INFO8010: Semantic Segmentation of Depth Images for Robotic Grasping

Joachim Coline[1]

[1] *joachim.coline@student.uliege.be (s155505)*

In this paper we present a complete pipeline for building and training a fully convolutional neural network that is able to perform binary semantic segmentation of depth images. In particular, we focus on the problem of distinguishing objects from the background in depth data collected from a Time-Of-Flight (TOF) 3D sensor. Pixel-wise object detection is especially useful in robotic grasping tasks where correctly understanding a scene is key to reducing the risks of unwanted interactions. To tackle the problem of labelling many depth images by hand, we introduce a method for automatically generating many labelled depth images through simulations of both the scene and the noise pattern of the sensor. We show that under certain assumptions regarding the environment, our model is able to fulfill its task with great success on new synthetic images, but has yet to be improved for correctly dealing with real captures.

Keywords: deep learning, semantic segmentation, fully convolutional neural network, robotic grasping, depth data, Time-Of-Flight sensor

I. INTRODUCTION

The autonomous robotic manipulation of objects in a real-world setting is a challenging task because it requires the system to constantly adapt to changes or perturbations in the environment. Furthermore, to be successful, it should also be able to generalize to previously unseen objects. Machine learning techniques provide a way to tackle this problem and their recent outstanding performance in many applications explains why they became trendy in robotics research.

Among the problems encountered in automated grasping, there is the correct perception of the environment: it is crucial for the robot to be able to reliably distinguish target objects from the rest of the scene. Indeed, in doing so, potential unwanted interactions can be avoided, and the space search for an optimal grasp can be greatly reduced. Since 3D sensors are a common solution for identifying three-dimensional patterns in the environment, there is a clear motivation in being able to remove the background signal from the depth data that they collect, in an automatic fashion.

The objective of this work is to build and train a neural network that fulfills at best these requirements, i.e. that segments at best any input depth data into two possible categories — object or background. The whole procedure is broken down into the following steps which are successively detailed in this paper: synthetic dataset generation, neural network design and training, and performance evaluation on both simulated and real images. We also provide some suggestions for future work.

II. RELATED WORK

The problem of image semantic segmentation has been successfully addressed using deep learning models in the recent era. Hundreds of methods have been introduced by the scientific community, backed up with their performance on popular datasets. A review of state-of-the-art segmentation models for general RGB settings is suggested in [1, 2]. Since the depth data considered in this work can be represented in a 2D image, any choice in these models would be relevant to our task.

Regarding the problem addressed, to the best of our knowledge the general task of distinguishing previously unseen objects from the background in raw depth data has never been tackled in the literature. The closest work that we found was [3] in which the objective of the authors is very similar to ours but they restrict the environment to a bin filled with industrial parts of known shape. In their setting, the sensor is fixed and oriented downward, and for the segmentation task they implement an encoder-decoder model using a pretrained VGG architecture for the encoder part and deconvolution layers for the decoder part. Because their methodology and implementation show convincing results, we follow a similar approach for building the neural network in this work.

It is worth noting that other researchers have addressed similar problems but with slight variations. In [4], mask R-CNN architectures designed for RGB images are adapted to depth images for category-agnostic object instance segmentation, and in [5] both RGB and depth data are considered. In our model, we consider using depth data only, as it reduces the amount of information that the robot has to deal with for performing a grasp.

III. METHODS

The steps that were undertaken for building the model are described in the subsequent sections. In section III A, the assumptions that were drawn to orient the work toward a specific setting are specified. In section III B, the approach adopted for generating a custom dataset is described. In section III C, the implemented neural network architecture is presented. Finally, in sections III D and

III E we explain how the model is trained and evaluated.

A. Narrowing the scope

Segmenting objects in a depth map is a very general task: many types of background, objects and sensor data may arise. In order to expect good performance from an image segmentation model, it is thus of natural concern to impose restrictions on certain parameters. In this work we focus on a particular setting imposed by a project in robotics which is currently run by research engineers at University of Liège, that is, INTEGRIA. The aim of this project is to integrate artificial intelligence solutions in robotic grasping tasks.

