 Copepods

Copepods are a group of crustecean zooplankton mostly feeding on phytoplankton constituting the majority of zooplankton in the oceans. Being food for larger organisms, they constitute the most important link between pelagic producers and higher levels in the food web [36]. Many species exhibit diel vertical migration (DVM) to avoid predators, the presence of DVM is strongly related to their size [37]. The presence of copepods has also been linked to production of toxins by phytoplankton such as *Alexandrium sp.* and *Pseudo-nitzschia seriata* [34, 38].

The network analyzing this sample has been trained on one frame at a time while also removing the running mean, as in notebook 2.5 without the cropping of the images, for the purpose of testing generality. In figure 3.13 the tracking of the network and TrackMate is shown. As seen in the video this frame contains 9 misclassifications where debris has been labeled as plankton for TrackMate while all found plankton are true positives for the network. The ones the network detects that TrackMate misses are baby copepods.

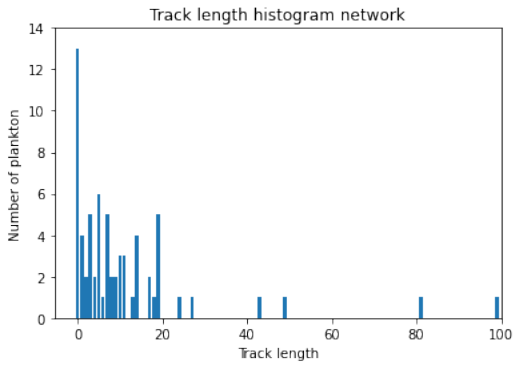


(a) Tracking of network trained on one frame at a time with running mean removed from each image. Video.

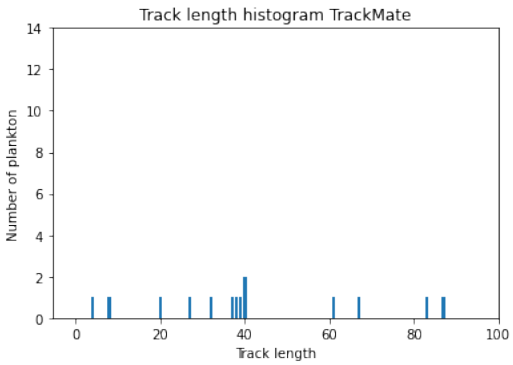


(b) Tracking of TrackMate. Video.

Figure 3.13: Comparison between one of the trained networks and TrackMate on the same frame.



(a) Track length distribution for the network.



(b) Track length distribution for TrackMate.

Figure 3.14: Track length distributions of the network and TrackMate for the methods used in figure 3.13.

In figure 3.14 the distribution of track lengths of the network and TrackMate is shown. TrackMate has 4 tracks lasting all the 100 frames while our network has most of its tracks lasting 1-20 frames. The high number of tracks lasting throughout the video for TrackMate is likely due to consistent misclassification and mislinking as seen in the video. In figure 3.15 the number of plankton found each frame by the network and TrackMate is shown. The number of identified plankton by the network each frame varies a lot compared to TrackMate. The number of misclassifications TrackMate has in figure 3.13 indicate that this relatively smooth curve is a result of consistent misclassification with the peaks being a result of actual plankton. The lower amount of misclassification and higher amount of found baby copepods by

the network indicate that the network can achieve higher sensitivity for plankton without increased risk for misclassification.

