Dataset creation for estimating human pose from depth images using Convolutional Neural Networks

Both Eyes Open

Bård-Kristian Krohg

Thesis submitted for the degree of
Master in Informatics: Robotics and Intelligent Systems
60 credits

Institute for informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2021

Dataset creation for estimating human pose from depth images using Convolutional Neural Networks

Both Eyes Open

Bård-Kristian Krohg

Dataset creation for estimating human pose from depth images using Convolutional Neural Networks

http://www.duo.uio.no/

Printed: X-press printing house

Dataset creation for estimating human pose from depth images using Convolutional Neural Networks

Bård-Kristian Krohg

15th November 2021

Abstract

This work is part of a larger project that explores the possibility of bringing robotics into geriatric care. The goal of that project is to create a robotic system that can assist in optimizing the use of caregivers, so they are used where they are needed.

This work introduces *DepthPose* – a system that solves human pose estimation in 3D for full skeletons. The system is lightweight and can be deployed on mobile units.

Convolutional Neural Networks have been used for solving object recognition in 2D images with great success. This work aims to use the same techniques to extract 3D human poses from depth images in real-time. Two multi-staged CNNs, one to encode the location of each joint and another to encode the association between the joints, provide initial poses and locations for perceived people. The locations are then refined using a novel articulation network.

Preface

First, I would like to thank my supervisors Jim Tørresen and Ryo Kurazume, for their support, guidance, and patience during the development of this project. Second, my HR manager Marit Flendstad Kruse, for her assistance in letting me combine work with the writing of this thesis. Last, I would like to thank everyone at the Kurazume-lab for their welcome and help during my stay at Kyushu University, and my friends and family for proofreading and encouragement.

The subtitle, Both Eyes Open, has a double meaning: In this paper, we will explore the world in 3D. The biological way to achieve depth vision is by using two eyes, hence both eyes open. The other way to interpret the subtitle is tied with the small figure on the front page. In Japanese, the saying translated as "Both Eyes Open" refers to the Daruma figure.

When one is working toward a goal, such as completing a thesis, one can purchase a Daruma figure from a temple. When bought, the Daruma has blank eyes - they are closed. The buyer then paints in one eye, asking for the Darumas help in completing their goal. In exchange for the Darumas help, the buyer promises to paint in the other eye. One significant detail about the Daruma is that it is weighted on the bottom. If it should ever falter and fall over, it will right itself back up and continue on its way to completing the goal.

The ability to right yourself up for every setback has been of particular inspiration to me during my work on this thesis. This is why I have placed the figure on the front page; as a personal helper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As life expectancy increases in Norway, so does the population who needs geriatric care either at home or in a geriatric facility. According to a communal health survey [17], it is projected that the increasing senior population in Oslo will lead to increased pressure on the healthcare services. To mitigate the need for help at home, it is important to encourage health-promoting activities and uncover what and where preventative measures are needed. The Multimodal Elderly Care System (MECS) is proposed as a solution where data gathered in users' homes guide where and what help is needed from healthcare personnel – be it preventative or acute.

The MECS is envisioned as a small, mobile unit that can be introduced to any home. This eliminates the user from the operational loop of the unit, making the data gathering process robust to forgetfulness or physical ability of the user. One of the key information points gathered by the MECS will be the users' physical pose, which informs the system of the following:
Body language, enabling the possibility of smooth Human-Robot Interaction (HRI) when moving around or determining the intent of the user.
Physical state, informing the MECS whether the user is in need of acute help.
History of natural posture, which could be invaluable information for a physical therapist or doctor to develop personalized training programs preventing muscle degradation.
Activity recognition, helping the MECS decide on what actions to execute. For example, reminding the user to take their prescribed

medication if they forget.

A 2D system could capture all these key points; however, a 2D representation of a pose would not be robust to different viewing angles. A 3D representation could define the origin of two poses' coordinate system to the same landmark in each pose, making it is easy to compare the two poses by, for example, the euclidian distance between the poses' corresponding landmarks.

Estimating human pose in 3D, or Motion Capture (mocap) is a well-known area of research. However, mocap is expensive and requires a large amount of physical hardware. The industry standard is to use an elaborate mocap studio that requires multiple expensive cameras, a large area, and specialized software. Therefore the application areas for motion capture are currently mostly limited to research, movie-, and videogame-making. The mocap problem deals with finding a representation of an actor, creature, or object that can be used in animation. This representation is often a *rigged* skeleton with bones that define the movement of the animated character [21]. The movements of the computerized rig are mapped to the movements of an observed actor in the real world and recorded.

1.1 Research Goals

This work explores the possibility of using a single depth camera to solve the mocap problem of estimating human pose in 3D. The resulting method needs to be lightweight and fast enough to be viable in a mobile unit with limited processing capability. The resulting method must also be accurate enough for healthcare personnel or the MECS itself to extract reliable and useful information about the pose.

The method presented in this thesis will therefore be evaluated using the following goals:

1. Propose a lightweight system for recognizing human pose in 3D based on depth images.

2. Design and develop an architecture that can be deployed to systems with limited hardware, and provide useful information.

3. Evaluate the performance of the system and find room for improvements for it.

1.2 Contributions

The main contribution of this work, a system called *DepthPose*, – is the proposal of using a shallow CNN trained on depth images, in combination with a novel articulation network to define human pose in 3D. The CNNs main task is to propose a set of poses present in the depth image. These poses are refined in the articulation network, which allows for a shallower CNN architecture. This leads then to fewer parameters in the system, which leads to a more lightweight method that should preform faster on limited hardware.

Chapter 2

Background

The classical approach to extract useful information from an image has been to find mathematical definitions for features that describe the information. The features could, for example, be lines, circles, or edges. Circles can be used to find coins, where the diameter denotes the value. Lines can be used to find how many fenceposts are in a fence. Traditional mathematical models include the Hough Transform [13] for detecting lines or circles, the Sobel operator to detect gradients, and in return, edges, or the Gray Level Co-occurrence Matrix for detecting texture features. In common for all these methods is that they are well defined and find precisely *one* type of feature that was previously specified. At a higher level, hand-crafted feature descriptors have been made. They look for a specific set of features in an image to identify unique locations. Some well-known examples are the SIFT [20], SURF [2] and ORB [25] feature descriptors. They have been used successfully in applications such as combining images into panoramas or combining different views to a 3D model, also known as Structure from motion [32].

Hand-crafted features have also been used in machine learning scenarios. As an example, Haar-like features were demonstrated to efficiently find faces in [33]. Here, the machine learning model decided which of a given set of features to use to classify whether the frame contained a face or not. In using both the Sobel operator and using Haar-like features, a filter is passed over the image. In the case of the Sobel operator, this is an actual convolution of the image with the filter, whereas Haar-features are extracted using a sliding window. This is the intuition that is used in Convolutional Neural Networks, discussed next.

2.1 Convolutional Neural Networks

First introduced by Fukushima [9] CNNs encode spatial information without being affected by shifts in the position of that information. In other words, they look at collections of *spacially connected pixels*. CNNs use filters, or kernels, to extract features that help them to "understand" an image. The deeper the network is, the more complex features emerge. In addition, the *receptive field*, the patch of the original image that affects the feature, becomes larger.

Instead of using hand-crafted features, deep CNNs define the features they use when they are trained. This is one of the reasons why CNNs needs fairly large training sets, as well as taking a long time to train. Therefore, a popular approach is to reuse the first few primitive layers (the first layers encountered in a forward-pass) from other models. This is known as *Transfer Learning*. It can be accomplished by first training a neural network for a specific task, for example, recognizing a handful of different types of objects in an image. Then, the primitive layers with their parameters are "chopped off" the network. Since these layers only have learned primitive features, these learned features can be input and fine-tuned to specialize a different network, which in turn needs less training time since it has already learned the simple features. Another advantage of this is that the two networks can be trained on vastly different datasets. If one application area has an abundance of training data, the first network could be trained on this. As long as the general input is the same, namely the number of channels and the values presented, the first few layers can be easily fine-tuned to a new application area.

When designing a CNN, it is often helpful to have in mind exactly what one can imagine should be detected in each layer. In 2D images, the first few layers of a deep CNN have been shown to often mimic the behavior of the simple handmade feature extractors such as the ones discussed above. However, such features can be quite different in 3D. Instead of edge-detectors, one can imagine plane detectors or other detectors unique to 3D objects. Additionally, 3D depth images only have a single channel, whereas most 2D CNNs have been trained on 3-channel RGB images. [29] argues that it is, therefore, better to train such networks from scratch.

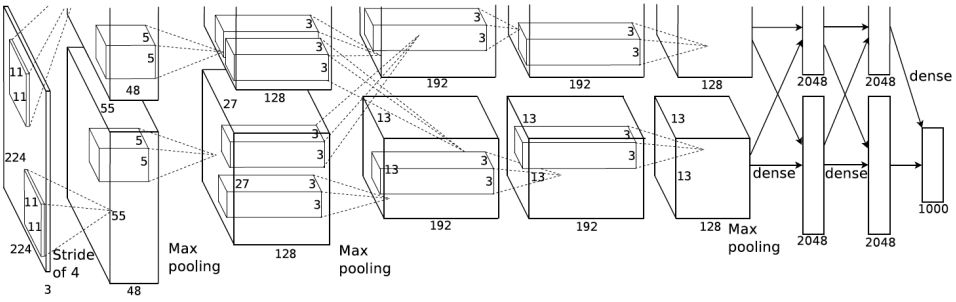


Figure 2.1: Illustration of a Convolutional Neural Network. The figure is borrowed from AlexNet [18]

2.2 Depth Images

In contrast to color, or RGB images, depth images contain information about the distances. Each pixel contains a measure of the distance from the camera x,y plane to the real world point projected through that pixel.

2.3 Pose Estimation

Human Pose Estimation is an area well-researched area because of its many applications. Much of the research focuses on finding pose in RGB images, where hand-crafted features such as parallel edges [22, 24], silhouette features [11], or pictorial structures [7, 8], have been used to suggest candidates for limbs, joints, and so on. From such evidence, constrained kinematic models, tree graphs, or decision trees [27], are used to construct the poses with varying degrees of success. With such top-down approaches, will the runtime of the algorithm increase when multiple people are present in the image.

With the increasing popularity of CNNs in recent years, many techniques utilizing them have also been developed for human pose estimation [31]. A particularly popular approach is to use a deep CNN to produce heat-maps that suggest candidate locations for various body parts [34, 35].

Much research has been done in estimating human pose in two dimensions, as quite large datasets have been made such, as the MPII, or the Human 3.6M datasets [1, 14].

The other approach is to use object recognition to find key features for

the whole image. Then the recognized landmarks are combined to build up the people instances.

