CHALMERS
UNIVERSITY OF TECHNOLOGY

# Tracking marine microorganisms using deep learning

Development of a software for tracking plankton based on predictions from neural networks

Master's thesis in Physics

Hillevi Wachtmeister

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

# Tracking marine microorganisms using deep learning

Development of a software for tracking plankton based on predictions from neural networks

Hillevi Wachtmeister

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2020

Tracking marine microorganisms using deep learning
Development of a software for tracking plankton based on predictions from neural networks
HILLEVI WACHTMEISTER

© HILLEVI WACHTMEISTER, 2020.

Supervisor: Daniel Midtvedt, Assistant Professor, Soft Matter Lab, Physics Department, Gothenburg University

Examiner: Giovanni Volpe, Professor & Group Leader, Soft Matter Lab, Physics Department, Gothenburg University

Master's Thesis 2020
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Plankton, more or less real. Image by Alizz Axberg, cartoon by Stephen Hillenburg from `https://www.pngfuel.com/free-png/oasuz`.

Tracking marine microorganisms using deep learning
Development of a software for tracking plankton based on predictions from neural networks
Hillevi Wachtmeister
Department of Physics
Chalmers University of Technology

## Abstract

The goal of this project is to develop a software that can be used to study swimming patterns of marine microorganisms. The software is based on a neural network, which is trained to recognise different types of plankton. The predictions from the network are then used to find the positions of the plankton and track their movements.

The project is divided into two parts. First, videos containing only one type of plankton, *Lingulodinium polyedra* and *Alexandrium tamarense*, respectively, are analyzed. A type of neural network, called U-net, is trained to segment the input images into background and plankton sections. From the segmented images, positions can be obtained and then connected to form a trajectory for each plankton. The drift of the plankton movements is calculated and subtracted from the trajectories, and finally the speed and net displacement are calculated. The results from the single plankton experiments are compared to a previous analysis that was made using an algorithmic method.

Secondly, videos containing two types of plankton are analyzed containing the phytoplankton *Strombidium arenicola* and *Rhodomonas baltica.* The segmented images, obtained from the U-net, consists of an additional plankton section for the second type of plankton present in the experiment.

The analysis of the single plankton experiments yields longer and fewer trajectories using the U-net method, compared to the previous results using the algorithmic method. This indicates that the U-net method detects plankton at more positions, and is therefore able to track each plankton for a longer time, compared to the algorithmic method. The multi-plankton experiments prove the network's ability to distinguish and track multiple plankton at the same time.

Keywords: deep learning, U-net, plankton, tracking, drift compensation.

## Acknowledgements

I would first of all like to thank my supervisor, Daniel Midtvedt, for all his guidance and help during this project. I would also like to thank Erik Selander from the Department of Marine Sciences, University of Gothenburg, for his feedback on the software and insight in the biological application. Finally, I would like to thank my examiner Giovanni Volpe, for the opportunity to engage in this project. To everyone else who was in some sense involved in this project, whether it was proof reading the manuscript, calming me down or cheering me up, you have my deepest gratitude.

Hillevi Wachtmeister, Gothenburg, June 2020

# Contents

# List of Figures

# List of Algorithms

# 1 Introduction

Being able to understand and predict how things move is an important part of many sciences, in everything from nanotechnology to astrophysics. To track and follow something may seem like a simple task for the brain; one can easily imagine a cat staring at a moving laser dot on the wall, or a human following the path of a bird in the sky. Such tasks do not pose much problem for our brains. However, if we ask a computer to simulate this action, it turns out it is not that simple. ”How do we tell the computer what to follow?” and ”How will it identify the same element when it is moving between frames?” These are just two of the many questions we need to answer before we proceed.

Fortunately, there have been great progress in this field. A method developed by Crocker and Grier in 1995 [1], locates particles by finding local maxima in an image. The method TrackMate, developed by Tinevez et al.[2], is more reliant on the user finding the right filters to distinguish the particles. However, what these, and other frequently used methods, have in common is their algorithmic approach. As a consequence, these do not work well when perfect conditions, such as homogeneous lighting, spherical particles and no drifting in and out of focus, do not apply. This is especially a problem at low signal-to-noise ratios.

In the field of biology, the limits of the algorithmic tracking methods becomes apparent [3, 4]. As an example, marine microorganisms, such as plankton, quite often need to be tracked in large volumes, which makes them drift in and out of focus. Tracking different kinds of plankton in a shared volume also present a challenge.

What if the computer could learn how to recognise certain elements in an image? What if there was a way to make the computer act as a brain? Deep learning is a novel approach to the tracking problem which does not rely on algorithms. Instead of us humans giving the computer the rules for finding plankton, the computer teaches itself. The deep learning approach has previously been used with great success in the field of particle tracking [5–7].

The purpose of this project is to develop a software which, based on the predictions obtained from a deep learning network, can track different types of plankton. The predictions are obtained from the Deeptrack 2.0 framework [8], which is developed by a student at the Soft Matter Lab, University of Gothenburg. Deeptrack 2.0 makes it possible to simulate images of particles resembling plankton and train a network to recognise them. The tracking software is developed with the future users, the Department of Marine Science at University of Gothenburg, in mind, who also carried out the experiments used in this project. These experiments produced several videos of different plankton under different circumstances. The goal is to be able to analyse the swimming pattern of the plankton, such as their speed and net displacement.

The project is divided into two parts; single and multiple plankton tracking. Both experiments focused on the effects copepodamides had on the swimmingbehaviour of different phytoplankton. Copepodamides are polar lipids emitted by a group of zooplankton, copepods, which the phytoplankton can detect. The single plankton experiments uses phytoplankton, *Lingulodinium polyedra* and *Alexandrium tamarense* respectively, exposed to different concentrations of copepodamides. A network is trained on simulated images of the plankton and the predictions from the network are then used to find positions and build traces for each plankton present in the experiment. The single particle experiment had previously been analyzed using an algorithmic method and these results are compared with the ones obtained through the deep learning method.

The multiple plankton experiments uses two types of phytoplankton, *Strombidium arenicola* and *Rhodomonas baltica*. The different plankton are simulated simultaneously by the network, and traces can be built for the two species separately. The software also incorporates a drift compensation feature.

# 2 Theory

## 2.1 Machine learning

Traditionally, humans provide a computer with a set of rules, algorithms, to solve problems. Imagine the box in Figure 2.1 to be a computer with a set of algorithms. When feeding an input to the box, it will perform some tasks defined by the algorithms and produce an output. This works well when we already know how a system behaves, or if a task is simple. As an example, algorithmic approaches work well for detecting spherical particles in the same focal plane under homogeneous lighting conditions. However, when the problem becomes more complicated, like finding particles in low signal-to-noise ratios or finding non-spherical particles, it can be simpler to have the computer make its own rules [9].

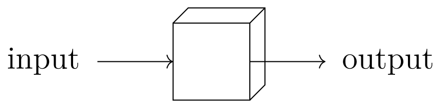


**Figure 2.1:** A simplified image of how an algorithm works. An input is first sent in to a box. The box, or algorithm, applies some function on the input and returns an output.

In machine learning, the box in Figure 2.1 is practically empty to start with. However, we already know the inputs and outputs for a number of cases. By letting the computer guess the output from each input and correcting its mistakes, the computer will learn more and more about how the input and output are connected. The goal is to provide the computer with enough known inputs and outputs such that it is finally able to understand a system well enough to predict how it will behave. Simply put, machine learning is letting the computer learn from its mistakes.

### 2.1.1 Neural networks

A neural network is a type of machine learning algorithm which takes inspiration from the biological networks of neurons found in the brain [10]. Just like a brain, a neural network consists of several neurons that are connected. An example of a neuron in a neural network can be seen in Figure 2.2. Each input is weighted differently and the output is a function of the weighted sum of the inputs, which can be written as the function below. A bias is also included in the output. The purpose of the bias is to allow the model to be more flexible in its predictions. The bias is usually set to 1.

$$\text{output} = f(\mathrm{w}_1 \cdot \text{input}_1 + \mathrm{w}_2 \cdot \text{input}_2 + \mathrm{w}_3 \cdot 1)$$

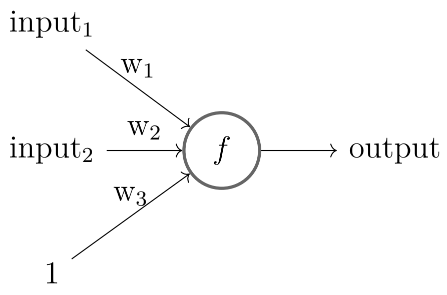


**Figure 2.2:** Schematics of a neuron with inputs, bias, weights w, activation function f and output. The output is the function $f$ of the weighted sum of the input and bias.

The function $f$ inside the neuron is called an activation function. The purpose of an activation function is to add a non-linearity to the network [11]. The operations executed by the network, like the weighted sum, are linear. However, most real world problems are not. It is therefore necessary to use a non-linear activation function since we are trying to get the network to recognise realistic problems. There are several different activation functions, some of the most used are Sigmoid, tanh and rectified linear unit (ReLU). The related functions can be seen below together with the associated curve in Figure 2.3. Note that the bias in the input allow the activation function to move right or left.

| Sigmoid | tanh | ReLU |
|---|---|---|
| $f(x) = \sigma(x) = \dfrac{1}{1+e^{-x}}$ | $\tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ | $R(x) = max(0, x)$ |

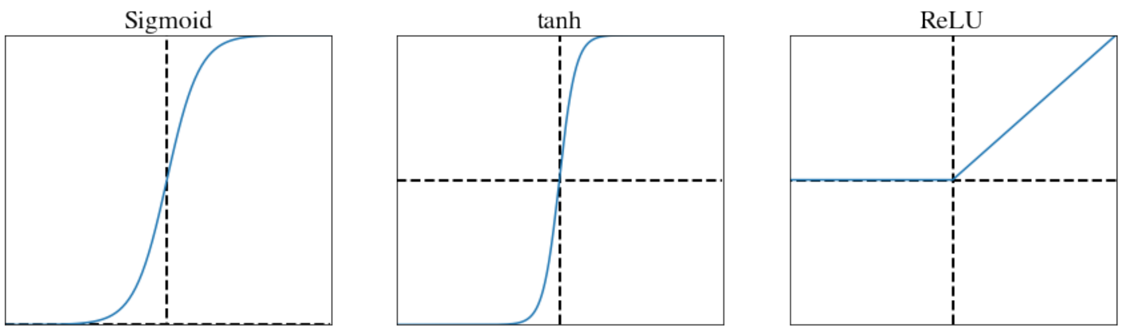


**Figure 2.3:** The curves of some of the most commonly used activation functions Sigmoid, tanh and rectified linear unit, ReLU.

In a neural network, there are several neurons which are connected through layers. Figure 2.4 shows a network with three layers. The input layer and the hidden layer have three neurons each, while the output layer have two neurons. The input, output and activation function are all fixed, which means that the only variables that can be trained by the network are the weights, which connects the neurons.

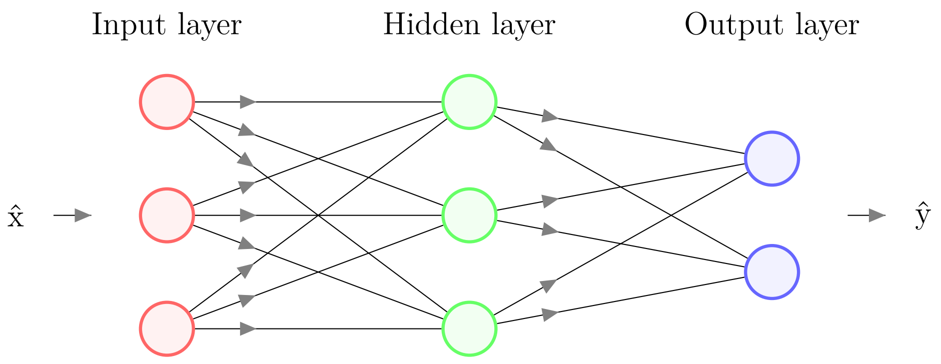


**Figure 2.4:** The forward feeding step of the neural network. An input x̂ is sent into the input layer. Every neuron in the input layer is connected to every neuron in the hidden layer. Every neuron in the hidden layer has an activation function and will return the function of the weighted sum from the input layers and the connected weights to the output layer. The output layer finally returns an output ŷ.

The learning of the network is done in two steps, which are repeated multiple times. In the beginning, the weights connecting the neurons are randomly assigned. An input vector x̂ is fed forward through the network, as seen in Figure 2.4. Note that the input layer does not perform any operations, but is simply passing the input through to the hidden layer. The nodes in the hidden layer each have an activation function and will preform this operation on the weighted sum from each of the input nodes. The hidden layer passes the new values through to the output layer, which will give an output vector ŷ. Because the weights are assigned randomly, we expect the first output to be very far some the true target $\hat{y}_0$.

The next step is to calculate the total error by comparing the output ŷ to the target $\hat{y}_0$. The total error is then propagated back through the network, as seen in Figure 2.5. During the back propagation, the weights are adjusted to reduce the error. The adjustment is done through gradient descent to minimize the error. After the weights are adjusted, a new input and output pair will be used. The new input and output will go through the same process as the first example, but the weights will be the ones adjusted from previous examples. This way, the error will slowly decrease while the network learns more and more.

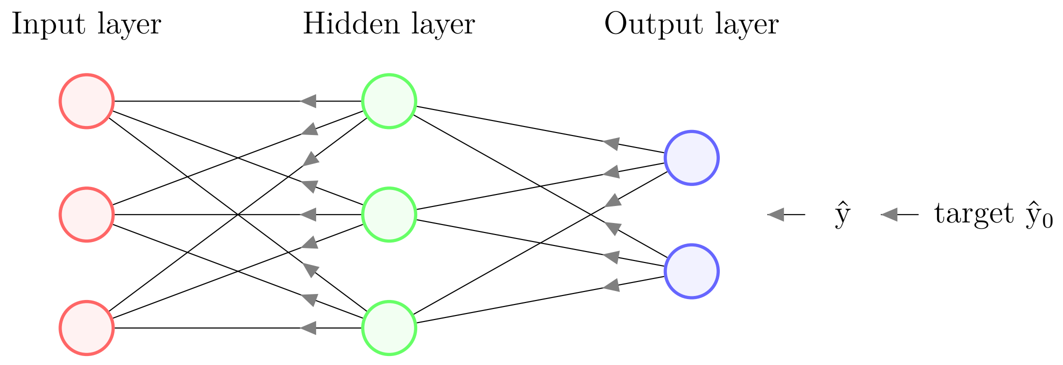


**Figure 2.5:** The back propagation step in the neural network. The total error is calculated by comparing the output ŷ and the target $\hat{y}_0$. The error is then propagated back through the network adjusting the weights connecting the neurons to reduce the error.

### 2.1.2 Convolutional neural networks

A convolutional neural network has proven to work well for image recognition and classification [12, 13]. An example of a convolutional neural network can be seen in Figure 2.6. This network consists of four operations which take us from an image to a label, these steps are explained in this section.

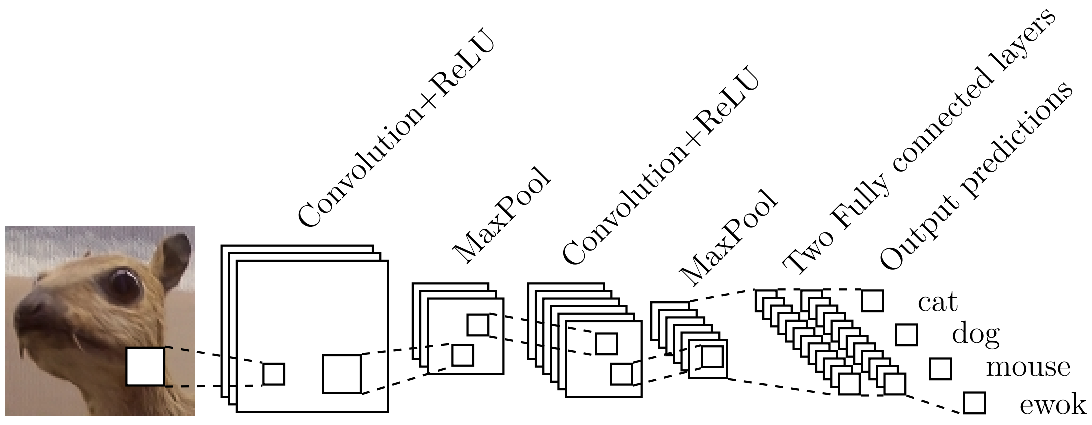


**Figure 2.6:** An example of a simple convolutional neural network consisting of two convolutional steps with three layers. Two ReLU steps, which introduces a non-linearity to the network. Two maxPooling layers downsamples the network and finally there are two fully connected layers which returns an output vector representing the labels. Original image from [14].

The first step in a convolutional neural network is a convolution. A convolution acts as a type of filter that runs over the input image to extract different features from the image [10]. There are different kinds of filters, such as edge detection filters or filters which sharpens the image. Examples of filters and their effects on the image can be seen in Figure 2.7 [15]. The filter is run over the the image with a specific step size. The size of the filter and its step size is defined by the user. However, the matrix values of the filter is learnt by the network through training.

