CHALMERS
UNIVERSITY OF TECHNOLOGY

Real-Time Multi-Object Tracking and Segmentation with Generated Data using 3D-modelling

Master's thesis in Complex Adaptive Systems

OLLE FAGER

PHYSICS DEPARTMENT

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Real-Time Multi-Object Tracking and Segmentation with Generated Data using 3D-modelling

OLLE FAGER

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2021

Real-Time Multi-Object Tracking and Segmentation with Generated Data using 3D-modelling
OLLE FAGER

Supervisor: Giovanni Volpe, Physics Department
Examiner: Giovanni Volpe, Physics Department

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

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Real-Time Multi-Object Tracking and Segmentation with Generated Data using 3D-modelling

OLLE FAGER
Department of Physics
Chalmers University of Technology

Abstract

Multi-Object Tracking and Segmentation (MOTS) is an important branch of computer vision that has applications in many different areas. In recent developments these methods have been able to reach favorable speed-accuracy trade-offs, making them interesting for real-time applications. In this work different deep learning based MOTS methods have been investigated with the purpose of extending the DeepTrack framework with real-time MOTS capabilities. Deep learning methods rely heavily on the data on which they are trained. The collection and annotation of the data can however be very time-consuming. Therefor, a pipeline is developed and investigated that automatically produces synthetic data by utilizing 3D-modelling. The most accurate tracker achieves a MOTSA score of 94 and the tracker with the best speed-accuracy trade-off achieves a MOTSA score of 88. It is also observed that satisfactory results can be achieved in most situations with a quite general data generation pipeline, indicating that the developed pipeline could be used in different scenarios.

Keywords: deep learning, neural networks, multi-object tracking and segmentation, synthetic data, PointTrack, SipMask.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Multi-Object Tracking (MOT) is an important branch of computer vision that deals with the task of identifying multiple objects in a video and tracking their movement. It is useful in many different applications such as video analysis, robotics and human-computer interaction. For the most part existing methods track with bounding boxes [1, 2, 3], meaning that each object is represented with a surrounding box. However recently methods have started using segmentation of the objects, instead representing the objects with masks on a pixel-level [4, 5, 6]. This kind of MOT is called Multi-Object Tracking and Segmentation (MOTS). With the MOTS approach the objects are then more finely represented. This makes it easier for the tracker to discriminate between objects, particularly in situations where they get close to each other or overlap. It can also be desirable in applications where visualizing the tracking is key.

The task of producing masks for each object in an image is called instance segmentation. It is this part of the MOTS task that during inference is the most time-consuming. For a MOTS method to reach real-time speeds it is therefor important to use an efficient instance segmentation method. Instance segmentation can be split up into two categories, two-stage and single-stage. Two-stage methods [7, 8] usually produce object proposals in one stage and then in a second stage perform classification and produce the masks. One-stage methods [9, 10] however find different ways of skipping the object proposal step, enabling them to do classification and produce masks in one stage. Because of this difference one-stage methods are generally much faster than two-stage methods but also lag in accuracy. In recent developments however, one-stage methods have been able to reach favorable speed-accuracy trade-offs [11, 12, 13].

A trackers performance does not just rely on the method used but also heavily on the data on which it is trained. Therefor it is crucial to provide high quality and abundant training data. This however poses a problem, as manual annotation of MOTS data quickly becomes very time consuming. To combat this, various methods for generating synthetic data can be used. One such method that has

shown promising results utilizes 3D-modelling software [14, 15]. By recreating the objects and the environments in 3D, different settings can automatically be varied and the data automatically annotated. In this way large amounts of data can be produced in a short amount of time.

1.2 Aim

This thesis has two primary aims. One is to extend the DeepTrack framework [16] with state-of-the-art real-time MOTS capabilities. The other is to create a data generation pipeline that as input takes 3d-models of the objects to be tracked and produces annotated training data for these objects. Then combining these two the aim is to end up with a quality tracker with real-time MOTS capabilities that is also easy to train.

2

Theory

2.1 Artificial neural networks

As the name suggests artificial neural networks (ANN), also called just neural nets, are inspired by the neural networks found in animal brains. Although ANN's are much simpler they still share some structural similarities. A biological neural network is built up of four main components, neurons, axons, synapses and dendrites. The neurons are the nodes of the neural network. Each neuron gets input, electrical signals, from a large number of the other neurons via its dendrites. From these an output is produced and sent out through the neurons axon. The axon then branches out and ends up in many different synapses, which connect to dendrites of other neurons. In this way the biological neural network forms a very intricate network of electrical signals. In artificial neural networks the neurons analogously receives input from multiple other neurons and produces a single output which is sent out to many other neurons. The connections between neurons are however not as complex. Although ANN's vary in complexity in this regard the most common structure is a layered structure where neurons in one layer only receive input from the immediately preceding layer and only outputs to the immediately following layer, see figure 2.1. Such networks are also called feed forward networks, stemming from its one directional property.

The first layer of a neural net is called the input layer. This layer feeds the input data to the network, such as an image. The last layer is called the output layer, which outputs the final result. The most simple neural net consists of only an input layer and an output layer. However, most commonly there are also intermediate layers called hidden layers. One hidden layer is for example needed to solve problems that are not linearly separable. Only one hidden layer can however for some problems lead to a large number of neurons, making the network inefficient. In some of these problems the number of neurons can be reduced, and thus the networks efficiency increased, by increasing the number of hidden layers. Networks with multiple hidden layers are also called deep neural networks.

The signals in an ANN are real numbers and analogous to the varying strength of synaptic couplings each connection has a real valued weight. Each neurons output is

computed by first combining the input signals and then feeding the combined input through a function called the activation function. The input signals are combined through a weighted sum. To this, a neuron-specific threshold is also added. So, if the output neuron is denoted by index i and the input neurons with index j the output O_i becomes

$$O_i = g(b_i) \quad \text{with} \quad b_i = \sum_j w_{ij} x_j - \theta_i$$

where w_{ij} is the connection weight, x_j the input value, θ_i the threshold and g the activation function. What activation function to use depends on the task at hand. A common one that is biologically inspired is the ReLU activation function. It is defined as

$$g(b) = \max(0, b)$$

and is based partly on that biological neurons only activate when the input strength reaches a certain threshold and partly on that the output strength of a biological neuron increases as the input strength increases.

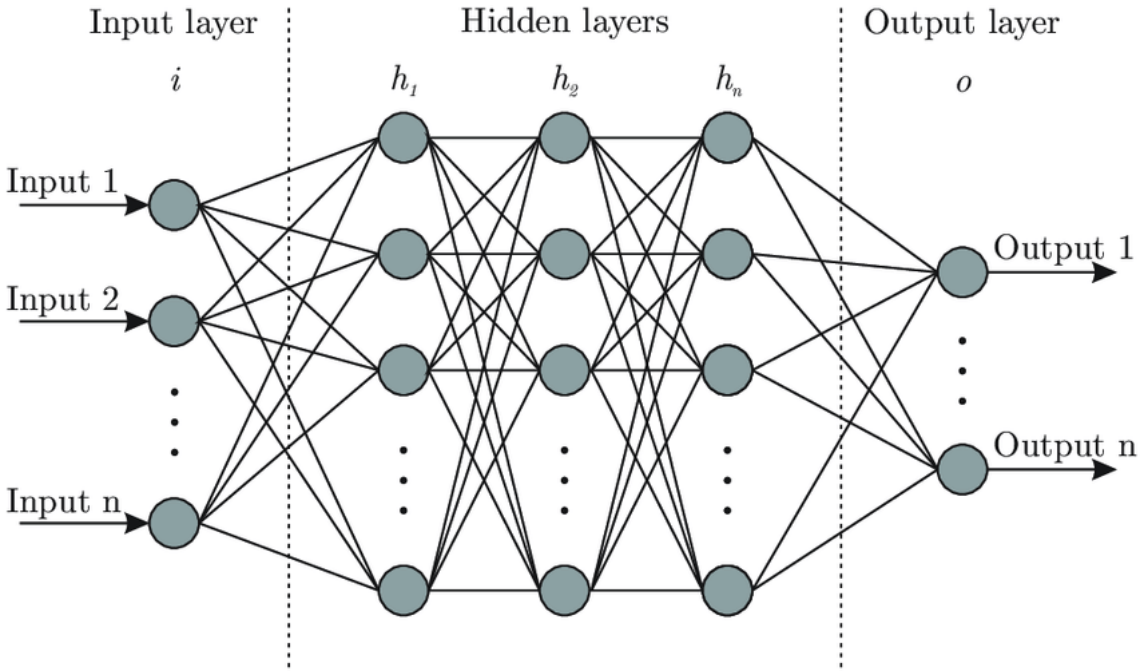

Figure 2.1: Artificial neural network architecture. Image reference: [17]

Starting at the input layer the outputs of the neurons are forwarded to the next and eventually the output layer is reached which produces the output of the neural net. So the final output is determined by all the wieghts w_{ij} and thresholds θ_i in

the network. These values are therefor the ones that are adjusted during training. How to adjust the values is most often determined by gradient-based methods and backpropagation. A differentiable loss function for the output is defined of which the gradient is calculated with respect to each weight and threshold by utilizing the chain rule. Each gradient then tells how much and in what direction to adjust the respective weight or threshold. To control the size of the adjustments each gradient is multiplied by a factor η called the learning rate. So for a weight w_{ij} the corresponding adjustment δw_{ij} becomes

$$\delta w_{ij} = -\eta \frac{\partial H}{\partial w_{ij}}$$

where H denotes the loss function. A high learning rate will speed up the training process but might lead to missed optimal states as large steps at each iteration are taken, thus decreasing accuracy. A low learning rate will slow down the training process but will also increase the potential of finding optimal states, thus increasing accuracy. The learning rate can be static for the entirety of the training process, but different methods for varying it during training are also frequently used. One such method is momentum, which increases the learning rate if the gradient direction has been the same for a long time and decreases it otherwise. Using such methods can improve the speed-accuracy trade-off of training.

2.2 Convolutional neural networks

Convolutional neural networks (CNN) are a type of ANN that are designed for the task of image recognition. As ANN's they are also inspired by biological neural networks, in this case the neural networks found in the visual cortex of animal brains. Applying a fully connected network to image data quickly becomes impractical as the number of neurons and connections quickly increases when the size of the images increases, leading to slow training and overfitting. By building the assumption that the input are images into the architecture, CNN's are able to greatly decrease the required number of parameters needed.

A CNN consists of one or more convolutional layers. The input to and output from a convolutional layer are images, most often 3-dimensional where the third dimension encodes the color of the images. Each pixel of each color channel are represented by a neuron which connect to the neurons of the convolutional layer. Unlike a fully connected layer each neuron in a convolutional layer is only connected to a set of neurons in the preceding layer. Also, all neurons in a convolutional layer share the same weights. This is realised with what is called a kernel. A kernel computes the convolution of a specific area of the image. This area is defined by a 2-dimensional rectangular field (usually square) called the receptive field. The kernel is then moved along the image producing outputs on its different parts which together form the kernels feature map. Each specific area of the image that the kernel operates on corresponds to a specific neuron in the convolutional layer and

thus defines the sets of neurons that the convolutional layer neurons are connected to. So the kernels feature map is actually the output of the convolutional layer. This is depicted in figure 2.2a. The weights of the connections are a part of the kernel, thus making all neurons in the convolutional layer share the same weights. This aspect of the convolutional layer drastically decreases the number of parameters needed as compared to a fully connected layer. However, a convolutional layer usually has multiple kernels, as shown in figure 2.2b, but despite this the parameter decrease is still significant. The kernel can be viewed as a feature extractor, where its weights define which feature of the image it will extract. The idea of using multiple kernels is then that the different kernels will learn to detect different features of the image which creates a better understanding of the image.