INTEGRIA researchers are considering placing a 3D sensor at the tip of a robotic arm for capturing depth information about targets that are close to the gripper. In this context, we can expect the background to be relatively close to the sensor. Furthermore, it is assumed that the scene contains objects that are placed in clutter on a table, and that the task of the robot is to successively grasp them one after the other, irrespective of where it places them afterward. Hence, the grasping movement is mainly vertical or close to vertical, so that the sensor may be assumed to be oriented downward.

Since the sensor itself shapes the distribution and dimensions of the input data, it is also important to identify its nature. Among the different possibilities for 3D sensing, Time-Of-Flight (TOF) cameras are a popular solution. To capture depth, these devices measure the round trip time of an artificial light signal from the sensor to the target and back — hence their name. Because a TOF camera is available and operational at the INTEGRIA research unit, its characteristics will be used for generating the data. The device is a *CamBoard pico flexx* development kit from PMD Technologies and is shown in Figure 1. It has a working range of 0.1 to 4m and a resolution of 224x171 pixels which makes it an adequate choice for our application. Indeed the working range is suitable for realistic grasping tasks and, although high resolution is always preferred, considering these particular specifications allows to reduce the complexity of the input domain so that data generation and model training, which are both costly operations, can be performed efficiently. Lastly, the availability of this device makes it possible to confront the segmentation model to real data. Such assessment, yet purely qualitative, is important because it determines the feasibility of implementing the virtually-built segmentation model in a real-world setting, which is the end purpose of this work.

B. Synthetic data generation

Regardless of the implemented neural network, a set of input-output examples is required for training our model. In the context of binary segmentation, this set is given by pairs of 2D maps, one of which is the depth data captured by the sensor, and the other is an image containing two possible labels per pixel, that is, 0 if the corresponding input pixel belongs to the background, and 1 if it is part of an object. Ideally, these pairs should be generated using real data because, as in any supervised learning problem, we want the training set to be a good representation of what can be expected in a previously unseen sample.

FIG. 1: CamBoard pico flexx by PMD Technologies [6].

However, this process would be hard to put into practice, as it would require to capture and manually label thousands of different scenes, which is not feasible in a reasonable amount of time. Furthermore, the resulting dataset would be bound to contain mistakes in the labels due to unforced human error. One possible approach for tackling this problem is to generate synthetic data: by virtually building a scene, placing some objects and performing depth captures, the whole dataset generation task can be fulfilled in an automatic fashion. Such method indeed overcomes the practical issues of generating a dataset with a real scene, but also comes at the cost of being noise-free. In fact by running simulations in which the geometry of the scene is known exactly, the behaviour of the input data can be expected to be too perfect and thus to differ a lot from real images in terms of measurement error — which is not wanted.

One way to take sensor-specific noise into account is to simulate not only the scene, but also the sensor model itself. To this end, the BlenSor framework is a powerful tool [7]. BlenSor is an open source Blender package that is meant for simulating the output of the major ranging technologies (Time-of-Flight, Line Laser and Rotating Laser Scanners). It thus constitutes a natural choice for our problem. Furthermore, all tasks can be automated via Python scripting inside the Blender API, such that the whole dataset can be created by running a single script.

The method is the following. After loading a plane which will serve as ground, a scene is successively built, scanned, then deleted. This process is repeated a number of times corresponding to the desired size of the dataset. Building the scene consists in importing a set of 3D objects (between 0 and 20) and placing them above the ground. The objects are selected randomly among a list of 10,809 models downloaded from open-access online ressources [8–10]. Such large variability in object shape is important for generalizing the segmentation task to previously unseen objects. In each scene, the size, posi-

tion and orientation of each item are the realisation of random variables that are drawn in uniform probability distributions which are described in Table I. Once done, the camera is also placed, then all objects are dropped to the ground using the physics simulation tool provided by Blender. To ensure that the objects remain at the same location after a fall, they are assigned high values of friction. Once the physics simulation is done, the last frame of the episode is selected, and the camera performs a scan by taking into account the specifications of the pico flexx. After storing the result, objects are removed from the scene, and the process starts over.