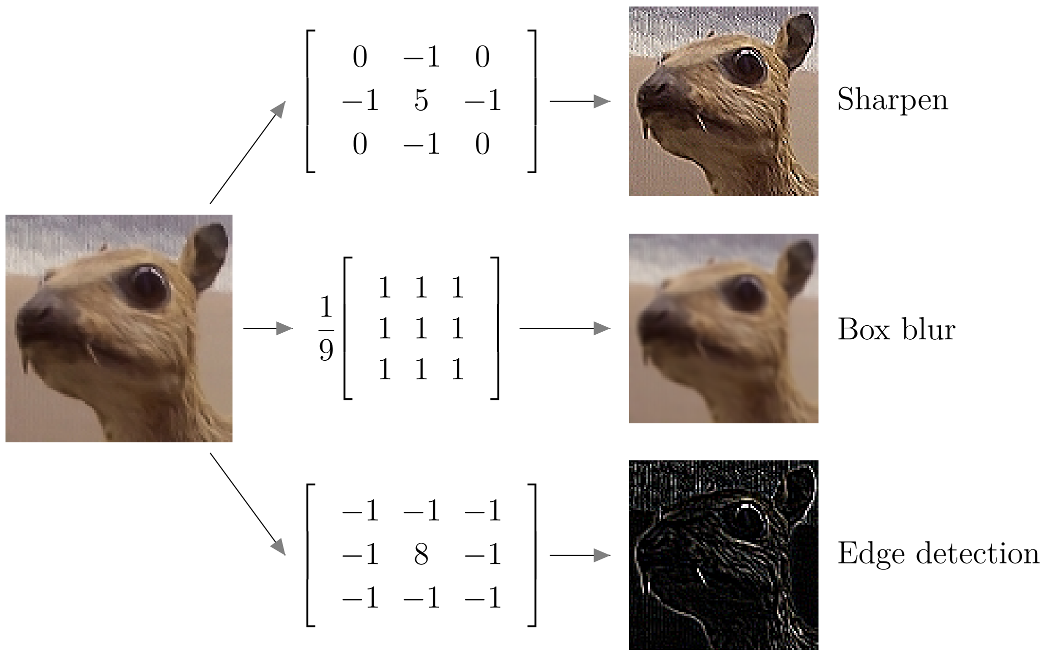

$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
Sharpen
$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
Box blur
$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
Edge detection

**Figure 2.7:** Examples of different filters and their effects on the original image. The sharpen filter sharpens the features of an image. The box blur filter takes a pixel and average it to the neighbouring pixels. The edge detection filter extracts edge features from an image. Original image from [15].

The next step is the introduction of non linearity, which comes in the form of an activation function. In the example in Figure 2.6, this step comes directly after each of the convolutional steps. The activation function used in this example, ReLU, replaces all the negative values in the image matrix with zeros. The effects a convolution and a ReLU can have on an image can be seen in Figure 2.8[14].

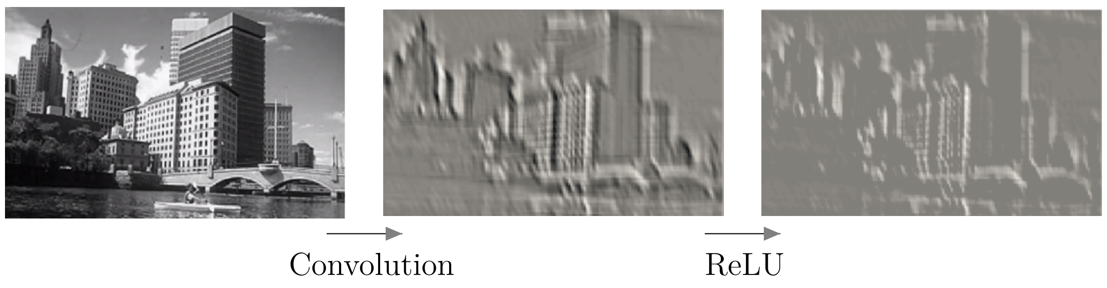


**Figure 2.8:** Example of the effects of a convolution filter on an image, followed by the effects of the activation function ReLU, which replaces all negative values with zeros. Original image from [14].

After each convolution and ReLU layers, a max pooling layer is introduced. A max pooling layer downsamples the image, an example of which can be seen in Figure 2.9. In this example, a 2x2 matrix is moving over the image with a step size of two, returning the largest number of the four numbers affected by the filter. The max pooling layer reduces the dimension of the image while keeping the most important information [10].

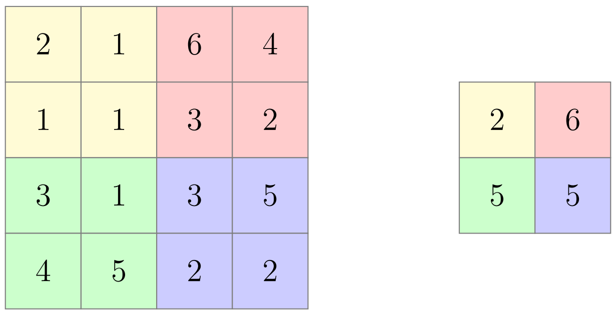


**Figure 2.9:** An example of how a max pooling layer of size 2x2 and step size 2 works on a 4x4 matrix. The maxpooling layer takes the maximum value of the 2x2 matrix as a new value.

The last steps in the convolutional neural network are the fully connected layers and the output predictions. These steps are equivalent to the fully connected layers described in section 2.1.1. In the example in Figure 2.6, the output is a vector with predictions for each label.

The trainable values in a convolutional neural network are the weights, but also the values of the elements in the convolutional filters. However, the user will have to specify the number of filters. In the example in Figure 2.6, there are three convolutional filters in each convolution step. The size and the step size of the filters will also have to be determined by the user, as well as the overall architecture of the network, i.e. how many layers there are and where to put each layer.

A convolutional neural network is trained by feeding it some known input images and output vectors. It works the same way as a simple neural network does, as described in section 2.1.1. However, when back propagating, both the weights and the matrix values in the convolutions will be adjusted to compensate for the errors.

### 2.1.3 U-net

As stated in section 2.1.2, the convolutional neural network works well for recognition and classification of images. However, a limitation with this network is segmentation [16]. In several fields, such as biology, diagnostics and self driving cars, not only the classification of the image is of interest, but also exactly where in the image an object is present [17]. In semantic segmentation, each pixel is labeled with its associated object. In Figure 2.10, every pixel is labeled with either a cat label or a background label.

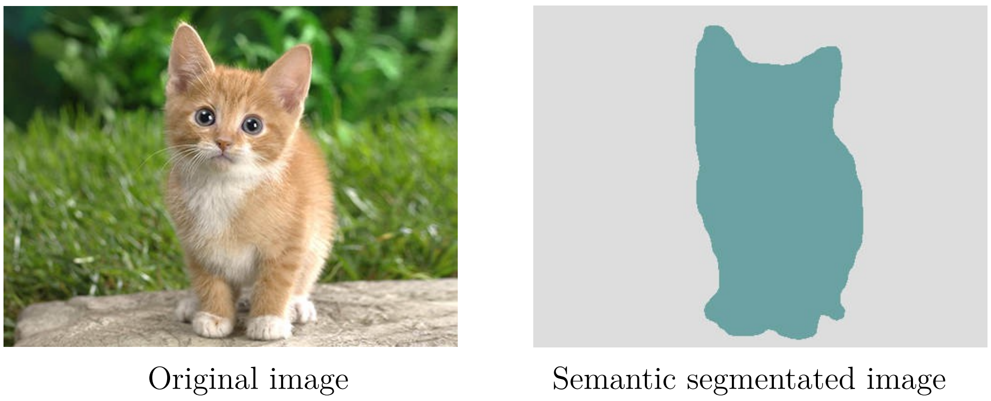


**Figure 2.10:** An image of a cat and the same image segmented into background and cat pixels. Original image from [18].

Even though there have been some success in segmentation using convolutional neural networks [19], a network that handles segmentation better than the convolutional neural network is a U-net [16]. A convolutional neural network takes an image and downsamples it to a vector, which does not necessarily give any information about the exact position of the object. A U-net, on the other hand, takes and image, converts it into a small 2D array and then converts that array back to an image. This means that every pixel in the image will be assigned a label.

The U-net has a downsampling part and an upsampling part, which are symmetrical. This gives the U-net a u-shaped architecture, hence the name. The upsampling and downsampling parts are also connected through bridges. Consequently, information earned through each downsampling step is used to upsample the image and the spatial information is maintained. An image of a U-net can be seen in Figure 2.11, where the u-shaped architecture is apparent. The left side of the network is downsampling the image, just like a convolutional neural network does. It consists of several convolutional layers, followed by ReLU functions and max pooling layers.

The upsampling part of the U-net, the right side, consists of several upsampling steps, which are followed by convolutions. An example of an upsampling process can be seen in Figure 2.12. In this example, a 2x2 image is upsampled to a 3x3 image. First, the smaller image is padded, which means that the original image is extended by adding extra elements or pixels. Usually, this is done by adding zeros

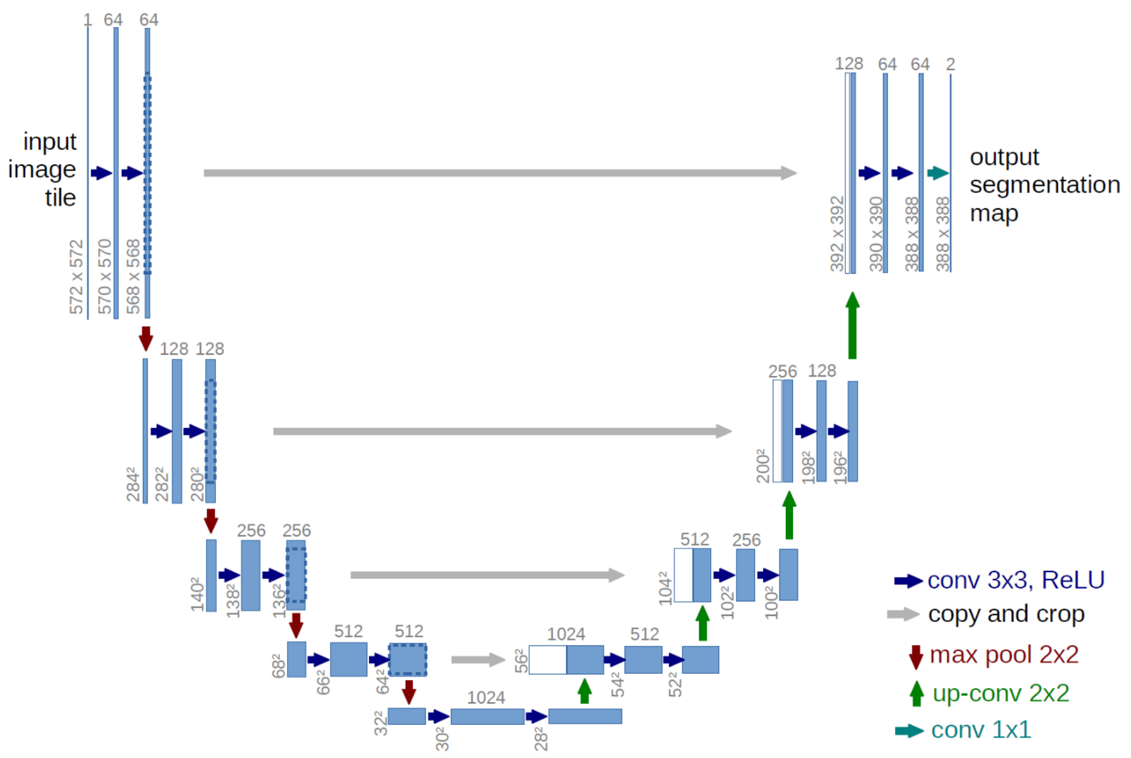


**Figure 2.11:** The U-net architecture. The left side contracts the image in the same way as a convolution network does, while the right side expands the image. The output of the U-net is one or several segmented versions of the original image. Original image from [16].

in the padding spots. Then the convolution filter is ran over the padded image and each step produces a value in the output. It is important to note that the convolutional filter used in the upsampling is the same filter used in the corresponding downsampling step. This way, the important features that were found in the downsampling are used to reconstruct the image. The last step in the U-net upsampling path is to make sure that the number of segmented images matches the number of labels.

To train the U-net, images and their known segmentation are used as inputs and outputs, respectively. To calculate the errors, each pixel is considered individually. For every pixel, we want to know the probability of it belonging to each class. Every pixel needs to lie in a class, i.e. the total probability needs to be equal to 1, and to force this, a soft max function is used. This function takes a vector, i.e the value of that pixel in each class, and normalize it into a probability distribution. The values of the vector have values within [0,1] and the total sum of the values is 1. This gives us a probability of each class for every pixel.

To calculate the total error we use a weighted cross entropy loss function:

$$WCE(p, \hat{p}) = -(p \cdot \beta_1 log(\hat{p}) + (1 - p) \cdot \beta_2 log(1 - \hat{p}))$$

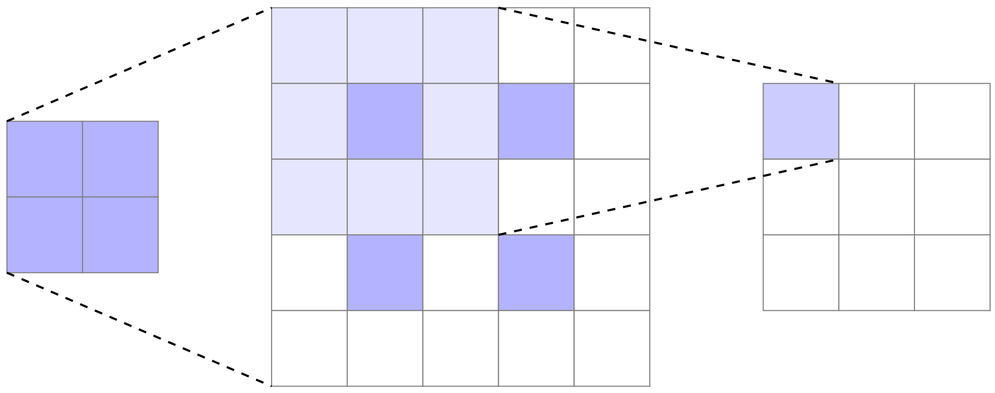


**Figure 2.12:** Upsampling of an image. This upsampling takes a 2x2 matrix and upsamples it to a 3x3 matrix by first padding it and then running a 3x3 convolution over it with a step size of one. Each step the convolution results in a new value in the final upsampled image.

In this function, $p$ is the true output, $\hat{p}$ is the predicted output, and $\beta_1$ and $\beta_2$ are the weights. This loss function is used because different classes may be more present than others, i.e. there is usually more background than objects in an image. To make sure that the network does not label all pixels as one, the network is punished differently for different predictions. For example, the punishment, or the total error, for labeling an object pixel as background is worse than labeling a background pixel as an object. Especially when segmenting images of particles, the background pixels are over represented compared to the particle pixels. To avoid the network averaging every pixel to background, we use a weighted cross entropy loss function and the weights decide how important each of the classes are.

## 2.2 Plankton

The primary movements of the plankton, on a grander scale, comes from the ocean currents. Nevertheless, this does not mean that plankton have no local movement. On the contrary, most plankton have the ability to swim and research their environment, although be it quite a limited area. The swimming pattern of the plankton can be quantified by analyzing the gross and net displacement, as well as the speed [20]. Figure 2.13 shows the gross displacement as $L$, and the net displacement as $l$. Plankton can recycle the water area they roam by reducing the net displacement of their movements, thereby reducing the rate at which predators may be encountered. However, this also means that the encounter rate of food or potential mates is also reduced. Increased swimming speed also increase the risk of running into predators [21].

Plankton can be divided into two groups, phytoplankton, the primary energy source of which is photosynthesis, and zooplankton, which survive by eating other plankton. A large group of zooplankton that preys on phytoplankton is called copepods. Copepods are blind, except for light detection, so in order to find prey and mates,

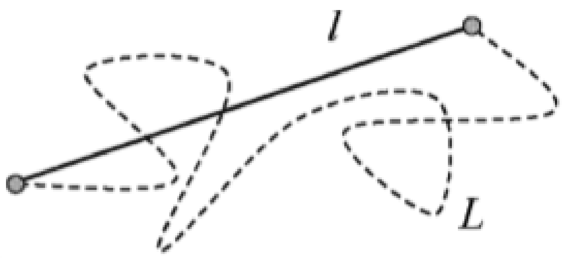


**Figure 2.13:** Net displacement, $l$, and gross displacement, $L$, of a plankton trajectory. Image from [20].

they rely on non-visual senses [22]. To find mates, Bagøien et al. [22] showed that the females send chemical signals for the males to track.

Another chemical that the copepods, both male and female, emit is a group of polar lipids called copepodamides [23]. However, the objective for emitting copepodamides is still unclear. Selander et al. speculated that the copepodamides might have something to do with the metabolism of the copepod, since the concentration of copapodamides lowered when the copepods were starved [23].

Furthermore, the phytoplankton, preys of the copepods, can detect the copepodamide and adapt their behavior to avoid getting eaten. In 2011 Selander et al. showed that the phytoplanktons changed their chain lengths when detecting copepodamides [21]. The longer the chain, the faster they can swim, which increases the risk of encountering a copepod. The longer the chain also means a larger mass of preys in the same place, which increases the risk of detection.

Phytoplankton can increase their production of algal toxins when detecting copepods, as shown by Selander et al. [23, 24]. Lindström et al. also showed the increase of bioluminescens in certain phytoplankton when exposed to copepodamides [25]. This study also showed a tendency towards longer net displacements and higher swimming velocities in single phytoplankton, when subjected to copepodamides. However, the result was not statistically significant.