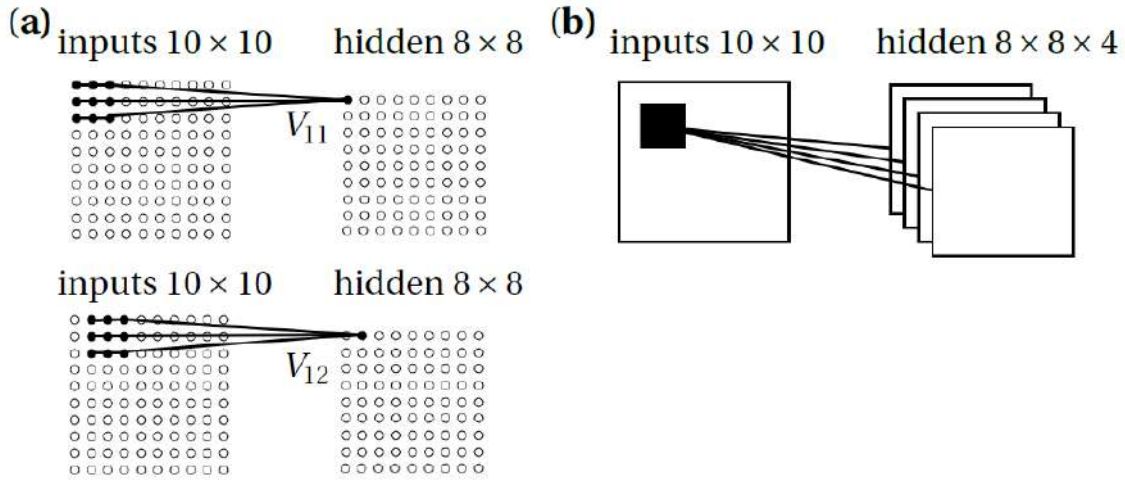

Figure 2.2: **(a)** Depiction of how a kernel processes the input image to produce the output, the feature map, in a convolutional layer. Here the input is a 10x10 image and the kernels receptive field a 3x3 square which produces an 8x8 image. **(b)** Shows an example where the convolutional layer has 4 kernels, thus producing 4 different feature maps. Image reference: [18]

2.3 Residual neural networks

The power of deep neural networks lies in their ability to solve complex problems. The different layers can be thought of handling different levels of the inputs features. Increasing the number of layers therefor enables the network to recognize more intricate features. However, it also comes with some problems, mainly two. One is that the deepest layers (layers closest to the input layer) might learn very slowly. This is because of the chain rule that is used when calculating the gradients in backpropagation. The gradients in a deep layer are effectively the products of many gradients and when the gradients are small they get exponentially smaller with the depth of the layer. This problem has therefor been named the *vanishing gradient problem*. The other issue that has been observed with deep neural nets is that as they are made deeper the accuracy starts to saturate and then even degrade. This problem is however a bit surprising as added layers should be able be constructed

as identity mappings, $f(x) = x$, and therefor in the worst case just lead to no improvement.

Residual neural networks (ResNet) [19], or residual nets, mitigate both of these problems. Having a base architecture that of a feed forward network, residual nets adds identity skip connections. These connections skip layers, usually two or three, and do not have any weights. They are parallel identity mappings of the deeper layers output. One such connection is shown in figure 2.3. If the mapping that the skipped layers would fit without the skip connection is denoted $H(x)$, they now with the skip connection fit the residual $F(x) = H(x) - x$, hence the name. The idea is that in the cases when an identity mapping is preferred, fitting the residual will be easier as the weights of the layers can simply be driven to zero.

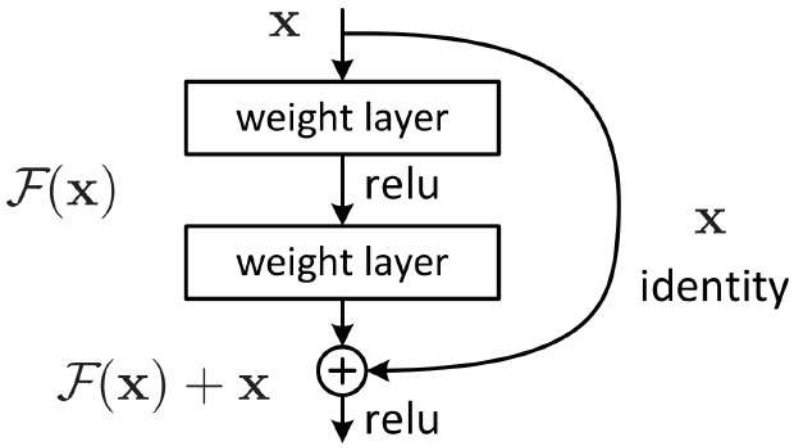

Figure 2.3: Residual neural network identity skip connection. Image reference [19]

Residual nets also deal with the other main problem, which is that in deeper networks the accuracy starts to saturate and then even degrade. In the residual net paper they propose that the problem might actually come from the fact that deep neural nets have difficulty to learn identity mappings. So by making it easier to learn identity mappings this problem is alleviated.

2.4 ERFNet - Efficient Residual Factorized Network

The ERFNet [20] is a convolutional neural network whose design is able to improve the speed-accuracy trade off for segmentation tasks. It does this by first implementing the design strategy of residual neural networks, adding identity skip connections. This improves the accuracy as explained in section 2.3 but the non-bottleneck and bottleneck residual blocks proposed in [19], see figure 2.4, still has a problem. As the network gets deeper the accuracy gains are not very large while the networks speed is still significantly reduced. To this end the ERFNet introduces a new residual

block called non-bottleneck-1D. The non-bottleneck block consists of two 2D convolutional layers, where 2D refers to the fact that the kernels are 2-dimensional. The non-bottleneck-1D block replaces each 2d convolutional layer with a combination of two 1D convolutional layers. This reduces the number of parameters needed and also enables more non-linearities to be included, effectively increasing the speed of the network while still maintaining equal accuracy to the non-bottleneck block. By building its architecture with these blocks ERFNet is thus able to produce accurate segmentations in real-time (over 83 FPS with a single Titan X GPU).

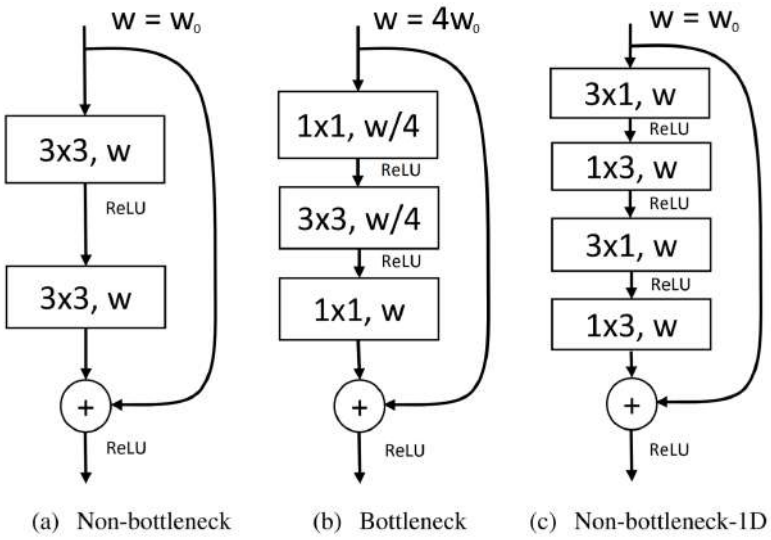

Figure 2.4: (a) and (b) depicts the original residual blocks proposed in [19] and (c) the new residual block used in ERFNet. Each layer is denoted by its kernel size a×b and w denotes the number of feature maps inputed to the layer. Image reference [20]

2.5 SpatialEmbedding

SpatialEmbedding is a single-stage and proposal-free instance segmentation method. It produces masks based on spatial information and is able to do so very accurately due to its novel loss function. Then they couple this with a fast architecture, the ERFNet encoder decoder architecture, enabling this method to also reach high speeds.

The method is based on the idea of grouping pixels to each instances centroid. To this end a 2D vector pointing towards an instance centroid is learned for each pixel. A common way to learn these offset vectors is by using a simple regression loss directly on them. This leaves the computation of centroids and assignment of centroids to pixels completely to post-processing at inference time. The centroid assignment can however be incorporated into the loss function by instead using a hinge loss. This incorporates a fixed margin around the centroid into the loss, within which the pixels offset vectors are forced to point. Having a fixed margin however

comes with the problem that, to be able to discern all objects, it has to be chosen in correspondence to the smallest object. This negatively affects the ability to learn larger objects because pixels further away from its centroid will have difficulty to point inside the margin. Therefor, what SpatialEmbedding does is that it instead of a fixed margin incorporates a flexible margin. So for each instance a specific margin is learned. This improves the ability to learn instances of different sizes as the margin should be proportional to the size of the instance. Seed maps for each class is also produced, in which each pixel is assigned a score based on how close its offset vector is to pointing at the instance centroid.

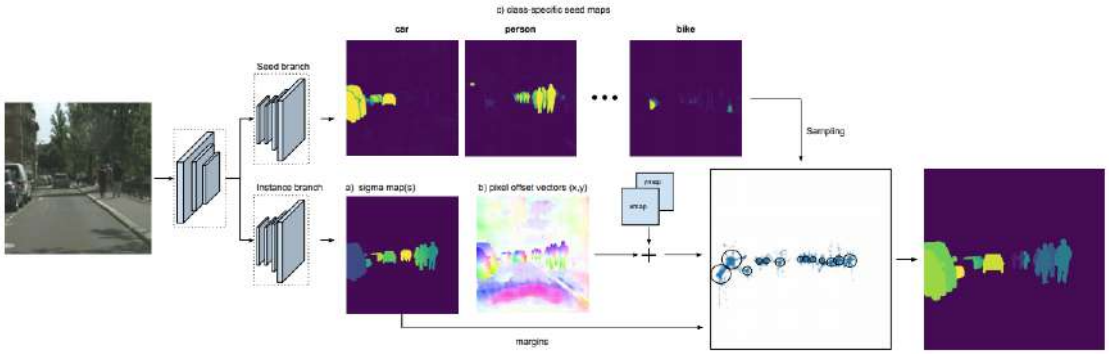

Figure 2.5: Architecture of SpatialEmbedding. At the base an ERFNet is used with a split decoder, creating two branches. One branch produces cluster margins and offset vectors. The margins sizes are shown on a color scale, where yellow is small and blue is big. The offset vectors are visualized by color-coding their angles. The other branch produces seed maps for each class. Here each pixel gets a score based on how close its offset vector is to pointing at the instance centroid. Yellow denotes a high score and blue low. The information from the two branches is then used to cluster pixels into masks. Image reference: [12]

As seen in figure 2.5 the ERFNet decoder is split into two branches. One branch produces the seed maps and the other the margins and the offset vectors. Then using the extracted information the pixels are clustered into masks. This is done separately for each class. First a centroid is sampled from the classes seed map by choosing the pixel with the highest seed score. Then pixels are clustered to this centroid using the corresponding margin. Finally the clustered pixels are removed from the seed map and the process is then repeated until all seeds are masked.

2.6 PointTrack

PointTrack [21] is built on the single-stage instance segmentation method called SpatialEmbedding [12]. The utilization of this method is what enables PointTrack to reach near real-time tracking. The main part of this tracking framework however, lies in the way it produces its instance embeddings. Unlike most other MOTS methods which in different ways uses convolutional networks, PointTrack extracts instance

embeddings by representing an image with 2D point clouds and describing each point with a number of descriptors. Given an instance with a segment and a bounding box the bounding box is first slightly enlarged equally in all four directions to include more of the surrounding environment. Then two point clouds are created, one with points from inside the segment and one with points from the area surrounding the segment. These point clouds are then processed in separate branches. This separation of the segment and its surroundings avoids the mixing of the two that occurs in convolution based methods, thus improving the discriminatory ability of the embedding. Each point in the segment point cloud is described by its offset, which is its position relative to the center of the segment point cloud, and its RGB color. Each point in the surroundings point cloud is, beyond the descriptors of the segment point cloud, also described by its category. The possible categories are the segmentation classes and also a background class. In the respective branches the points and its descriptors are then combined and their embeddings learned. Finally the embeddings from the two branches and also an embedded position of the instance in the image are concatenated and the instance embedding learned.