Object largest dimension [cm]	$\in [4, 10]$
Object x-, y-position [cm]	$\in [-7.5, 7.5]$
Object distance to ground [cm]	$10k,\ k \in \mathbb{N}_0$
Object euler angles [rad]	$\in [0, \pi]$
Camera x-, y-position [cm]	0
Camera distance to ground [cm]	$\in [20, 40]$
Camera rotation around z-axis [rad]	$\in [-\pi, \pi]$
Camera rotation around x-, y-axis [rad]	$\in [-\frac{\pi}{18}, \frac{\pi}{18}]$

TABLE I: Camera and object parameters in each scene. The chosen values reflect what can be expected in a realistic setting. Objects are assigned different distances to ground so that they don't overlap upon import.

The data generation process is illustrated in Figure 2. It is important to note that although the implementation of the described algorithm is straightforward, its execution time is somewhat tedious. Indeed, one drawback of the BlenSor package is that is not meant for real-time processing. Scanning the scene thus turned out to be an expensive operation, requiring a couple of seconds to execute. In addition, it was observed that object imports could also slow down the process, according to the size of the file. The combined effect of these difficulties is bothersome for generating a large dataset but can be tackled by running the script in a background process.

Two last remarks shall be made regarding the described method. Firstly, a Blender addon (downloaded from [11]) was installed to allow the import of `.off` files. Secondly, some of the categories in the downloaded object datasets were not considered, because they were observed to be very flat or to contain very flat parts. This is indeed not wanted, as it could potentially cause a large part of the object to be literally undistinguishable from the background in the depth data — leading to inconsistencies in the corresponding pair of images.

C. Neural network architecture

Based on the successful implementation of an encoder-decoder model for segmenting depth data in [3], it was decided to follow a similar approach. A fully convolutional network [12] is thus considered and implemented

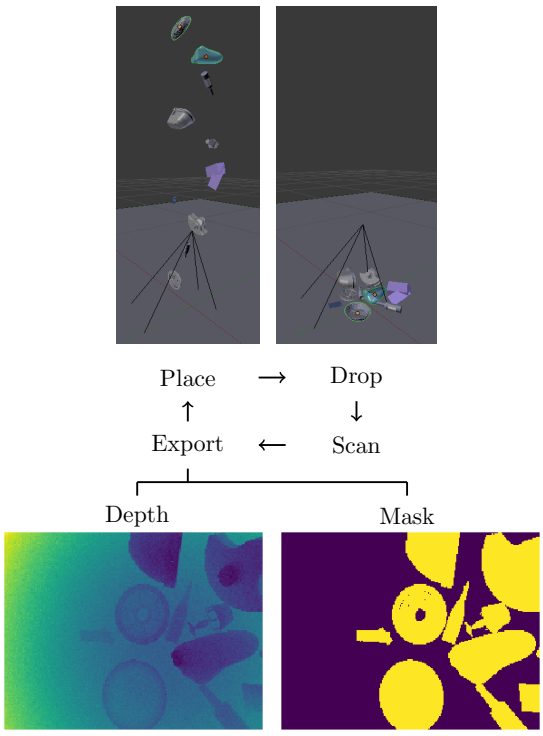


FIG. 2: Illustration of the proposed synthetic data generation method.

from scratch using the PyTorch framework [13]. Its architecture is illustrated in Figure 3. The encoder part consists of the first 31 layers of a VGG16 model whose weights were pretrained on ImageNet. This choice is motivated by three reasons. Firstly, using transfer learning techniques is known to be a way to both reduce required training time and achieve better results [14]. Secondly, VGG networks have proven to be powerful, light feature extractors from their performance in classification accuracy in the context of the Large Scale Visual Recognition Challenge 2014 [15]. Finally, the best results were achieved by using a VGG16 encoder in the original paper on fully convolutional networks for semantic segmentation [12].