It is important to note that the photosynthesis in phytoplankton is responsible for about 50% of the worlds oxygen production [26]. Algal toxins are also something that can greatly affect humans, both directly but also through our fish and shellfish consumption [27]. An understanding of how phytoplankton behave when threaten is therefore of great importance.

# 3
# Methods

## 3.1 Training the network

A network with a U-net architecture is trained to recognise different types of plankton. In order to provide sufficient data for the training process, the training is performed on simulated images of particles, made to resemble plankton as closely as possible. The following sections will describe how these images are simulated and how the model is evaluated on both simulated and experimental data.

### 3.1.1 Simulating images

In the real images, the plankton are blurry and out of focus. This makes them look like dots, as seen in Figure 3.1. This is used to our advantage, because it is easier to simulate a geometric dot than it is to simulate an organism with legs and internal structures. The simulated images are generated through several steps, which are explained in this section. Both the simulations and the training of the network are done using the software Deeptrack 2.0 [8].

The first step is to define what we want to simulate. This is done by specifying several parameters about the particle or plankton. The particle is assigned a random position, x and y coordinates, withing the range of the size of the image plane. This gives each particle a new, random position on the image. A distance from the focal plane is defined, which makes the model able to recognise particles, even if they are not at the perfect distance from the camera. This distance is given as a range, above and below the focal plane. The plankton are simulated as ellipses with two radii, which are both given as ranges, to simulate different sizes and shapes of the same type of plankton. This is especially important for plankton, which are naturally more elliptically shaped, because if they are swimming towards or away from the camera, the shape will change. They are also given a random rotation, as to simulate different angles the particle can be viewed from. A refractive index is defined, which should be the difference between the medium in which the plankton is emerged, usually water, and the plankton itself. An absorption coefficient is also defined. Finally, the simulated particle is assigned a type to make it easier to distinguish them later on. Two types of particles can be simulated in the same image at the same time.

The second step is to define the optics. This includes defining the resolution of the camera, to simulate the one used in the real experiment. The magnification and the numerical aperture, NA, is also specified and included in the image simulation.

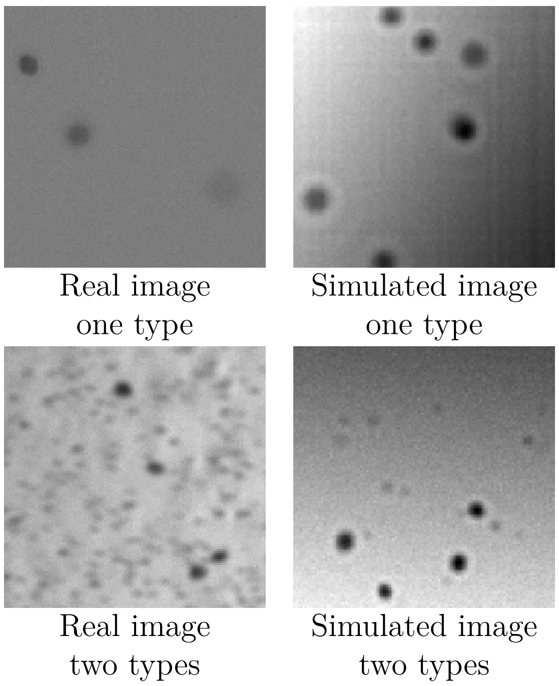


**Figure 3.1:** Examples of real and simulated images of plankton with one type of plankton and two types of plankton, respectively. All of the images are 128x128 pixels in size.

The wavelength of the light illuminating the particle is specified. However, because visible light has a range of wavelengths, the simulations are a combination of wavelengths within a range of $\lambda$=[400nm, 700nm]. These are all combined in a later step. Finally, an illuminating gradient is included, to simulate different inhomogeneous lightning, that could be present in the real images.

The third step is to add some noise. Noise with a Poisson distribution is therefore added to the image. The signal-to-noise ratio of the noise can be simulated as a range, to be able to distinguish particles in many different noise conditions.

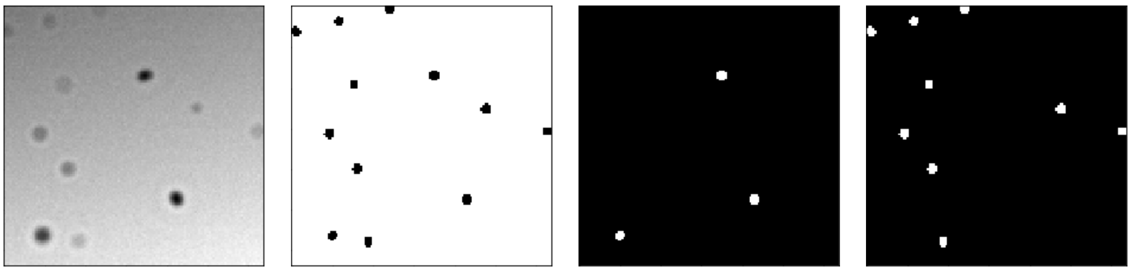
The three steps described above are combined to form an image, which should include everything the model needs to learn how to recognise a specific particle type. A sample is defined as the number of small particles and the number of large particles that should be present in each image. For each step in wavelength range, the optics is incorporated into the sample. When the whole wavelength range has been assessed, the individual sample-optics parts are summed together to form an image that should represent particles illuminated by real light.

The last step is to create the target images, which are segmented versions of the simulated image. These are the outputs from the model, and also serve as the target which the network should predict. Each simulated image is segmented into three parts: the background, particle 1 and 2, as seen in Figure 3.2. Every pixel is assigned a label, according to its associated layer, and each pixel can only be assigned one of the three labels.

Examples of the real images and simulated versions can be found in Figure 3.1. To save time, each simulated image is re-used 8 times by flipping it right to left, up and down and diagonally. Each pixel is also normalized. The model is then trained by giving the network batches of 8 images (or one image flipped around 8 times), the number of batches is defined by the stepsize, i.e 20 steps equals 160 images in total. The total number of images the model needs to be trained depends on the difficulty of the image we want to simulate. However, the images are not given all at once, but in batches, to keep the network from averaging them all and not learning anything.

For images in which the particles are very densely positioned, it is possible to upsample the images to twice, or even four times the size of the original. This makes it possible for the network to distinguish particles that are very close to each other. Because of the padding in the extra upsamling step, there are simply more pixels present between each of the particles, which makes it easier for the network to distinguish them.

It is important to note that the simulated particles are in a size range of a few hundred nanometer, while the real plankton are of 20-40 micrometer. It is therefore not the actual size of the plankton that should be simulated, but the number of pixels occupied by the plankton in the real image.



**(a)** Simulated image **(b)** Background **(c)** Particle 1 **(d)** Particle 2

**Figure 3.2:** Example of a simulated image with two kinds of plankton. The simulated image is segmented into three layers, the background, particle 1 and particle 2. The white areas are the pixels that belong to the corresponding class.

## 3.2 Tracking

The tracking of the particles is divided into two steps. First, finding the particle positions in each frame using a connected component algorithm. Secondly, connecting the positions between each frame, and building a trace using linear sum assignment. Both are described in this section.

### 3.2.1 Finding the positions

By feeding the video to the trained network, three lists of predicted images are obtained. One containing the segmented background layer, one containing the segmented particle 1 layer and containing the segmented particle 2 layer. For the network targeted at the single plankton type experiments, the particle 2 list will only contain black images since there are only one type of plankton present in the experiment. In the network targeted at the multiple plankton type experiments, both the particle 1 list and the particle 2 list contained plankton.

To find the position of the plankton in each frame, a connected component algorithm is used. This algorithm was designed by Rosenfeld and Pfaltz in 1966 [28] and connects neighbouring pixels of the same values. In this case, the algorithm connects pixel values which are all above a specific limit. The center of each connected patch is found using the Python function ”regionprops” together with the attribute ”centroid”. The algorithm returns a list of every plankton’s positions in each frame.

**Algorithm 3.1** Positions. An algorithm, which uses the Python function ”label” to label each connected region in the image with pixel values above a specific limit. The function ”regionprops”, used with the attribute ”centroid”, will return the positions of the center of each connected region.

```
function GET_POSITIONS(image, limit)
    blobs = image pixels > limit
    all_labels = label(blobs)
    regions = regionprops(all_labels)
    create empty list of positions
    for probs in regions do
        y, x = probs.centroid
        add y and x to list of positions
    end for
    return positions
end function
```

### 3.2.2 Building traces frame by frame

The next step in the tracking process is to connect the positions in each frame to the positions in the next frame, thereby building a trace. This is done using algorithms 3.2 and 3.3.

The positions are connected using a linear sum assignment algorithm [29]. This algorithm is built on minimizing a cost matrix. An example of a cost matrix, and how it works, can be seen in 3.3. In this example, the rows of the cost matrix (a, b, c, d) are called the workers, while the columns (p, q, r, s) are called jobs. The aim is to connect the workers and the rows so that the sum of the elements, the total cost, is minimized. In the example in Figure 3.3, the sum of the highlighted elements gives a total cost 23. This is, however, not minimized, since it is possible to choose a better worker-job combination, which yields a lower cost. The purpose of the linear sum assignment is to give the combination of rows and columns which yields a minimized cost.

$$C(i,j) = \begin{array}{c|cccc|} & \text{p} & \text{q} & \text{r} & \text{s} \\ \text{a} & 1 & 2 & 3 & 4 \\ \text{b} & 2 & 4 & 6 & 8 \\ \text{c} & 3 & 6 & 9 & 12 \\ \text{d} & 4 & 8 & 12 & 16 \end{array}$$

**Figure 3.3:** Demonstration of linear sum assignment. The matrix C is the cost matrix which consists of workers (a,b,c,d) and jobs (p,q,r,s) with their corresponding values. Each worker is connected to one job. In this example, the total cost of the worker-job combination (highlighted in blue) is 23. The linear sum assignment will combine the workers and the jobs to minimize this sum. Example from [30].

In our case, the cost matrix is a matrix with the position of frame $i$ as rows, and the positions in frame $i + 1$ as columns. The elements in the matrix are the distances between each of the positions in the different frames. The linear sum assignment therefore gives a minimized cost of distances.

Algorithm 3.3 uses the linear sum assignment to connect positions in one frame with the next, making sure that the positions are not too far away. Distances longer than 20 pixels are replaced by a large number, to make sure that these are not connected to any positions later. Connections that exceeds a certain threshold are deleted.

The two algorithms 3.2 and 3.3 are used together. Algorithm 3.3 decides which positions are to be connected and returns the built-on-traces to Algorithm 3.2. Algorithm 3.2 checks which traces are continuing to be built upon, and sends those back into Algorithm 3.3. The finished traces are saved for later. After all frames have been run through, Algorithm 3.2 returns a list of all traces found in the video.

**Algorithm 3.2** This algorithm is to be used together with algorithm 3.3. It takes positions, traces and frame number. It starts new traces, or adds new positions to already begun traces. The traces are built according to certain conditions, such as the proximity of the new position to the old one and the speed of the object from one frame to another.

```
1: function TRACE(new positions, trace, frame)
2:     if length of trace > 0 then
3:         create list of trace positions
4:         for trace in traces do
5:             last position = list of positions in previous frame
6:             if len(trace) > 2 then
7:                 last positions += mean(diff(trace[position i, position i+1])
8:             end if
9:             add last positions to list of trace positions
10:        end for
11:        distance matrix = distance(new positions, trace positions)
12:        distance mattrix[distance mat > 20] = 100000
13:        row, col = linear sum assignment(distance mat)
14:        isinf = nonzero(distance mat[row, col] > 10000)
15:        row = delete(row, isinf)
16:        col = delete(col, isinf)
17:        for c, r in (col, row) do
18:            add frame and new positions to traces
19:        end for
20:        new positions = delete(new positions, row)
21:    end if
22:    for i in range(len(new positions)) do
23:        add frame and new positions[i] to traces
24:    end for
25: end function
```

**Algorithm 3.3** An algorithm, which uses both algorithm 3.1 and 3.2 to build a trace. It takes a video as a list and a limit and returns a list of traces, one for each object traced in the video.

```
function GET_PARTICLE_TRACE(video, limit)
    create list of lists for begun traces
    create list of completed traces
    for frame in video do
        new particles = get_positions_video_frame(frame, limit), from 3.1
        trace(new particles, begun traces, frame), from 3.2
        for trace in begun traces do
            if current frame not in trace and length of trace >= 10 then
                add trace to list of completed traces
            end if
        end for
        keep updated traces in begun traces list and re-run trace function
    end for
    return list of competed traces + begun traces
end function
```

### 3.2.3 Optimization of limit

The limit, or threshold, introduced in the connective components algorithm, can be set to anything the user finds suitable. However, to reduce the user bias, an optimized limit is introduced. The goal is to get as long particle traces as possible, while introducing few new traces. The number of traces is compared to the average length of all traces. Both values are normalized and the limit, which gives the largest difference between the two, is used. A pseudo code of the algorithm, which gives the optimized limit, can be found in Algorithm 3.4.

**Algorithm 3.4** Limit. An algorithm, which takes a list of traces as input and returns the optimized limit. The number of traces and the average length of all the traces is calculated and normalized. When no average value can be calculated, i.e. no traces were found, the algorithm will throw an exception. The optimized limit is given as the one that gives the largest difference between both of the normalized values.

```
 1: function GET_LIMIT(list of traces)
 2:     create two empty lists for x and y
 3:     limit_test = value of limits you want to test
 4:     for i in limit_test do
 5:         try y= mean length of particle traces
 6:         except IndexError break
 7:         add y to y-list
 8:         x = number of particle traces
 9:         add x to x-list
10:         normed_x = i/sum(x) for i in x
11:         normed_y = i/sum(y) for i in y
12:     end for
13:     limit = limit_test(argmax(normed_y-normed_x))
14:     return limit
15: end function
```

## 3.3 Drift compensation

To avoid edge effects in the experiments, the plankton need to be observed in a relatively large container. This might lead to unwanted drift in the trajectories, which will interfere with the speed and displacement calculations. To reduce the drift effects, we use the drift compensation function developed by Allan et al. for the TrackPy package in Python [31]. This function computes an overall ensemble drift function for all plankton in the experiment. This is done by calculating the displacement of each plankton between each frame and thereby create a function of the overall drift. This drift function is subtracted from the original tracks to obtain a drift compensated dataset. A pseudo code of how this function works can be found in Algorithm 3.5.

**Algorithm 3.5** Drift. A function which takes a dataframe with x, y, frame and particle as columns and computes an overall ensemble drift of the trajectories.

```
function COMPUTE_DRIFT(dataframe)
    for p, t in dataframe.groupby('particle')]) do
        delta = difference in distance between frames in dataframe
    end for
    delta = keep only delta where frame difference = 1
    dx = mean(delta)
    x = cumulative sum (dx['x', 'y'])
    return x
end function
```

## 3.4 Calculations for swimming behaviours

The most important features to consider when analyzing the swimming behaviour of the plankton are the speed and the net displacement. The following sections describe how these can be calculated from the positions.

### 3.4.1 Speed

From the positions and traces of the plankton, found in section 3.2, the speed of each of the plankton can be calculated. The dataframe is first split into smaller dataframes, one for each plankton. The distance between the position in the current and the previous frame is calculated. To get the speed in millimeter per second, instead of pixels per second, the distance is divided by the number of pixels per millimeter. Finally, the distance is divided with the number of frames per second. The distance is then appended to the dataframe as a column named "speed [mm/s]". A pseudo code can be found in Algorithm 3.6.

**Algorithm 3.6** Speed. An algorithm which calculates the speed of a plankton from the positions x and y in each frame.
fps = frames per second, ppmm = pixels per mm.

```
function GET_SPEED(dataframe, fps, ppmm)
    create empty list of speed
    for particle in dataframe, split dataframe into separate particle frames do
        xy = particle frame[['x', 'y']]
        a = xy[:-1]
        b = roll(xy, -1, axis=0)[:-1]
        dxy = linalg.norm(a - b)/ppmm
        speeds = divide(dxy, fps)
        convert speed list to dataframe and add as column to particle frame
    end for
    concatenate particle frames to one
    return dataframe with speed
end function
```

### 3.4.2 Net displacement

The net displacements of the plankton trajectories are found from the positions of each trace. As seen in Figure 2.13 in section 2.2, the net displacement is the straight line distance from the start position to the end position of the trace.

To find the net displacement for each plankton at all times, the input dataframe is first split into smaller dataframes, one for each plankton. The distance between a plankton at frame $j$ and a plankton at frame $i - j$ are calculated and added to an empty list. The mean value of this list is then added to a list of distances. This is to make sure that the frame difference is the same for all plankton. The distance list is then converted and appended to the original dataframe.

**Algorithm 3.7** Net displacement. This algorithm takes a dataframe with columns 'x', 'y', 'frame' and 'particle' and returns the original dataframe with the added column 'net displacement'. The net displacement is calculated by calculating the distance between positions in frame j and frame i-j.

```
function GET__NET__DISPLACEMENT(dataframe, ppmm)
    create empty list of net displacements
    for particle in dataframe, split dataframe into separate particle frames do
        create empty list of distances
        for j in particle frame do
            create empty, temporary list
            for i in particle frame - j do
                x0 = particle frame['x'].values[i]
                y0 = particle frame['y'].values[i]
                x1 = particle frame['x'].values[i+j]
                y1 = particle frame['y'].values[i+j]
                dist = $sqrt((x1 - x0)^2 + (y1 - y0)^2)/ppmm$
                add dist to temporary list
            end for
            add mean of temporary list to distance list
        end for
        convert distance list to dataframe and add as column to particle frame
    end for
    concatenate particle frames to one
    return dataframe with net displacement
end function
```

## 3.5 Plankton experiments

The details of the experiments conducted on single type plankton by the the Department of Marine Science can be found in the paper Lindström et al. from 2017 [25]. In short, two types of phytoplankton, *Lingulodinium polyedra* and *Alexandrium tamarense*, were exposed to different amounts of copepodamides and 4 replicates of each experiment were conducted. Each experiment was filmed for 1 minute with 5 frames per second. There was one control group, one exposed to 10pM copepodamide and one exposed to 100pM copepodamide.