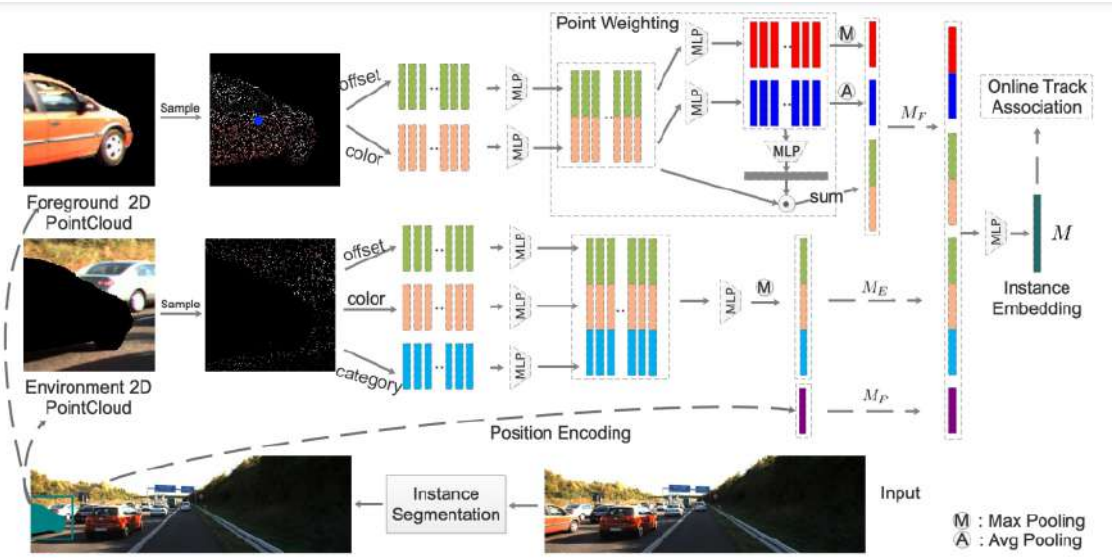

Figure 2.6: Outline of PointTrack. First instance masks are produced by the instance segmentation method. Then for each instance an embedding is constructed through 2d point clouds. Points inside the instance are separated from points in the surroundings and are processed in two different branches. The results from the two branches and also an encoded position of the instance are then combined into the final instance embedding. Image reference: [21]

To then perform instance association between frames a similarity measure S is defined. Given two segments C_i and C_j and their embeddings M_i and M_j their similarity is given by

$$S(C_i, C_j) = D(M_i, M_j) + \alpha U(C_i, C_j)$$

where D represents the euclidian distance and U the mask IOU. The similarities between all the instances in the current frame and the last instances of a set of saved tracks are then calculated and the Hungarian algorithm used to match the instances to each other. Each instance id has its track and the last instance of each track is kept in memory. If the instance id does not appear in a certain amount of consecutive frames it is considered to have gone out of frame and is removed from memory. There is also an association threshold that the similarity score have to be above in order to be matched. If a certain instance is not matched a new instance id is assigned to the instance and thereby a new track is also started.

2.7 SipMask

SipMask [13] is first and foremost an instance segmentation method but a version adapted for the MOTS task is also available. SipMask's instance segmentation is single-stage, making it a fast method capable of reaching real-time inference. It is based on the ideas of a different one-stage instance segmentation method called YOLACT [11]. The way YOLACT works is by dividing the job into two parallel branches. In one branch a fully convolutional network is used to create k "prototypes" the size of the image. This is similar to semantic segmentation, however k does not have to equal the number of classes and the loss is not calculated on the result of this branch but on the final result. This means that the prototypes are not directly linked to the actual instances. Exactly what they represent is not explicitly defined. The task of the other branch is to produce a set of k mask coefficients for each predicted bounding box. From these two branches the final result is then achieved by a linear combination of the prototypes and mask coefficients. SipMask also produces category-independent prototypes like YOLACT or as they call them, basis masks. However, instead of just a single set of mask coefficients for each bounding box SipMask produces multiple sets of coefficients for different sub-regions of each bounding box. Each predicted bounding box is split into a grid of sub-regions. To get a good trade-off between speed and accuracy SipMask uses a 2x2 grid, resulting in 4 sets of mask coefficients per bounding box. So for every bounding box first different parts of the mask are constructed through linear combination of basis masks and mask coefficients which then are combined to produce the final mask.

The SipMask architecture is shown in figure 2.7. It has a backbone consisting of a residual network (ResNet) [19] with a feature pyramid network (FPN) [22]. The output from the FPN then goes to two fully convolutional branches, the mask-specialized classification branch and the mask-specialized regression branch. The mask-specialized regression branch outputs the bounding boxes and the basis masks. The mask-specialized classification branch calculates classification scores on the bounding boxes regressed by the mask-specialized regression branch and the coefficients for the different sub-regions of each bounding box. The coefficients and the basis masks are then linearly combined as described above to produce the final result.

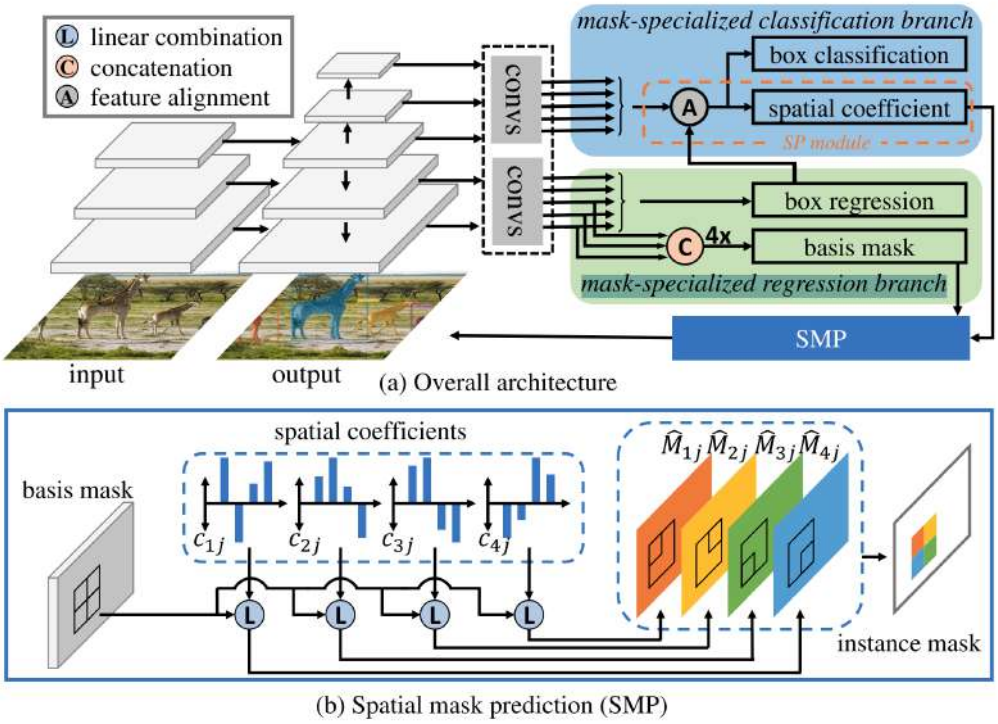

Figure 2.7: (a) SipMask architecture. The input image is fed into the backbone consisting of a residual net with a feature pyramid network. The output from the backbone then goes into two fully convolutional branches. The mask-specialized regression branch outputs bounding boxes and the basis masks. The mask-specialized classification branch takes the bounding boxes and calculates their classification scores and sub-region specific coefficients. (b) Here the linear combination of coefficients and basis masks is shown. The blue bars indicate coefficient values. In this example the bounding box is split into a 2x2 grid of sub-regions. Each set of coefficients is linearly combined with the basis masks and the results are then combined to produce the final instance mask.

In its adaption to MOTS SipMask adds another branch in parallel to its other two branches. This branch is also fully convolutional and produces feature vectors for all bounding boxes. Given a bounding box, its feature vector is created by extracting the center most value of the bounding box from all feature maps outputted by the convolutional network.

In the first frame all detected instances are assigned an instance id and for each instance id a new track is started. In each following frame matching is performed between the detected instances in the current frame and the last recorded instance of each track. The last recorded instance of each track is always kept in memory. The similarity measure between an instance I_i^c in the current frame with feature vector $\mathbf{v}_i^c$ and an instance I_j^p from the past tracks with feature vector $\mathbf{v}_j^p$ is given by

$$S = \log\left(\frac{\exp(\mathbf{v}_i^c \cdot \mathbf{v}_j^p)}{\sum_j \exp(\mathbf{v}_j^p)}\right) + c_1 C_i + c_2 U(I_i^c, I_j^p) + c_3 L(I_i^c, I_j^p) \tag{2.1}$$

where C_i is the classification score of the instance in the current frame, U the bounding box IoU and L is either 0 or 1 depending on if the instances labels match. Each instance in the current frame is matched to the instance giving the maximum similarity score S. If two instances match with the same instance, the one with the highest similarity score is chosen.

2.8 Metrics

2.8.1 Intersection over Union

The Intersection over Union (IoU) is a metric used to measure the similarity of predicted and ground truth instances. As shown in figure 2.8 the IoU of two instances is given by the area of their intersection divided by the area of their union. In the figure the IoU of two instances bounding boxes is shown, the IoU of two masks can however also be calculated in the same way.

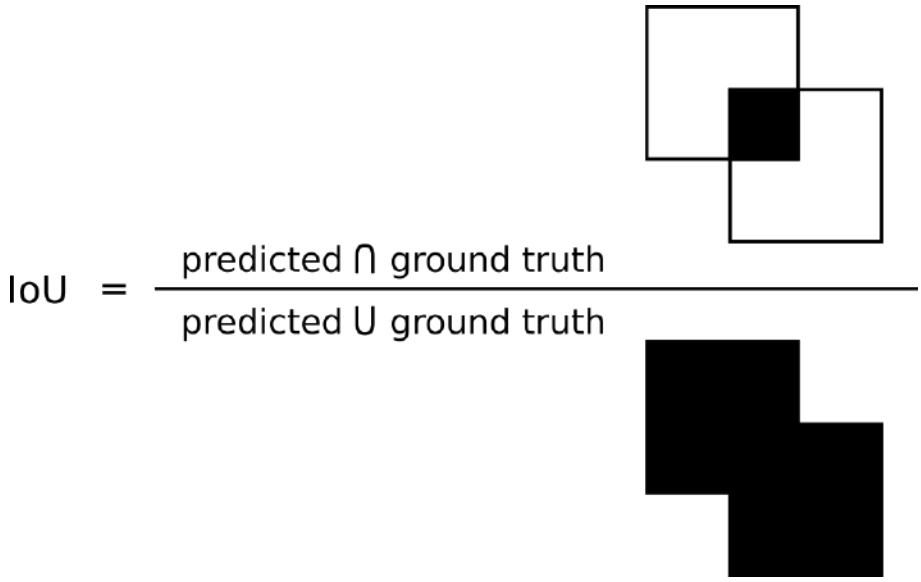

Figure 2.8: Definition of IoU metric visualized with bounding boxes.

2.8.2 Average precision

Average precision (AP) is the standard metric used for object detection tasks. However, it does not measure tracking performance but only detection performance. So, in this case it was used to evaluate the instance segmentation parts of the trackers. To describe this metric two more basic metrics, precision and recall, need to first be described. These metrics combine true positives (TP), false positives (FP) and false negatives (FN) in different ways. Precision is given by

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Thus, what precision tells us is how many of all positive detections are actually positive. This measure penalizes false positives and is therefor useful when it is important to avoid false positive detections. Recall is given by

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Thus, what recall tells us is how many of the actual positives are detected. It penalizes false negatives and is therefor useful when it is important to avoid false negatives.

Detected instances are classified as either true positives, false positives or false negatives based on their mask IoU's with the ground truth instances. In the COCO API REF, which is used to calculate the AP scores, the classification is performed by iterating through the set of detected instances and comparing them to all ground truth instances. In each iteration the detected instance is matched with the previously unmatched ground truth that it has the larges IOU with, as long as the IoU is also above a certain threshold. A matched detection is classified as a true positive and an unmatched detection as a false positive. A ground truth instance that have not been assigned any detection is classified as a false negative. The COCO API provides AP metrics with three different IoU thresholds. One, denoted AP@50, is calculated with an IoU threshold of 0.5. Another, denoted AP@75, is calculated with an IoU threshold of 0.75. The last one, denoted AP@[50:95], is actually an average of 10 AP metrics with IoU thresholds in the range 0.5 to 0.95.