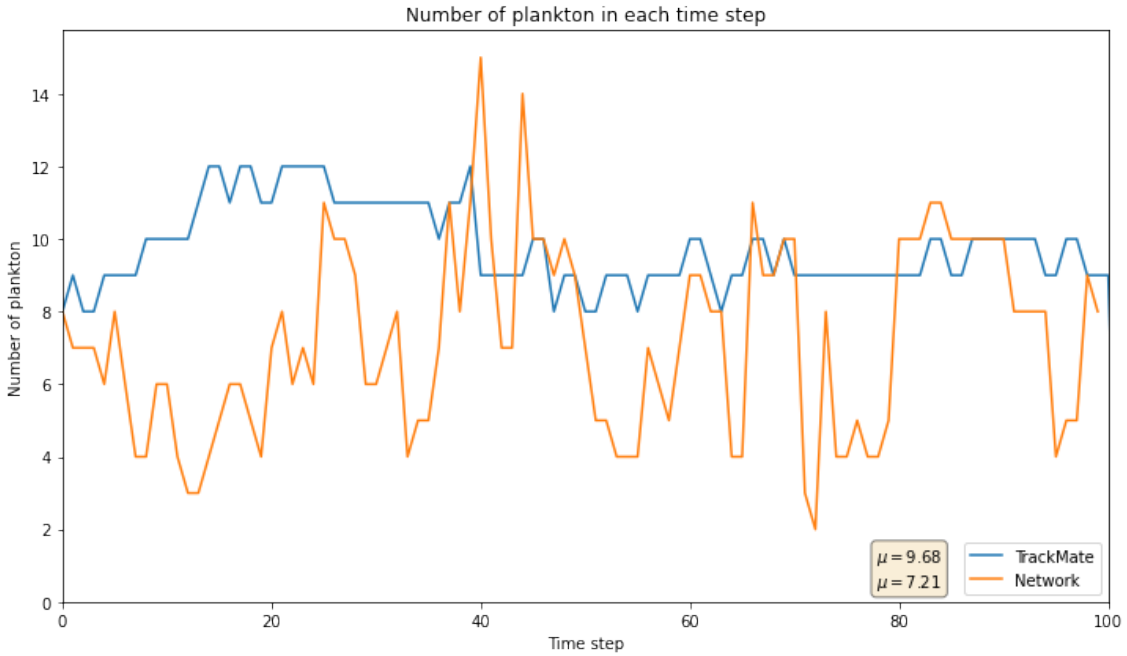
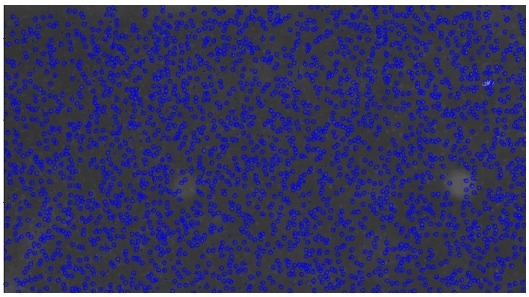


Figure 3.15: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained frame by frame with the running mean removed. The mean number of positions found by each method is displayed next to the legend.

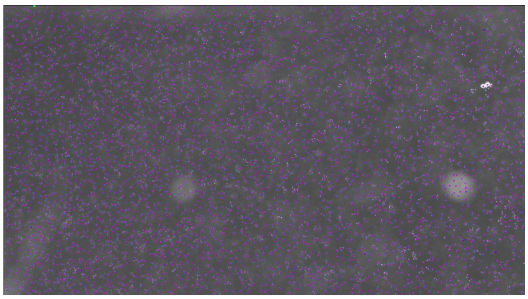
3.6 *Oxyrrhis marina*

Oxyrrhis marina is a heterotrophic dinoflagellate commonly cultured and used experimentally to study planktonic processes, similar to rats and mice. This is because they are easily obtained, identified and reared, but also because they have unusual cytological and genetical properties making them interesting subjects for studying evolutionary patterns and genome organization. Their size is typically 20-30µm in length. [39, 40]

The network used to analyze this sample is the one used in figure 3.1a trained to differentiate between small and large plankton frame by frame, as in notebook 2.4. For this sample the segmentation of the "small plankton" was used. To compensate for our network being trained on black particles and this being a dark-field microscopy the images were inverted by changing their sign. In figure 3.16 a comparison between the tracks made from TrackMate and our network is shown.

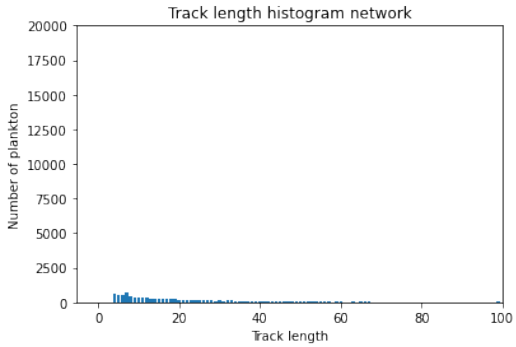


(a) Tracking of network trained frame by frame. Video.

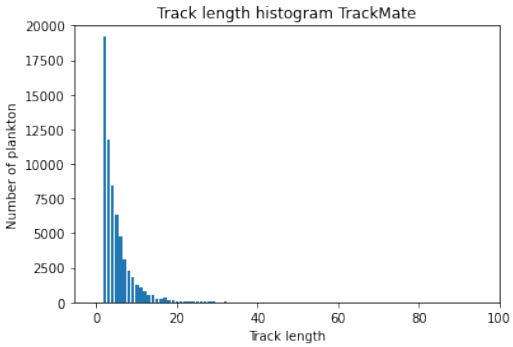


(b) Tracking of TrackMate. Video.