In [4], the second approach is used. Two CNNs, one for landmark localization and the other for recombination, are trained to estimate human pose. One of the networks produces a *confidence map* for each joint. Each pixel in the confidence map contains the probability, or confidence, that the pixel is part of a person's joint. The other creates Part Affinity Field (PAF)s, which is a map of vectors pointing in the direction of one of M limbs. [1] These maps are assembled by bipartite matching to create the 2D skeletons observed in the scene.

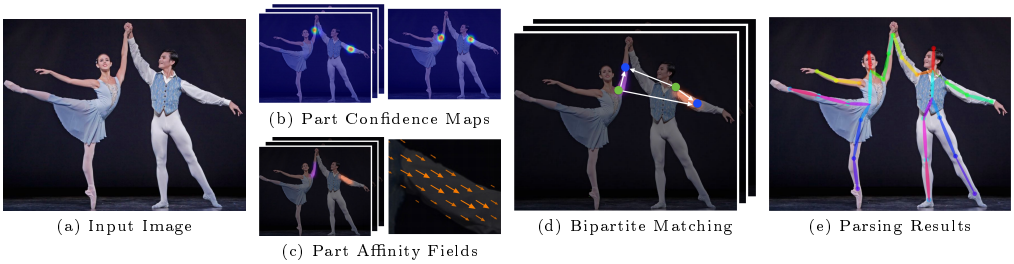


Figure 2.2: The pipeline used in OpenPose [3, 4]. The input image 2.2a is fed into the two networks, which produce joint detections in confidence maps 2.2b and PAFs 2.2c. Bipartite matching is performed in 2.2d, to determine which detected joints should be connected by a limb. 2.2e shows the finished results.

Using the method described in [4], no information is given for joints that are not accurately detected. If, for example, an extremity is occluded together with half of the connecting limb, the extremity will not be part of the output skeleton, even if the joint could be extrapolated from the parts of the limb that is visible. This is also true for undetected joints in the middle of a joint chain. The joint could be extrapolated using the surrounding joints. The problem with joint-extrapolation happens because of the bipartite matching, which does not work if any joint is missing.

[1] This work will stick to the convention of using the term *limb* to describe any *connection between any pair of body landmarks*. The body landmarks will be termed *joints*.

Chapter 3

Datasets

This chapter presents an overview of the datasets suitable for human pose estimation in depth images. The datasets are evaluated according to closeness to real-world conditions, the amount of data, and the accuracy of the recorded poses.

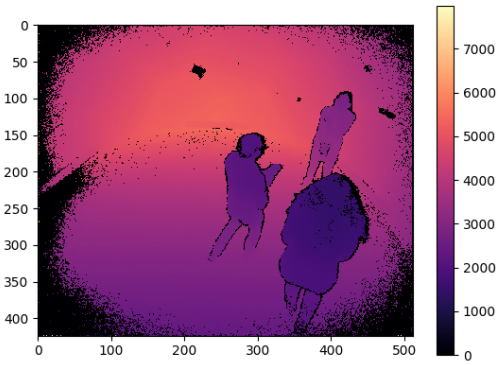
3.1 The Panoptic Studio Dataset

The dataset used in this work was created from source material provided by the Panoptic Studio [15, 16]. Calibration files, predicted ground-truth skeletons, and depth images were downloaded using a modified version of the download script provided by the dataset. The GNU parallel program [30] was used to download it at a faster rate. Some problems were experienced, as the Panoptic Studio server was quite unreliable, which resulted in significant delays, as well as corrupted files and, in return, a smaller dataset than desired.

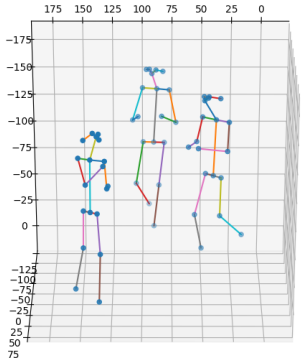
The ground truth positions for the skeletons in the dataset were obtained using 2D human pose estimation on the HD, VGA, and Kinect cameras in the Panoptic Studio, and triangulating the matcing joints.

A set of five complete sequences were set aside as the test set to ensure that the network would not recognize any similar frames or body positions from previous sets. The dataset contains samples from a wide variety of activities, view-angles, the number of people in the scene, and body compositions.

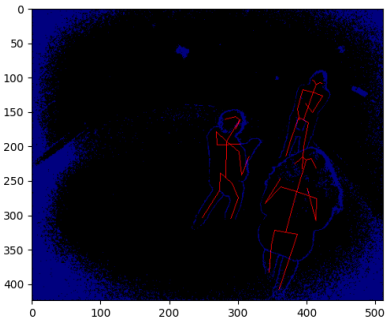
The sequences in the dataset were recorded at approximately 30



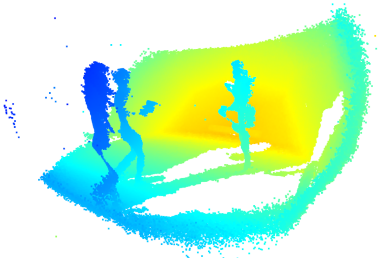
(a) Depth image from KINECTNODE6.



(b) Skeletons defined in the world coordinate system.



(c) Skeletons reprojected to the depth image



(d) The depth image projected to a point cloud.

Figure 3.1: Combination of datapoints for sequence 160226_haggling1, frame 144 in the Panoptic Studio Dataset [16].

fps. All frames and world coordinate positions at an approximate single timestep are hereby referred to as a *frame*. To allow for poses to change, two samples were extracted per second. Each sequence was recorded by ten different Kinects from a height of 1 and 2 meters above the ground. This results in a total number of 10x the amount of samples listed in Table 3.1.

	Category	Sequence Name	Duration (MM:SS)
Training / Validation	Range of Motion	171204_pose1	17:30
	Range of Motion	171204_pose2	22:30
	Range of Motion	171204_pose3	5:00
	Range of Motion	171204_pose5	15:00
	Range of Motion	171204_pose6	12:50
	Range of Motion	171026_pose1	13:20
	Range of Motion	171026_pose3	4:20
	Social Games	160226_haggling1	8:00
	Social Games	160422_haggling1	8:00
	Social Games	160422_ultimatum1	15:00
	Musical Instruments	160906_band1	1:00
	Musical Instruments	160906_band2	5:00
	Musical Instruments	160906_band3	5:00
	Toddler	160906_ian2	5:00
	Others	170915_office1	3:00
	Others	160906_pizza1	5:00
	Dance	170307_dance5	6:40
Test	Range of Motion	171204_pose4	17:30
	Range of Motion	171026_pose2	9:00
	Social Games	160224_haggling1	5:00
	Toddler	160906_ian1	2:00
	Others	170407_office2	3:00

Table 3.1: Sequences used for training, validation and testing

3.2 Dataset Augmentations

The coco19 points from the pose directories were reprojected into the depth image using the perspective camera model

$$\boldsymbol{u} = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_{3\times3} & \boldsymbol{t}_{3\times1} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \tilde{\boldsymbol{X}} \tag{3.1}$$

The matrix K is the camera intrinsics and describes how to transform the point from the normalized image-plane to the pixel coordinates in the image.

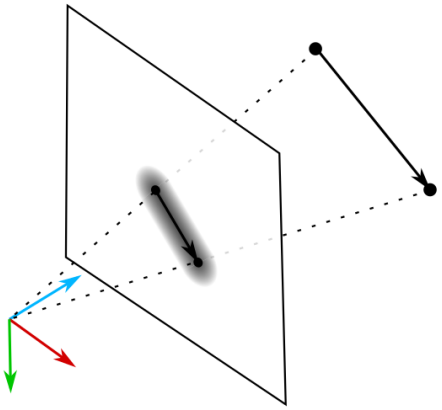
$$K = \begin{bmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

where f_u, f_v is the focal lengths in column/row direction, s is the skew and c_u, c_v is the center of the image in column/row direction.

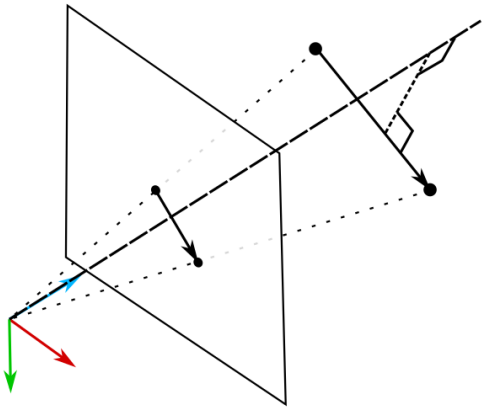
In the dataset, each sequence provided a file containing the K matrices for all the sensors in an unordered list. It was therefore assumed that the placement in the list corresponded to the number of the sensor. However, the extrinsic parts of these did not yield correct results. Another set of extrinsic rotation/translation matrices were provided in a different file where the sensor number *was* identified. Still, these extrinsic parameters were calibrated for the color camera of each sensor. Since the depth camera and color camera of the Kinect are placed approximately 20cm from each other, a vector $\begin{bmatrix} 0.02 \\ 0 \\ 0 \end{bmatrix}$ was added after calculating the extrinsic part of the model to compensate. The resulting projections can be seen in figure 3.1c. Still, the K matrices had to be sourced from the unordered list and introduced uncertainty about whether the datasets will be accurate. Since the Kinect cameras are the same model, and after all very similar, this uncertainty were chosen to be ignored.

$$K(x,y) = \begin{cases} \frac{2}{\pi\sigma^2}(1 - ((\frac{x}{\sigma})^2 + (\frac{y}{\sigma})^2)) & \text{if } |(\frac{x}{\sigma})^2 + (\frac{y}{\sigma})^2| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Target limb-maps for each frame were created by projecting all instances of a certain type of limb onto an empty matrix of the same dimensions as the depth frame, illustrated in Figure 3.2b. Because of the

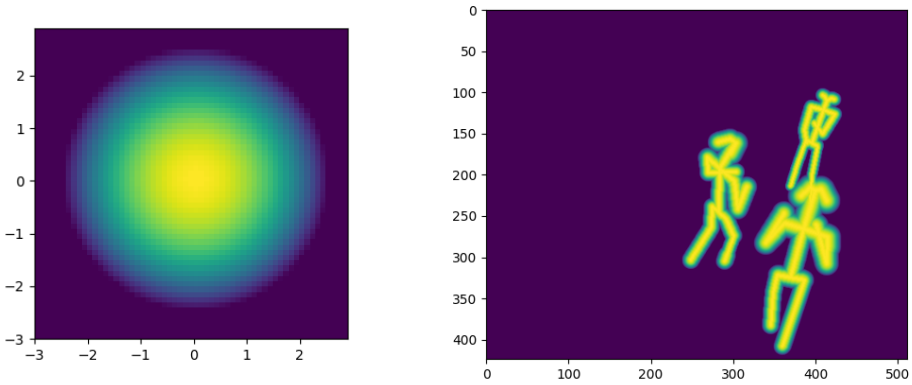


(a) A limb is projected onto the image plane



(b) The beam cast through a pixel in the image plane

Figure 3.2



(a) A 2D Epanechnikov [6] kernel.