To calculate AP the detected instances classification scores are also used. These are the scores that tell the confidence of the detected instances class assignment. A list is created of all the detected instances in all images sorted based on classification score in descending order. Iterating through this list, a precision and recall pair is calculated at each iteration. These scores are calculated based on the previous detections in the list. So for precision the total number of positives detected is equal to the number of previous detections in the list. Iterating through the list the recall will never decrease because the number of actual positives does not change. The precision will vary however, going up when a true positive is introduced and going down when a false positive is introduced. This zig-zag pattern is then smoothed out so that the precision never increases. Finally the AP score is obtained by sampling 101 equally spaced recall values, extracting the corresponding precision values and calculating the average of these extracted precision values. This procedure effectively penalises false detections with high classification score. The best possible AP score is 1 and is achieved when all the true positives also are the detections with the highest classification scores.

2.8.3 MOTSA and sMOTSA

To evaluate the results of the trackers two common metrics for the MOTS task are used. These metrics are built by in different ways utilizing the more basic metrics, true positives, false positives, false negatives and id switches (IDs). To calculate these more basic metrics the predicted masks need to be assigned to the ground truth masks. The MOTSA and sMOTSA metrics assumes that each pixel is assigned only one label. Therefor a predicted mask can simply be assigned to a ground truth mask as long as the IoU is above 0.5. So, each successful assignment is counted as a true positive, each unassigned predicted mask is counted as a false positive and each unassigned ground truth mask is counted as a false negative. These three metrics however only measure the performance of the instance segmentation. To also measure the tracking performance, id switches are included. Given two ground truth masks in consecutive frames whose instance id's are the same, an id switch occurs when the instance id of the corresponding predicted masks switches.

MOTSA: MOTSA is an accuracy metric and is defined by

$$\text{MOTSA} = \frac{\text{TP} - \text{FP} - \text{IDS}}{\text{M}}$$

where M denotes the number of ground truth masks.

sMOTSA: The name sMOTSA stands for soft MOTSA and it comes from that this metric uses a soft TP, $\widetilde{\text{TP}}$. The difference between TP and $\widetilde{\text{TP}}$ is that instead of just counting successful assignments the $\widetilde{\text{TP}}$ metric sums the actual IoU's of the successful assignments. The sMOTSA metric is then given by

$$\text{sMOTSA} = \frac{\widetilde{\text{TP}} - \text{FP} - \text{IDs}}{\text{M}}$$

where M denotes the number of ground truth masks.

3

Method

The Multi-Object Tracking and Segmentation (MOTS) methods and the data generation pipeline were investigated on a test setup made to resemble a scenario of sausages on a commercial grill. Being able to track the sausages in this scenario could for example make it possible to keep track of how long the sausages have been laying on the grill, which in turn could help avoiding sausages laying on the grill for too long. In the following sections it is described how data for this scenario was generated and how the different tracking methods were tested.

3.1 Data generation

The data was generated using the open-source 3D-creation suite Blender. It comes with a powerful Python API which gives the user control of many properties related to scene creation in the suite. Utilizing this API, the scripts for automatic scene creation were written. The last component of the data generation pipeline was the bpycv python package. This package enabled the automatic annotation. From a rendered blender scene this package can produce different kinds of annotations. One of those are mask annotations which were used in this case, see figure 3.1.

Figure 3.1: Result from automatic annotation using the bpycv package. The different shades of grey represents different instance id's.

3.1.1 Sausage models

As a sausage has a simple shape and not a lot of different details the sausage model was created from scratch. At the ground level a 3D-model is made up of a number of connected polygons in 3D-space. This basic building block is called the mesh. First a low-poly mesh, a mesh made up of a small amount of polygons, of the basic shape of the sausage was created. Then, to add more realism some irregularities were added to the surface through sculpting on a high-poly version of the mesh. However, to save on rendering time the high-poly mesh was not used directly but instead transformed into a high-resolution normal map. A normal map is an RGB-image where each pixels RGB-value represents the 3-dimensional direction of the surface. When the normal map is applied to the low-poly mesh the actual shape of the model is not changed but it will tell the render engine that light should interact with the object as if it was. To add some variation, when creating each sausage the strength of the normal map was varied.

Next, color was added to the sausage. As sausages come in different colors and also can change color while being cooked, variability was prioritized over realism for this. Instead of painting a number of different color maps, an automatic generation process was created. First a color was randomly chosen from three defined color ranges, one with more orange, one with more pink, and one darker. This color was then combined with a noise image to introduce some variation before being applied to the model.

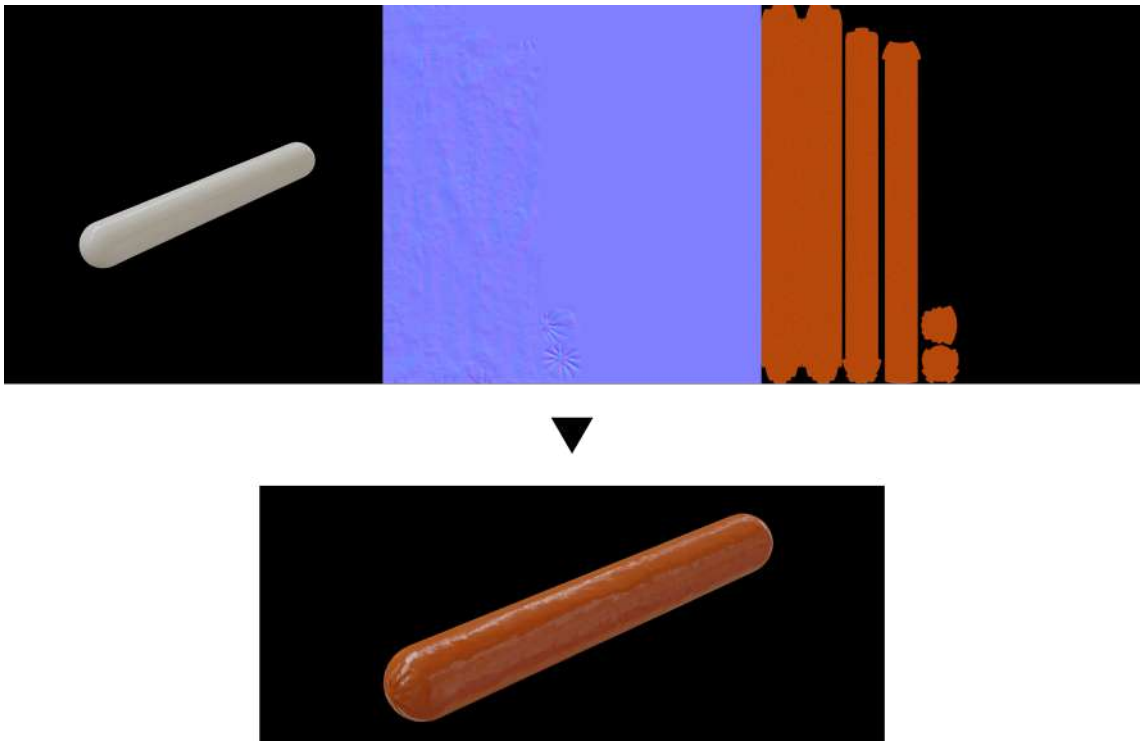

Figure 3.2: Example of sausage model composition. A color map and a normal map is combined with a low-poly mesh to produce the final sausage model.

Commonly also a roughness map is used to create the texture of the model. The roughness map is a grayscale image where the pixel values define how sharp reflections of the surface should be. However, a sausage consists of only one "materia" and should therefor reflect light in the same way all over its surface. So no roughness map was used but a single roughness value was set for the whole model. This value was however varied from one sausage to another, as different sausages can vary in reflectiveness.

3.1.2 Scene environment

The sausages were placed on a slab with a simple flat metallic surface, see figures 3.4 and 3.5. This "stage" was chosen as to mimic the metallic counter used in the test videos, see figure 3.8, which in turn was chosen to mimic the metallic surface of a commercial sausage grill.

Figure 3.3: Examples of the HDRI's used for lighting the scenes and as background.

In order to get realistic lighting of the scenes, HDRI's were utilized. HDRI stands for High Dynamic Range Imaging and are panoramic images that apart from color information also contains luminance information. As good quality HDRI's requires specific equipment to capture, freely available HDRI's were instead obtained from the website HDRI Haven. More specifically, it was the HDRI's from the Indoor package that were used, which included 104 images. Some examples of these HDRI's are shown in figure 3.3. Even though some images might not exactly resemble a store, the luminance information can still be relevant. When creating a scene an HDRI as well as its strength was chosen at random. For most images the HDRI was not only used for lighting the scene but also as a background. The idea was that this introduces some negative information to the models, which should help them to not detect background objects as sausages.

3.1.3 Negative objects

In the test videos the sausages were moved around with the help of a tong, see figure 3.8. This introduced objects that had similarities with the sausages. The tong was not very similar in color but had a similar shape, fingers could be similar in shape and also quite similar in color and an arm could be quite similar in color but not as much in shape. To combat the detectors detecting these objects as sausages, models of these objects were included in some images. These objects are called negative objects as they are not supposed to be detected.

For the hand/arm two different models were used. As a hand is quite complicated to create from scratch, free models from the website Turbosquid was instead obtained. These models however only included the meshes so color was added afterwards. As these were just used as negative models they were only painted to adequately resemble a Hand/arm. The tong was created from scratch and also in a rather simplistic manner as to just resemble a tong.

3.1.4 Image data

Even though generating synthetic data using 3d modelling removes the need for manual annotations, some degree of manual labor might be introduced if the scenes are to be made to look more like the real scenario. To investigate the impact of how general a scene is a couple of different datasets were created.

Figure 3.4: Examples of image data created with the more general scene where sausages are dropped onto the stage.

The most general scene was created by simply dropping a number of sausages onto the stage. For this Blenders physics engine was used. Each sausage was first placed at a set height and at a random position within a square in the xy-plane. They were also randomly rotated from their base orientation. The sausage model was oriented so that the long axis of the sausage was parallel with the y-axis. So, from this orientation the sausages were then rotated a random amount around the y-axis and the z-axis, both within 360°. With the sausages placed at their initial positions a gravity simulation was then started where the sausages were simulated as rigid objects. After a while the simulation was stopped and an image rendered. This method managed to place the sausages in the scene in a realistic and very simple way. Examples of the scene are shown in figure 3.4. However, the sausages on a grill are usually placed in a more ordered fashion. So for this specific scenario it might not be a sufficient method.

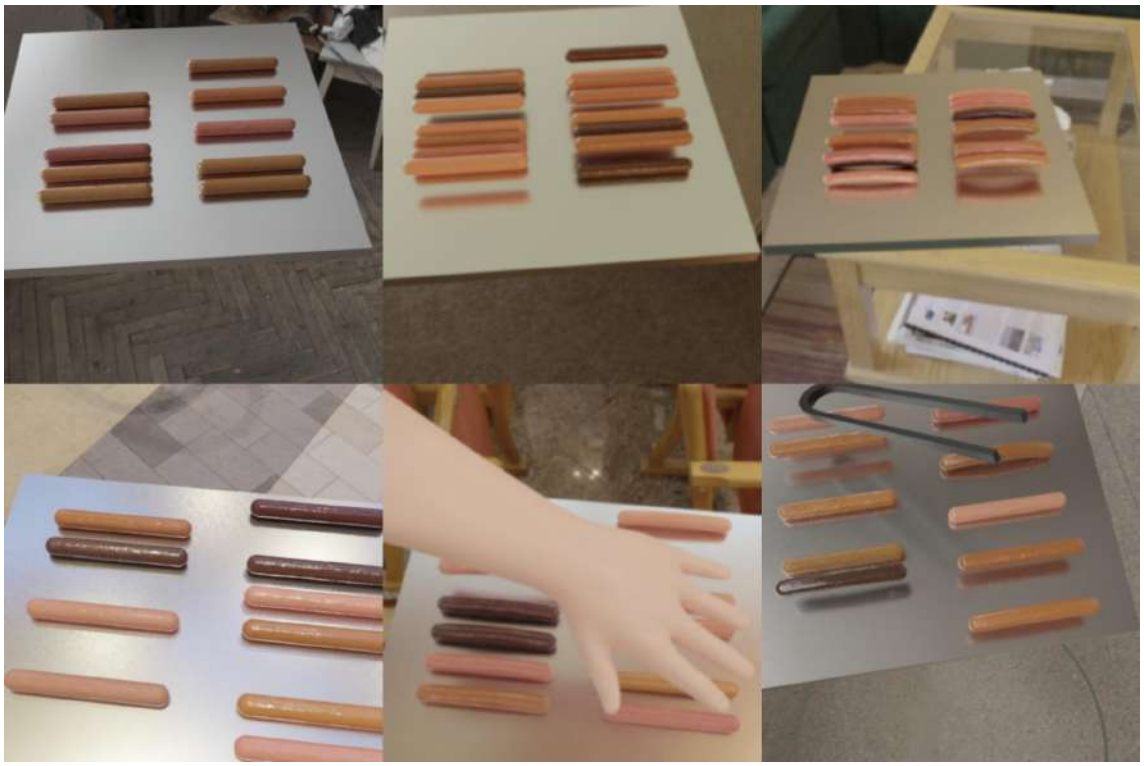

Figure 3.5: Examples of image data created with the more specific scene where sausages are placed onto the stage in an ordered fashion.