Furthermore, an experiment with multiple types of plankton was carried out, containing pythoplankton *Strombidium arenicola* and *Rhodomonas baltica*, the latter in a high concentration. Two experiments were conducted, with and without exposure to copepodamides. These were recorded for 1 minute with a frame rate of 10 frames per second, and with 6 replicates of each experiment.

### 3.5.1 Previous analysis using TrackMate

The swimming pattern of the single type plankton was analyzed using TrackMate plugin for Fiji ImageJ [2]. The net displacement was calculated using all traces. The mean speed of the plankton were analyzed using only traces longer than 5 seconds. The same parameters was used when analyzing the speed and displacement when using the Deeptrack method [8].

## 3.6 Testing the model

Before the model can be tested on experimental data, it is first tested on simulated images to evaluate its ability to find and distinguish different particles. The model is tested in five different categories. First, how much training is needed for acceptable results. Secondly, how much noise can be added. Thirdly, how large an illumination gradient the image can have. Finally, how large of a difference between sizes of particles and how big of a shape difference the network can distinguish between. For each of the tests, one, or several, new networks are trained. The parameters of each of the models trained can be found in table 3.1.

**Table 3.1:** Parameters for each model. The amount of training is set in batches, 1 batch has 8 simulated images. The noise is the signal-to-noise ratio. The gradient is a range of 0 increases with higher numbers. If only one number, the particles are spherical, if two, the particle is elliptical. The unit of the particle size is in pixels.

| | Training | Noise | Gradient | particle 1 | particle 2 |
|---|---|---|---|---|---|
| Training | - | 50-100 | 0-0.2 | 8 | 4 |
| Noise | 4000 | 3-100 | 0-0.2 | 8 | 4 |
| Gradient | 4000 | 50-100 | 0-1.5 | 8 | 4 |
| Size | 4000 | 50-100 | 0-0.2 | 8-12 | 4-8 |
| Shape | 4000 | 50-100 | 0-0.2 | 8 | $r_1$=8, $r_2$=4-8 |

Other parameters, such as the depth of the particle and the refractive index, were kept constant or in the same range.

The model is also evaluated on experimental data. It is tested on the multi type plankton experiment containing both *Strombidium arenicola* and *Rhodomonas baltica*. However, only a single frame is used to test the model. Because there is no ground truth available to compare the models ability to find particles with, the different particle types are counted manually. The image used for this is a cropped version of the whole video frame and can be seen in Figure 3.4. The network is trained using two types of simulated particles that are made to resemble the two plankton present in the experiment. Since the presence of the smaller plankton *Rhodomonas baltica* is so dense, the network is upsampled to twice the size of the original image. The model is trained using 4000 batches of images. The number of correct particle positions is compared to the number of incorrect positions, found by the model.

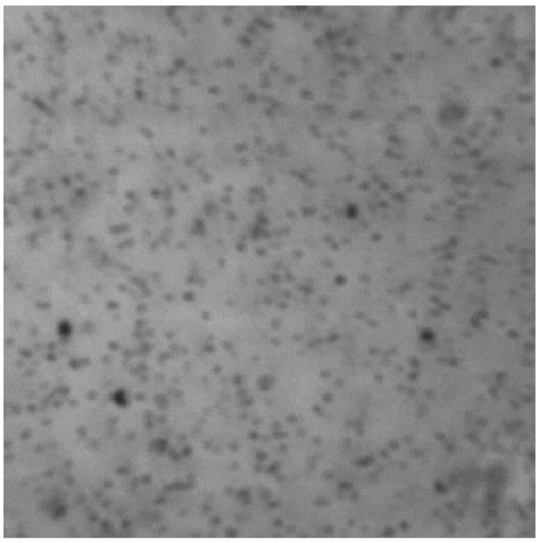


**Figure 3.4:** A cropped version of the frame used to evaluate the model on experimental data. The larger dots are *Strombidium arenicola*, and the smaller dots are *Rhodomonas baltica*.

# 4
# Results

## 4.1 Model evaluation

The model was evaluated by testing its capability to find and distinguish different particles under different conditions, as described in section 3.6. The results of the different assays can be found in the following sections.

### 4.1.1 Amount of training

To test the amount of training needed for acceptable results, the same network was trained using different number of input images. Each network was trained by simulating two types of particles. The larger simulated particle was twice as big as the small, and there were 3-5 particles of each type in each image. A signal-to-noise ratio in a range of 50-100 and a light gradient of 0-0.2 was also added to the image, the details of the network can be found in table 3.1. The results from training on 100, 400 and 2000 batches of images can be seen in Figures 4.1, 4.2 and 4.3, respectively. One batch of images equals 8 single images, which are fed to the network at the same time. All results shown in Figures 4.1, 4.2 and 4.3, used constant parameters; a gradient of 0.1, a signal to noise ratio of 100 and constant particle sizes.

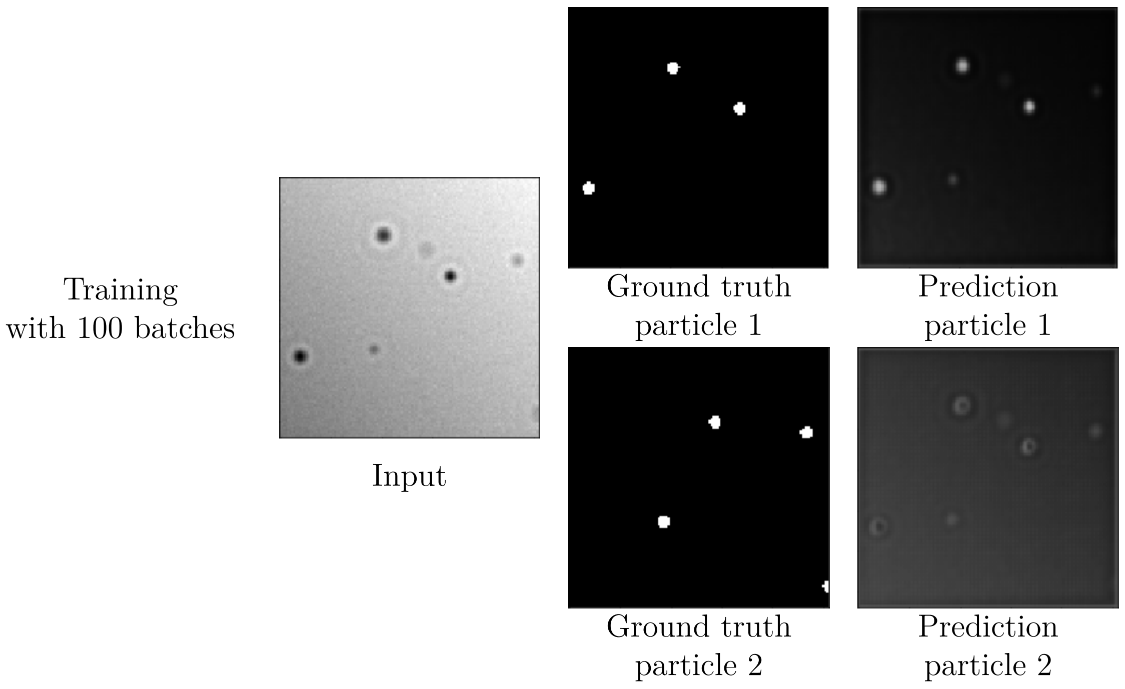


**Figure 4.1:** Input image of two spherical particle types where the larger was twice as big as the smaller one. The true and predicted segmented images of particle 1 and 2. The predicted images were obtained from 100 batches of training images.

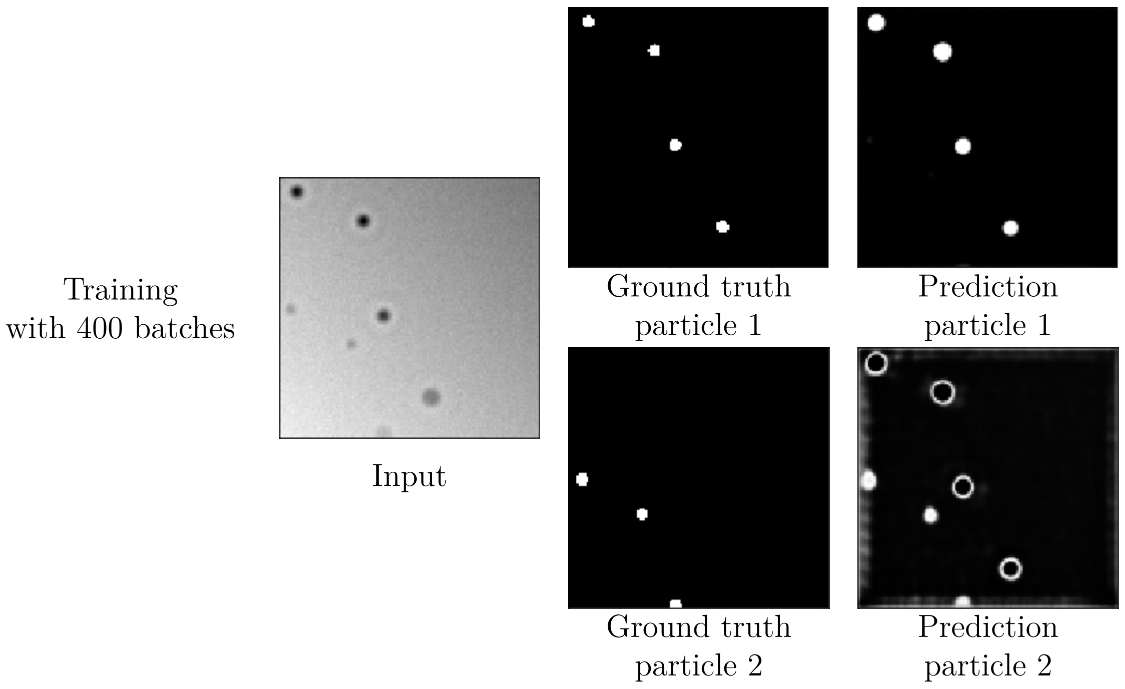


**Figure 4.2:** Input image of two spherical particle types where the larger was twice as big as the smaller one. The true and predicted segmented images of particle 1 and 2. The predicted images were obtained from 400 batches of training images.

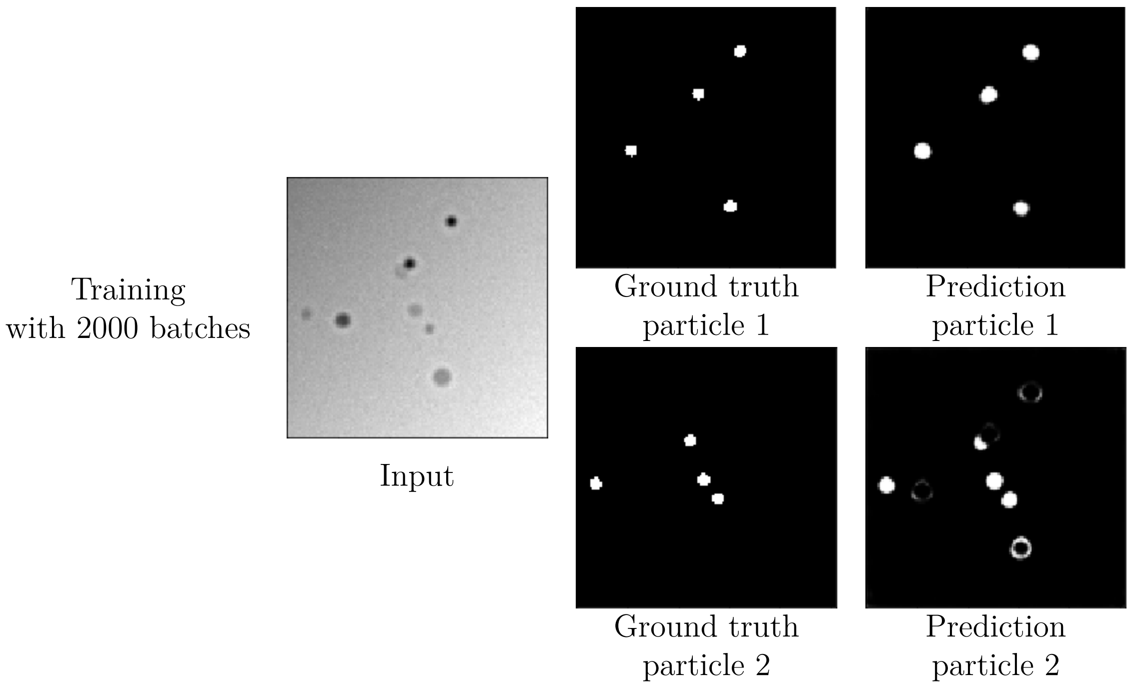


**Figure 4.3:** Input image of two spherical particle types where the larger was twice as big as the smaller one. The true and predicted segmented images of particle 1 and 2. The predicted images were obtained from 2000 batches of training images.

### 4.1.2 Noise

To test the model's capability to find particles under different noise conditions, a new network was trained. This network was trained with two types of spherical particles, the larger twice the size of the small. The signal-to-noise ratio was in a range of 3-100 and lighting gradient in a range of 0-0.2 was added to the simulated images. The network was trained with 4000 batches of image and the details can be found in table 3.1. The trained network was tested on signal-to-noise ratios 25, 10 and 5. The results of these tests can be found in Figure 4.4, 4.5 and 4.6, respectively. In these results, only the noise was varied and the particle size was kept constant, the large at a radius of 8 and the small at a radius of 4 pixels. The light gradient of the images was also kept constant at 0.1.

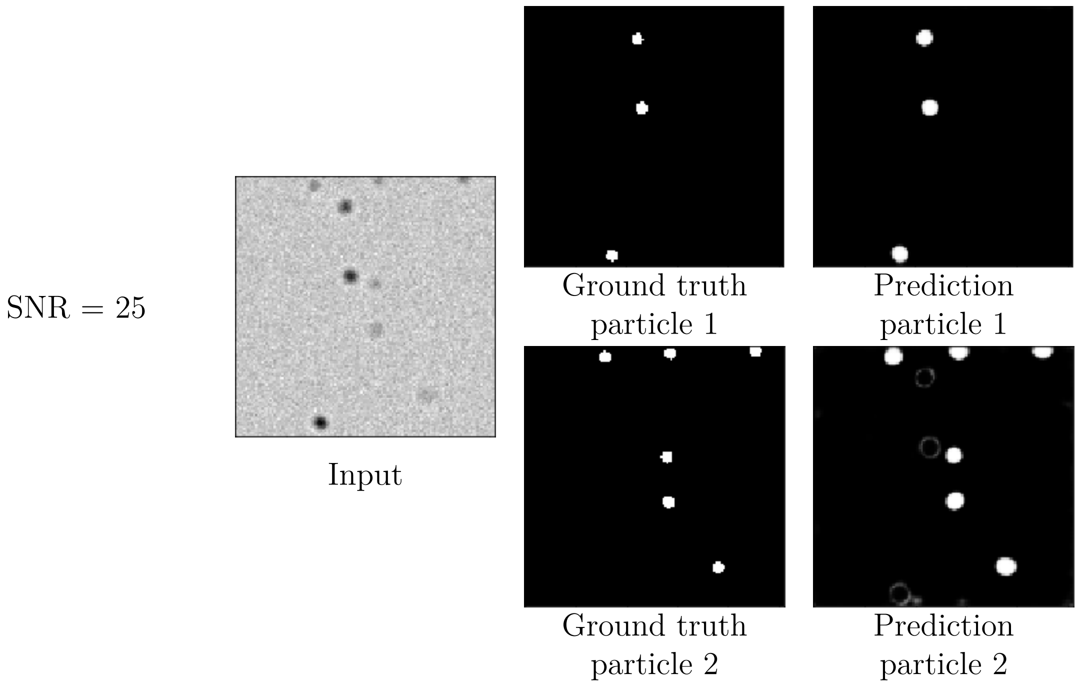


**Figure 4.4:** Input image with two types of particles with a signal-to-noise (SNR) ratio of 25. The true segmentation of the image and the predicted segmentation of each of the two particles.