Regarding the decoder, it is built with five deconvolution layers, after each of which ReLU and batch normalization are successively applied. It also uses predictions from earlier layers according to the FCN-8s architecture described in [12]. More precisely, the feature maps from the last three max pool operations of the encoder are combined with the outputs of the first three transposed convolutions (see Figure 3). These fusions allow to retrieve and take advantage of spatial and contextual information that is lost in the down-sampling process. The last layer of the network is a convolutional layer whose purpose is to reduce the number of channels to 1. This allows to output a single scalar prediction for each input

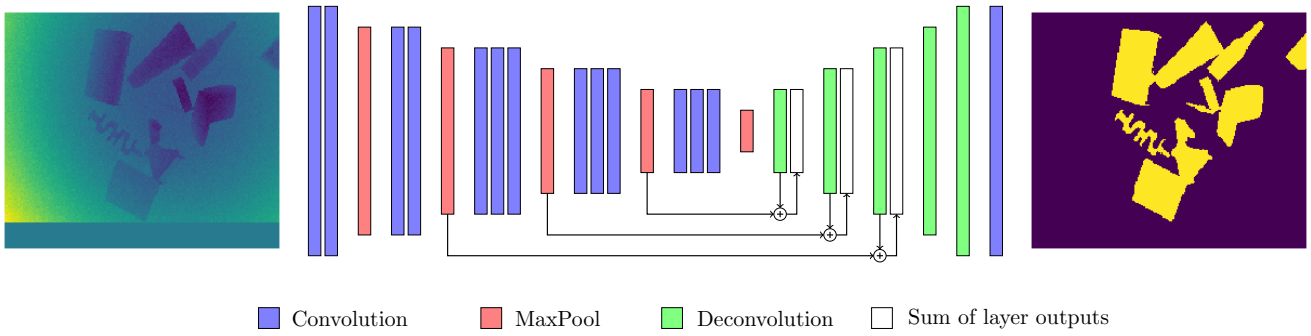


FIG. 3: Illustration of the implemented FCN-8s architecture.

Layer	Output Shape	# of param.
Conv2d-1	[-1, 64, 224, 192]	1,792
ReLU-2	[-1, 64, 224, 192]	0
Conv2d-3	[-1, 64, 224, 192]	36,928
ReLU-4	[-1, 64, 224, 192]	0
MaxPool2d-5	[-1, 64, 112, 96]	0
Conv2d-6	[-1, 128, 112, 96]	73,856
ReLU-7	[-1, 128, 112, 96]	0
Conv2d-8	[-1, 128, 112, 96]	147,584
ReLU-9	[-1, 128, 112, 96]	0
MaxPool2d-10	[-1, 128, 56, 48]	0
Conv2d-11	[-1, 256, 56, 48]	295,168
ReLU-12	[-1, 256, 56, 48]	0
Conv2d-13	[-1, 256, 56, 48]	590,080
ReLU-14	[-1, 256, 56, 48]	0
Conv2d-15	[-1, 256, 56, 48]	590,080
ReLU-16	[-1, 256, 56, 48]	0
MaxPool2d-17	[-1, 256, 28, 24]	0
Conv2d-18	[-1, 512, 28, 24]	1,180,160
ReLU-19	[-1, 512, 28, 24]	0
Conv2d-20	[-1, 512, 28, 24]	2,359,808
ReLU-21	[-1, 512, 28, 24]	0
Conv2d-22	[-1, 512, 28, 24]	2,359,808
ReLU-23	[-1, 512, 28, 24]	0
MaxPool2d-24	[-1, 512, 14, 12]	0
Conv2d-25	[-1, 512, 14, 12]	2,359,808
ReLU-26	[-1, 512, 14, 12]	0
Conv2d-27	[-1, 512, 14, 12]	2,359,808
ReLU-28	[-1, 512, 14, 12]	0
Conv2d-29	[-1, 512, 14, 12]	2,359,808
ReLU-30	[-1, 512, 14, 12]	0
MaxPool2d-31	[-1, 512, 7, 6]	0
ConvTranspose2d-32	[-1, 512, 14, 12]	2,359,808
ReLU-33	[-1, 512, 14, 12]	0
BatchNorm2d-34	[-1, 512, 14, 12]	1,024
ConvTranspose2d-35	[-1, 256, 28, 24]	1,179,904
ReLU-36	[-1, 256, 28, 24]	0
BatchNorm2d-37	[-1, 256, 28, 24]	512
ConvTranspose2d-38	[-1, 128, 56, 48]	295,040
ReLU-39	[-1, 128, 56, 48]	0
BatchNorm2d-40	[-1, 128, 56, 48]	256
ConvTranspose2d-41	[-1, 64, 112, 96]	73,792
ReLU-42	[-1, 64, 112, 96]	0
BatchNorm2d-43	[-1, 64, 112, 96]	128
ConvTranspose2d-44	[-1, 32, 224, 192]	18,464
ReLU-45	[-1, 32, 224, 192]	0
BatchNorm2d-46	[-1, 32, 224, 192]	64
Conv2d-47	[-1, 1, 224, 192]	33