Figure 3.16: Comparison between one of the trained networks and TrackMate on the same frame.



(a) Track length distribution for the network.



(b) Track length distribution for TrackMate.

Figure 3.17: Track length distributions of the network and TrackMate for the methods used in figure 3.16.

In figure 3.17 the track length distribution of the two methods is shown. It looks like the network performs better but it is impossible to tell whether this is true or not since such a plankton-dense sample is prone to mislinking. Note that the histogram of the network has the same axes as the histogram for TrackMate, the highest bar is ≈700 plankton high. As seen in both figure 3.17 and 3.18 the algorithm, on average finding 3173 plankton per frame, finds more plankton in the images than the network averaging 2363 plankton each frame. This could be explained by the fact that simply inverting a dark-field image might not be a good enough estimation of a bright-field image for the network to handle it well. Other things being different between the simulated images and sample is the difference in density of plankton, the difference in size and the difference in contrast of the plankton and the background.

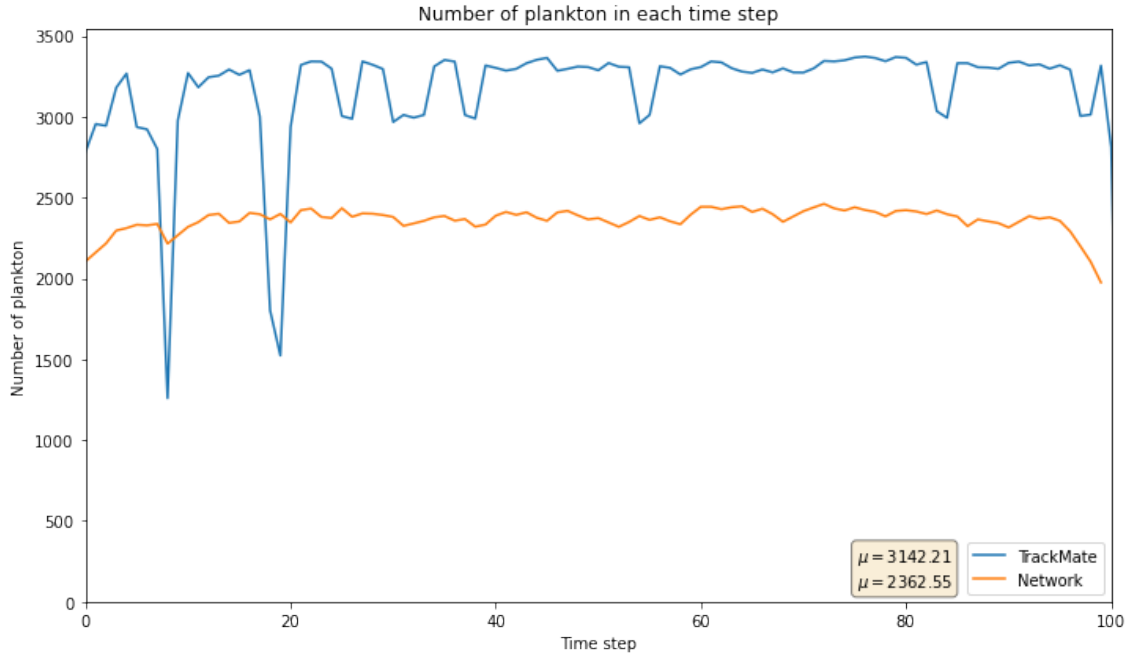
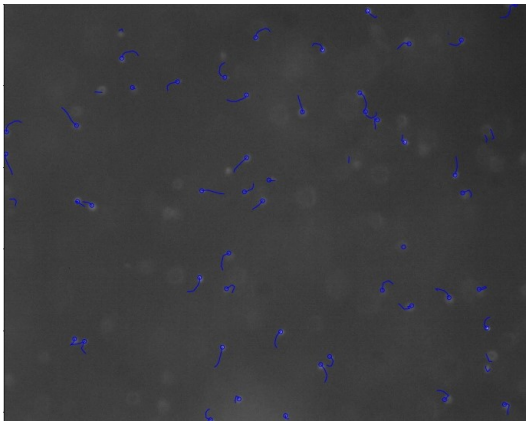