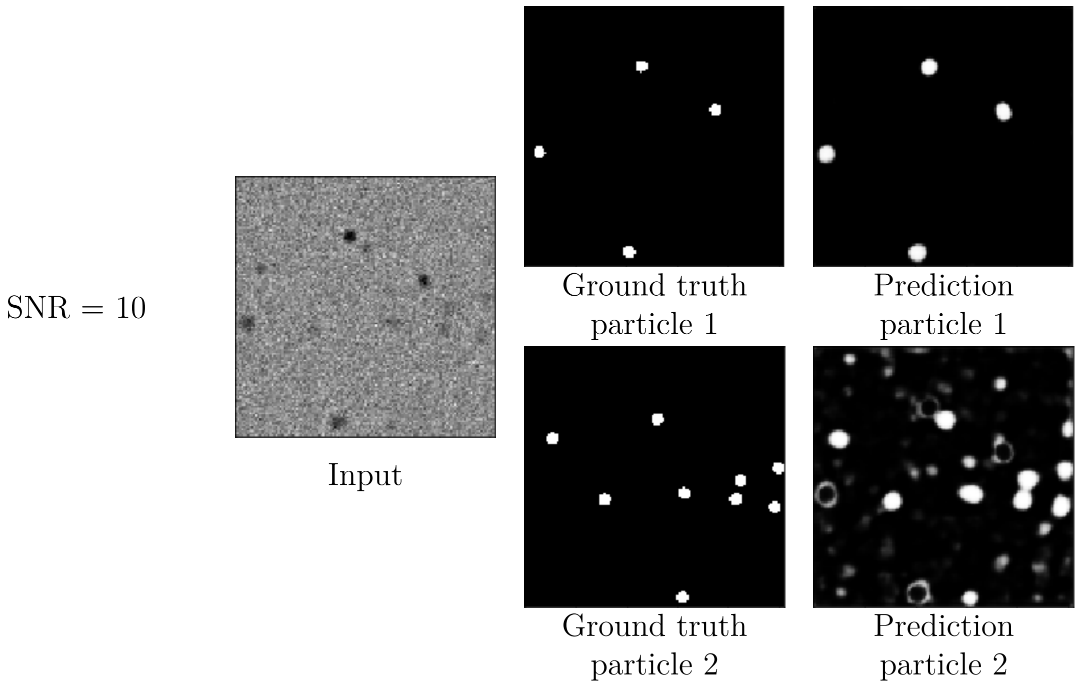


**Figure 4.5:** Input image with two types of particles with a signal-to-noise (SNR) ratio of 10. The true segmentation of the image and the predicted segmentation of each of the two particles.

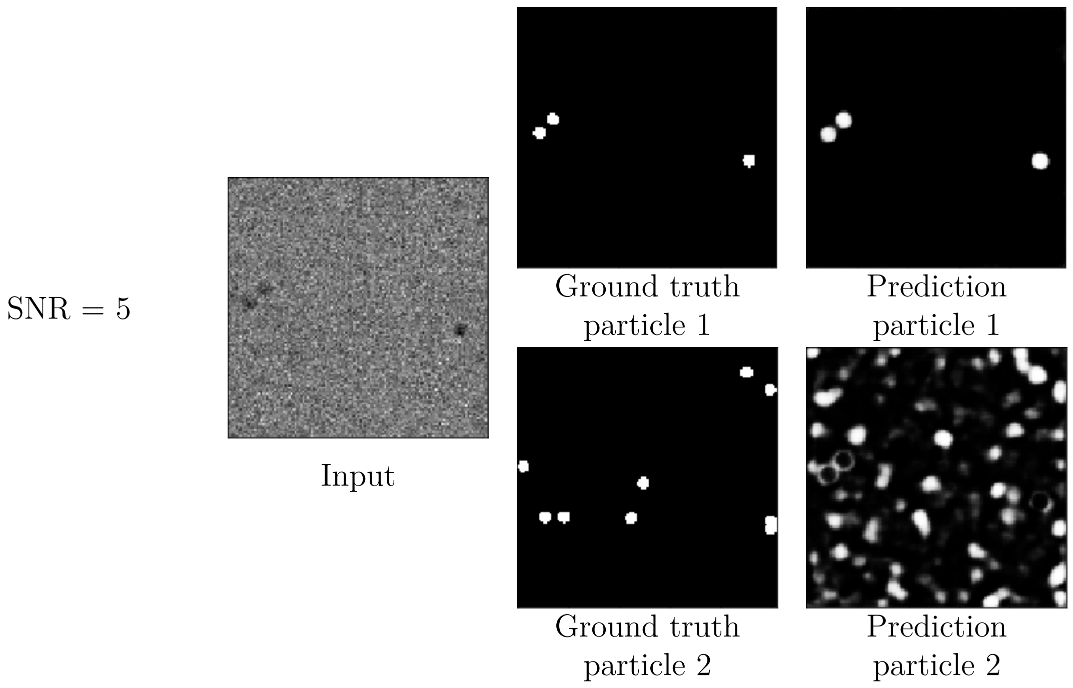


**Figure 4.6:** Input image with two types of particles with a signal-to-noise (SNR) ratio of 5. The true segmentation of the image and the predicted segmentation of each of the two particles.

### 4.1.3 Inhomogeneous lighting conditions

The model's capability to find and distinguish particles in inhomogeneous lighting conditions, a new network was trained using 4000 batches of images. This network was trained using images with two different types of particles, a large and a small, with the larger being twice the size of the small. A signal-to-noise ratio, in a range of 50-100, was added to the images. The gradient used on the training images was in a range of 0-1.5. The details of this network can be found in table 3.1. The results presented in this section had constant noise parameters, signal to noise ratio 50, and constant particle sizes, 8 and 4 pixels. Lighting gradients with intensity 0.5, 0.9 and 1.5 were tested, and the results can be found in Figure 4.7, 4.8 and 4.9 respectively.

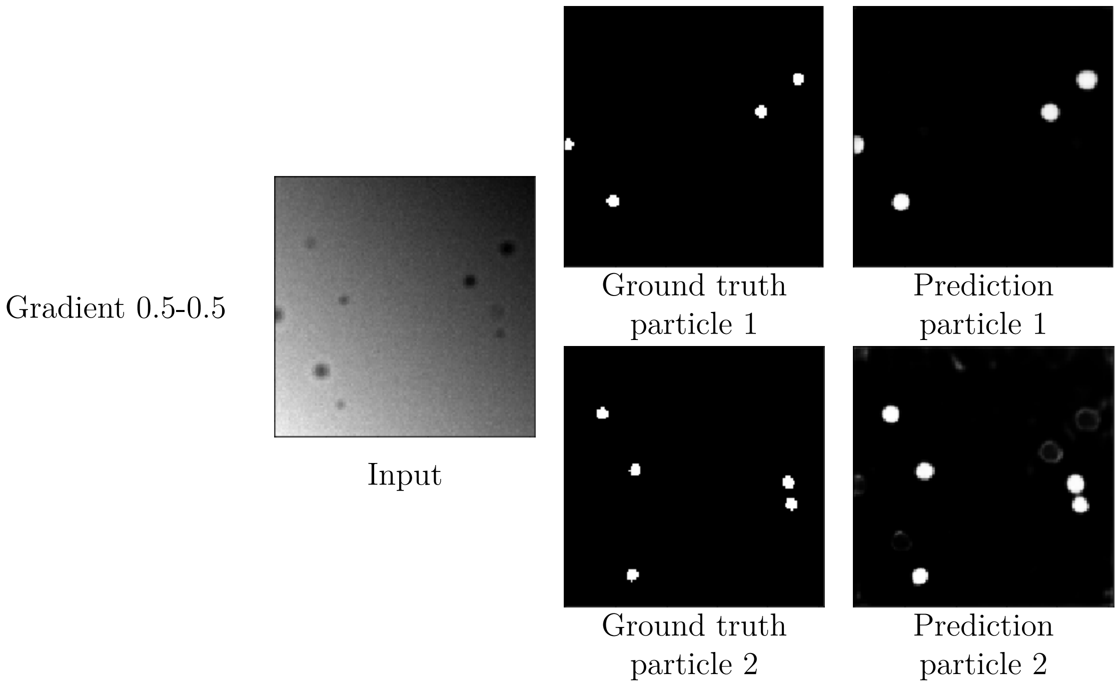


**Figure 4.7:** Input image with two types of particles with lighting gradient of 0.5-0.5. The true segmentation of the image and the predicted segmentation of each of the two particles. The noise is kept constant at a signal-to-noise ratio of 50.

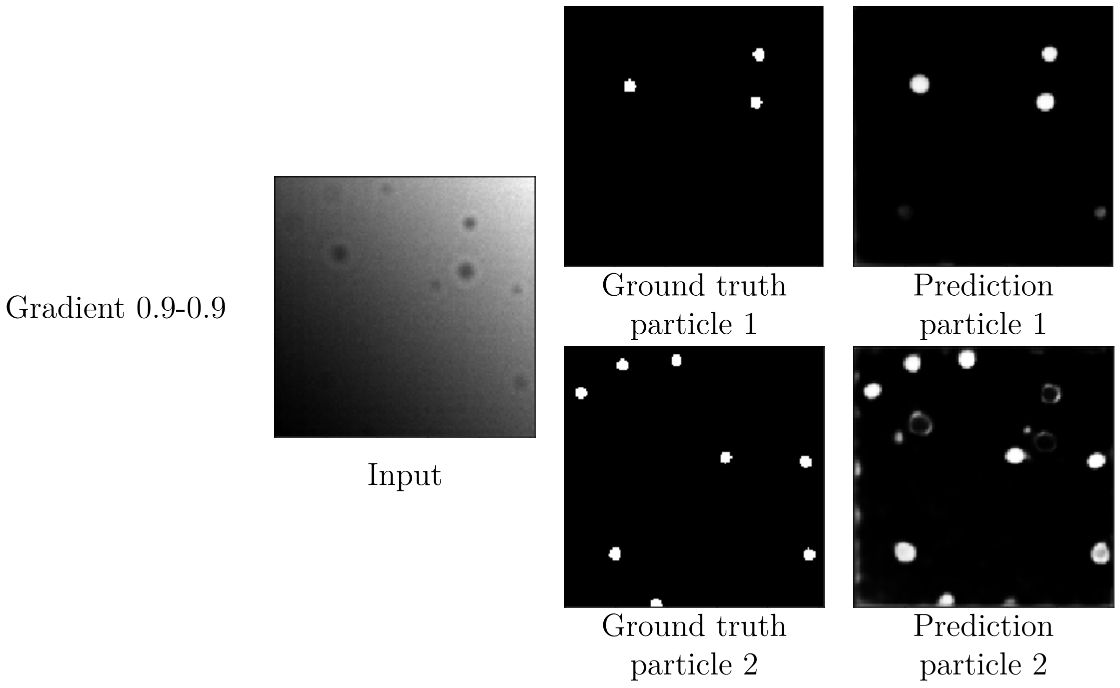


**Figure 4.8:** Input image with two types of particles with lighting gradient of 0.9-0.9. The true segmentation of the image and the predicted segmentation of each of the two particles. The noise is kept constant at a signal-to-noise ratio of 50.

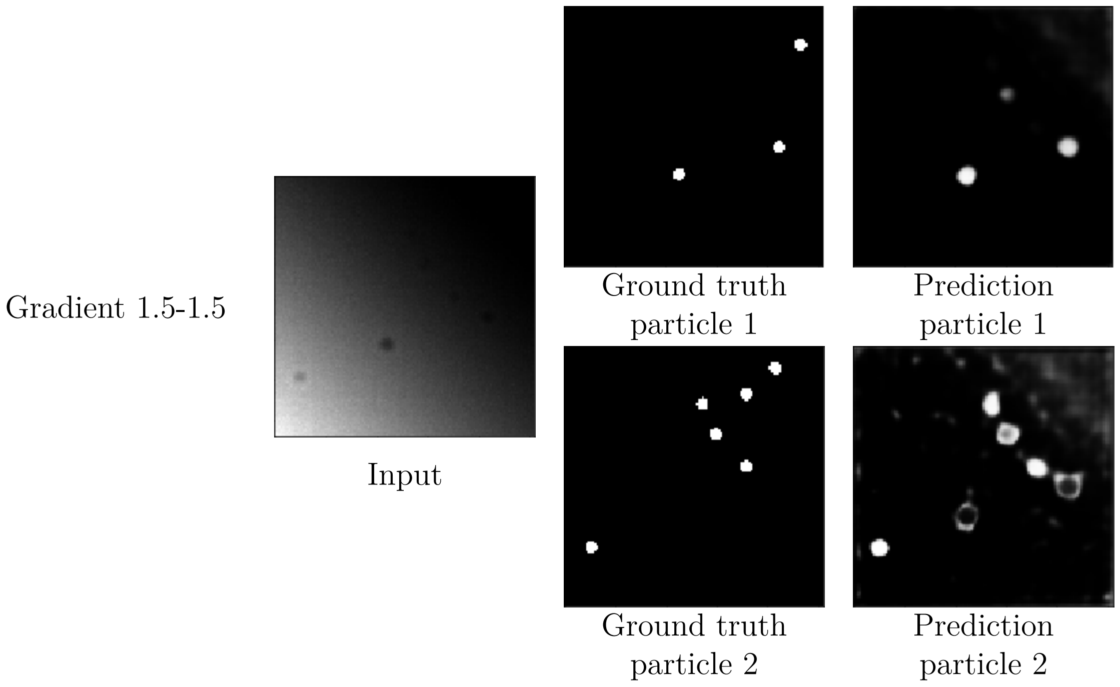


**Figure 4.9:** Input image with two types of particles with lighting gradient of 1.5-1.5. The true segmentation of the image and the predicted segmentation of each of the two particles. The noise is kept constant at a signal-to-noise ratio of 50.

### 4.1.4 Size difference

The capability of the model to distinguish between different particle sizes was tested by simulating spherical particles of different sizes. A new network was trained, which parameters can be found in table 3.1. Summarizing, the network was trained using 4000 batches of simulated images. Each simulated image had two types of spherical particles present. The larger particle was simulated with a radius in a range of 8-12 pixels, while the small particle was simulated with a radius of 4-8 pixels. Other parameters regarding the particles were kept the same and constant for both types of particles. A signal-to-noise ratio of 50-100 was added to the images, as well as a lighting gradient of 0-0.2. Differences of 25%, 15% and 5% between the large and the small particles were tested. The result of these analyses can be found in Figure 4.10, 4.11 and 4.12 respectively.

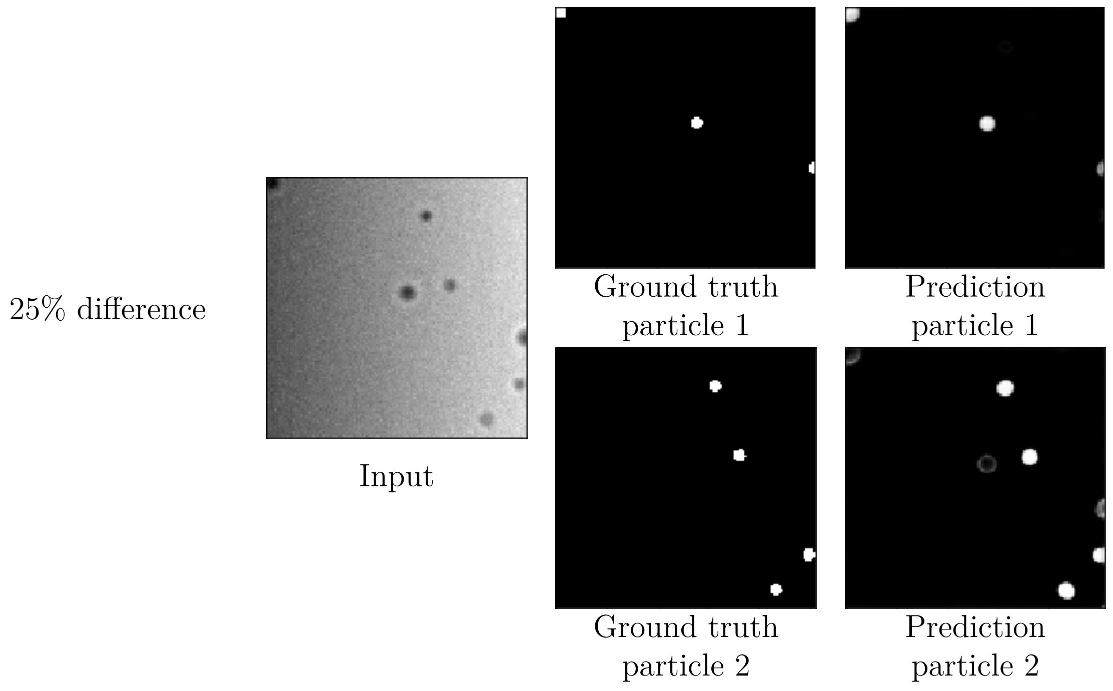


**Figure 4.10:** Input image with two types of particles with a size difference of 25%. The true segmentation of the image and the predicted segmentation of each of the two particles. The noise is kept constant at a signal-to-noise ratio of 50.