(b) Variable kernel width

Figure 3.3: Kernel variation based on distance

discrete nature of a depth image, projecting the 3D limb into the image will yield a set of pixels. For each pixel point along the line created by the projected limb, the depth was calculated by finding the closest points of the infinite 3D line created by the limb and the projected beam from the camera coordinate system through the 3D point formed by the pixel. The closest points of the resulting skewed 3D lines was calculated by solving the linear equation set 3.6, and plugging the results in to the corresponding line-equations.

A 2D Epanechnikov Kernel with a σ varying based on the distance were then used to calculate the magnitude for each of the 3D vectors pointing in the direction of the limb in question. The distance used to calculate the width of the kernel were stored for later comparison. Where limbs would occlude each other, the values for the pixel with the closest point were retained. The reason for varying the σ was to simulate that the shell of points around a limb would appear smaller at a greater distance. Because a normal epanechnikov kernel would change the max value of the center of the line, the equation was simplified to a quadratic function 3.4.

$$K(x,y) = \begin{cases} 1 - ((\frac{x}{\sigma(x,y)})^2 + (\frac{y}{\sigma(x,y)})^2) & \text{if } |(\frac{x}{\sigma(x,y)})^2 + (\frac{y}{\sigma(x,y)})^2| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

$$d = \begin{cases} \max(\|\mathbf{j_1}\|, \|\mathbf{j_2}\|) & \text{if} \|\mathbf{v}\| > \max(\|\mathbf{j_1}\|, \|\mathbf{j_2}\|) \\ \min(\|\mathbf{j_1}\|, \|\mathbf{j_2}\|) & \text{if} \|\mathbf{v}\| < \min(\|\mathbf{j_1}\|, \|\mathbf{j_2}\|) \\ \|\mathbf{v}\| & \text{otherwise} \end{cases} \tag{3.5}$$

$$\begin{aligned} L_1 = (l_{1x}, l_{1y}, l_{1z}),\ u_1 = (u_{1x}, u_{1y}, u_{1z}) \\ L_2 = (l_{2x}, l_{2y}, l_{2z}),\ u_1 = (u_{2x}, u_{2y}, u_{2z}) \end{aligned}$$

$$\begin{aligned} &\begin{bmatrix} u_{1x}^2 + u_{1y}^2 + u_{1z}^2 & -(u_{1x} * u_{2x} + u_{1y} * u_{2y} + u_{1z} * u_{2z}) \\ u_{1x} * u_{2x} + u_{1y} * u_{2y} + u_{1z} * u_{2z} & -(u_{2x}^2 + u_{2y}^2 + u_{2z}^2) \end{bmatrix} \\ &= \begin{bmatrix} u_{1x}(l_{2x} - l_{1x}) + u_{1y}(l_{2y} - l_{1y}) + u_{1z}(l_{2z} - l_{1z}) \\ u_{2x}(l_{2x} - l_{1x}) + u_{2y}(l_{2y} - l_{1y}) + u_{2z}(l_{2z} - l_{1z}) \end{bmatrix} \end{aligned} \tag{3.6}$$

Equation 3.5 describes how σ changes by the value in the depth image D. The constant c is chosen by the width of the limb in pixels at a certain distance.

The resulting magnitudes for the limbs are illustrated by the declining intensity in figure 3.3b. The produced sample tensors are illustrated in figure 3.4. For each sequence, a "long" tensor is made. The first slice of the tensor is the depth image, of size $1 \times w \times h$. Next, the limb maps are stored. They contain the decomposition of the vector, x, y, z in each pixel. This part of the sample, therefore, has the size $3M \times w \times h$ for M limbs. Last is the Joint Maps, which contain both the distance to the center of the joint and the confidence for each pixel. The confidences are also simulated by placing a kernel made by equation 3.4 at the projected location for the joint, with a σ based on the distance from the camera to the joint. This results in the size $2N \times w \times h$ for N joints. In addition to the target maps in figure 3.4, the resulting dataset also contains a vector with the position of all joints in the image in the coordinate frame of the camera. This is to provide ground truth targets for the articulation network if the joint is occluded in the joint map.

A single tensor of samples was created for each sequence. At training time, all sequences except the test sequences were combined to a single dataset which was shuffled.

The raw dataset is captured at 30 fps for each of the ten kinects in the Panoptic Studio. To not capture similar poses in the dataset, three

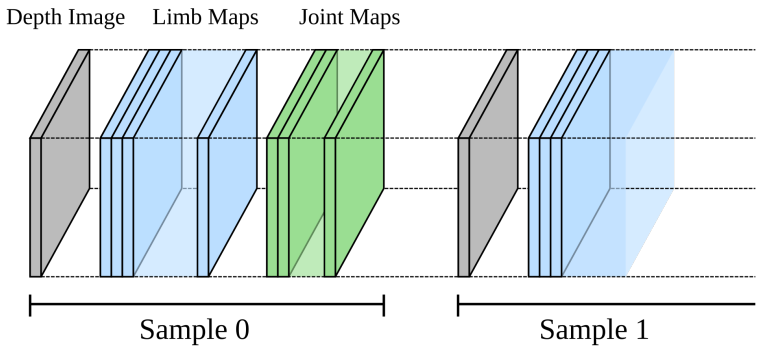


Figure 3.4: The sturcture of the created dataset

frames are sampled from each kinect per second. In addition, the order the kinects are sampled is shuffled after each sampling round.

Chapter 4

DepthPose

The DepthPose here introduced is a complete system for extracting human 3D poses from a single depth image in real-time[1]. It is specifically tailored for use in robotic applications where computational resources are limited. Because the MECS is envisioned to be a mobile unit, DepthPose has to be robust to be able to see the person from different viewing angles despite possible occlusions between the person and the depth-sensor. The architecture is inspired by and builds on the architecture in OpenPose [3]. The general pipeline of DepthPose follows the OpenPose pipeline in figure 2.2 closely, with the addition of a novel Articulation Network that refines the poses after the bipartite matching.

One of the goals for the MECS project is to look for patterns that could lead to worsening or more dangerous living conditions. To that end, Human Activity Recognition (HAR) is implemented with the purpose of tracking a user from day to day. Representing the pose in 3D will simplify application areas such as HAR, because a 3D representation of a skeleton can be defined by a coordinate system constrained to any two connected limbs from the observed skeletons. This means that two different skeleton observations can be represented by a common reference point. Had the skeletons been represented in 2D, the same skeleton seen from two different angles could look vastly different, even if referenced from the same limb. Comparing the two 2D poses will therefore be a more difficult problem than comparing the same poses referenced from a common 3D coordinate system.

[1] Real-time is defined as being capable of processing at least 30 depth frames per second on the specified minimum hardware requirements.

4.1 Architecture

The system pipeline is outlined in figure 4.1. As in [3], two stages are used to extract the pose from an image. In each stage, a Recurrent Neural Network (RNN) architecture iteratively refines the output from the network at that stage. This iterative architecture was inspired by Convolutional Pose Machines [35].

At timestep $t = 0$ in the first stage, a set of depth features are created by the first CNN, *df*. These are stacked with the *limb-maps* produced by the next CNN, *lp*. At $t = 0$ these limb-maps will be initialized to a known value, $\mathbf{0}$, so only the learned bias weights influence *lp* at this timestep. *lp* will then refine these limb-maps at subsequent timesteps $0 < t \leq T_P$.

At timestep $t = T_P + 1$ the refined "first guess" limb-maps from *lp* are again stacked with the depth features from *df*, and used as inputs to *jp* which produces a set of likely *joint-maps*. These joint-maps are then passed to the Assembly function, which performs the bipartite matching algorithm and constructs a set of skeletons. An instance of the Articulation Network is created for each of the detected skeletons. The Articulation Networks refine the poses for each of the detected skeletons. The refined skeleton poses are then projected onto a set of limb-maps which is used instead of the "first guess" limb-maps from *lp*, in successive timesteps $T_P + 1 < t \leq T_P + T_C$.

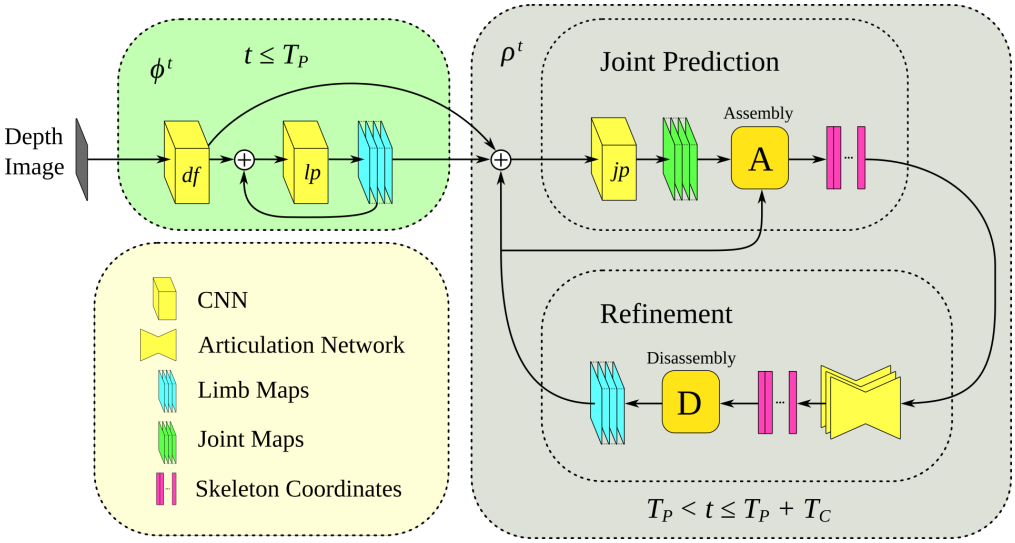


Figure 4.1: Main architecture, as in [4] two recurrent stages, ϕ^t and ρ^t are used to iteratively refine the limb and joint locations.

4.1.1 Depth feature extraction

Figure 4.2: Example of an RGB image on the left vs. a comparative depth image of the same scene on the right. Excerpt sourced from [5].