To investigate this, a scene with sausages placed in a more ordered fashion was created. Instead of dropping the sausages onto the stage they were placed at specific positions. These positions were however still chosen randomly. A grid of predefined positions were created, from which the sausages positions were randomly chosen. This grid was made up of two columns of equally spaced slots. The sausages were also only rotated along the y-axis, so that all the sausages were aligned. Examples of this scene are shown if figure 3.5.

Other versions of the ordered scene were also created. One more difficult version, where the sausages were placed closer together and also some sausages slightly above ground. This resulted in scenes where the sausages occlude each other in such ways that they become quite difficult to distinguish between. Another version included bent sausages. In this scene half of the sausages were bent by a random amount.

For both the unordered and ordered scenes, scenes with the negative objects were also created. These were in all cases placed above the sausages at a specific height and randomly placed in the xy-plane. Their orientation were also set at random.

3.1.5 Video data

To train the part of the models that performs the tracking, videos of the sausages moving were needed. For this a lot could be done in terms of realism. To keep it at a reasonable level however, rather simplistic animations were created. Sequences from the different types of videos created and described below are shown in figure 3.6.

The main movements to replicate in the investigated scenario were the addition and removal of sausages. These were created in two different scenes, one more general and one more specific. In the more general scene the sausages start positions were set by dropping the sausages onto the stage. This was done using Blenders physics engine. Each sausage was first placed at a set height and at a random position within a square in the xy-plane. They were also randomly rotated from their base orientation. The sausage model was oriented so that the long axis of the sausage was parallel with the y-axis. So, from this orientation the sausages were then rotated a random amount around the y-axis and the z-axis, both within 360°. With the sausages placed at their initial positions a gravity simulation was the run for a short amount of time where the sausages were simulated as rigid objects. From this simulation the start positions of the sausages lying on the stage were then obtained. However, as sausages also were to be added to the stage, two sausages were also positioned off the stage outside the field of view of the camera. Sausages were then animated one at a time. Either one was removed from the stage or one was added. When removing a sausage, the top most sausage was always selected to avoid animating sausages going through each other. The animation was then made up of two parts. First the sausage was moved only upwards a certain distance, then it was moved in a straight line towards the camera until no longer in the cameras field of view. When adding a sausage, the sausage was first moved in a straight line to a random position above the other sausages. Then, to again avoid animating the sausage going through another sausage, the sausage was dropped using a gravity simulation.

In the more specific case, the start positions of the sausages starting on the stage were set by placing the sausages in an ordered fashion on the stage. Instead of dropping the sausages onto the stage they were placed at specific positions. These positions were however still chosen randomly. A grid of predefined positions were

created, from which the sausages positions were randomly chosen. This grid was made up of two columns of equally spaced slots. The sausages were also only rotated along the y-axis, so that all the sausages were aligned. The removal and addition of sausages were then apart from a few differences animated in a similar way. For the removal, as sausages in this case don't lie on top of each other, the sausage to remove was instead simply chosen at random. For the addition, the position to move the sausage to was not chosen randomly within the space of the stage but instead within the set of free slots. Also, the sausage was not dropped onto the stage but rather placed.

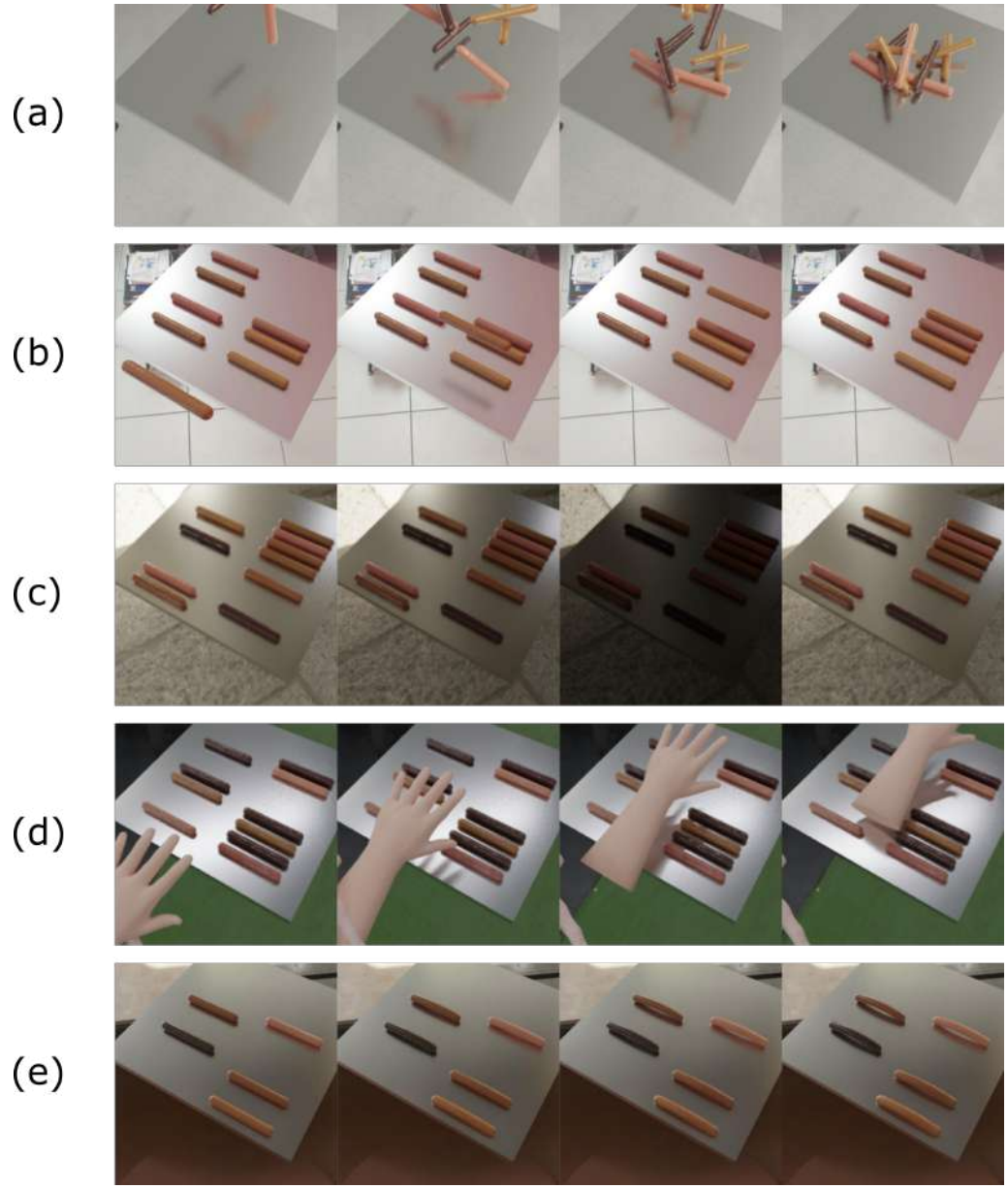

Figure 3.6: Examples of sequences from the different types of videos. (a) Sausages falling onto the stage. (b) Adding a sausage to the stage. (c) Changing HDRI strength. (d) Hand moving over the sausages. (e) Bending sausages.

These video types described above mimic the addition and removal of sausages, but the movements are very straight and does not have a lot of variety. To introduce some more general movements, videos of dropping the sausages onto the stage were also created. The sausages were dropped onto the stage in much the same way as described earlier. However for these videos the sausages were initially placed at different heights in order to create space between the sausages as they fell onto the stage. The videos included the sausages falling onto the stage and then also them moving around on the stage for a while until the movement slowed down.

The aforementioned video types only concern movement. Tracking however also includes other aspects such as color and shape. In the investigated scenario the color of the sausages can for example change when a shadow is cast on them and the sausages might bend slightly when being picked up, changing their shape. To improve the trackers association ability, videos for these two described situations were created. To replicate the sausages being shaded, sausages were placed on the stage and kept stationary while animating the HDRI strength. To replicate the sausages being bent, sausages were placed on the stage and then only the actual bending of the sausages was animated.

Even if the sausages actual shape does not change, its visible shape can change by something occluding a part of it. Such changes are to an extent already accounted for by sausages moving in front of each other in the videos that animate sausage movement. However, the sausages in the real scenario can of course be occluded by other objects than the sausages themselves. So to add some more occlusion data for the tracker to go on, videos of the hand models moving back and forth over stationary sausages were also created. These videos were made for both the more general case of sausages being randomly dropped onto the stage and the more specific case of sausages being placed onto the stage in an ordered fashion.

3.1.6 Camera and post-processing

To introduce some variation in the viewing angles of the scenes the camera was positioned randomly to an extent. The cameras position along the x-axis was fixed while its position along the y-axis and z-axis was varied randomly. The camera was also always rotated to point towards the stage. To get a good balance between rendering speed and quality the images were rendered at a resolution of 640x640.

Compared to the real captured videos the rendered images were sharper and more saturated. Therefor the rendered images were also post-processed to look more like the captured videos. This was done partly by just desaturating the images a constant amount and partly by adding a varying amount of gaussian blur to the images. The amount of gaussian blur was varied because the video quality in the captured videos varies depending on for example lighting. A comparison of a rendered image, post-processed image and captured image is shown in figure 3.7.

Figure 3.7: Demonstration of the post-processing step. On the left is the rendered image. In the middle is the post-processed (blur and desaturation) image. For comparison, an image from the captured videos is also included to the right.

3.2 Tracking methods

Two real-time multi object tracking and segmentation methods were investigated for this task, PointTrack and SipMask. Apart from applying both methods separately, combinations of the two methods were also investigated.

3.2.1 PointTrack

PointTrack consists of two separate parts, an instance segmentation network and an embedding extractor. The instance segmentation network, SpatialEmbedding, was trained on the image data and the embedding extractor on the video data.

SpatialEmbedding was trained for 80 epochs at a learning rate of $5 \cdot 10^{-4}$ for the first 50 epochs and $5 \cdot 10^{-5}$ for the last 30 epochs. The training was performed in batches with a batch size of 6, which was the maximum size possible given the memory capacity of the GPU. Starting the training from a version of SpatialEmbedding pretrained on the Cityscapes dataset [23] was also investigated.

The embedding extractor was trained for 15 epochs at a learning rate of $5 \cdot 10^{-4}$ for the first 10 epochs and $5 \cdot 10^{-5}$ for the last 5 epochs. The training of this network was also performed in batches. PointTrack constructs these batches from track ids. Each different object has its unique track id. PointTrack gathers all the track ids from all of the training videos and then picks random track ids from this collection to form a batch. From a specific track id a sample is taken by randomly choosing three equally spaced frames. The spacing is randomly chosen between 1 and 5, where 1 corresponds to three consecutive frames. The loss used is margin based hard triplet loss. This kind of loss benefits from larger batch sizes as it increases the probability of including harder triplets. Therefor the maximum possible batch size given the GPU memory capacity was chosen, which was 25.