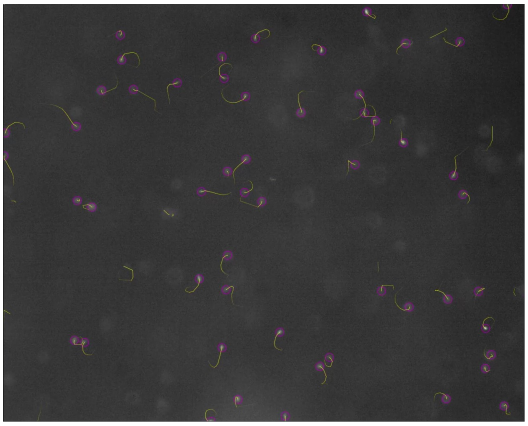
Figure 3.18: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained frame by frame. The mean number of positions found by each method is displayed next to the legend.

3.7 *Oxyrrhis marina* higher magnification

For this sample a network trained on the differences between the images has been used as shown in notebook 2.6, the same network and weights as in figure 3.1c. In figure 3.19 in the comparison between the network and TrackMate they give an even performance with TrackMate picking up slightly more plankton.

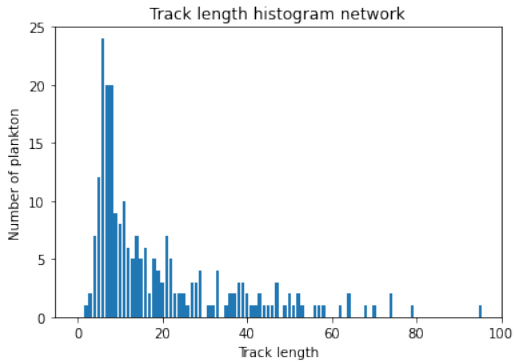


(a) Tracking of network trained on the differences between three images in sequence. The order of subtraction was reversed to invert the images.

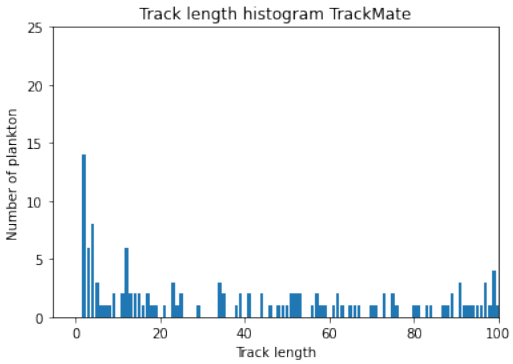


(b) Tracking of TrackMate.

Figure 3.19: Comparison between one of the trained networks and TrackMate on the same frame.



(a) Track length distribution for the network. Video.



(b) Track length distribution for TrackMate. Video.

Figure 3.20: Track length distributions of the network and TrackMate for the methods used in figure 3.19.

It can be seen in figure 3.20 that the tracks produced by the network are more densely distributed at lengths between 0-30 time steps while TrackMate has more tracks longer than 60 time steps. In figure 3.21 the number of plankton caught in each frame by the network and TrackMate is shown. Their performance is very similar with means at 46 and 51 identified plankton for the network and the algorithm. This could mean that a change of sign is enough to close the gap between a dark-field and bright-field image through the eyes of our network, in which case this could be implemented when generating training data. This can also mean that the difference in performance in figure 3.16 is due to the other things mentioned.