TABLE II: Detailed layout of the model.

pixel. Note that the absence of a last sigmoid activation function is intentional as it allows to compute the loss in a numerically more stable way (see section III E). This additional function is however required for prediction, so that the ouput lies in between 0 and 1, depicting the probability that a pixel belongs to an object.

The total number of parameters of the model is 18,643,713. However, not all of them are set to be trainable. Indeed the encoder defined by the pretrained VGG16 architecture is used as a fixed feature extractor. Only the decoder is trained from scratch, resulting in a total of 3,929,025 trainable parameters. A detailed list of these entities is provided in Table II.

D. Input preprocessing

Prior to feeding the network with scans, some preprocessing is required. Indeed the raw depth data are stored in 224×171 images, but the VGG network requires a three-channel input with dimensions that must be downscalable five times by a factor of 2. To overcome these problems, the input depth image is replicated over three channels, and zero-padding is applied along each dimension that is not a multiple of 32. This allows to prevent distortion of the images, so that the shape characteristics of depth data are preserved. The extra zeros are simply ignored — both during training and predicting tasks.

In addition to these operations, the input is also standardized, i.e. it is scaled so that it has zero mean and unit variance. The result is an input tensor of size `[3,224,192]`, the size of which is transformed through the layers according to the mechanisms presented in Table II. A visual example of a preprocessed input is visible in the left image of Figure 3.

E. Training

a. Initialization For training the model, the trainable parameters are assigned random values according to the default settings provided by PyTorch, that is, Kaiming initialization [16].

b. Loss function The loss function that is minimized during training is the *binary cross-entropy* (BCE) loss function $\mathcal{L}$ defined by

$$\mathcal{L}(\hat{y}, y) = -\left[y \log \hat{y} + (1-y) \log(1-\hat{y})\right] \qquad (1)$$

where $y \in \{0, 1\}$ is the ground truth label (equal to 0 if the pixel belongs to the background and to 1 if it belongs to an object) and where $\hat{y} \in [0, 1]$ is the value predicted by the network. In practice, a modified version of Equation 1 is implemented: this is because $\hat{y}$ is the result of a sigmoid activation function, meaning that taking its logarithm is an unnecessary reciprocal operation that could be avoided. Instead of considering the sigmoid and the BCE loss separately, they can be combined together to come up with a simplified expression that reduces the amplification of numerical errors. This is what is done in the PyTorch class `torch.nn.BCEWithLogitsLoss` [13] which defines its loss function as

$$\mathcal{L}(x, y) = -\left[y \log \sigma(x) + (1 - y) \log\left(1 - \sigma(x)\right)\right] \quad (2)$$

where x is the output of the network prior to applying the sigmoid activation function (i.e. the so-called *logit*) and $\sigma(.)$ is the sigmoid function defined by

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad (3)$$

The use of Equation 2 as loss function for training the network justifies the absence of a sigmoid activation function in the last layer of the model presented in Table II, which is however required when performing a prediction.