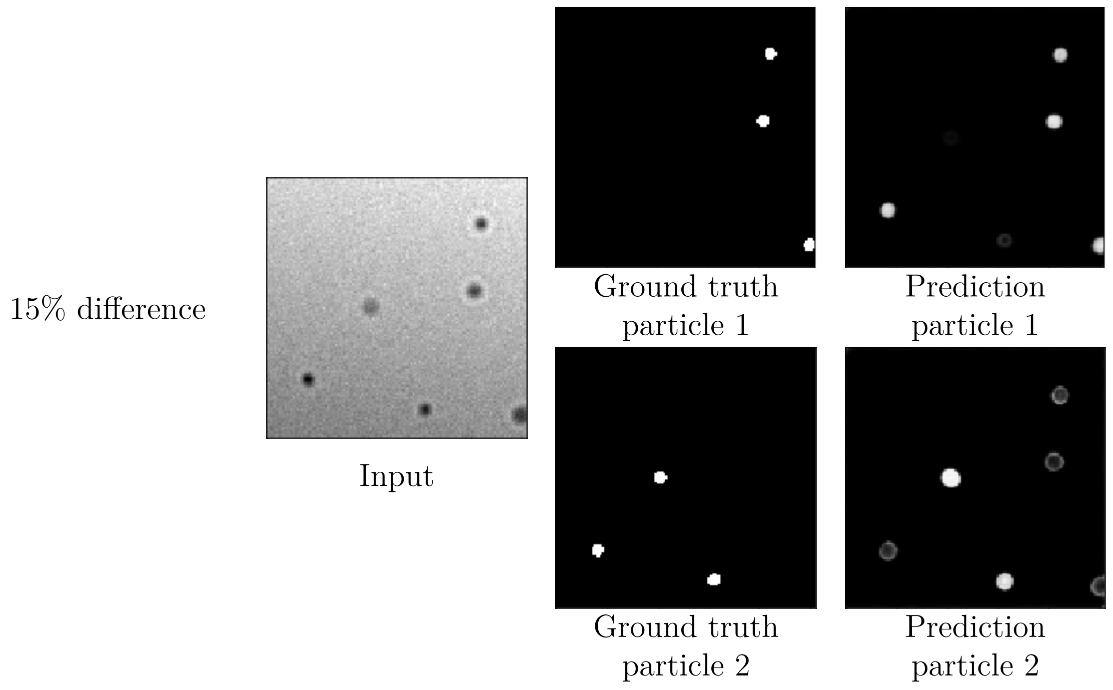


**Figure 4.11:** Input image with two types of spherical particles with a size difference of 15%. The true segmentation of the image and the predicted segmentation of each of the two particles.

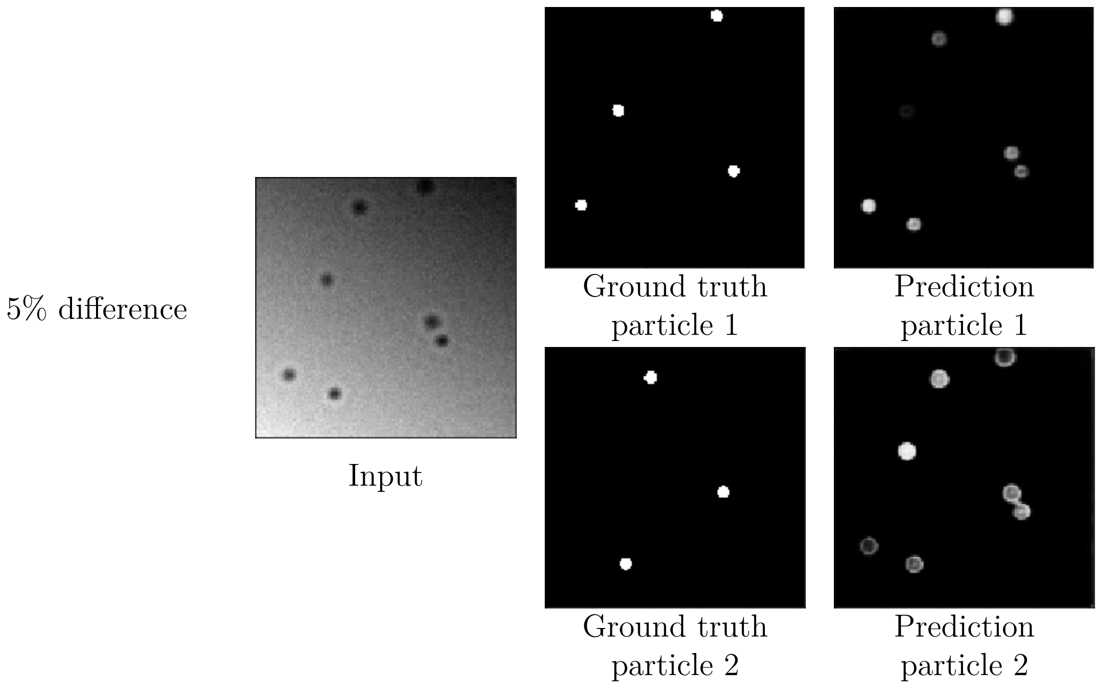


**Figure 4.12:** Input image with two types of spherical particles with a size difference of 5%. The true segmentation of the image and the predicted segmentation of each of the two particles.

### 4.1.5 Shape difference

To test the model's ability to distinguish different shapes of particles, a new network was trained using the parameters found in table 3.1. In short, two types of particles were simulated. Spherical particles with a radius of 8 pixels, and elliptical particles with a length of 8 and a width 4-8 pixels were present in each simulated image. The network was trained using 4000 batches of images. A signal-to-noise ratio of 50-100 was added to the images, as well as a lighting gradient of 0-0.02. The network was tested on two different cases. The first with particle 1 as a spherical particle with a radii of 8 pixels and particle 2 as an elliptical particle with radii of 8 and 2 pixels. The second case had the same shape of particle 1 but particle 2 was elliptical with radii 8 and 4 pixels. The results of these assays can be found in Figure 4.13 and 4.14.

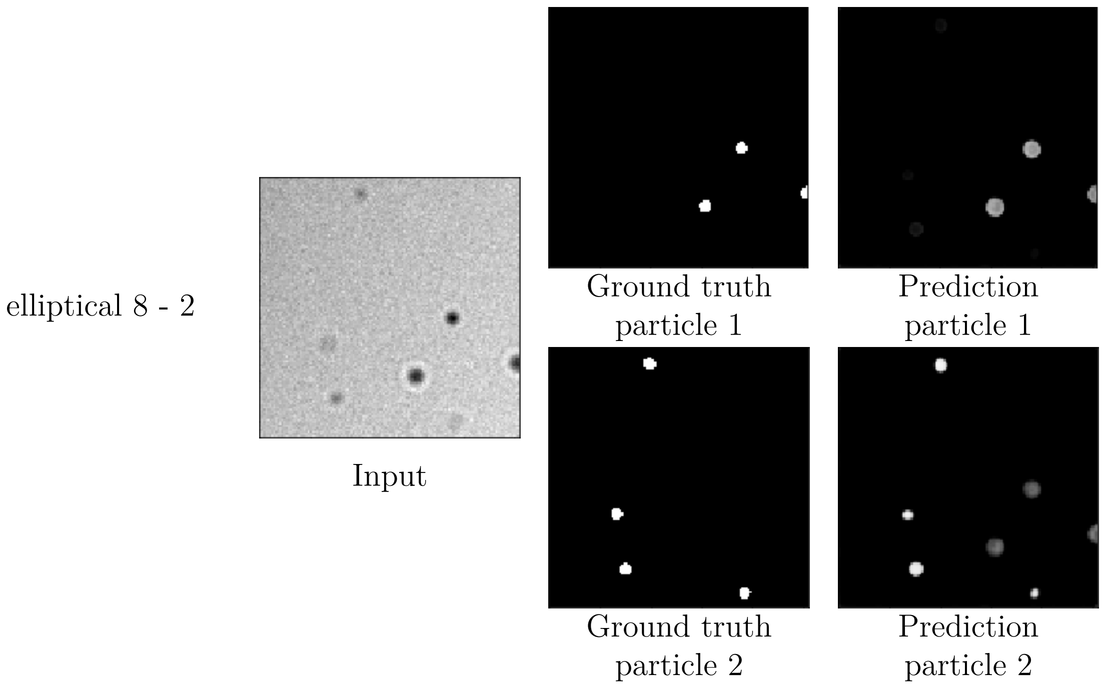


**Figure 4.13:** Input image of two types of particles with different shapes. Particle 1 is spherical and particle 2 is elliptical with radii 8 and 2 pixels.

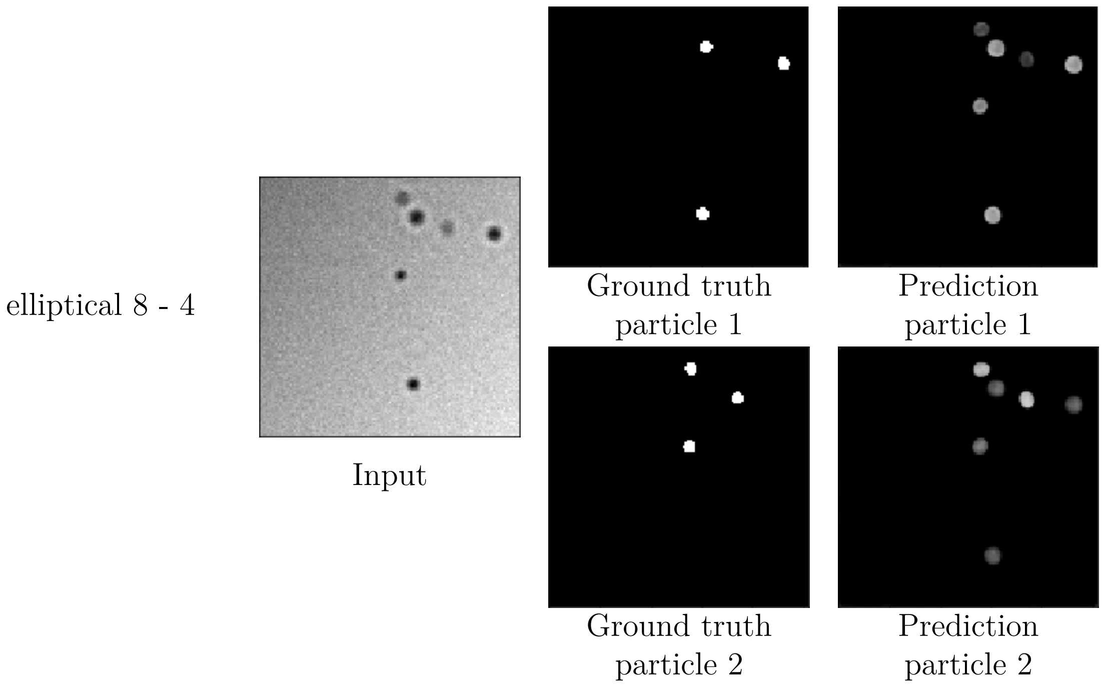
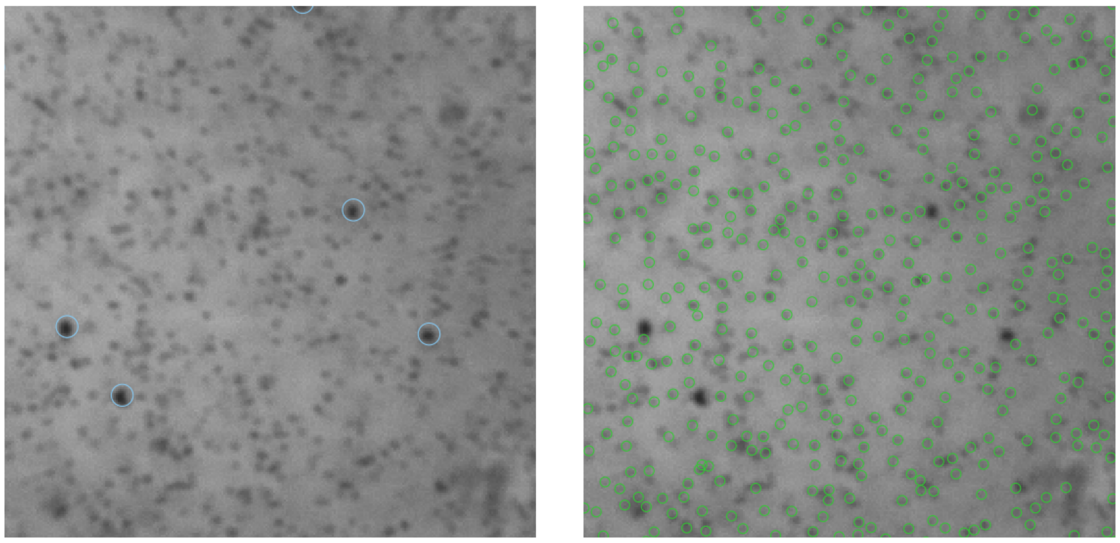


**Figure 4.14:** Input image of two types of particles with different shapes. Particle 1 is spherical and particle 2 is elliptical with radii 8 and 4 pixels.

## 4.2 Evaluation of model on experimental data.

The results from the evaluation of the model on experimental data can be found in Figure 4.15. In Figure 4.15a, we see the positions of the larger plankton, *Strombidium arenicola*, and in Figure 4.15b, the positions of the smaller plankton, *Rhodomonas baltica*. To find the positions, the connected components algorithm was used, as seen in algorithm 3.1. The limit described in this algorithm was for this experiment set to 0.99 for both plankton.



**(a)** Positions of the larger plankton *Strombidium arenicola*, found through the network.

**(b)** Positions of the smaller plankton *Rhodomonas baltica*, found through the network.

**Figure 4.15:** The model's predicted positions of particle 1 and 2 at a limit of 0.99.

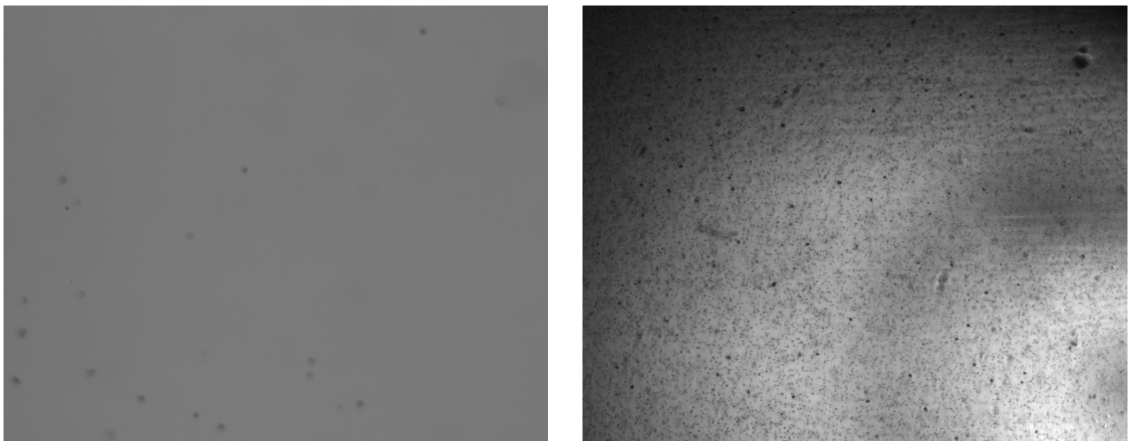
Comparing the network's positioning and labeling of the plankton with the manually labeled plankton, gave the results seen in table 4.1. Summarizing the results, the network managed to find 100% of large *Strombidium* at the correct positions. The network found 415 the smaller *Rhodomonas*, of which 245 were correctly labeled and positioned, and 170 incorrectly labeled. These 170 were labeled as *Rhodomonas* by the network, while the manual labeling had labeled those positions as background. The manually labeling had 120 *Rhodomonas*, which the network labeled as background.

**Table 4.1:** The model's predictions of the plankton compared to the manually labeled plankton. SM, RM and BM are manually labeled *Strombidium*, *Rhodomonas* and background respectively. SN, RN and BN are the networks labeling of the same classes.

| | | | |
|---|---|---|---|
| **SN** | 4/4 | 0 | 0 |
| **RN** | 0 | 245/365 | 170 |
| **BN** | - | 120 | - |
| | **SM** | **RM** | **BM** |

## 4.3 Testing software on experimental data

There were two experiment used to test the softwares capability to track plankton. The first experiments contained only one type of plankton, *Lingulodinium* and *Alexandrium* respectively, and an example of a frame from these experiments can be seen in Figure 4.16a. The second experiment contained two types of plankton *Strombidium arenicola* and *Rhodomonas baltica*, an example of a frame from these experiments can be seen in Figure 4.16b.



**(a)** Frame from experiments containing only *Lingulodinium*

**(b)** Frame from experiments containing both *Strombidium arenicola* and *Rhodomonas baltica.*

**Figure 4.16:** Examples of frames from the experiments used to analyze the swimming behaviour of different types of plankton.

### 4.3.1 Single type plankton tracking

The experiments containing only one type of plankton were analyzed in two parts. First, the two methods TrackMate and Deeptrack were compared. Secondly, the drift compensated tracks were analyzed. The results from both queries can be found in the sections below.

The number of particle traces, as well as the average length of the traces, were calculated for both the TrackMate and the Deeptrack method. The calculations were executed using all four replicates for each experiment. The number of traces can be found in table 4.2, and the average length in table 4.3. There were on average 32% fewer traces and the length of the traces were on average 10.6% longer using the Deeptrack method, compared to TrackMate.

**Table 4.2:** A table which displays the number of plankton traces present during each experiment. The number is the sum of four replicates for each experiment.

| | Copepodamides [pM] | TrackMate | Deeptrack |
|---|---|---|---|
| | 100 | 321 | 270 |
| *Lingulodinium* | 10 | 460 | 398 |
| | 0 | 241 | 222 |
| | 100 | 706 | 367 |
| *Alexandrium* | 10 | 587 | 399 |
| | 0 | 538 | 346 |

**Table 4.3:** A table which displays the average length of all traces during each experiment. Each experiment was 300 frames long and the length is the average number of frames each plankton was tracked.

| | Copepodamides [pM] | TrackMate | Deeptrack |
|---|---|---|---|
| | 100 | 177.84 | 188.70 |
| *Lingulodinium* | 10 | 193.12 | 208.65 |
| | 0 | 180.32 | 205.53 |
| | 100 | 167.23 | 194.22 |
| *Alexandrium* | 10 | 169.08 | 188.47 |
| | 0 | 171.82 | 199.88 |

The mean net displacement and standard deviation for both the previous TrackMate analysis and the Deeptrack analysis can be found in figure 4.18. A histogram comparing the average speed for the traces found with TrackMate and Deeptrack can be found in figure 4.17.