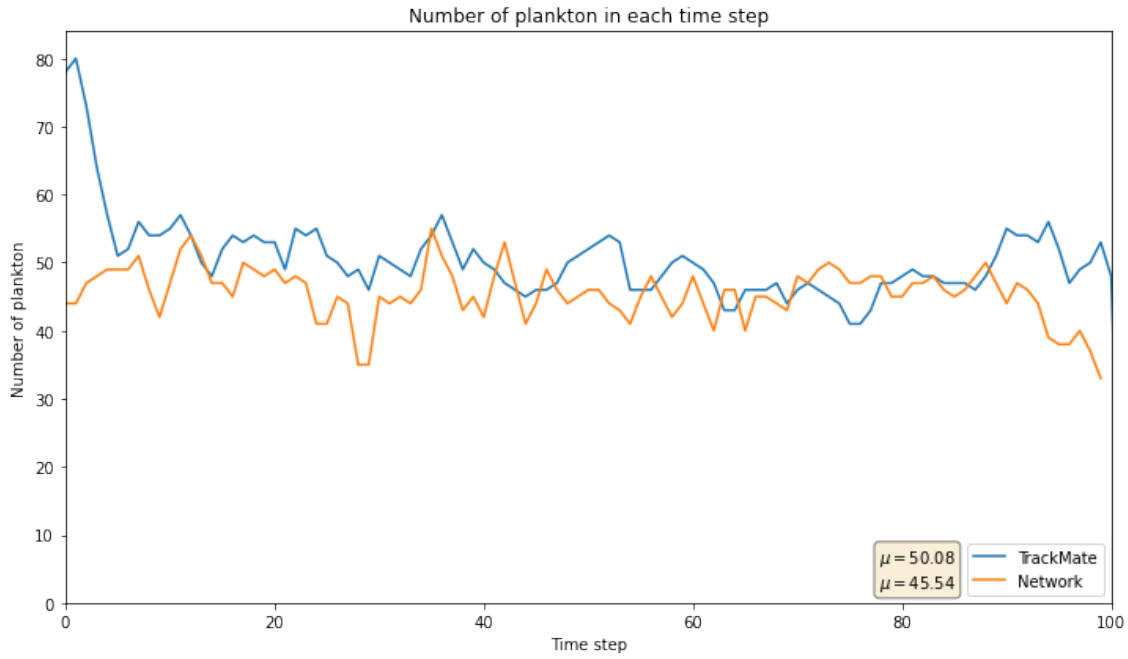


Figure 3.21: The number of found positions in each frame by the network and TrackMate. **Blue line**: Trackmate. **Orange line**: Network trained on the differences in a sequence of length 3. The mean number of positions found by each method is displayed next to the legend.

4
Conclusions

We have used DeepTrack 2.0 to simulate particles with features chosen to resemble images of plankton, not taking into account organelles or other heterogeneities in or on the plankton. These simulated images were used in different ways to build training sets which were then used to train U-nets. The created network-based software outperforms the algorithm-based software TrackMate, both with regards to number of found particles and linking of their positions, on the plankton samples the training data was made to resemble. The better performance of linking the particles can likely be attributed to the network finding more plankton but more importantly also having fewer misclassifications which makes the linking process easier. This can be seen in figure 3.1, and with regards to misclassifications especially in figures 3.4 and 3.13 where TrackMate makes many misclassifications. This shows that DeepTrack 2.0 with the added functions advantageously can be used to train networks with the purpose of tracking plankton.

The trained networks were tested on other samples than the ones the training data was designed for to test how well the networks generalize to samples they are unprepared for. The results of the networks vary from outperforming TrackMate (figure 3.13), to being similar (figure 3.19) and to being outperformed by TrackMate (figures 3.7, 3.10 and 3.16) with regards to number of found plankton. The reason why the network outperforms TrackMate on tracking the copepods is likely due to this sample offering the same challenges as the sample of the salmon lice (i.e. a messy background with few plankton moving in an irregular fashion). This results in TrackMate mislabeling debris as plankton while our software is resistant to this type of error since we remove the running mean from each frame. The irregular motion of the copepods could also add to the advanced linking algorithms of TrackMate backfiring since they try to predict the next position based on previous positions. However, this doesn't mean that the network-based method would mislabel the debris as plankton if the background wasn't removed, since we have shown in figure 3.1 that it is possible to train a network to only detect things moving or visually resembling plankton. The samples where TrackMate outperformed our software are the ones containing especially small or blurry plankton since the networks hasn't been trained for these features. However, some initial tests has shown that training a network on blurry plankton increases the performance on these samples (these results will be shown in a later paper). This shows that the networks are able to generalize fairly well to samples deviating from the training set but not necessarily to a degree higher than TrackMate in its current stage.

We tried a few differently structured image stacks in the training sets while tracking the initial samples and they allowed the networks to pick up different features. This can be seen in figure 3.1. The networks used in images 3.1a and 3.1b were mostly trained to segment particles visually resembling plankton and thus they label all black dots as plankton, rarely even debris. The networks in images 3.1c and 3.1d had a higher emphasis on finding what moves between the images and they find the faster moving small plankton as well as some very blurry big plankton. From this we conclude that a network is highly adaptable to learn different things depending on how the training data is structured. This knowledge can be utilized to further influence the network to look for the information most easily attainable in the sample.