3.2.2 SipMask

SipMask exists in two versions, one only for instance segmentation and one adapted for tracking. The instance segmentation version can then also be used with different backbone networks. Four different backbones, all residual networks, were investigated, ResNet50, ResNet101, ResNet50-Deform and ResNet101-Deform. The number in the names (50 and 101) denotes the depth of the network and "Deform" stands for that deformable convolutional networks [24] are used. When the deformable backbone is used the method is instead called SipMask++, which apart from a different backbone also implements a mask scoring strategy.

The instance segmentation versions was trained on the image data. The learning rate schedule adopted by SipMask was used. This included a warm-up period, steps and the usage of momentum. A warm-up period of 500 iterations was used, starting at a third of the desired learning rate. The learning rate was warmed up to a value of 10^{-3}, stepped down to 10^{-4} at epoch 8 and finally stepped down again for the last 2 epochs at epoch 11 to 10^{-5}. The training was performed in batches with a batch size of 6, which was the maximum size possible given the memory capacity of the GPU.

The different backbones used were all versions pretrained on the ImageNet dataset [25]. Versions of SipMask pretrained on the COCO dataset [26] were also available. Starting the training from these pretrained versions of SipMask was investigated.

In the SipMask version adapted for tracking the only difference is an added parallel branch. Therefor the learned parameters from the instance segmentation version was first loaded and then frozen, so that only the tracker part was trained on the video data. The tracker was trained with a similar learning rate schedule. First the learning rate was warmed up to a value of $2 \cdot 10^{-3}$ for 1000 iterations, starting from 1/80 of the final value. The learning rate was then stepped down to $2 \cdot 10^{-4}$ at epoch 8 and to $2 \cdot 10^{-5}$ at epoch 11. The training was stopped after 12 epochs. The training was performed in batches with a batch size of 6.

3.2.3 Scenario-specific robustness component

To increase the trackers robustness for the investigated scenario a new component was added to the instance association part of the tracker. Specifically this component was created to reduce the number of id switches. An id switch occurs when given two ground truth masks in two consecutive frames whose instance id's are the same, the instance id of the corresponding predicted masks switches. This can happen if for example two objects pass over each other. In such a situation it can be difficult for the tracker to know which object is which and might thereby assign wrong ids. However, this component can also help in situations where during an occlusion of an object the id of that object gets wrongfully assigned to another instance, possibly leading to the occluded object being assigned a new id when revealed again.

This component took advantage of the fact that in the investigated scenario only

one object (sausage) was moved at a time. First two states that each instance of a track could occupy were defined. These two states were moving and stationary, and each instance could only occupy one of the states at a time. An instance was defined to be stationary if for 5 consecutive pairs of frames the mask IOU of the instances with the same id in each pair was larger than 0.9. For an instance to be assigned the moving state, the change in bounding box size and bounding box position between the instance and the instance with the same id in the previous frame had to fulfill certain criteria. The change in bounding box width and height had to both be below 7 pixels and the change in bounding box position (distance between the top left corners) had to be above 6 pixels. The criteria on bounding box size was set because otherwise size changes could be considered movement. When the size of a bounding box changes, the positions of all points that can be used as reference for movement such as corners and the center also changes. However, this criteria also meant that a moving instance could be considered not moving if at the same time its shape changed, orientation changed or parts of it were occluded. For this scenario this did not pose much of a problem because the objects does not change their shape or orientation significantly.

When an instance became stationary it was saved in memory. In each frame the detected instances, after being assigned id's, were then compared to these saved stationary instances. If a detected instance and a saved instance had an IOU larger than 0.5, their id's were checked and if the id's were different the detected instances id was changed to the id of the saved instance. To avoid an instance moving over one of the saved instances and thereby changing its id, the id was only changed if the detected instance was not in the moving state. After changing the id of the detected instance it had to be checked if any of the other detected instances had been assigned the switched to id, i.e. if two instances switched id's with each other. If so was the case the former id of the first instance was assigned to that instance.

3.3 Testing

3.3.1 Gathering test videos

A test setup resembling the actual scenario was created. To resemble the sausage grill a metallic counter was used. Sausages were then moved around in frame and also moved in and out of frame with a tong. In an actual store the most convenient way of mounting the camera would be to mount it on the wall behind the grill. Therefor the camera used in the test setup was mounted on the wall behind the counter. The videos were captured on a Nikon D3200 camera with a resolution of 1920x1080 at 25 frames per second. The video was then cropped down to exclude as much unnecessary information as possible. Finally the cropped down video was resized to match the resolution of the training images and videos. Some example frames from the videos are shown in figure 3.8.

A part of one of the captured videos (500 frames) was also annotated. The annotations were made in a semi-supervised manner. Masks were first created with one of

the trackers and then manual refinements of the produced masks were made. This annotated sequence was then used to evaluate the trackers performances.

Figure 3.8: Example frames from the test videos.

3.3.2 Speed evaluation

The inference speed of the trackers were measured on a single Nvidia Geforce GTX 970. A GPU can exist in different power states and when not being used it can go into a state of lower power. Therefor, to ensure that the GPU was not in a low power state, a warm-up of 10 iterations was performed before the actual measurements were taken. To perform the measurements the pytorch package was used. Specifically the function `torch.cuda.synchronize()` was used to synchronize the CPU with the GPU, making sure that the timing was not stopped before the processes on the GPU were finished.

In PointTrack the instance segmentation and the tracking is split into two separate parts. So for this method the speeds of the separate parts were measured. SipMask exists in two versions, one that only does instance segmentation and one that also does tracking. Both were however measured because a method using SipMask's instance segmentation and PointTrack's tracking was also investigated. The instance segmentation methods were tested on the image data and the tracking methods on the video data. As the inference speeds varied from one pass through the models to another the final reported inference speed for each method is the average inference speed over the respective datasets.

4

Result

4.1 Instance segmentation methods

The accuracy (mask AP) of the different investigated instance segmentation methods on the test data is shown in table 4.1. For all methods better accuracy was observed when starting from a pretrained version and only finetuning on the synthetic data compared to training from scratch on the synthetic data. According to the AP@[50:95] metric all pretrained methods perform quite equally. Looking at the AP@50 and AP@75 metrics however, SpatialEmbedding performs better than all the other methods.

Table 4.1: Accuracy (mask AP) of the investigated instance segmentation methods. Three different accuracy metrics are used to evaluate the methods. Pretrained versions achieve higher accuracies in all cases. According to AP@[50:95] all methods perform quite equally. According to the metrics AP@50 and AP@75 however, SpatialEmbedding performs the best.

method	pretraining	AP@[50:95]	AP@50	AP@75
SipMask-ResNet50	COCO	0.842	0.924	0.9
SipMask-ResNet50	No	0.799	0.926	0.878
SipMask-ResnNet101	COCO	0.843	0.960	0.939
SipMask-ResnNet101	No	0.736	0.963	0.785
SipMask++-ResNet50	No	0.755	0.951	0.883
SipMask++-ResNet101	No	0.718	0.960	0.824
SpatialEmbedding	Cityscapes	0.839	0.979	0.969
SpatialEmbedding	No	0.796	0.954	0.936

The inference speeds of both the instance segmentation methods and the trackers were measured. In table 4.2 we can see that SpatialEmbedding was significantly slower than all the SipMask variants with an inference speed of 224 ms/frame. SipMask++-ResNet50 was measured to be the fastest method at an inference speed of 85 ms/frame. SipMask-ResNet50 was however not too far away, measured to have an inference speed of 103 ms/frame. The SipMask method with the deeper backbone, SipMask-ResNet101, was slower than the other SipMask methods. It was

however still faster than SpatialEmbedding.

Table 4.2: Inference speed of the instance segmentation methods. SipMask++-ResNet50 is the fastest while SpatialEmbedding is the slowest.

method	speed (ms/frame)
SpatialEmbedding	224
SipMask-ResNet50	103
SipMask-ResNet101	155
SipMask++-ResNet50	85
SipMask++-ResNet101	115

4.2 Tracking methods

The accuracies (MOTSA) of the trackers are shown in table 4.3. The best accuracy is achieved with the combination of SpatialEmbedding and the PointTrack tracker and the worst with the SipMask tracker. Out of the trackers combining SipMask and PointTrack, the one using SipMask++-ResNet50 performs the best.

The inference speeds of the trackers are shown in table 4.4. The speed of the tracker part of PointTrack was measured to be 40 ms/frame. To obtain the tracking speed of the methods using PointTracks tracker, the speed of respective instance segmentation method and the speed of PointTracks tracker were simply added. From this we can then see that the SipMask tracker was the fastest, with an inference speed of 100 ms/frame, and the original PointTrack tracker with SpatialEmbedding was the slowest, with an inference speed of 264 ms/frame.

Table 4.3: Accuracy of the trackers. The different methods are denoted according to "instance segmentation method" + "tracking method". For all instance segmentation methods the pretrained version was used. The best accuracy is achieved with the combination of SpatialEmbedding and the PointTrack tracker and the worst with the SipMask tracker.

method	MOTSA	sMOTSA	TP	FP	IDS
SpatialEmbedding + PointTrack	94	85	1154	53	5
SipMask-ResNet50 + PointTrack	77	67	1146	243	8
SipMask-ResNet101 + PointTrack	84	74	1125	138	6
SipMask++-ResNet50 + PointTrack	88	76	1096	59	5
SipMask++-ResNet101 + PointTrack	86	71	1098	93	4
SipMask-ResNet50 + SipMask	71	62	1078	238	10

Table 4.4: Inference speed of the trackers. The different methods are denoted according to "instance segmentation method" + "tracking method". The SipMask tracker is the fastest, while the combination of SpatialEmbedding and the PointTrack tracker is the slowest.

method	speed (ms/frame)
SpatialEmbedding + PointTrack	264
SipMask-ResNet50 + PointTrack	143
SipMask-ResNet101 + PointTrack	195
SipMask++-ResNet50 + PointTrack	125
SipMask++-ResNet101 + PointTrack	155
SipMask-ResNet50 + SipMask	100

4.3 Training data

Below the impact of training on different kinds of data is presented. For these investigations PoinTracks instance segmentation method, SpatialEmbedding was used.

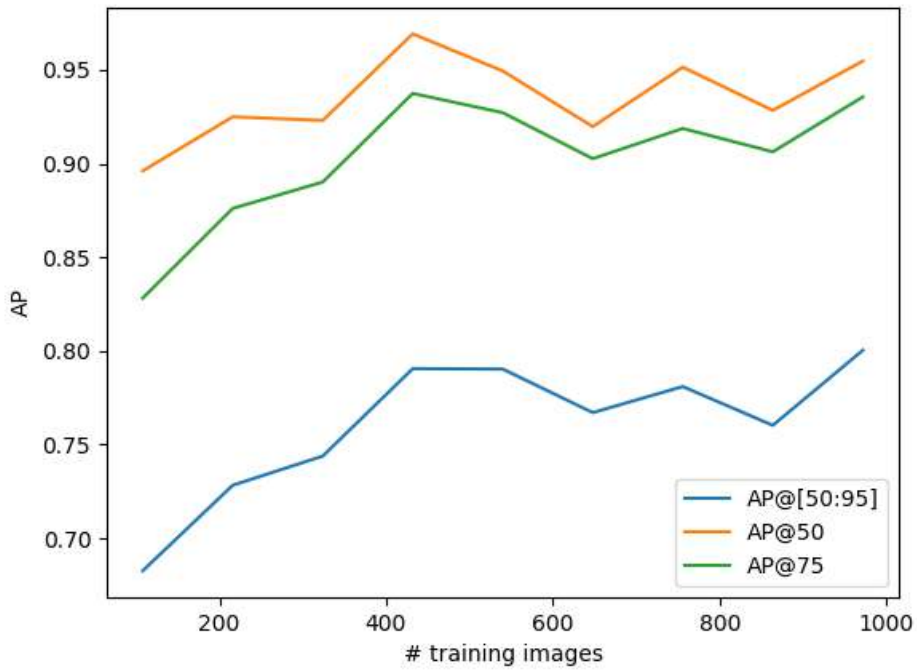

Figure 4.1: Impact of the amount of training data. The instance segmentation network used is PointTracks, SpatialEmbedding. The different curves shows the result from three different mask AP metrics. The accuracy improves up until around a dataset size of 400, where the curve flattens out.