The main task for the CNN *df* is to find useful depth features for the subsequent networks. As illustrated in Figure 4.2, the information in depth images is not as dense as in RGB images. With the goal of object recognition, networks that are trained on RGB images could rely on features such as colors and textures. For example, the blue shirt vs. the khaki pants or the checkerboard pattern of the floor. However, none of those features are present in the depth image. In both images, edges could be extracted to indicate the outline of objects. Indeed, the edges detected in the depth image could be more useful for this purpose than in the RGB image because they would represent the actual 3D boundary between two objects. A quick example could be two boxes of the same color placed partly behind each other. In an RGB image, they could appear to be part of the same object, whereas a sharp edge would separate them in a depth image. Still, much of the information in depth images have to rely on gradients and other features at a larger scale. It can be speculated that representations for planes geometric or organic shapes will be learned.

Figure 3.1d shows that depth images can also be represented as points in a volume. However, to use such a representation of the data as input to a 3D CNN, a limit would have to be placed on the observed depth to define a fixed input size. This would lead to a sparse representation of the data, where many convolutions would contain empty space. Since no information can be gathered in the occluded parts of the scene anyway, a single channel depth image is a more succinct representation of the input.

As discussed in Section 2.1, the feature extraction parts of a network can be trained on large, unrelated datasets. However, no models that were pre-trained on depth images were found, so applying transfer learning for this part of the network was not an option. This part of the model is therefore not necessarily optimal, and having only seen the inside of a dome, it might not have found features that would perform better in a real-world environment.

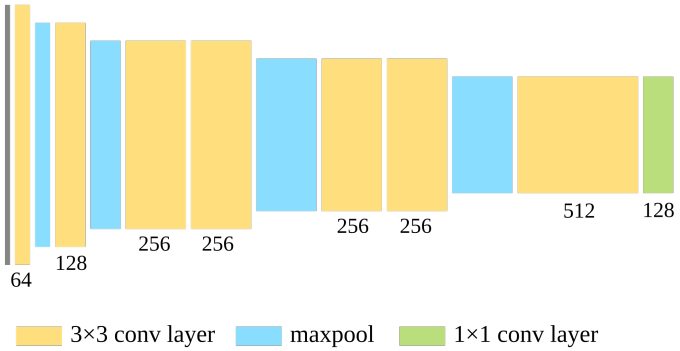


Figure 4.3: Depth feature extractor network. The maxpool layers has size 2×2 and stride 2. Each convolutional layer uses the Rectified Linear Unit (ReLU) activation function.

The OpenPose architecture reuses the first 10 layers of VGG-19 (E configuration) [28]. Therefore, this part of the network takes inspiration from it as well. VGG-19 is constructed using several 3×3 convolutional layers with ReLU [23] activations, and maxpool operations at certain depths. By only using enough 3×3 convolutional layers, the number of learnable parameters are kept down, while preserving the receptive field of a larger kernel (not stacked)[2]. This is ideal for keeping the network as small, and thus as fast as possible. However, since the information at a small scale can be sparse in depth images, dilation is used to obtain a larger receptive field for each convolutional layer. This is different from pooling layers, because they emit which *feature* in a certain layer has the strongest, weakest or calculate what the average response to the input was. On the other hand, a dilated convolutional layer *creates* features for a larger receptive field, and does not downscale the spacial dimensions of the feature tensor, if same-padding is utilized. The first convolutional layer

[2]For an input and output consisting of two channels the learnable parameters can calculated: One 7×7 kernel $\rightarrow 7^2C^2 = 49C^2$ parameters. Three stacked 3×3 kernels $\rightarrow 3(3^2C^2) = 27C^2$ parameters.

in Figure 4.3 uses dilation of 4 to compensate for the lack of small-scale features in the depth image.

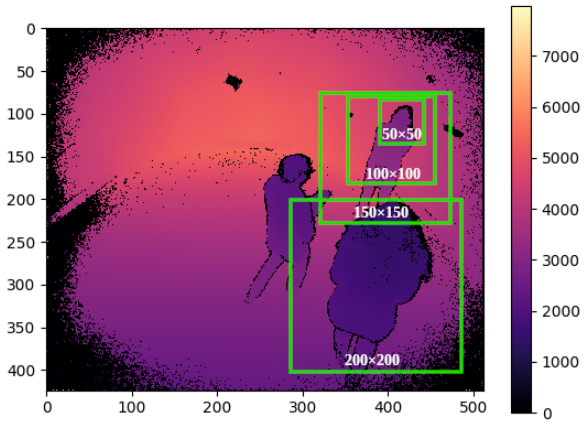


Figure 4.4: Relevant receptive fields in a depth image. Fields are shown as green boxes, with sizes placed near the edge.

4.1.2 Limb- and Joint-Maps

In [3, 35] it is observed that the receptive field of the convolutions are important to establish long-range inferences about body landmarks. This might be because the network learns about the natural symmetries in the human body, and can therefore more easily predict the relationship between different body parts.

In the revisited version of OpenPose, the 7×7 convolution blocks in the pipeline are replaced by convolution blocks of three 3×3 filters, where the emissions from each filter is concatenated at the end of the block. This preserves the receptive field of the convolution block, while reducing the number of learnable parameters, as noted in [28]. In addition, by concatenating the result from each layer at the end of the block, a type of residual network [12] is created, mitigating the vanishing gradient problem.

DepthPose uses three such convolution blocks, with a two final 1×1 convolution layers with dropout between them. This final feature vector is then upsampled through a transposed convolution layer. The output of this is a tensor of $w \times h \times N$ where N is the number of feature maps needed for the network. Since the limb and joint maps look for the same kind of

long-reaching features, the architecture is similar.

The limb-map network produce tensors $3M \times w \times h$ where M is the number of limbs in a skeleton. Each limb has 3 components, as they encode the vector pointing in the direction of the limb at that coordinate.

The joint-maps are tensors $2N \times w \times h$ where N is the number of joints in a skeleton. The tensors encode the depth of the limb at that location, as well as the confidence.

4.1.3 Articulation network

The articulation network solves the problem where two people occludes the other. If a joint of the same body part for two people is co-linear to the camera, this joint can not be represented in the joint-map for the occluded person.

The articulation network is stacked on top of the part-detection network and its main role is to refine the limb lengths and angles between each joint. Each of the detected persons are passed through the articulation network, which leads to a bit more complexity and runtime for the network based on the number of people. However, since the network has so comparatively few inputs, and is quite shallow, preformance is not expected to suffer notably.

The coordinates and confidences for each joint (if not detected, confidence is 0) is the input to the network. The network will try to find out what the positions of joints with low confidences, or no detections, should be. It is hypothesized that this network will learn things like symmetry (left and right limbs should have the same length), proportionality (limbs should be proportional to each other), possible articulations, and natural poses.

The architecture is visualized as a simple, almost fully connected neural network. Since each joint has four properties, (x, y, z, c), these are input to a single neuron in the network. The layers after this is however fully connected.

4.2 Assembly

To assemble the poses, candidate points for joints are found by dilating each joint map to find candidate opints. Each of the candidate points are matched up with the corresponding candidate joints following the limb

graph. Then, the line integral 4.1 (line sum, since the sampled pixels are discrete) is taken along the line between the candidate points, however the values for the integral is taken from the corresponding predicted limb map for the joints.

$$S = \sum_{i=0}^{K} \cos(\theta) \tag{4.1}$$

θ is the angle between the the vector defined by the candidate points and the angle of the vector at the points of the limb-maps along the line i, K. The highest scoring integral is the one chosen as the 'correct' connection. After all the limbs have gotten candidate joints, the predicted skeletons are sent to the articulation network.

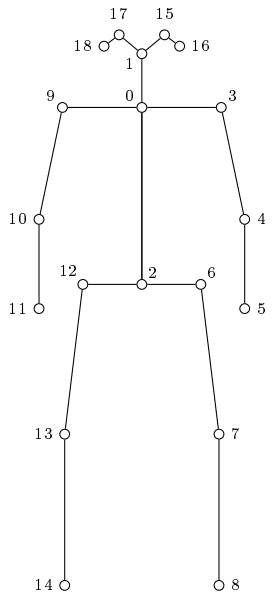


Figure 4.5: Numbering for detected landmarks/keypoint markers.

ID	Description	Std.Coord.
0	Neck	(0.00, 2.34)
1	Nose	(0.00, 3.05)
2	Middle hip	(0.00, 0.00)
3	Left shoulder	(1.05, 2.34)
4	Left elbow	(1.36, 0.86)
5	Left wrist	(1.36, -0.32)
6	Left hip	(0.78, 0.00)
7	Left knee	(1.02, -1.98)
8	Left ankle	(1.02, -3.98)
9	Right shoulder	(-1.05, 2.34)
10	Right elbow	(-1.36, 0.86)
11	Right wrist	(-1.36, -0.32)
12	Right hip	(-0.78, 0.00)
13	Right knee	(-1.02, -1.98)
14	Right ankle	(-1.02, -3.98)
15	Left eye	(0.30, 3.30)
16	Left ear	(0.50, 3.15)
17	Right eye	(-0.30, 3.30)
18	Right ear	(-0.50, 3.15)

Table 4.1: Numberings, names/descriptions and standard coordinates for recognized landmarks

4.3 Training

Chapter 5

Experiments

A sample run of 20 frames (by 10 kinects) were created from the dataset. The total running time was 79 seconds where eight maps failed due to no skeletons being present in the frame. These samples would have been good to include in the dataset, as well as frames from outside the studio. Extrapolating from these numbers a sequence length of 11000 frames (or 6 minutes at 30 fps), would take approximatley 366 hours to run. The data extraction method is in other words in need of a serious optimization.

Had the dataset been finished, the experiments that should have been run on the architecture are the following:

- Compare the learnable parameters between the openpose architecture and the depthpose architecture.
- Calculate the euler distance between joint-locations output by the pipeline and the ground-truth locations (skeleton matching must be preformed, since it is not guaranteed that the IDs for the skeletons would match)
- Compare accuracy and complexity with different values for T_P and T_C
- Assess the usefullness of the articulation network by comparing the joint locations after refinement with joint locations output at $t = T_P + 1$, where just limb-maps from the first stage are used in the prediction.
- Could T_P be reduced to achieve usefulness of the articulation

network?

5.1 Results

Chapter 6

Conclusions

The data extraction method needs to be severely optimized before it can run on a larger dataset.

Currently no conclusions can be drawn about the proposed architecture due to the lack of experiments. However, It should be possible to use the articulation network to provide complete poses, even if key landmarks are occluded. I addition, the articulation network should provide enough information in the refinement loop to allow for a shorter number of loops to be done.