5
Outlook

Functions and features to add and build on in this project for a more complete software.

5.1 Short term

Functions and features that could likely be added with relative ease.

5.1.1 Blurrier and bigger plankton

As can be seen in figure 3.7 the network fails to find the plankton that are very out of focus, this is probably because the network hasn't been trained to do so. Adding the option to blur the particles would improve the classification on these images.

5.1.2 Add debris

To further increase the resistance to misclassification by the network it could be beneficial to add noise to the background of the simulations. This would also let us increase the sensitivity to particles allowing us to detect more plankton without an increase in misclassifications.

5.1.3 Quantifying results

Currently we have few ways to measure the results and compare them with TrackMate (or other software). One very simple way to quantify results is to feed the segmented images as input to TrackMate to eliminate differences of tracing algorithms. [21] In the initial stages of development it is easy to determine what works or not, but as our software improves it becomes harder to see which of two working results is better. Therefore ways to quantify the results becomes key to perfecting the software.

5.2 Long term

Functions and features to add that likely requires more effort.

5.2.1 Make the output of the network separate the particles

When two particles overlap in the image their segmentation will be a white blob consisting of the two circles conjoined because this is what the label looks like if two simulated particles overlap. It would be preferred if they could be separated somehow, one first approach to this could be to make a separation of the particles in the label of the simulation. [41]

5.2.2 Dark field microscope

Sometimes a dark field microscope is used to analyze a sample. Dark field microscope simulations currently isn't an option in the software, but can to some degree be dealt with by inverting the image by putting a minus-sign in front of it. It might be beneficial to add a digital dark field microscope to DeepTrack 2.0, but it probably isn't a top priority since we could get good results by simply inverting the images. [42]

5.2.3 Attention maps

Transformer networks has shown promising results in image recognition where it makes the network "pay attention" to areas of importance and suppress irrelevant or misleading areas. In our case an "area of importance" would be the latest position a plankton was found in and an area of irrelevance would be a chunk of debris for instance, it would be interesting to try to integrate an attention network with the U-Net. [43]

5.2.4 LSTM-unit for sequences

The plankton don't always move between each and every frame in a video making the method of using the differences between frames useless, but until it moves it stays at the position it moved to in earlier an frame. If one could save this position as input to the current frame where it is stationary it could vastly improve the performance of the network. This could be done if the differences between the frames where used as input to an LSTM-unit which then could be trained to remember where the plankton moved to most recently and update this position throughout the video. [44]

5.2.5 RNN to assign positions of plankton

Currently the output of the U-Net is treated using algorithms to extract the positions. Algorithms are also used to create the tracks from these positions and assign them to separate plankton. Using an RNN or some other type of network with memory to deal with this could improve the extraction of positional data, since the next position of a plankton depends on the previous one. [45]

5.2.6 Automatize simulation of particles

One big drawback with these methods is that the user has to select parameters and functions to make the simulated particles visually similar to the plankton. One way to automatize this could be to use a stochastic optimization algorithm, eg. genetic algorithm or particle swarm optimization. The function to optimize would then be a function calculating the difference between a simulated plankton and a real plankton. The input to this function would be all parameters used to change the appearance of a simulated particle and a few images of plankton. The manual part of this solution would be to crop out a few plankton from an image of the sample. [46]

Bibliography

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[2] Andrew S. Brierley. Plankton. *Current Biology*, 27(11):R478–R483, 2017.

[3] Colomban De Vargas, Stéphane Audic, Nicolas Henry, Johan Decelle, Frédéric Mahé, Ramiro Logares, Enrique Lara, Cédric Berney, Noan Le Bescot, Ian Probert, et al. Eukaryotic plankton diversity in the sunlit ocean. *Science*, 348(6237), 2015.

[4] Christopher P Lynam, Mark J Gibbons, Bjørn E Axelsen, Conrad AJ Sparks, Janet Coetzee, Benjamin G Heywood, and Andrew S Brierley. Jellyfish overtake fish in a heavily fished ecosystem. *Current biology*, 16(13):R492–R493, 2006.

[5] Jeffrey M Leis. Are larvae of demersal fishes plankton or nekton? *Advances in marine biology*, 51:57–141, 2006.

[6] Yadigar Sekerci and Sergei Petrovskii. Mathematical modelling of plankton–oxygen dynamics under the climate change. *Bulletin of mathematical biology*, 77(12):2325–2353, 2015.