In figure 4.1 the impact of the dataset size is shown. The dataset sizes ranged from 108 to 972. The accuracy improves quite rapidly from a datasize of 108 to a datasize of around 400. Then it flattens out while also showing some variation.

The impact of introducing negative data (images with hands and tongs) was investigated. For this 6 datasets with different sizes were created. The datasets were created by first taking a set of 600 images with no negative objects and then adding different amounts of negative data. The amount of negative data ranged from 0 to 600 images resulting in datasets with proportions of negative data ranging from 0% to 50%.

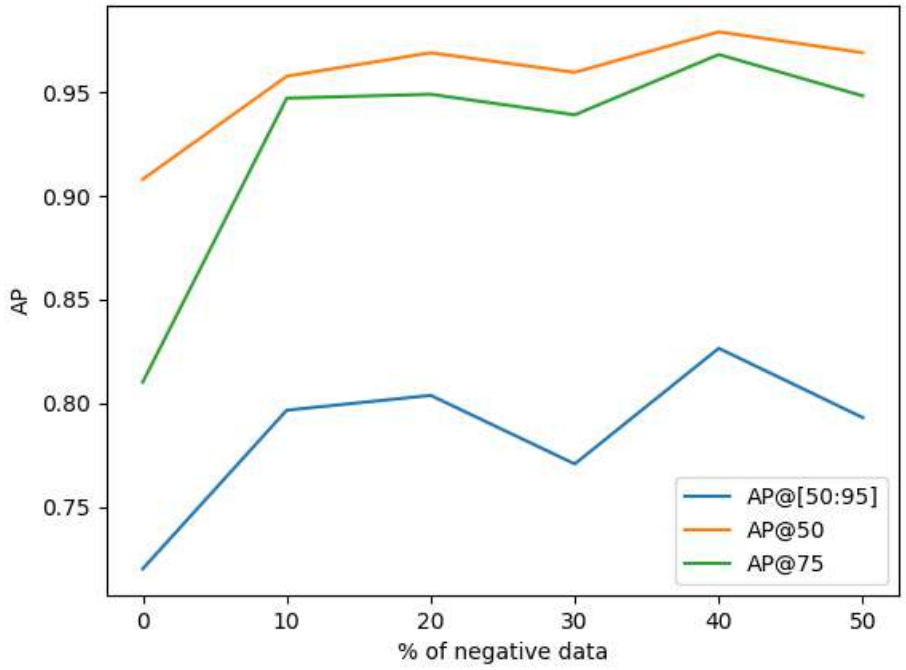

Figure 4.2: Impact of introducing different proportions of negative data (images with hands and tongs) into the training data. The instance segmentation network used is PointTracks, SpatialEmbedding. The different curves shows the result from three different mask AP metrics. Introducing negative data increases the accuracy, but increasing the proportion of negative data is observed to not have much effect.

The result is shown in figure 4.2. Here it is observed that the model clearly benefits from the introduction of negative data. Only a small amount is needed however. Most of the improvement occurs when going from 0% to 10% of negative data. From 10% to 50% little improvement is observed. It is also observed that the metrics AP@[50:95] and AP@75 improves more than the metric AP@50 when introducing negative objects. In figure 4.3 qualitative results of the introduction of negative objects is shown for two scenarios. Figure 4.3 (a) shows a scenario where an arm is first detected as a sausage and then is not detected. Figure 4.3 (b) shows a scenario

where when a sausage is held by a tong it is first split into two masks and then is masked with one single mask.

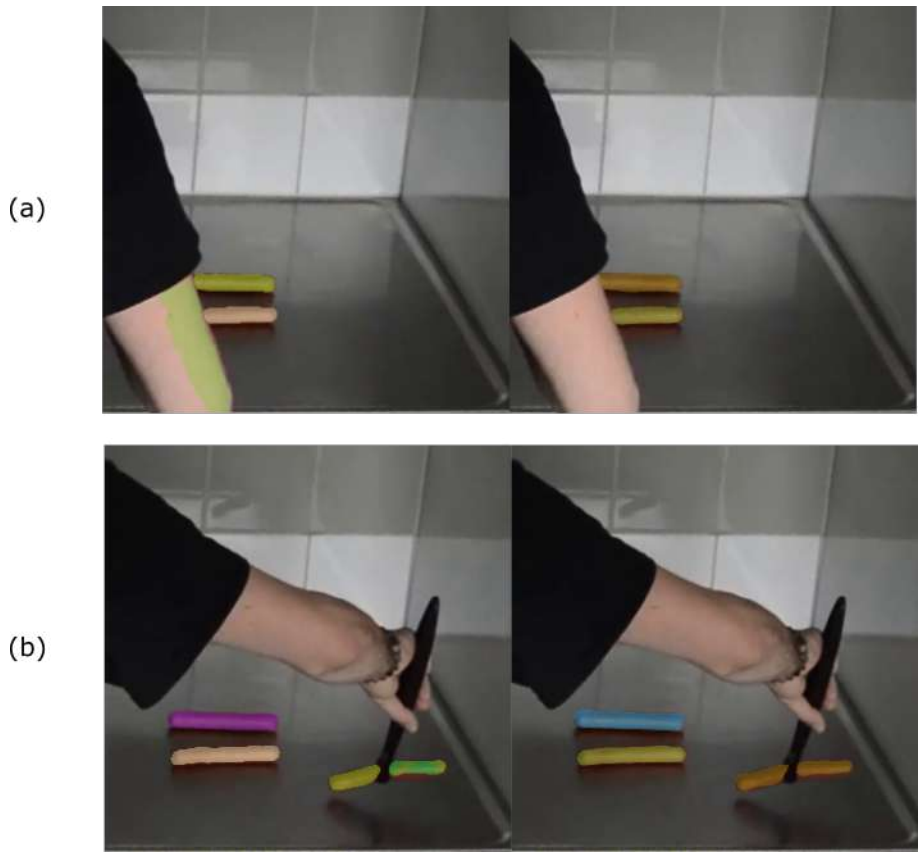

Figure 4.3: Improvement of the produced masks when introducing negative data in two scenarios. Images to the left shows result with no negative training data and images to the right with negative training data. Different colors represent different instance id's. (a) The arm goes from being detected as a false positive to not being detected. (b) A sausage goes from being split into two instances by the tong to being detected as just one instance.

Three different datasets were created, one more general with sausages being randomly dropped onto a stage (see figure 3.4), one more specific with sausages being placed in a more ordered fashion (see figure 3.5) and one complete dataset including both of these types of data. The accuracy achieved when training on each of these datasets was then investigated. In table 4.5 the obtained mask AP's are shown. From this it is obtained that only training on the general data achieves better accuracy than only training on the specific data, while the best result is achieved when training on the complete dataset. In figure 4.4 qualitative results are shown for a couple of more challenging scenarios. When only training on the specific dataset it is observed that the detector has trouble with rotated sausages, with sausages occluding each other and also in low light scenarios. Only training on the general dataset achieves much better results in these scenarios, sometimes achieving comparable results to training on the complete dataset.

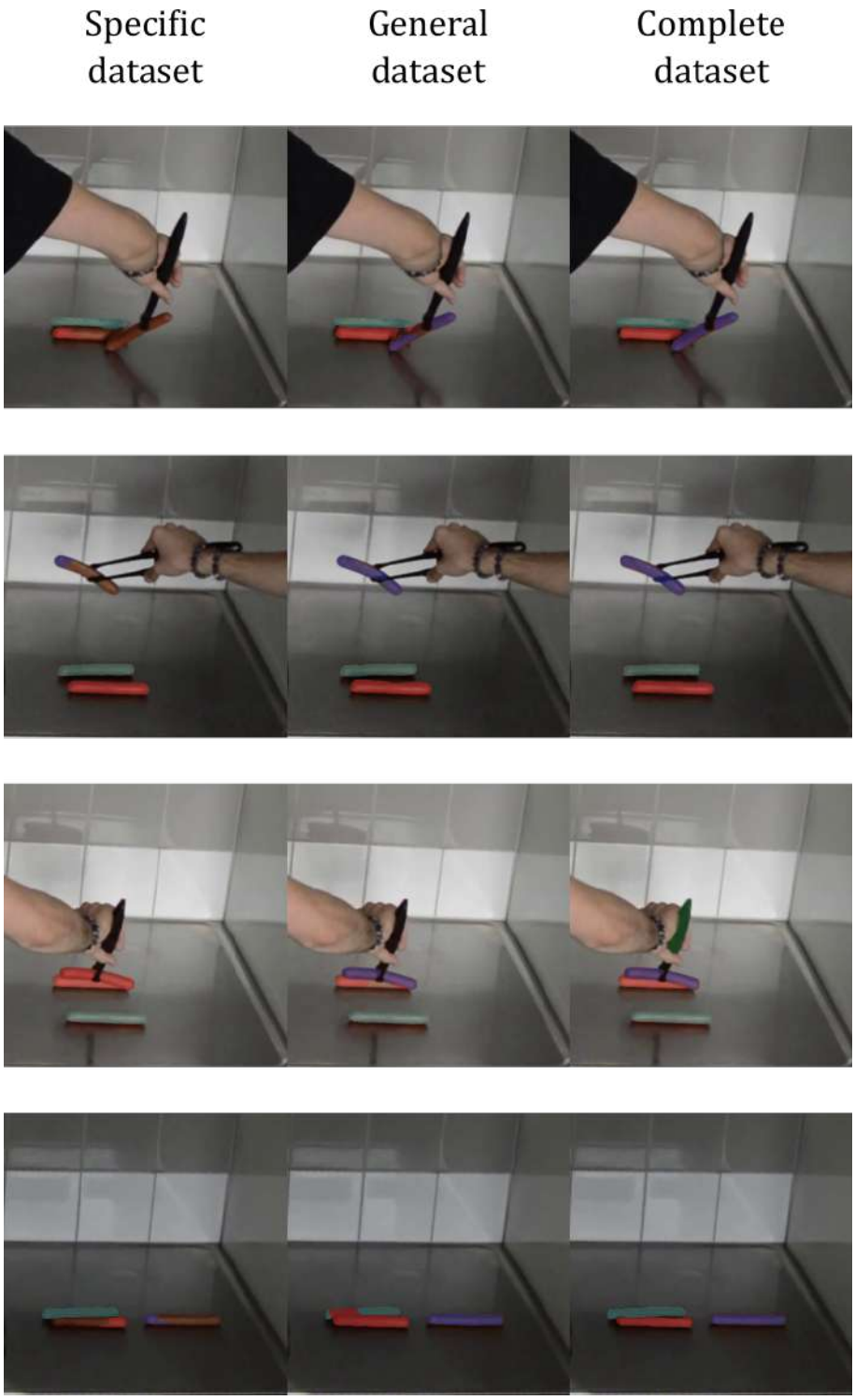

Figure 4.4: Comparison of qualitative results when training on different types of data. For this comparison four more challenging scenarios are chosen. The instance segmentation network used is PointTracks, SpatialEmbedding. In these scenarios only training on the specific data is not sufficient. Only training on the general data however sometimes yields comparable results to training on the complete dataset.

Table 4.5: Accuracy (mask AP) of training on different types of data. The instance segmentation network used is PointTracks, SpatialEmbedding. The complete dataset includes both the specific and the general data. Examples of the specific data is shown in figure 3.5 and the general data in figure 3.4. Training on the general data achieves better accuracy than training on the specific data, while the highest accuracy is achieved with the complete dataset.

dataset	AP@[50:95]	AP@50	AP@75
Specific dataset	0.655	0.812	0.760
General dataset	0.786	0.942	0.919
Complete dataset	0.839	0.979	0.969

The PointTrack tracker was also trained on each of these different dataset types. Additionally it was also trained on a complete dataset without negative data. In all cases the same instance segmentation method was used, SpatialEmbedding trained on the complete dataset. The result is shown in table 4.6. Only training on specific data yields the worst result and training on the complete dataset without negative data yields the best result.