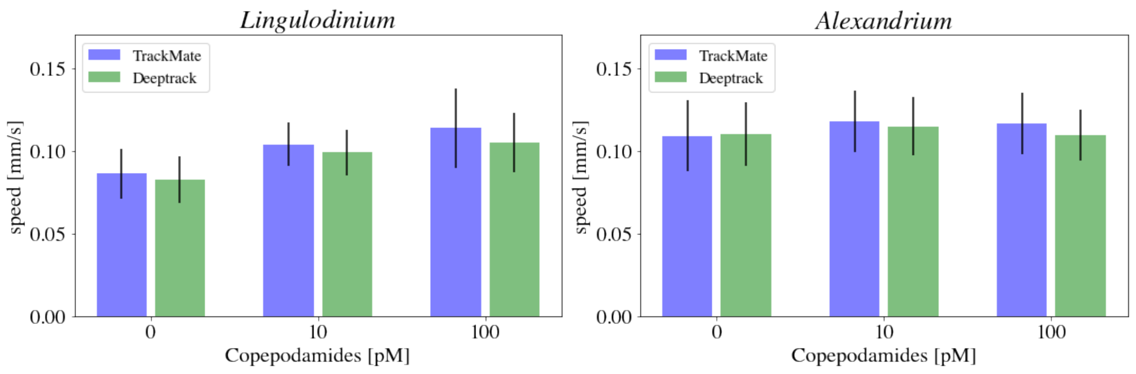


**Figure 4.17:** The average speed calculated from traces longer than 5 seconds from both TrackMate and Deeptrack. The black bars represent the standard deviation.

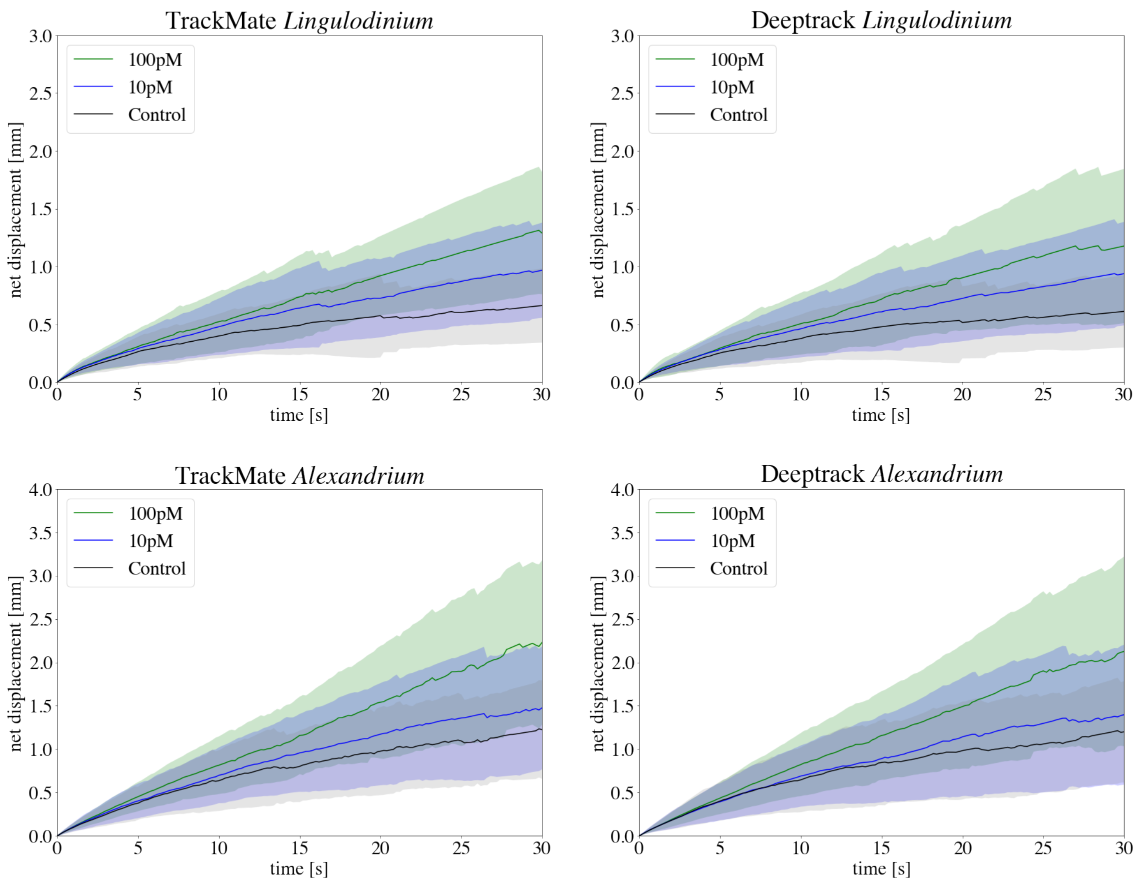


**Figure 4.18:** The net displacement of the different plankton types using the TrackMate and Deeptrack method. The average net displacement for each plankton was calculated from the first frame the plankton is present in the video. The lines represent the mean net displacement and the shaded areas represent the standard deviations.

### Drift compensated traces

The traces found using Deeptrack were analyzed both in their original state and when drift compensated. An example of traces in both original and drift compensated state can be found in figure 4.19.

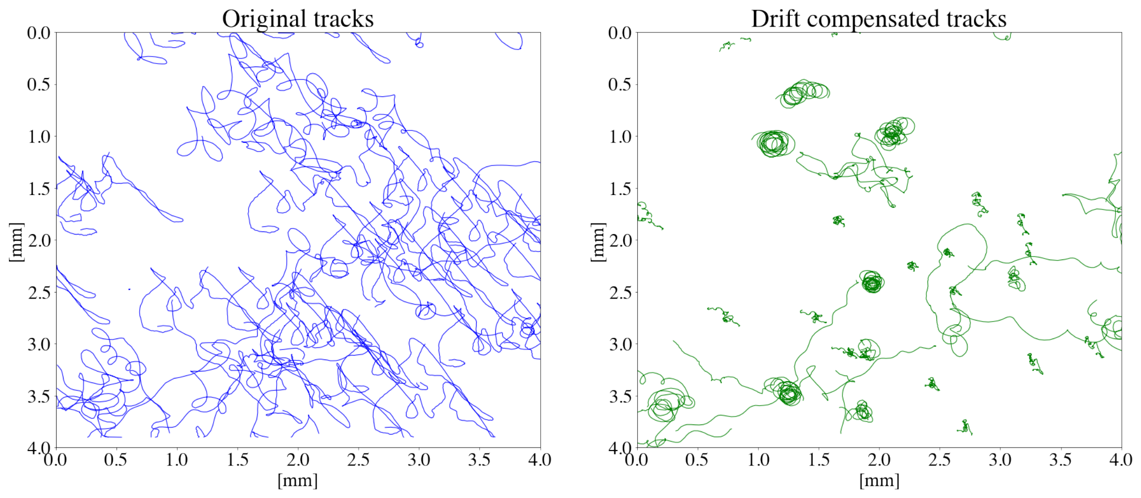


**Figure 4.19:** Examples of original traces and drift compensated traces. The right figure (green tracks) is the drift compensated version of the left (blue tracks). This is replicate 2 of the *Lingulodinium* control experiment.

The net displacement of the plankton, as well as the speed, were analyzed. A figure showing the average net displacement for the first 30 seconds of the traces can be found in figure 4.21. A histogram, comparing the average speed of the original and the drift compensated tracks, can be found in figure 4.20.

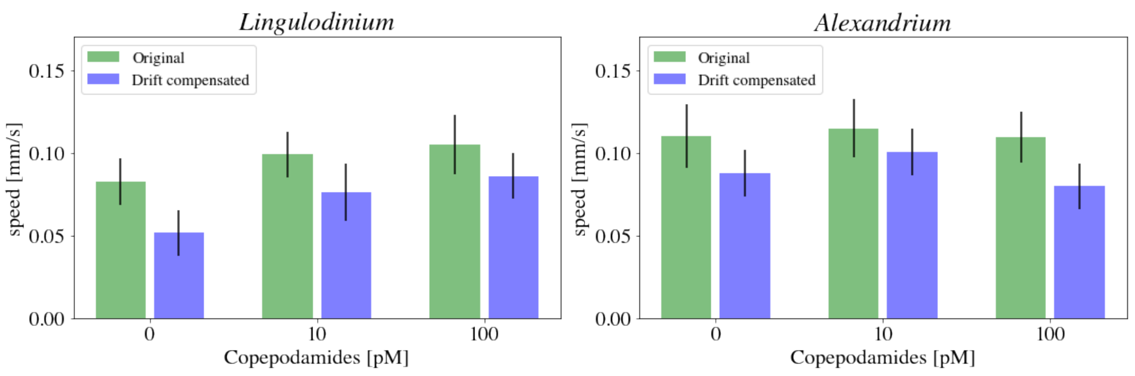


**Figure 4.20:** A histogram representing the mean speed of the trajectories from the different experiments. The original traces (green) and the drift compensated traces (blue). The black bars represent the standard deviation.

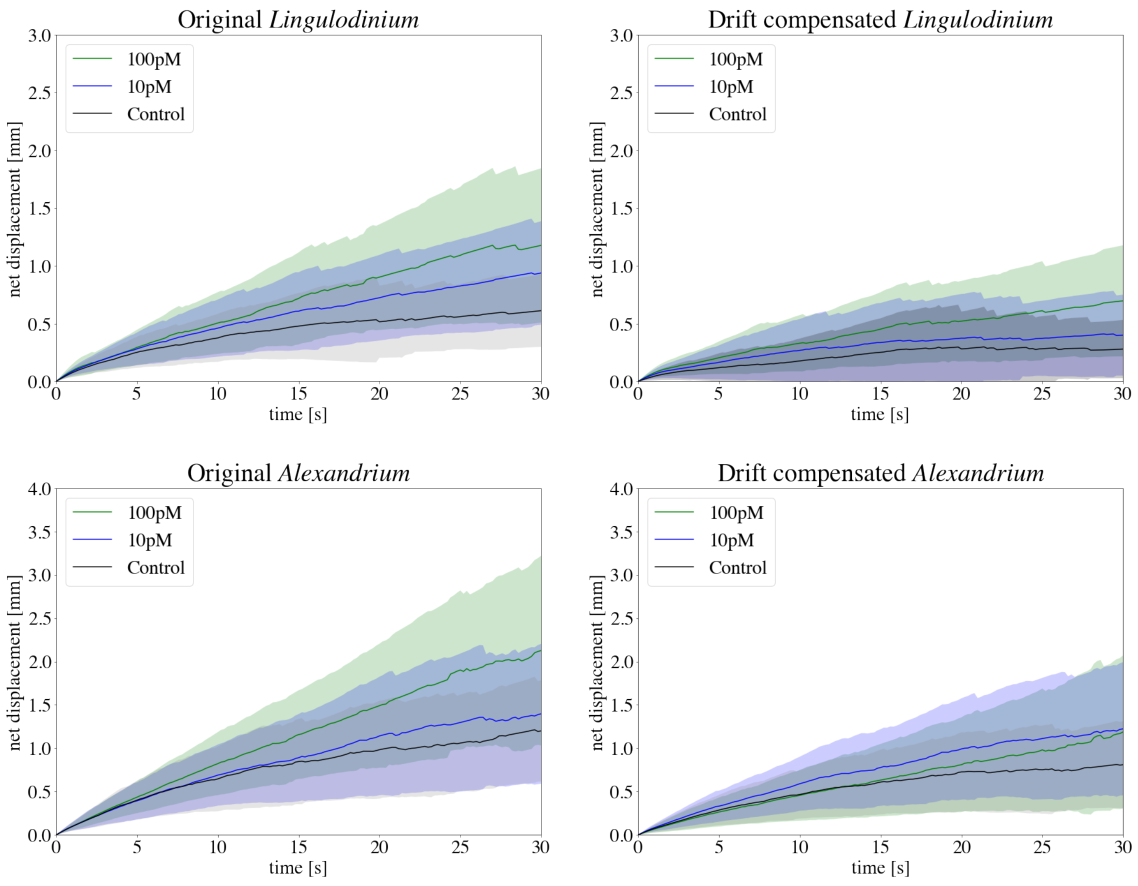


**Figure 4.21:** The net displacement of the different plankton types for both the original and the drift compensated traces. The solid lines are the mean net displacement calculated from all four replicates. In each subplot, the three lines correspond to the different amount of copepodamides the two plankton types were exposed to. The shaded areas represent the standard deviation.

### 4.3.2 Multiple type plankton tracking

An example of a frame from the videos from the multi type plankton experiments can be seen in Figure 4.16b. Only the larger plankton, *Strombidium*, was tracked. The net displacement of both the original and the drift compensated traces can be seen in Figure 4.22. The result of the analysis of the average speed can be seen in Figure 4.23.

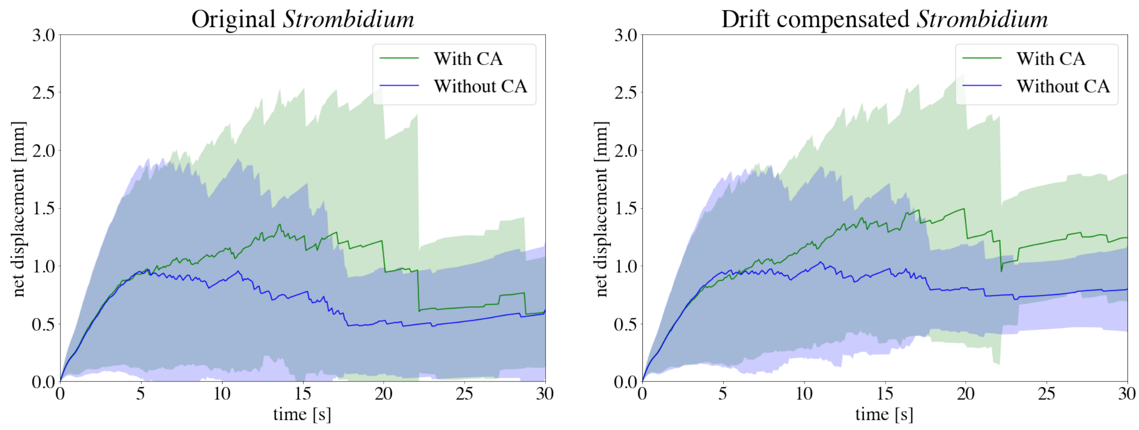


**Figure 4.22:** Net displacement of *Strombidium* with and without the additions of copepodamides (CA). The left figure shows the original tracks and the right the drift compensated. The solid lines represent the mean net displacement and the shaded areas represent the standard deviation.

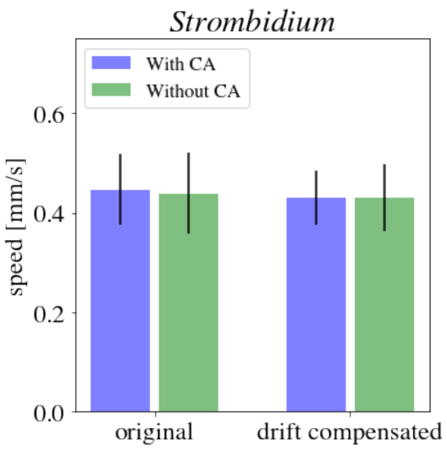


**Figure 4.23:** The mean speed of *Strombidium* with and without the addition of copepodamides (CA) on both the original and the drift compensated traces. The black bars represent the standard deviation.

# 5 Discussion

### Model evaluation

The model was evaluated by testing its capability to find and distinguish particles under difficult imaging conditions.

Section 4.1.1 describes the result from the training evaluation, where the same network was trained using different number of input images. Comparing the results from the 2000 batch training in Figure 4.3 and the 100 batch training in Figure 4.1, we can see a significant difference in the predictions of the two particles. In the 2000 batch training, the network manages to label each particle correctly with a high prediction. However, in the 100 batch training, the predictions for particle 1, which is the larger of the two particles present, are still correct, though not as accurate as the longer trained network. Particle 2, the smaller of the two particles present, is not given correct labels. Ergo, the predictions after 100 batches of input data is not accurate enough to be able to correctly distinguish between the small and the large particle.

Considering the 400 batch training results in Figure 4.2, we can see that the results have dramatically improved. The network's learning rate is not linear and during the first batches of input data, the loss decreases the most, as described in section 2.1.1. This raises a question; how much training is enough? Which brings us to the importance of avoiding overfitting the model. The goal is to train the model enough to recognise the particles, but not too much so that the model won't recognise a particle if it is slightly outside the range on which the model was trained. This becomes even more important for experimental data, which will be discussed more in detail later on.

The networks capability to distinguish particles in difficult noise and lightning conditions were tested and the result can be found in sections 4.1.2 and 4.1.3, respectively. What is worth noting is that the network is able to predict the larger particle, particle 1, correctly in all of the signal-to-noise ratios. However, at a signal-to-noise ratio of 5, seen in Figure 4.6, the particle 2 predictions are too far from the truth, to the point that they are no longer useful. In Figure 4.5, we can see that, even though there are some false predictions, the highest predictions matches the truth. The result in Figure 4.5 can therefore be used, if the limit is chosen accordingly. The network performance on images with inhomogeneous lighting conditions were also successful in the cases wth lower gradients found in Figure 4.7 and 4.8. When the gradient became larger, see Figure 4.9, the network had difficulties both finding the positions and distinguish the particles in the darkest part of the image.