[7] Paul Falkowski. Ocean science: the power of plankton. *Nature*, 483(7387):S17–S20, 2012.

[8] Mark C Benfield, Philippe Grosjean, Phil F Culverhouse, Xabier Irigoien, Michael E Sieracki, Angel Lopez-Urrutia, Hans G Dam, Qiao Hu, Cabell S Davis, Allen Hansen, et al. Rapid: research on automated plankton identification. *Oceanography*, 20(2):172–187, 2007.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[10] Saga Helgadottir, Aykut Argun, and Giovanni Volpe. Digital video microscopy enhanced by deep learning. *Optica*, 6(4):506–513, Apr 2019.

[11] Frank Cichos, Kristian Gustavsson, Bernhard Mehlig, and Giovanni Volpe. Machine learning for active matter. *Nature Machine Intelligence*, 2(2):94–103, 2020.

[12] Thomas Kerr, James R Clark, Elaine S Fileman, Claire E Widdicombe, and Nicolas Pugeault. Collaborative deep learning models to handle class imbalance in flowcam plankton imagery. *IEEE Access*, 8:170013–170032, 2020.

[13] Ketil Malde and Hyeongji Kim. Beyond image classification: zooplankton identification with deep vector space embeddings. *arXiv preprint arXiv:1909.11380*, 2019.

[14] Simon-Martin Schröder, Rainer Kiko, and Reinhard Koch. Morphocluster: Efficient annotation of plankton images by clustering. *Sensors*, 20(11):3060, 2020.

[15] Alessandra Lumini, Loris Nanni, and Gianluca Maguolo. Deep learning for plankton and coral classification. *Applied Computing and Informatics*, 2020.

[16] Haiyong Zheng, Ruchen Wang, Zhibin Yu, Nan Wang, Zhaorui Gu, and Bing Zheng. Automatic plankton image classification combining multiple view features via multiple kernel learning. *BMC bioinformatics*, 18(16):1–18, 2017.

[17] Benjamin Midtvedt, Saga Helgadottir, Aykut Argun, Jesús Pineda, Daniel Midtvedt, and Giovanni Volpe. Quantitative digital microscopy with deep learning. *arXiv preprint arXiv:2010.08260*, 2020.

[18] Clemens Bechinger, Roberto Di Leonardo, Hartmut Löwen, Charles Reichhardt, Giorgio Volpe, and Giovanni Volpe. Active particles in complex and crowded environments. *Reviews of Modern Physics*, 88(4):045006, 2016.

[19] Suyog Dutt Jain, Bo Xiong, and Kristen Grauman. Fusionseg: Learning to combine motion and appearance for fully automatic segmentation of generic objects in videos. In *2017 IEEE conference on computer vision and pattern recognition (CVPR)*, pages 2117–2126. IEEE, 2017.

[20] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] Jean-Yves Tinevez, Nick Perry, Johannes Schindelin, Genevieve M Hoopes, Gregory D Reynolds, Emmanuel Laplantine, Sebastian Y Bednarek, Spencer L Shorte, and Kevin W Eliceiri. Trackmate: An open and extensible platform for single-particle tracking. *Methods*, 115:80–90, 2017.

[22] Hui Kong, Hatice Cinar Akakin, and Sanjay E. Sarma. A generalized laplacian of gaussian filter for blob detection and its applications. *IEEE Transactions on Cybernetics*, 43(6):1719–1733, 2013.

[23] Getting started with trackmate. `https://imagej.net/Getting_started_with_TrackMate.html#Choosing_a_detector`. Accessed: 2021-06-01.

[24] Khuloud Jaqaman, Dinah Loerke, Marcel Mettlen, Hirotaka Kuwata, Sergio Grinstein, Sandra L Schmid, and Gaudenz Danuser. Robust single-particle tracking in live-cell time-lapse sequences. *Nature methods*, 5(8):695–702, 2008.

[25] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.

[26] Trackmate algorithms. `https://imagej.net/TrackMate_algorithms.html#Principle`. Accessed: 2021-06-01.

[27] John O Corliss. *The ciliated protozoa: characterization, classification and guide to the literature*. Elsevier, 2016.

[28] Anna Arias, Erik Selander, Enric Saiz, and Albert Calbet. Predator chemical cue effects on the diel feeding behaviour of marine protists. *Microbial Ecology*, pages 1–9, 2021.