Chapter 7

Future Work

- Find a more accurate pose from a temporal algorithm that takes input from a series of depth images from a single viewpoint.
- Combine multiple viewpoints for a more accurate pose.
- Human activity recognition using 3D pose provided by the method proposed in this paper.
- Train the network over a larger dataset in unstructured environments and with multiple people present.

Appendices

A Depth sensors

Since depth sensors are widely used in different robotics applications for tasks such as SLAM, odometry and object detection, we selected this as our main source of information for monitoring the user. There are mainly three different technologies to choose from: *Structured light*, *Time-of-Flight (ToF)* and *Stereo vision*.

A.1 Stereo vision

uses two cameras that are observing parts of the same scene. In commercial packages the cameras are usually calibrated, so we have measurements to put into the camera matrix as well as the rotation and translation between the two camera matrices.

However, to get a 3D structure, we need to find common feature points between the two cameras. To do this, we can use various feature descriptors such as ORB, SWIFT and SURF. When good matches has been found between the images, we measure the disparity between the points, and triangulate the distance. The depth measurements for the rest of the image are then calculated by matching pixels close to the found featurepoints.

Since this is an optically based technology, it will work well in well-lit scenes that contain many unique featurepoints. If we operate in an homogenous environment with few, or similar textures it will be difficult to find featurepoints to map the environment. An example of this could be on the seabed or inside buildings with limited light conditions, for example during a blackout.

A.2 Structured light

uses a projected pattern of light points onto the scene which is registered by a calibrated camera. Usually, the projected light pattern and camera operate in the infrared part of the electromagnetic spectrum[1]. This means that in locations where one can expect a lot of IR radiation, this technology will not work very well. Since the IR radiation from the sun usually is much stronger than the one emitted from the projector on the sensor, this technology will not work well outside in well-lit conditions. It will however work inside and in conditions where no external light source are provided.

[1] The Microsoft Kinect V2 sensor uses a wavelength of 827-850nm, according to [10, Chapter 4.1]

In addition, since the light is structured and the sensor is calibrated, we can skip the step where we find common featurepoints to triangulate the distance which we have to do in the stereo vision case.

A.3 Time-of-Flight

cameras uses the known constant of c to calculate distances in the image, by measuring the time a light-pulse emitted from the camera uses to be reflected onto the camera sensor. For example, Microsofts Kinect v2 uses a specialized ToF-pixel array in conjunction with a timing generator and modulated laser diodes to obtain per-pixel depth images [26].

As with structured light sensors, this is susceptible to interferrence from external light sources, or specular surfaces, and has limited range because of light fall-off. However, since the distance calculations are timing based, we can obtain framerates up to 30 fps in the Kinect v2 sensor [19].

B Robotic Operating System

In order to make the system easier to use and available to as many platforms as possible, it was decided to create it for the Robotic Operating System (ROS). ROS is a collection of libraries and a runtime environment making communication with different modules and programs on the robot possible.

C Classifiers

write a bit about what classifiers are, and how we use them to find the different keypoints in the image.

D Dataset Download

```
1 #!/bin/bash
2
3 # This script creates a list of wget commands and downloads
      them in parallel:
4 # ./getParallel.sh -n [numThreads] -s "sequence1 sequence2
      sequence3 ..."
5 numThreads=1
6 numKinectViews=10
7 sequences="none"
```

```
8  currDir=$(pwd)
9
10 if command -v wget >/dev/null 2>&1; then
11     WGET="wget -c"
12     mO="-O"
13 else
14     echo "This script requires wget to download files."
15     echo "Aborting."
16     exit 1;
17 fi
18
19
20 while getopts "n:s:k:" args
21 do
22     case $args in
23   n)
24       numThreads=$OPTARG
25       ;;
26   s)
27       IFS=', ' read -r -a sequences <<< $OPTARG
28       echo "Downloading ${#sequences[@]} sequences"
29       ;;
30   k)
31       numKinectViews=$OPTARG
32       if [ $numKinectViews -gt 10 ] || [ $numKinectViews -lt 1
             ]; then
33     echo "Invalid number of kinect views. Must be in interval
           [1,10]"
34     echo "Aborting"
35     exit 1;
36       fi
37       ;;
38   *)
39       echo "Did not understand input arguments."
40       exit 0
41     esac
42 done
43
44 declare -a wgetURIList
45
46 if [ ${#sequences[@]} == 0 ]; then
47     echo "need input sequences"
48     exit 0
49 fi
50
51 for name in "${sequences[@]}"
52 do
53     mkdir "$name" ###################################
54     # Adding download for panoptic calibration data
```

```
55      wgetURIList+=("$WGET $mO $name/calibration_$name.json http
            ://domedb.perception.cs.cmu.edu/webdata/dataset/$name/
            calibration_$name.json")
56      # Adding download for kinect calibration data
57      wgetURIList+=("$WGET $mO $name/kcalibration_$name.json http
            ://domedb.perception.cs.cmu.edu/webdata/dataset/$name/
            kinect_shared_depth/kcalibration_$name.json")
58      # Download synctables data
59      wgetURIList+=("$WGET $mO $name/synctables_$name.json http
            ://domedb.perception.cs.cmu.edu/webdata/dataset/$name/
            kinect_shared_depth/synctables.json")
60      wgetURIList+=("$WGET $mO $name/ksynctables_$name.json http
            ://domedb.perception.cs.cmu.edu/webdata/dataset/$name/
            kinect_shared_depth/ksynctables.json")
61
62      # Skipping download RGB videos
63      ##########################
64      # Download kinect depth videos
65      ##########################
66      nodes=(1 2 3 4 5 6 7 8 9 10)
67      for (( c=0; c<$numKinectViews; c++ ))
68      do
69    # Create subfolder
70    mkdir -p $name/kinect_shared_depth/KINECTNODE${nodes[c]} #
          ###################################
71    fileName="kinect_shared_depth/KINECTNODE${nodes[c]}/depthdata
          .dat"
72    wgetURIList+=("$WGET $mO $name/$fileName http://domedb.
          perception.cs.cmu.edu/webdata/dataset/$name/$fileName ||
          rm -v $name/$fileName")
73    # Delete if file is blank: || rm -v $name/$filename"
74      done
75
76      ##########################
77      # Download skeleton data
78      # Coco 19 keypoint definition
79      # by VGA index
80      ##########################
81      wgetURIList+=("$WGET $mO $name/hdPose3d_stage1_coco19.tar
            http://domedb.perception.cs.cmu.edu/webdata/dataset/
            $name/hdPose3d_stage1_coco19.tar || rm -v $name/
            hdPose3d_stage1_coco19.tar")
82      # Delete if file is blank: || rm -v $name/
            hdPose3d_stage1_coco19.tar
83  done
84
85
86  # Downloading all files.
```

```
# printf "%s\n" "${FILES[@]}" | xargs -i mv '{}' /path/to/destination
echo "Num Threads: $numThreads"
parallel --bar -P $numThreads ::: "${wgetURIList[@]}"
```

E Dataset Extraction

Listing 1: camera

```
import numpy as np
from geometry import Point2D, Point3D


class Camera:
    def __init__(self, k_matrix, m_matrix):
        self.im_cols = 512
        self.im_rows = 424
        self.k = k_matrix
        self.m = m_matrix
        self.c_x = k_matrix[0, 2]
        self.c_y = k_matrix[1, 2]
        self.f_x = k_matrix[0, 0]
        self.f_y = k_matrix[1, 1]

    def project_point(self, point: Point2D, img):
        """ Project a point to 3D coordinates """
        x = (point.x - self.c_x) * img[point.row, point.col] / self.f_x
        y = (point.y - self.c_y) * img[point.row, point.col] / self.f_y
        z = img[point.row, point.col]
        return x, y, z

    def normalized_image_point(self, point: Point2D) -> Point3D:
        """ Project a point into the normalized image plane """
        x = (point.x - self.c_x) / self.f_x
        y = (point.y - self.c_y) / self.f_y
        z = 1
        return Point3D(x, y, z)

    def project_frame(self, depth_frame):
        """ Project all points in a depth frame into 3D coordinates """
        points = np.array([self.project_point(Point2D(ix, iy), depth_frame)
                           for iy, ix in np.ndindex(depth_frame.shape)])
        return points
```

```
35
36      def reproject_point(self, point: Point3D) -> Point2D:
37          """ Find the 2D point of the 3D point """
38          coord = np.array([[point.x], [point.y], [point.z]])
39          extrinsic = np.matmul(self.m, np.append(coord, 1).
                transpose())
40          extrinsic = np.matmul(np.eye(3, 4), extrinsic.transpose
                ())
41          extrinsic = np.array([extrinsic[0] / extrinsic[2],
                extrinsic[1] / extrinsic[2], 1])
42          # adding translation, because color_sensor:
43          extrinsic = np.add(extrinsic, np.array([.02, 0, 0]))
44          intrinsic = np.matmul(self.k, extrinsic.transpose())
45          column, row = intrinsic[0], intrinsic[1]
46          return Point2D(int(column), int(row))
47
48      def point_in_cam(self, point: Point3D) -> Point3D:
49          coord = np.array([[point.x], [point.y], [point.z]])
50          extrinsic = np.matmul(self.m, np.append(coord, 1).
                transpose())
51          extrinsic = np.matmul(np.eye(3, 4), extrinsic.transpose
                ())
52          return Point3D(extrinsic[0], extrinsic[1], extrinsic
                [2])
```