Recall that $\mathcal{L}$ is not computed for the outputs corresponding to the zero-valued pixels that were added during preprocessing of the input (see section III D). Ignoring these extra entries is done simply by indexing the predictions of interest only.

c. Optimizer For minimizing the objective function given by Equation 2 over the training set, the Adam optimization algorithm is considered. It is an adaptative learning rate method, meaning that each parameter updated during training has its own learning rate. These individual learning rates are computed using bias-corrected estimations of the first and second moments of the gradient of $\mathcal{L}$, according to the mechanisms presented in [17]. For training our network, the hyperparameters are set to their default recommended values. The exponential decay rates for the moment estimates are thus set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and the ϵ parameter for avoiding dividing by zero is set to 10^{-8}. Regarding the step size α, it is set to 10^{-5}.

d. Training scheme For learning the weights of the model, a dataset of 10,000 pairs of images generated using the method presented in section III B is considered. It is split into three parts: a training set (7,000 pairs) , a validation set (2,000 pairs) and a test set (1,000 pairs). The model is fit by running the algorithm over 100 epochs of the training set, using a batch size of 16. To prevent overfitting, the mean loss over the validation set is computed after each epoch, such that the weights that minimize this loss are kept and assigned to the final model. This method is referred to as *early stopping*. The test set is used to evaluate the model on previously unseen data. For a clear understanding of the proposed training scheme, a summary is provided in the pseudocode of Algorithm 1.

ALGORITHM 1: Training algorithm overview.

```
Initialize model and Adam optimizer parameters
Set number of epochs and batch size
for each epoch do
  for each training batch do
    Perform a forward pass with the input data
    Compute L using Equation 2
    Compute gradient of L w.r.t. all trainable parameters
    Update all trainable parameters using Adam optimizer
  end for
  Compute and store the mean loss L* over validation set
  if L* < all previous L* then
    Update stored model weights
  end if
end for
Assign stored model weights to final model
Assess model on test set
```

In addition to the above procedure, we also investigate how both the dataset size and using variants of the VGG16 encoder may impact performance. To this end, the learning phase is repeated again with a new dataset of 100,000 pairs of images (generated over the course of three weeks), and for modifying the model, pretrained VGG11 and VGG19 encoders are considered. These encoders are similar to VGG16 but differ in the number of convolutional layers. For a detailed description of their architecture refer to [15].

e. Implementation The neural network is implemented from scratch in PyTorch and trained using a computer equipped with a Geforce GTX 1080 TI GPU.

f. Evaluation metrics To assess the model performance on test data, we consider using the following metrics which are applied to each data sample:

- **Mean pixel accuracy**: the mean pixel accuracy is defined as the percent of pixels that were correctly classified in the image.
- **Intersection over union (IoU)**: the IoU metric is specific to each possible output mask (here either object or background). It is defined as the ratio between the number of pixels whose prediction correctly corresponds to the target mask divided by the number of pixels that either belong to the target mask or are predicted as such (or both). It can be understood as an overlap score between the predicted and the target mask. In our case study we only report the IoU for objects.
- **Precision**: the precision score for object prediction is defined as the proportion of pixels that were correctly predicted as objects among all pixels that were predicted as objects.
- **Recall**: the recall score for object prediction is defined as the proportion of pixels that were correctly

predicted as objects among all pixels that actually belong to an object.

Note that to compute these quantities, the predictions of the network must be turned into binary values. This is done by thresholding each output according to a decision boundary which we set to 0.5. In other words, prediction values which lie in under 0.5 are set to 0 (background prediction), and all others are set to 1 (object prediction).