[29] Sabine Agatha. Morphology and ontogenesis of novistrombidium apsheronicum nov. comb. and strombidium arenicola (protozoa, ciliophora): a comparative light microscopical and sem study. *European Journal of Protistology*, 39(3):245–266, 2003.

[30] Tânia Gomes, Ana Catarina Almeida, and Anastasia Georgantzopoulou. Characterization of cell responses in rhodomonas baltica exposed to pmma nanoplastics. *Science of the Total Environment*, 726:138547, 2020.

[31] Martin J Fraunholz, Juergen Wastl, Stefan Zauner, Stefan A Rensing, Margitta M Scherzinger, and Uwe-G Maier. The evolution of cryptophytes. In *Origins of algae and their plastids*, pages 163–174. Springer, 1997.

[32] SC Johnson and LJ Albright. Development, growth, and survival of lepeophtheirus salmonis (copepoda: Caligidae) under laboratory conditions. *Journal of the Marine Biological Association of the United Kingdom*, 71(2):425–436, 1991.

[33] PA Heuch, JR Nordhagen, and TA Schram. Egg production in the salmon louse [lepeophtheirus salmonis (krøyer)] in relation to origin and water temperature. *Aquaculture Research*, 31(11):805–814, 2000.

[34] Erik Selander, Peter Thor, Gunilla Toth, and Henrik Pavia. Copepods induce paralytic shellfish toxin production in marine dinoflagellates. *Proceedings of the Royal Society B: Biological Sciences*, 273(1594):1673–1680, 2006.

[35] Shauna Murray, Uwe John, Henna Savela, and Anke Kremp. 4 alexandrium spp.: genetic and ecological factors influencing saxitoxin production and proliferation. In *Climate Change and Marine and Freshwater Toxins*, pages 133–166. De Gruyter, 2020.

[36] Jan Heuschele and Erik Selander. The chemical ecology of copepods. *Journal of plankton research*, 36(4):895–913, 2014.

[37] GC Hays, CA Proctor, AWG John, and AJ Warner. Interspecific differences in the diel vertical migration of marine copepods: the implications of size, color, and morphology. *Limnology and Oceanography*, 39(7):1621–1629, 1994.

[38] E Selander, EC Berglund, P Engström, F Berggren, J Eklund, S Harðardóttir, N Lundholm, W Grebner, and MX Andersson. Copepods drive large-scale trait-mediated effects in marine plankton. *Science advances*, 5(2):eaat5096, 2019.

[39] David JS Montagnes, Chris D Lowe, Emily C Roberts, Mark N Breckels, Dan E Boakes, Keith Davidson, Patrick J Keeling, Claudio H Slamovits, Michael Steinke, Zhou Yang, et al. An introduction to the special issue: Oxyrrhis marina, a model organism? *Journal of Plankton Research*, 33(4):549–554, 2011.

[40] Chris D Lowe, Patrick J Keeling, Laura E Martin, Claudio H Slamovits, Phillip C Watts, and David JS Montagnes. Who is oxyrrhis marina? morphological and phylogenetic studies on an unusual dinoflagellate. *Journal of Plankton Research*, 33(4):555–567, 2011.

[41] Ryoma Bise, Kang Li, Sungeun Eom, and Takeo Kanade. Reliably tracking partially overlapping neural stem cells in dic microscopy image sequences. In *MICCAI Workshop on OPTIMHisE*, volume 5, pages 67–77, 2009.

[42] J Pizarro, PL Galindo, E Guerrero, A Yáñez, MP Guerrero, A Rosenauer, DL Sales, and SI Molina. Simulation of high angle annular dark field scanning transmission electron microscopy images of large nanostructures. *Applied Physics Letters*, 93(15):153107, 2008.

[43] Saumya Jetley, Nicholas A Lord, Namhoon Lee, and Philip HS Torr. Learn to pay attention. *arXiv preprint arXiv:1804.02391*, 2018.

[44] Assaf Arbelle and Tammy Riklin Raviv. Microscopy cell segmentation via convolutional lstm networks. In *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pages 1008–1012. IEEE, 2019.

[45] Roman Spilger, Andrea Imle, Ji-Young Lee, Barbara Mueller, Oliver T Fackler, Ralf Bartenschlager, and Karl Rohr. A recurrent neural network for particle tracking in microscopy images using future information, track hypotheses, and multiple detections. *IEEE Transactions on Image Processing*, 29:3681–3694, 2020.

[46] Mattias Wahde. *Biologically inspired optimization methods: an introduction*. WIT press, 2008.