Listing 2: geometry

```
1  import math
2  import numpy as np
3
4
5  class Point2D:
6      def __init__(self, x, y):
7          self.x, self.y = x, y
8          self.row, self.col = y, x
9
10     def __str__(self):
11         return f'({self.x}, {self.y}), row: {self.row} col: {
               self.col}'
12
13
14 class Point3D:
15     def __init__(self, x, y, z):
16         self.x, self.y, self.z = x, y, z
17
18     def __str__(self):
19         return f'({self.x}, {self.y}, {self.z})'
20
21
```

```
22 class Line3D:
23     def __init__(self, *args, **kwargs):
24         self.__point_init_args = ['point_a', 'point_b']
25         self.__point_vec_args = ['point', 'vec']
26         if (all({x in kwargs.keys() for x in self.
               __point_init_args}) or
27                 (isinstance(args[0], Point3D) and isinstance(
                       args[1], Point3D))):
28             self.__point_initialization(*args, **kwargs)
29         elif (all({x in kwargs.keys() for x in self.
               __point_vec_args}) or
30                 (isinstance(args[0], Point3D) and isinstance(args
                       [1], Vec3D))):
31             self.__point_vec_initialization(*args, **kwargs)
32         else:
33             raise ValueError("Invalid arguments")
34
35     def __point_initialization(self, *args, **kwargs):
36         try:
37             assert kwargs[self.__point_init_args[0]].isinstance
                   (Point3D)
38             assert kwargs[self.__point_init_args[1]].isinstance
                   (Point3D)
39             point_a = kwargs[self.__point_init_args[0]]
40             point_b = kwargs[self.__point_init_args[1]]
41         except (KeyError, AssertionError):
42             point_a = args[0]
43             point_b = args[1]
44         self.pos = point_a
45         x = point_b.x - point_a.x
46         y = point_b.y - point_a.y
47         z = point_b.z - point_a.z
48         self.u = Vec3D(x, y, z, point_a)
49
50     def __point_vec_initialization(self, *args, **kwargs):
51         try:
52             assert kwargs[self.__point_vec_args[0]].isinstance(
                   Point3D)
53             assert kwargs[self.__point_vec_args[1]].isinstance(
                   Vec3D)
54             self.pos = kwargs[self.__point_vec_args[0]]
55             self.u = kwargs[self.__point_vec_args[1]]
56         except (KeyError, AssertionError):
57             self.pos = args[0]
58             self.u = args[1]
59
60     def __str__(self):
61         return f'pos: {self.pos}, unit_vector: {self.u}'
62
```

```
63
64  class Vec3D:
65      def __init__(self, x, y, z, pos=Point3D(0, 0, 0)):
66          self.x, self.y, self.z = x, y, z
67          self.pos = pos
68
69      def norm(self):
70          m = self.mag()
71          if m == 0:
72              raise RuntimeError("Cannot divide by zero.")
73          return Vec3D(self.x / m, self.y / m, self.z / m, pos=
                self.pos)
74
75      def mag(self):
76          return math.sqrt(self.x ** 2 + self.y ** 2 + self.z **
                2)
77
78      def __str__(self):
79          return f'pos: {self.pos}, components: ({self.x}, {self.
                y}, {self.z})'
80
81
82  def closest_points(line_a: Line3D, line_b: Line3D):
83      t_term_1 = line_a.u.x ** 2 + line_a.u.y ** 2 + line_a.u.z
            ** 2
84      s_term_1 = -(line_a.u.x * line_b.u.x + line_a.u.y * line_b.
            u.y + line_a.u.z * line_b.u.z)
85      t_term_2 = line_a.u.x * line_b.u.x + line_a.u.y * line_b.u.
            y + line_a.u.z * line_b.u.z
86      s_term_2 = -(line_b.u.x ** 2 + line_b.u.y ** 2 + line_b.u.z
             ** 2)
87      q_term_1 = (line_a.u.x * (line_b.pos.x - line_a.pos.x) +
88                  line_a.u.y * (line_b.pos.y - line_a.pos.y) +
89                  line_a.u.z * (line_b.pos.z - line_a.pos.z))
90      q_term_2 = (line_b.u.x * (line_b.pos.x - line_a.pos.x) +
91                  line_b.u.y * (line_b.pos.y - line_a.pos.y) +
92                  line_b.u.z * (line_b.pos.z - line_a.pos.z))
93      a = np.array([[t_term_1, s_term_1],
94                    [t_term_2, s_term_2]])
95      b = np.array([[q_term_1],
96                    [q_term_2]])
97      res = np.linalg.solve(a, b)
98      t = res[0][0]
99      s = res[1][0]
100     point_a = Point3D(line_a.pos.x + t*line_a.u.x,
101                       line_a.pos.y + t*line_a.u.y,
102                       line_a.pos.z + t*line_a.u.z)
103     point_b = Point3D(line_b.pos.x + s*line_b.u.x,
104                       line_b.pos.y + s*line_b.u.y,
```

```
105                         line_b.pos.z + s*line_b.u.z)
106     return point_a, point_b
107
108
109 def distance_to_3d_line(point: Point3D, line: Line3D,
        lim_points):
110     vec = Vec3D(point.x, point.y, point.z, Point3D(0, 0, 0))
111     beam = Line3D(vec.pos, vec)
112     closest, _ = closest_points(beam, line)
113     dist = math.sqrt(closest.x**2 + closest.y**2 + closest.z
            **2)
114     d_a = math.sqrt(lim_points[0].x**2 +
115                     lim_points[0].y**2 +
116                     lim_points[0].z**2)
117     d_b = math.sqrt(lim_points[1].x**2 +
118                     lim_points[1].y**2 +
119                     lim_points[1].z**2)
120     if dist > d_a and dist > d_b:
121         return max(d_a, d_b)
122     if dist < d_a and dist < d_b:
123         return min(d_a, d_b)
124     return dist
125
126
127 if __name__ == '__main__':
128     # Test closest points
129     l1 = Line3D(Point3D(0.0, 2.0, -1.0), Vec3D(1.0, 1.0, 2.0))
130     l2 = Line3D(Point3D(1.0, 0.0, -1.0), Vec3D(1.0, 1.0, 3.0))
131     a, b = closest_points(l1, l2)
132     assert (a.x, a.y, a.z) == (-1.5, 0.5, -4.0)
133     assert (b.x, b.y, b.z) == (0.0, -1.0, -4.0)
```

Listing 3: loader

```
1  import numpy as np
2  import json
3  import re
4  from camera import Camera
5  from os import listdir
6  from os.path import isfile, join
7
8
9  class DataLoader:
10     """
11     Object for loading data.
12     Data is accessed through frame- and kinect-number.
13     One DataLoader object is needed per sequence. Use of this
           class would
14     therefore be:
```

```
15      DataLoader sequence_name = DataLoader(data_path,
            sequence_name)
16      depth_image, _ = sequence_name.frame(idx)
17      """
18      def __init__(self, data_path: str, sequence: str):
19          self.depth_dir = f'{data_path}/{sequence}/
                kinect_shared_depth'
20          self.kinect_calib = f'{data_path}/{sequence}/
                kcalibration_{sequence}.json'
21          self.kinect_sync_table = f'{data_path}/{sequence}/
                ksynctables_{sequence}.json'
22          self.panoptic_calib = f'{data_path}/{sequence}/
                calibration_{sequence}.json'
23          self.panoptic_sync_table = f'{data_path}/{sequence}/
                synctables_{sequence}.json'
24          self.body_3d_scene_dir = f'{data_path}/{sequence}/
                hdPose3d_stage1_coco19'
25          self.sensors = None
26          self.color_sensors = None
27          with open(self.kinect_calib, 'r') as calib_file:
28              calib = json.load(calib_file)
29              self.sensors = calib['sensors']
30          with open(self.panoptic_calib, 'r') as calib_file:
31              calib = json.load(calib_file)
32              self.color_sensors = [c for c in calib['cameras']
                    if c['type'] == 'kinect-color']
33          # The kinect number mapping tells us which index in the
                self.sensors array
34          # the corresponding kinect node is.
35          self.kinect_number_mapping = [1, 2, 3, 4, 5, 6, 7, 8,
                9, 10]
36          self.kinect_nodes = ['KINECTNODE1', 'KINECTNODE2', '
                KINECTNODE3', 'KINECTNODE4',
37                               'KINECTNODE5', 'KINECTNODE6', '
                                    KINECTNODE7', 'KINECTNODE8',
38                               'KINECTNODE9', 'KINECTNODE10']
39          self.cameras = [Camera(*self._k_m_matrix(n)) for n in
                self.kinect_nodes]
40
41      def frame(self, idx: int, kinect_node: str):
42          """
43          Return depth image, and bodies and the camera at idx
                for the specified node.
44          Bodies are at format: [x0, y0, z0, score0, x1, y1, ...]
45          """
46          bodies, univ_time = self._bodies_univ_time(idx)
47          depth_idx = self._nearest_depth(univ_time, kinect_node)
48          depth_image = self._depth_image(depth_idx, kinect_node)
49          camera = self._get_camera(kinect_node)
```

```
50
51          return depth_image, bodies, camera
52
53      def min_max(self):
54          files = [int(re.findall(r'\d+', f)[1]) for f in listdir
                (self.body_3d_scene_dir)]
55          files.sort()
56          return files[0], files[-1]
57
58      def _get_camera(self, kinect_node: str):
59          return self.cameras[self.kinect_nodes.index(kinect_node
                )]
60
61      def _bodies_univ_time(self, idx: int):
62          """
63          Load the body3DScene file at the specified index
64          returns both an array of skeletons and the univ_time
65          for this idx
66          """
67          fpath = f'{self.body_3d_scene_dir}/body3DScene_{idx
                :08}.json'
68          with open(fpath, 'r') as scene_file:
69              metafile = json.load(scene_file)
70          univ_time = metafile['univTime']
71          bodies = [joint_array['joints19'] for joint_array in
                metafile['bodies']]
72          return np.array(bodies, dtype=float), univ_time
73
74      def _nearest_depth(self, univ_time: float, kinect_node: str
            ):
75          """
76          Find the nearest depth image to the univ_time for the
                given node
77          """
78          sync_table = None
79          with open(self.kinect_sync_table, 'r') as
                sync_table_file:
80              sync_table = json.load(sync_table_file)
81          timestamps = sync_table['kinect']['depth'][kinect_node
                ]['univ_time']
82          closest = min(range(len(timestamps)), key=lambda i: abs
                (timestamps[i] - univ_time))
83          return closest
84
85      def _depth_image(self, depth_idx: int, kinect_node: str):
86          im_cols = 512
87          im_rows = 424
88          f_size = im_cols * im_rows
89
```

```
90          fpath = f'{self.depth_dir}/{kinect_node}/depthdata.dat'
91          im = None
92          with open(fpath, 'rb') as s_file:
93              # offset is multiplied by 2 because uint16 takes
                    two bytes per number
94              a = np.fromfile(s_file, dtype=np.uint16, offset=2*
                    f_size*depth_idx, count=f_size)
95              a = a.reshape((im_rows, im_cols))
96              im = np.fliplr(a)
97          return im
98
99      def _k_m_matrix(self, kinect_node: str):
100         kinect_number = int(re.search(r'\d+', kinect_node).
                group())
101         sensor = self.sensors[self.kinect_number_mapping.index(
                kinect_number)]
102         sensor_name = f'50_{kinect_number:02}'
103         color_sensor = next(s for s in self.color_sensors if s[
                'name'] == sensor_name)
104         k_matrix = np.array(sensor['K_depth'])
105         m_matrix = np.concatenate((np.array(color_sensor['R']),
                 np.array(color_sensor['t'])), axis=1)
106         m_matrix = np.concatenate((m_matrix, np.array([[0, 0,
                0, 1]])), axis=0)
107         return k_matrix, m_matrix
```