Table 4.6: Tracking accuracy achieved when training the PointTrack tracker on different datasets. The same instance segmentation method is used in all cases, SpatialEmbedding trained on the complete dataset. Training solely on the specific data yields the worst result and training on the complete dataset without negative data yields the best result.

dataset	MOTSA	sMOTSA	IDS
specific dataset	93.41	84.73	9
general dataset	93.67	84.98	6
complete dataset without negative data	93.84	85.15	4
complete dataset with negative data	93.76	85.07	5

4.4 Scenario-specific robustness component

By adding an extra component to the tracking algorithm, id-switches were able to be caught and corrected. In figure 4.5 a few examples of corrections are shown. Two examples show situations where an id-switch occurs because a sausage is moved in front of another sausage. A third example shows a situation where the instance association fails because of a difference in lighting.

Three videos, each of a length around 40 seconds were captured. On these videos, three of the proposed methods were tested and qualitatively evaluated. The methods tested were SpatialEmbedding with the PointTrack tracker trained on the complete dataset, SpatialEmbedding with the PointTrack tracker trained on only the general dataset and SipMask++-ResNet50 with the PointTrack tracker trained on the complete dataset. With all three methods all id-switches were able to be corrected.

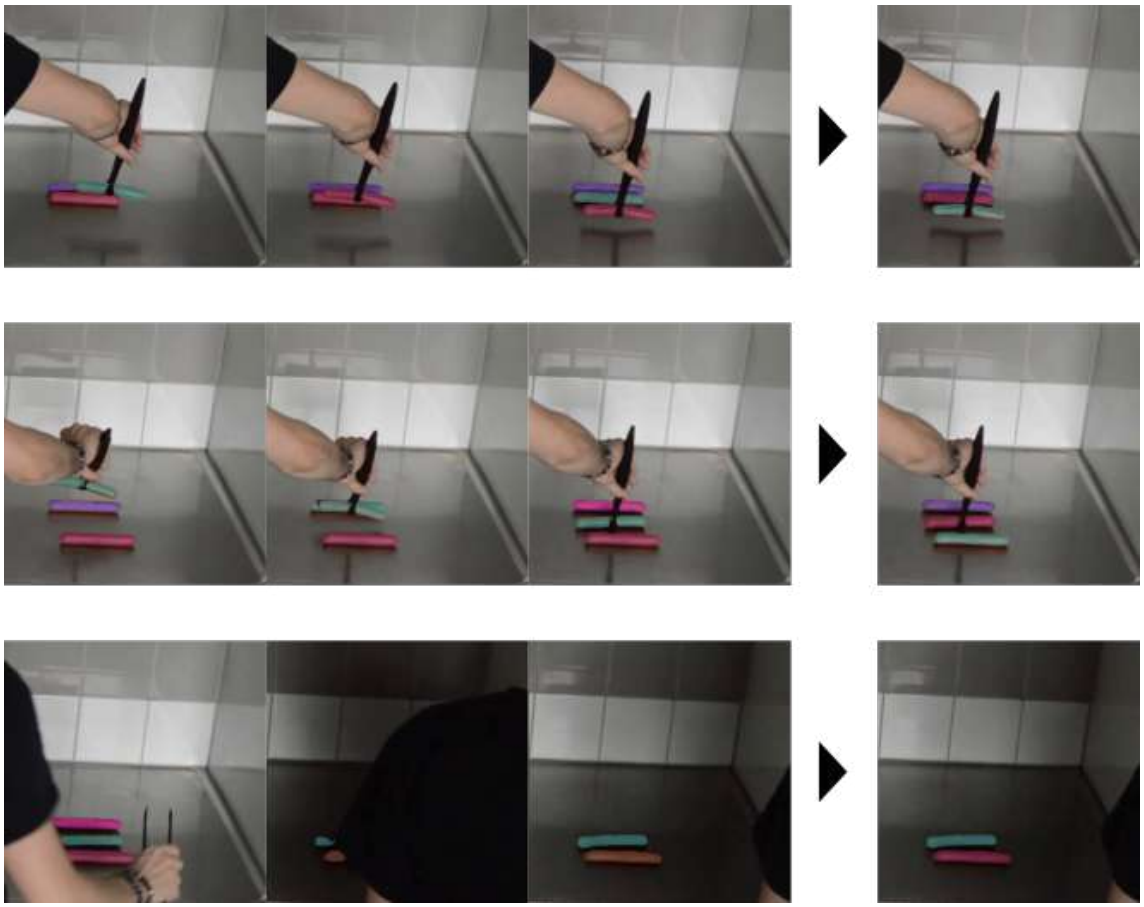

Figure 4.5: Examples of corrections made by the added tracking component. Two examples show a situation where an id-switch occurs because a sausage is moved in front of another sausage. A third example shows a situation where an id-switch occurs because of a change in lighting.

5

Discussion

5.1 Instance segmentation and tracking methods

In table 4.1 it is shown that all instance segmentation methods performed better when starting from a pretrained version. This was expected at least for the COCO dataset as it includes a lot of small objects and also some food items. So it includes data that is somewhat similar to the data in the investigated scenario. The CityScapes data however only includes objects found in cities like vehicles and pedestrians, so it is not very similar to the data in the investigated scenario. The reason for why pretraining on the CityScapes dataset still improves the performance is most likely that the convolutional networks are still able to take advantage of the learned filters that detect more general features such as edges.

From the same table it is observed that according to the AP@[50:95] metric all instance segmentation methods perform quite equally when pretrained, while according to the metrics AP@50 and AP@75 SpatialEmbedding performs better than all SipMask methods. This could be explained by the fact that the SipMask methods more frequently detect negative objects as sausages, as shown in table 4.3. When taking into account more precise metrics, like AP@[50:95] does, these false positives might not contribute as much to the final score as the total amount of false positives will increase with more precise metrics. Looking at the tracking performances which are measured by the MOTSA metric, differences are also observed. So in this scenario the AP@[50:95] metric does not seem to reveal the whole truth.

Looking at the speed measurements of the instance segmentation methods, table 4.2, it is seen that SpatialEmbedding is significantly slower than all of the SipMask versions. This could partly be explained by the way they produce their masks after retrieving information from an image. For SipMask it is a simple linear combination, while for SpatialEmbedding a clustering technique is used. SipMask++-ResNet50 is surprisingly found to be the fastest of them all. It is however still quite close to SipMask-ResNet50. As expected, SipMask-ResNet101 is found to be the slowest SipMask method due to its deeper backbone.

In table 4.4 the speed measurements of the trackers are shown. Here it is observed that the SipMask tracker is faster than all versions using the PointTrack tracker. It

is also noticed that SipMask with tracking actually seems to be slightly faster than just the instance segmentation part on its own. It is however unlikely that this is actually the case. It is more likely that this difference comes from that they are tested on different datasets. However, it still indicates that the tracking in SipMask is fast.

Combining the results on the speeds and accuracies of the methods it can first be stated that no method is both the fastest and the most accurate. The best compromise seems to be the tracker which uses the SipMask++-ResNet50 for instance segmentation and the PointTrack tracker for tracking. This tracker performs second best in terms of both accuracy and speed. However, depending on the hardware used, PointTrack with SpatialEmbedding might still be preferred as it is still a fast method and achieves the best accuracy.

5.2 Training data

In Figure 4.1 it is shown that the instance segmentation accuracy improves with larger datasets up to a point where the curve flattens out. The curve flattens out at around a dataset size of 400 images. So not a lot of data is needed for this task, which is not very surprising considering that only one type of object is being learned and that the appearance of this object does not vary too much. After flattening out some variation is still observed. This variation could be explained by the fact that the training data is more general than the test data. This means that some images might actually be detrimental to the test data and some might match very well. So when increasing the dataset size, how well the added data matches the test data might vary. Thus leading to varying accuracy.

From figure 4.2 and 4.3 it is seen that the inclusion of negative data is important for the detector to reach high accuracy. The detectors performance improves more on the more precise metrics AP@[50:95] and AP@75. This indicates what is also observed qualitatively that the introduction of negative data not only reduces the number of negative detections but also improves the masks of the positive detections.

In tables 4.5 and 4.6 it is observed that training solely on the specific dataset performs the worst. Even though this is the dataset that tries to replicate the real scenario the most, it lacks in generality. The largest drawback in this dataset is most likely that all sausages have the same orientation, which is not the case in the real scenario. Adding a random element to the choice of orientation could have been beneficial. Solely training on the general dataset however yields favorable results, which is promising as this method required the least manual effort. It could also easily be used with other types of objects. Training on the complete dataset performs the best, which indicates that the specific dataset still also contributes with some important information.

6

Conclusion

Multiple methods have been identified that in the investigated scenario are able achieve satisfactory results in most situations. Further, by introducing a new component to the tracking algorithm that takes advantage of the current scenario, satisfactory tracking results are also achieved in more difficult situations.

It has been shown that finetuning on solely synthetic data, generated through 3D-modelling, can be a viable option for the MOTS task. Moreover, good results can be achieved with a quite general data generation approach. However, to reach better accuracy some extra manual work might be needed.

Bibliography

[1] Yifu Zhang et al. "FairMOT: On the Fairness of Detection and Re-Identification in Multiple Object Tracking". In: *arXiv preprint arXiv:2004.01888* (2020).

[2] Zhongdao Wang et al. "Towards real-time multi-object tracking". In: *arXiv preprint arXiv:1909.12605* 2.3 (2019), p. 4.

[3] Peng Chu et al. "TransMOT: Spatial-Temporal Graph Transformer for Multiple Object Tracking". In: *arXiv preprint arXiv:2104.00194* (2021).

[4] Paul Voigtlaender et al. "Mots: Multi-object tracking and segmentation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 7942–7951.

[5] Lorenzo Porzi et al. "Learning multi-object tracking and segmentation from automatic annotations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 6846–6855.

[6] Aljoša Ošep et al. "Track, then decide: Category-agnostic vision-based multi-object tracking". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 3494–3501.

[7] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[8] Shu Liu et al. "Path aggregation network for instance segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8759–8768.

[9] Enze Xie et al. "Polarmask: Single shot instance segmentation with polar representation". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 12193–12202.

[10] Pedro O Pinheiro et al. "Learning to refine object segments". In: *European conference on computer vision*. Springer. 2016, pp. 75–91.

[11] Daniel Bolya et al. "Yolact: Real-time instance segmentation". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 9157–9166.

[12] Davy Neven et al. "Instance segmentation by jointly optimizing spatial embeddings and clustering bandwidth". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 8837–8845.

[13] Jiale Cao et al. "SipMask: Spatial Information Preservation for Fast Image and Video Instance Segmentation". In: *arXiv preprint arXiv:2007.14772* (2020).

[14] Ankur Handa et al. “Understanding real world indoor scenes with synthetic data”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4077–4085.

[15] Nathan Clement et al. “Synthetic Data and Hierarchical Object Detection in Overhead Imagery”. In: *arXiv preprint arXiv:2102.00103* (2021).

[16] Benjamin Midtvedt et al. “Quantitative digital microscopy with deep learning”. In: *arXiv preprint arXiv:2010.08260* (2020).

[17] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. “Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks”. In: *Energy and Buildings* 158 (Nov. 2017). DOI: 10.1016/j.enbuild.2017.11.045.

[18] Bernhard Mehlig. “Machine learning with neural networks”. In: (2020).

[19] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[20] Eduardo Romera et al. “ERFNet: Efficient Residual Factorized ConvNet for Real-Time Semantic Segmentation”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.1 (2018), pp. 263–272. DOI: 10.1109/TITS.2017.2750080.

[21] Zhenbo Xu et al. “Segment as points for efficient online multi-object tracking and segmentation”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 264–281.

[22] Tsung-Yi Lin et al. “Feature pyramid networks for object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125.

[23] Marius Cordts et al. “The cityscapes dataset for semantic urban scene understanding”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3213–3223.

[24] Jifeng Dai et al. “Deformable convolutional networks”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 764–773.

[25] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[26] Tsung-Yi Lin et al. “Microsoft coco: Common objects in context”. In: *European conference on computer vision*. Springer. 2014, pp. 740–755.