The network successfully distinguishes particles of different sizes, as long as the size difference is sufficiently large. The evaluation of the network's capability to distinguish particles of different sizes was made on spherical particles with 25%, 15% and 5% size difference. As seen in Figure 4.10, a 25% size difference between the particles was an easy task for the network, and all particles were labeled correctly. However, the predictions for the particles with a 15% difference had some of the particles labeled incorrectly, as seen in Figure 4.11. A smaller size difference proved even more difficult and at a 5% difference the network could not tell the two particle types apart, as seen Figure 4.12.

Similarly, the network managed to correctly label the particles with a shape difference of a spherical particle with a radius of 8 pixels and an elliptical particle with radius 8 and 2 pixels, as seen in Figure 4.13. When the elliptical particle became more rounded, with radii of 8 and 4 pixels, the network could no longer tell them apart, as seen Figure 4.14.

It is worth noting that the particles were not all located at the focal plane, but could be randomly placed 10 pixels above or below it. When the particles are very similar and further form the focal plane, they are hard to distinguish, which is proven by the results from the size and shape difference. The more similar they are, the harder they are to distinguish, but when the particles are also out of focus, it is an even bigger challenge. When simulating the particles, one have to consider how much the real particles are placed above and below the focal plane to reduce the risk of false predictions of the experimental data.

It is important to note that to quantitatively simulate experimental images of plankton, the detailed internal structure of the plankton needs to be incorporated into the simulation. Further, since the plankton are significantly larger than the wavelength of the illuminating light, the light propagation through this complex structure would have to be simulated. In this project, the approach has been to generate images which resemble the experimental images of plankton, rater than to provide exact simulations. Generalizing the larger plankton with the synthetic simulations provides the model with enough input data for training, but this also presents a user bias. Firstly, making the simulations look as much as the original image as possible puts more pressure on the user to make sure that the same number of pixels are occupied as a plankton in both the real and simulated image. Instead of giving the model the exact size of the plankton to be simulated, the user have to manually count pixels and then tweak the simulations accordingly.

Secondly, we are trying to simulate a plankton, which has a size much larger than the light propagating through it. However, what we are actually simulating are particles, which are in the same size range as the light's wavelength. Since the light is not going to be propagated the same way through both of the particle types, it puts a larger responsibility on the user to fine-tune the parameters to make the simulated image resemble the real one as much as possible. This also present a user bias.

This leads us to the next problem; overfitting. As stated previously, it is easy to assume that the longer a network is trained, the better it will perform. However, we are not simulating the exact versions of the plankton in the images, but synthetic versions that do not scatter light the same.

If we train the network too much, it may stop to recognise the plankton, since they are not perfectly represented by the simulated images. Even though the user is able to tweak the parameters very well, it is still not going to be the exact replica. In Figure 3.1, we can see that the simulations does not exactly match the real image. These simulations gave a good result after being trained with 4000 batches of images. The results might not have been the same if the model was trained for a longer period of time. Another problem with overfitting is that the user have to be careful and include a quite broad range of particle sizes to account for the fact that plankton, from a viewers perspective, will change size. An elliptical plankton will look elliptical when viewed from above, but viewed from the front, it might look spherical. This has to be included in the simulations, and the risk of overfitting the model has to be considered. Unfortunately, too wide a range of particle sizes might affect the model's ability to distinguish the different types.

### Evaluation on experimental data

The evaluation of the model on experimental data yielded some very good results when it came to finding and labeling the larger plankton *Strombidium arenicola* correctly. In table 4.1 and Figure 4.15a, we saw that the network managed to correctly label all of the *Strombidium*. This indicates that the network can be used to specifically track one, larger subspecies despite the presence of a large amount of smaller plankton of a different species. This made it possible for us to obtain the results from the multiple plankton experiment described in section 4.3.2.

The network did not perform as well when it came to the detection of *Rhodomonas baltica*. Because the *Rhodomonas* were very densely dispersed in the volume, the network struggled to detect each single plankton. In cases where there were two, or more, smaller plankton very close together, the network would label all of them as only one particle. In some cases, the network would also label part of the background as a particle.

It is important to note that the ”correct” positions are determined manually. In Figure 3.4, it is easy to see which dots are the larger *Strombidium*, but to manually distinguish and label the small *Rhodomonas* is difficult. The network found 415 positions belonging to *Rhodomonas*. Of these 415 positions, 245 were ”correct”, according to the manual labeling. Furthermore, 170 of the positions, found by the network were manually labeled as background. Even though it was easy to spot where the network failed, in the cases where it labeled many particles as one, there is no real way for us to see if a position manually labeled as background is correct or not. Furthermore, the network labeled 120 of the manually labeled *Rhodomonas* as background. Most of these lables came from positions where the density of the plankton was high and only one of the plankton were labeled. The manual labeling was done before the networks predictions were seen, to limit the bias.

**Testing software on experimental data**

The tracking software was tested on two experiments containing different types of plankton. This section will discuss the results from the single type plankton experiment, which had already been analyzed using the algorithmic method TrackMate, and the multiple type experiment.

From the single type plankton experiment, both the number of traces and the mean length were obtained using both the previously used TrackMate method, and the Deeptrack method. The analysis of the number of traces, as seen in table 4.2, showed that the Deeptrack method had 32% fewer traces. The average length of the traces, as seen in table 4.3, was 10.6% longer, the traces was otherwise the same. Longer, but fewer traces suggest that the Deeptrack method is more capable to keep tracking the plankton and not lose them, when they swim slightly out of focus.

Figure 4.18 shows the net displacement, where we can see that the results obtained from both methods, for both of the plankton types *Lingulodinium* and *Alexandrium*, are very similar. The mean speed obtained from the same queries, seen in Figure 4.17, shows that the Deeptrack method results in a lower mean speed, compared to the TrackMate results. The speed was only calculated on traces longer than 5 seconds. Since the Deeptrack method got more traces that exceeded this length, the mean speed was calculated on more traces using this method. This might be the reason for lower mean speeds in this case, simply because it is more accurate. Note that this is not the case for the control group of the *Alexandrium* experiment.

The single type plankton experiments had a lot of drift in them, meaning that the medium, in which the plankton were suspended, had a large movement, causing more drift. To compensate for this, the software included a drift compensating feature, as described in section 3.3. In Figure 4.19, the original traces can be seen to the left (blue) and the drift compensated tracks to the right (green). The mean speed of the original and the drift compensated traces can be seen in Figure 4.20. Not surprisingly, the subtracted drift reduces the average speed of the plankton. The net displacement of the traces are also lowered, as seen in Figure 4.21. The original traces showed a tendency for the plankton to have a longer net displacement the more copepodamides were present in the experiment. However, in the drift compensated *Alexandrium*, this is not the case. Instead of the experiments with 100pM copepodamides having the longest net displacement, the 10pM experiment tends to have a longer net displacement.

In the multiple type plankton experiment, containing both *Strombidium arenicola* and *Rhodomonas baltica*, only *Strombidium* was analyzed. This was because of the networks inability to correctly label the smaller *Rhodomonas.*

In Figure 4.16b, the larger dots are *Strombidium* and the smaller dots are *Rhodomonas*. The net displacement of the traces for both the experiment, with and without copepodamides, can be seen in Figure 4.22. Both the original and the drift compensated traces can be seen in this figure. Surprisingly, the drift compensation resulted in a longer net displacement, especially for the experiment without copepodamides. In the single plankton experiments, the drift compensation yielded a shorter net displacement. The larger *Strombidium arenicola* were not the only plankton present in the experiments, but they were the only ones used for the drift compensation calculations. If the movements of the plankton were random, the drift compensation would have been none. The drift function, which is subtracted from the traces, is calculated on the average movement of each particle between each frame, as described in section 3.3. Nevertheless, the net displacement figures suggest that the movement is not random, since the two figures in 4.22 are different. To acquire a more accurate drift function, both the larger *Strombidium* and the smaller *Rhodomonas* should have been included in the calculations. However, because of the high density of the *Rhodomonas* plankton in the experiment, it was not possible to get accurate traces from these.

As discussed in the evaluation of the model on experimental data, the model managed to label 67% of the small plankton correctly, according to the manual labeling. However, 40% of the total positions found by the network were false predictions. We therefore decided to only track the larger *Strombidium* in this experiment, and the movements of the *Rhodomonas* were not included in the drift calculations. As a result, the mean speed of the experiment showed very little difference between the drift compensated and the original traces. There was also a very small difference between the experiment with and without copepodamides, as seen in Figure 4.23.

The optimized limit was chosen as the one which provided the longest traces, while also providing as few traces as possible. By choosing the limit in this way, we ensure that the detections are robust to small changes in particle position relative to the focal plane, while minimizing the number of false predictions by done the network. On the other hand, this choice of limit may introduce a bias toward the particle sizes chosen in the simulations. As discussed previously, the simulations are not always the truth. Nevertheless, when looking at the results from the single plankton experiment, we see that the results are improved using Deeptrack compared to TrackMate. The traces we get from Deeptrack are longer and less prone to lose detections. However, the final results on the speed and the net displacement were marginal.

With results so similar, a new question is raised: is the Deeptrack method worth it? Both the Deeptrack and the TrackMate method took approximately the same time to analyze a video. Not including the training of a network, which has to be done for Deeptrack. When a network has been trained, it can be used for all the experiments, making the training a one time obstacle. The results yielded from the single plankton experiments might not have been superior enough to make it worth the endeavor of training. However, the possibilities of tracking multiple particles in much worse conditions is an ability which most algorithmic methods are unable to do. The difficulties of simulating the particles correctly is also a task that, similar to most other methods, becomes easier with practice.

This project has proven that the deep learning approach to tracking plankton is a successful method to use. Nonetheless, the results in this project gives rise to further questions. The network was not able to correctly label all of the smaller plankton in the dense, multiple type plankton experiment. In this project, the image was only successfully upsampled to twice the original size. It could be interesting to see if the network would have a greater success rate on labeling the smaller plankton, if the images were upsampled even more. Previous trials with upsamplings showed the greatest success with an upsampling of 4 times the original size. Unfortunately, this was not tried in this project, due to limited computer capacity.

Another interesting aspect to consider for future projects, is to implement another type of network to find the particles. A recurrent neural network, RNN, could be something to consider for tracking plankton. This network is able to consider the movement of the object simulated, which would be helpful when trying to track and predict the movement of plankton.

Furthermore, it is possible that the results could have been further improved, had the training images captured the light scattering properties of the plankton more precisely. An interesting future project could therefore be to see if this type of network could be trained using real images of plankton, instead of simulated ones.

# 6 Conclusion

In this project, a network with a U-net architecture was used to track and distinguish plankton of different types. The network was trained on simulated images of plankton. The output from the network were segmented images of the different plankton species present in the input image. These segmented maps were used to find the positions of the plankton with a connected components algorithm and the positions were connected between frames using a linear sum assignment algorithm.

Evaluation of the performance of the network on simulated images demonstrated the capability to find and distinguish different particles in difficult situations, such as low signal-to-noise ratios and similar particle sizes. Even the evaluation on experimental data showed the model's capability to distinguish particles. However, pinpointing the exact positions of individual particles in dense particle images proved difficult.

The tracking software was tested on a single type plankton experiment and the traces obtained from the analysis were compared with previous result using an algorithmic method. We could conclude that the Deeptrack method was slightly superior, yielding longer, but fewer tracks. The multiple type plankton experiment was successful and the software managed to trace large plankton in a sea of smaller ones. However, this assay also showcased the drift compensation complications which might occur if not enough particles are considered in the drift calculation.

The limitations of the model were discussed and an important point to be made is the complications of simulating larger plankton as small particles. One obstacle is for the user to tweak the simulations so that they resemble the plankton as much as possible. However, there will always be differences, so avoiding overfitting of the model is crucial.

Some ideas for future projects were discussed. Using a recurrent neural network, RNN, to track particles would be an interesting future project since this network also incorporates the movements of the particles when training. Another interesting project could be to expand the U-net so that it could be trained on real images of plankton, since the synthetic simulations caused some limitations with the current version.

# Bibliography

[1] J. C. Crocker and D. G. Grier, "Methods of digital video microscopy for colloidal studies", Journal of colloid and interface science **179**, 298–310 (1996).

[2] J.-Y. Tinevez, N. Perry, J. Schindelin, G. M. Hoopes, G. D. Reynolds, E. Laplantine, S. Y. Bednarek, S. L. Shorte, and K. W. Eliceiri, "Trackmate: an open and extensible platform for single-particle tracking", Methods **115**, 80–90 (2017).

[3] G. Bianco, M. T. Ekvall, J. Bäckman, and L.-A. Hansson, "Plankton 3d tracking: the importance of camera calibration in stereo computer vision systems", Limnology and Oceanography: Methods **11**, 278–286 (2013).

[4] J. C. Ho and A. M. Michalak, "Challenges in tracking harmful algal blooms: a synthesis of evidence from lake erie", Journal of Great Lakes Research **41**, 317–325 (2015).

[5] J. M. Newby, A. M. Schaefer, P. T. Lee, M. G. Forest, and S. K. Lai, "Convolutional neural networks automate detection for tracking of submicron-scale particles in 2d and 3d", Proceedings of the National Academy of Sciences **115**, 9026–9031 (2018).

[6] S. Helgadottir, A. Argun, and G. Volpe, "Digital video microscopy enhanced by deep learning", Optica **6**, 506–513 (2019).

[7] M. D. Hannel, A. Abdulali, M. O'Brien, and D. G. Grier, "Machine-learning techniques for fast and accurate feature localization in holograms of colloidal particles", Optics express **26**, 15221–15231 (2018).

[8] B. Midtvedt, *Soft matter lab/deeptrack 2.0*, 2020.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, `http://www.deeplearningbook.org` (MIT Press, 2016).

[10] A. Karpathy et al., "Cs231n convolutional neural networks for visual recognition", Neural networks **1**, 1 (2016).

[11] A. Oppermann, *Activation functions in neural networks*, (2019) `https://www.deeplearning-academy.com/p/ai-wiki-activation-functions` (visited on 05/13/2020).

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: towards real-time object detection with region proposal networks", in Advances in neural information processing systems (2015), pp. 91–99.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition”, Proceedings of the IEEE **86**, 2278–2324 (1998).

[14] U. Karn, *A quick introduction to neural networks*, (2016) `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/` (visited on 05/13/2020).

[15] Wikipedia, *Kernel (image processing) — Wikipedia, the free encyclopedia*, (2020) `https://en.wikipedia.org/wiki/Kernel_(image_processing)` (visited on 05/18/2020).

[16] O. Ronneberger, P. Fischer, and T. Brox, “U-net: convolutional networks for biomedical image segmentation”, in International conference on medical image computing and computer-assisted intervention (Springer, 2015), pp. 234–241.

[17] H. Lamba, *Understanding semantic segmentation with unet*, (2019) `https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47` (visited on 05/14/2020).

[18] H. Sankesara, *Unet, introducing symmetry in segmentation*, (2019) `https://towardsdatascience.com/u-net-b229b32b4a71` (visited on 05/15/2020).

[19] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, “Deep neural networks segment neuronal membranes in electron microscopy images”, in Advances in neural information processing systems (2012), pp. 2843–2851.

[20] A. W. Visser and T. Kiørboe, “Plankton motility patterns and encounter rates”, Oecologia **148**, 538–546 (2006).

[21] E. Selander, H. H. Jakobsen, F. Lombard, and T. Kiørboe, “Grazer cues induce stealth behavior in marine dinoflagellates”, Proceedings of the National Academy of Sciences **108**, 4030–4034 (2011).

[22] E. Bagoien and T. Kiorboe, “Blind dating—mate finding in planktonic copepods”, I. tracking the pheromone (2005).

[23] E. Selander, J. Kubanek, M. Hamberg, M. X. Andersson, G. Cervin, and H. Pavia, “Predator lipids induce paralytic shellfish toxins in bloom-forming algae”, Proceedings of the National Academy of Sciences **112**, 6395–6400 (2015).

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

[25] J. Lindström, W. Grebner, K. Rigby, and E. Selander, “Effects of predator lipids on dinoflagellate defence mechanisms-increased bioluminescence capacity”, Scientific reports **7**, 1–9 (2017).

[26] P. G. Falkowski, R. T. Barber, and V. Smetacek, “Biogeochemical controls and feedbacks on ocean primary production”, Science **281**, 200–206 (1998).

[27] J. H. Landsberg, “The effects of harmful algal blooms on aquatic organisms”, Reviews in Fisheries Science **10**, 113–390 (2002).

[28] A. Rosenfeld and J. L. Pfaltz, “Sequential operations in digital picture processing”, Journal of the ACM (JACM) **13**, 471–494 (1966).

[29] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment problems, revised reprint*, Vol. 106 (Siam, 2012).

[30] R. A. Pilgrim, *Munkres' assignment algorithm*, (2018) `http://csclab.murraystate.edu/~bob.pilgrim/445/munkres.html` (visited on 05/18/2020).

[31] D. Allan, C. van der Wel, N. Keim, T. A. Caswell, D. Wieker, R. Verweij, C. Reid, Thierry, L. Grueter, K. Ramos, apiszcz, zoeith, R. W. Perry, F. Boulogne, P. Sinha, pfigliozzi, N. Bruot, L. Uieda, J. Katins, H. Mary, and A. Ahmadia, *Soft-matter/trackpy: trackpy v0.4.2*, version v0.4.2, Oct. 2019.