Listing 4: target_maps

```
1  from util import kernel, line_a_b
2  from geometry import Point3D, distance_to_3d_line, Line3D
3  import numpy as np
4
5
6  def target_map(bodies, edges, camera, imsize):
7      rows, cols = imsize
8      bodies_2D = []  # pixel points for the edges
9      bodies_3D = []
10     for body in bodies:
11         bodies_2D.append([camera.reproject_point(Point3D(i[0],
                i[1], i[2]))
12                           for i in body.reshape(-1, 4)])
13         bodies_3D.append([camera.point_in_cam(Point3D(i[0], i
                [1], i[2]))
14                           for i in body.reshape(-1, 4)])
15     # Creating a array on format:
16     # array: [ body0: [ edge0: [im], edge1: [im], ...],
17     #          body1: [ edge0: [im], edge1: [im], ...],
18     #            ... ]
19     edge_array = create_edge_map(bodies_2D, bodies_3D, imsize,
```

```
        edges, camera)
20      joint_array = create_joint_map()
21      return edge_array, joint_array
22
23
24  def create_edge_map(bodies_2D, bodies_3D, imsize, edges, camera
        ):
25      rows, cols = imsize
26      # Saving the edges in an array that has the following
            dimensions:
27      # (bodyNum, edgeNum, dme(distance, magnitude, x, y, z),
            rows (height), cols (width))
28      edge_map = np.zeros((len(bodies_2D), len(edges), 5, rows,
            cols))
29      edge_array = np.zeros((len(bodies_2D), rows, cols))
30      idx = 0
31      for body2D, body3D in zip(bodies_2D, bodies_3D):
32          body_map = np.zeros((len(edges), rows, cols))
33          b_idx = 0
34          for edge in edges:  # Edge is at format (joint_a,
                joint_b)
35              # Draw the 2D edge, and constrain the interval:
36              # array of 2D pixels in the line
37              line_pixels = line_a_b(rows, cols, body2D[edge[0]],
                    body2D[edge[1]])
38              # Add the kernels to the pixel positions
39              # 1) Find the Line3D equation for the edge we're
                    currently at
40              # 2) Calculate the distance along the projected
                    line going through
41              #    each pixel to the Closest point on the Line3D
                    for the edge.
42              line_eq = Line3D(body3D[edge[0]], body3D[edge[1]])
43              edge_vector = line_eq.u
44              lim_points = (body3D[edge[0]], body3D[edge[1]])
45              edge_map = np.zeros((len(line_pixels), 2, rows,
                    cols))
46              # edge_map = np.zeros((len(line_pixels), rows, cols
                    ))
47              e_idx = 0
48              for pixel in line_pixels:  # pixel is 2D point
49                  # find distance to 3D line through the 2D pixel
50                  # find 3D location for pixel
51                  dist = distance_to_3d_line(camera.
                        normalized_image_point(pixel),
52                                             line_eq, lim_points)
53                  # create a kernel based on the distance to the
                        3D line
54                  # add that kernel to an output map, and add
```

```
                    that output map to the
55                  # result array
56                  sigma = 3000 // dist  # Found something that
                        seemed to work
57                  sigma = sigma if sigma % 2 != 0 else sigma+1
58                  sigma = max(sigma, 1e-10)
59                  # print(f'Sigma: {sigma}, Dist: {dist}')
60                  _, _, k = kernel(sigma)
61                  k_im = kernel_image(k, pixel, imsize)
62                  d_im = k_im.copy()
63                  d_im[d_im > 0] = 1/dist
64                  # edge_map[e_idx] = k_im
65                  edge_map[e_idx, 0] = k_im
66                  edge_map[e_idx, 1] = d_im
67                  e_idx += 1
68              edge_map = np.amax(edge_map, axis=0)
69              # Apply vectors (TODO)
70              body_map[b_idx] = edge_map[0]
71              b_idx += 1
72          body_map = np.amax(body_map, axis=0)
73          edge_array[idx] = body_map
74          idx += 1
75      edge_array = np.amax(edge_array, axis=0)
76      return edge_array
77
78
79  def create_joint_map():
80      pass
81
82
83  def kernel_image(k, pos, imsize):
84      img = np.zeros(imsize)
85      k_max = max(k.shape)
86      k_size = k_max // 2  # floor division
87      if k_size <= 1:
88          return img
89
90      # Assume pos.row/col is zero indexed
91      # irs: image row start, # kce: kernel col end ....
92      krs = -1 * min(pos.row - k_size, 0)
93      kre = k_max-1 + min(0, imsize[1]-1 - (pos.row+k_size)) + 1
            # Because indexing
94      kcs = -1 * min(pos.col - k_size, 0)
95      kce = k_max-1 + min(0, imsize[0]-1 - (pos.col+k_size)) + 1
96
97      irs = max(0, pos.row - k_size)
98      ire = min(pos.row + k_size, imsize[1] - 1) + 1
99      ics = max(0, pos.col - k_size)
100     ice = min(pos.col + k_size, imsize[0] - 1) + 1
```

```
101
102         img[ics:ice, irs:ire] = k[kcs:kce, krs:kre]
103         return img
```

Listing 5: visualization

```
 1 from matplotlib import pyplot as plt
 2 from target_maps import target_map
 3 from loader import DataLoader
 4 from geometry import Point3D
 5 from util import line_a_b
 6 import numpy as np
 7 import math
 8 import open3d as o3d
 9
10 edges = [
11     (0, 1), (0, 2), (1, 15), (15, 16), (0, 3), (3, 4), (4, 5),
12     (2, 6), (6, 7), (7, 8), (1, 17), (17, 18), (0, 9), (9, 10),
13     (10, 11), (2, 12), (12, 13), (13, 14)
14 ]
15
16
17 def show_depth_frame(kinect_node, idx, loader: DataLoader):
18     depth_frame, _, _ = loader.frame(idx, kinect_node)
19     plt.imshow(depth_frame, cmap='magma', interpolation='
           nearest')
20     plt.colorbar()
21     plt.show()
22
23
24 def show_depth_frame_as_pointcloud(kinect_node, idx, loader:
       DataLoader):
25     depth_image, _, camera = loader.frame(idx, kinect_node)
26     points = camera.project_frame(depth_image)
27
28     pcd = o3d.geometry.PointCloud()
29     pcd.points = o3d.utility.Vector3dVector(points)
30     o3d.visualization.draw_geometries([pcd])
31
32
33 def display_skeleton(kinect_node, idx, loader):
34     depth_image, bodies, camera = loader.frame(idx, kinect_node
           )
35
36     cmap = plt.get_cmap('jet')
37     rgba = cmap(depth_image)
38
39     target_edges, target_joints = target_map(bodies, edges,
           camera, depth_image.shape)
```

```
40      rgba = target_edges
41
42      plt.imshow(depth_image, interpolation='nearest', cmap='jet'
            )
43      plt.imshow(rgba)
44      plt.show()
45
46
47  def body_3d_coords(body_array):
48      coords = []
49      skeleton = body_array.reshape(-1, 4).transpose()
50      for i in skeleton:
51          print(i)
52          coords.append(Point3D(skeleton[i][0], skeleton[i][1],
                skeleton[i][2]))
53      return coords
54
55
56  def read_bodies(bodies):
57      skeletons = []
58      xs = np.array([], dtype=float)
59      ys = np.array([], dtype=float)
60      zs = np.array([], dtype=float)
61      for body in bodies:
62          body = np.array(body)
63          skeleton = body.reshape(-1, 4).transpose()
64          xs = np.concatenate([xs, skeleton[0, :]])
65          ys = np.concatenate([ys, skeleton[1, :]])
66          zs = np.concatenate([zs, skeleton[2, :]])
67          skeletons.append(skeleton)
68      return skeletons, zs, ys, zs
69
70
71  def skeleton_visualization_3d(loader: DataLoader, idx: int):
72      bodies, univ_time = loader._bodies_univ_time(idx)
73      skeletons, xs, ys, zs = read_bodies(bodies)
74
75      fig = plt.figure(figsize=(4, 4))
76      ax = fig.add_subplot(111, projection='3d')
77      # Plot landmarks
78      ax.scatter(xs, ys, zs)
79      # Draw lines between edges in each skeleton
80      for skeleton in skeletons:
81          for edge in edges:
82              coords_x = [skeleton[0, edge[0]], skeleton[0, edge
                    [1]]]
83              coords_y = [skeleton[1, edge[0]], skeleton[1, edge
                    [1]]]
84              coords_z = [skeleton[2, edge[0]], skeleton[2, edge
```

```
                    [1]]]
85                ax.plot(coords_x, coords_y, coords_z)
86        # Ensure equal axis
87        max_range = np.array([xs.max() - xs.min(), ys.max() - ys.
              min(), zs.max() - zs.min()]).max() / 2.0
88
89        mid_x = (xs.max() + xs.min()) * 0.5
90        mid_y = (ys.max() + ys.min()) * 0.5
91        mid_z = (zs.max() + zs.min()) * 0.5
92        ax.set_xlim(mid_x - max_range, mid_x + max_range)
93        ax.set_ylim(mid_y - max_range, mid_y + max_range)
94        ax.set_zlim(mid_z - max_range, mid_z + max_range)
95
96        plt.show()
97
98
99    def epanechnikov_2d(sigma, distance):
100       _x = np.arange(-distance, distance + 2, 1)
101       _y = np.arange(-distance, distance + 2, 1)
102       xx, yy = np.meshgrid(_x, _y)
103       z = ((2/(math.pi * sigma ** 2)) * (1 - ((xx/sigma)**2 + (yy
              /sigma)**2)))
104       z = np.where(z > 0, z, 0)
105
106       fig = plt.figure(figsize=(4, 4))
107       ax = fig.add_subplot(111)
108       ax.pcolor(_x, _y, z)
109       plt.show()
110
111
112   if __name__ == '__main__':
113       loader = DataLoader('../data', '160226_haggling1')
114
115       # epanechnikov_2d(5, 5)
116       # skeleton_visualization_3d(loader, 144)  # [x]
117       # show_depth_frame('KINECTNODE6', 144, loader)  # [x]
118       # show_depth_frame_as_pointcloud('KINECTNODE6', 144, loader
              )  # [x]
119       display_skeleton('KINECTNODE6', 144, loader)  # [x]
120       # skeleton = Skeleton(np.zeros(76))
121       # print(skeleton.joint_coordinates(SkeletonJoint.L_ELBOW))
```

Listing 6: util

```
1  import math
2  import numpy as np
3  from skimage import draw
4  from geometry import Point3D, Point2D, Line3D, Vec3D,
       distance_to_3d_line
```

```
5
6
7  def line_to_array(d_array: np.ndarray, edges):
8      """
9      Returns an a set of arrays of row/column pairs for the
           input edges. The input
10     edges are pairs of 2d points that a line should be drawn
           between.
11     """
12     edge_array = []
13     idx = 0
14     for edge in edges:
15         start, end = edge[0], edge[1]
16         rr, cc = draw.line(start[0], start[1], end[0], end[1])
17         rr_s, rr_e = constrain_interval(rr, d_array.shape[0])
18         cc_s, cc_e = constrain_interval(cc, d_array.shape[1])
19         start = max(rr_s, cc_s)
20         end = min(rr_e, cc_e)
21
22         if end != -1:
23             rr = rr[start:end]
24             cc = cc[start:end]
25             line_pixels = [(rr[i], cc[i]) for i in range(len(cc
                   ))]
26             edge_array.append(line_pixels)
27         idx += 1
28     return edge_array
29
30
31 def line_a_b(img_rows: int, img_cols: int, point_a: Point2D,
       point_b: Point2D):
32     """
33     Return the constrained line pixel locations for the line
           going from
34     point_a to point_b in the image.
35     """
36     rr, cc = draw.line(point_a.row, point_a.col, point_b.row,
           point_b.col)
37     rr_s, rr_e = constrain_interval(rr, img_rows)
38     cc_s, cc_e = constrain_interval(cc, img_cols)
39     start = max(rr_s, cc_s)
40     end = min(rr_e, cc_e)
41     if end != -1:
42         rr = rr[start:end]
43         cc = cc[start:end]
44         line_pixels = [Point2D(rr[i], cc[i]) for i in range(len
               (cc))]
45     return line_pixels
46
```

```
47
48 def constrain_interval(arr, max_val):
49     idx = 0
50     while (arr[idx] < 0 or arr[idx] > max_val - 1) and idx <
           len(arr) - 1:
51         idx += 1
52     if idx >= len(arr):
53         idx = -1
54     arr_start = idx
55
56     idx = len(arr) - 1
57     while (arr[idx] < 0 or arr[idx] > max_val - 1) and idx > 0:
58         idx -= 1
59     if idx < 0:
60         idx = -1
61     arr_end = idx
62     return arr_start, arr_end
63
64
65 def kernel(sigma):
66     _x = np.arange(-sigma, sigma+1, 1)
67     _y = np.arange(-sigma, sigma+1, 1)
68
69     xx, yy = np.meshgrid(_x, _y)
70     z = 1 - ((xx / sigma) ** 2 + (yy / sigma) ** 2)
71     z = np.where(z > 0, z, 0)
72     return _x, _y, z
73
74
75 def epanechnikov_2d(sigma, distance):
76     _x = np.arange(-distance, distance + 2, 1)
77     _y = np.arange(-distance, distance + 2, 1)
78     xx, yy = np.meshgrid(_x, _y)
79     z = ((2/(math.pi * sigma ** 2)) * (1 - ((xx/sigma)**2 + (yy
           /sigma)**2)))
80     z = np.where(z > 0, z, 0)
81
82     fig = plt.figure(figsize=(4, 4))
83     ax = fig.add_subplot(111)
84     ax.pcolor(_x, _y, z)
85     plt.show()
86
87
88 def add_max(org, kernels, centres):
89     """
90     Add the value of a kernel (of arbitrary size, on an
           arbitrary location),
91     to an array if the value in the array is not larger than
           the value in the filter.
```

```
92          """
93          return org
94
95
96      def add_kernel(org: np.ndarray, kernel: np.ndarray, center):
97          """
98          Add the value of the kernel (of arbitrary size, on the
                location 'center'), to an array
99          if the value in the array is not larger than the value in
                the kernel.
100         """
101         # place filter at the array, and exclude any locations that
                are out of bounds
102         c_r, c_c = center.row, center.column
103         k_r, k_c = kernel.shape
104         row_start = max(0, k_r/2 - c_r)
105         pass
106
107
108     if __name__ == '__main__':
109         from matplotlib import pyplot as plt
110
111         _x, _y, z = kernel(7)
112         print(z)
113         print(z.shape)
114         fig = plt.figure(figsize=(4, 4))
115         ax = fig.add_subplot(111)
116         ax.pcolor(_x, _y, z)
117         plt.show()
```

Bibliography

[1] Mykhaylo Andriluka et al. '2D Human Pose Estimation: New Benchmark and State of the Art Analysis'. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014.

[2] H. Bay, T. Tuytelaars and L. Van Gool. 'SURF: Speeded Up Robust Features'. In: *Computer Vision – ECCV 2006*. Ed. by A. Leonardis, H. Bischof and A. Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33833-8.

[3] Zhe Cao et al. *OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. 2019. arXiv: 1812.08008 [cs.CV].

[4] Zhe Cao et al. 'Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields'. In: *CVPR*. 2017.

[5] Maxime Devanne. '3D Human Behavior Understanding by Shape Analysis of Human Motion and Pose'. PhD thesis. Dec. 2015.

[6] V. A. Epanechnikov. 'Non-Parametric Estimation of a Multivariate Probability Density'. In: *Theory of Probability & Its Applications* 14.1 (1969), pp. 153–158. DOI: 10.1137/1114019. eprint: https://doi.org/10.1137/1114019. URL: https://doi.org/10.1137/1114019.

[7] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 'Pictorial Structures for Object Recognition'. In: *Int. J. Comput. Vision* 61.1 (Jan. 2005), pp. 55–79. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000042934.15159.49. URL: https://doi.org/10.1023/B:VISI.0000042934.15159.49.

[8] M.A. Fischler and R.A. Elschlager. 'The Representation and Matching of Pictorial Structures'. In: *IEEE Transactions on Computers* C-22.1 (1973), pp. 67–92. DOI: 10.1109/T-C.1973.223602.

[9] Kunihiko Fukushima. 'Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position'. In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251. URL: https://doi.org/10.1007/BF00344251.

[10] Silvio Giancola, Matteo Valenti and Remo Sala. *A Survey on 3D Cameras: Metrological Comparison of Time-of-Flight, Structured-Light and Active Stereoscopy Technologies*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-91761-0. URL: http://dx.doi.org/10.1007/978-3-319-91761-0.

[11] Grauman, Shakhnarovich and Darrell. 'Inferring 3D structure with a statistical image-based shape model'. In: *Proceedings Ninth IEEE International Conference on Computer Vision*. 2003, 641–647 vol.1. DOI: 10.1109/ICCV.2003.1238408.

[12] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[13] P. V. C. Hough. 'Method and means for recognizing complex patterns'. US 3 069 654A. 1960. URL: https://patents.google.com/patent/US3069654A/en.

[14] Catalin Ionescu et al. 'Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.7 (July 2014), pp. 1325–1339.

[15] Hanbyul Joo et al. 'Panoptic Studio: A Massively Multiview System for Social Interaction Capture'. In: 2017.

[16] Hanbyul Joo et al. 'Panoptic Studio: A Massively Multiview System for Social Motion Capture'. In: *ICCV*. 2015.

[17] Helseetaten Oslo Kommune. *Oslohelsa – Kortversjonen, Oversikt over helsetilstand og påvirkningsfaktorer*. June 2016. URL: https://www.oslo.kommune.no/getfile.php/13139280/Innhold/Politikk%20og%20administrasjon/Statistikk/Oslohelsa_kortversjon.pdf.

[18] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[19] Elise Lachat et al. 'Assessment and Calibration of a RGB-D Camera (Kinect v2 Sensor) Towards a Potential Use for Close-Range 3D Modeling'. In: *Remote Sensing* 7.10 (Oct. 2015), pp. 13070–13097. ISSN: 2072-4292. DOI: 10.3390/rs71013070. URL: http://dx.doi.org/10.3390/rs71013070.

[20] D. G. Lowe. 'Object recognition from local scale-invariant features'. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2.

[21] N. Magnenat-Thalmann, R. Laperrière and D. Thalmann. 'Joint-Dependent Local Deformations for Hand Animation and Object Grasping'. In: *Proceedings on Graphics Interface '88*. Edmonton, Alberta, Canada: Canadian Information Processing Society, 1989, pp. 26–33.

[22] Greg Mori and Jitendra Malik. 'Estimating Human Body Configurations Using Shape Context Matching'. In: *Proceedings of the 7th European Conference on Computer Vision-Part III*. ECCV '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 666–680. ISBN: 3540437460.

[23] Vinod Nair and Geoffrey E. Hinton. 'Rectified Linear Units Improve Restricted Boltzmann Machines'. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[24] D. Ramanan and D.A. Forsyth. 'Finding and tracking people from the bottom up'. In: *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* Vol. 2. 2003, pp. II–II. DOI: 10.1109/CVPR.2003.1211504.

[25] E. Rublee et al. 'ORB: An efficient alternative to SIFT or SURF'. In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571.

[26] John Sell and Pat O'Connor. 'XBOX One Silicon'. Hot Chips 25. Aug. 2013. URL: http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.10-SoC1-epub/HC25.26.121-fixed-%20XB1%2020130826gnn.pdf.

[27] Jamie Shotton et al. 'Efficient Human Pose Estimation from Single Depth Images'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.12 (2013), pp. 2821–2840. DOI: 10.1109/TPAMI.2012.241.

[28] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[29] Xinhang Song, Luis Herranz and Shuqiang Jiang. *Depth CNNs for RGB-D scene recognition: learning from scratch better than transferring from RGB-CNNs*. 2018. arXiv: 1801.06797 [cs.CV].

[30] O. Tange. 'GNU Parallel - The Command-Line Power Tool'. In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. DOI: http://dx.doi.org/10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[31] Jonathan Tompson et al. *Joint Training of a Convolutional Network and a Graphical Model for Human Pose Estimation*. 2014. arXiv: 1406.2984 [cs.CV].

[32] S. Ullman and S. Brenner. 'The interpretation of structure from motion'. In: *Proceedings of the Royal Society of London. Series B. Biological Sciences* 203.1153 (1979), pp. 405–426. DOI: 10.1098/rspb.1979.0006. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspb.1979.0006. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspb.1979.0006.

[33] P. Viola and M. Jones. 'Rapid object detection using a boosted cascade of simple features'. In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. 2001, pp. I–I.

[34] Keze Wang et al. 'Human Pose Estimation from Depth Images via Inference Embedded Multi-Task Learning'. In: *Proceedings of the 24th ACM International Conference on Multimedia*. MM '16. Amsterdam, The Netherlands: Association for Computing Machinery, 2016, pp. 1227–1236. ISBN: 9781450336031. DOI: 10.1145/2964284.2964322. URL: https://doi.org/10.1145/2964284.2964322.

[35] Shih-En Wei et al. 'Convolutional pose machines'. In: *CVPR*. 2016.