IV. RESULTS

We present our results using a quantitative then a qualitative approach. In the rest of the work, the generated dataset containing 10,000 samples is referred to as the small dataset, and the one containing 100,000 samples as the large dataset.

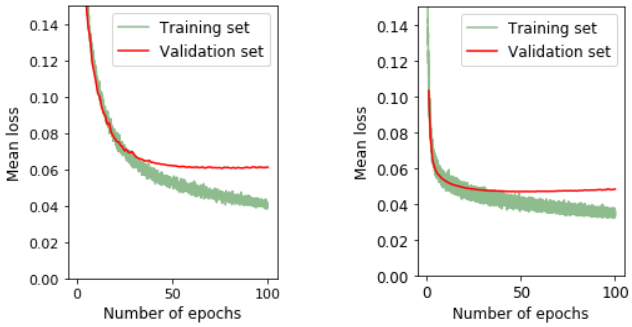
A. Quantitative analysis

By running our training algorithm separately on the small and on the large dataset with our model (FCN-8s architecture with VGG16 encoder), we get the results of Table III.

Training dataset size	Pixel acc.	IoU	Precision	Recall
7,000 samples	97.87	87.59	92.82	94.27
70,000 samples	**98.31**	**90.84**	**95.30**	**95.20**

TABLE III: Evaluation of the segmentation task performed by the final models on the test set of the large dataset. Each score is the mean value obtained for all test samples, expressed in %.

The numbers suggest that overall good performance is achieved by each model for the segmentation task. However, it can be observed that better results are obtained in using 70,000 samples, which highlights the importance of training the model on as much data as possible. It can thus be said that the costly generation of a very large dataset is worth the computational time and effort. As regards to the learning phase itself, we plot the evolution of $\mathcal{L}$ against training algorithm iterations in Figure 4. The curves suggest that the recommended hyperparameters of the Adam optimizer allow a reasonably fast convergence of $\mathcal{L}$ toward a minimum. By training the model on the small dataset, the loss function over the validation set is minimized after 75 epochs; on the large dataset it is minimized after 47 epochs. Note that in Figure 4 the green curve is only an estimation of the loss over the training set because its value is actually computed over one training batch at a time. This explains the stochasticity of the curve, which was smoothened using a moving average filter of size 50. The red curve is more stable because it represents a mean loss that is computed over the same validation set after each epoch. Regarding the required computation time, it took $\sim$ 1 hour to run the training algorithm on the small dataset, and $\sim$ 12 hours on the large dataset.



(a) Small dataset (b) Big dataset

FIG. 4: Evolution of the mean value of $\mathcal{L}$ measured on the training and validation sets during learning.

The effect of modifying the architecture of the encoder is shown in Table IV where we also report the mean time required for a forward pass. This computation time was computed using the GPU by feeding 10,000 times the network with randomized input and taking the average time required. It can be seen that using VGG11 as pre-trained encoder slightly improves performance on all aspects. Indeed all VGG11 scores are greater than or close to the best scores obtained with the other architectures. Furthermore, it requires the least amount of time for a forward pass, which can be very beneficial for the integration of our model in the real-time application of robotic grasping.

Model	Pixel acc.	IoU	Precision	Recall	Forward pass
VGG11	98.35	**91.80**	**96.07**	95.40	**1.31ms**
VGG16	98.31	90.84	95.30	95.20	1.66ms
VGG19	**98.36**	91.56	95.80	**95.43**	1.86ms

TABLE IV: Effect of changing the encoder architecture on overall performance of the model measured on the test set of the large dataset. Each score is the mean value obtained for all test samples, expressed in %.

B. Qualitative analysis

Figure 5 provides examples of previously unseen depth data predicted with our model (FCN-8s architecture with VGG16 encoder) trained on the large dataset. The correct predictions on top of Figure 5 show that in some cases the neural network is able to segment the depth data with remarkable accuracy. However, in others, the predicted masks may contain unacceptably large areas of error, as can be seen in the bottom images of Figure 5. Although these poor predictions are not frequently met

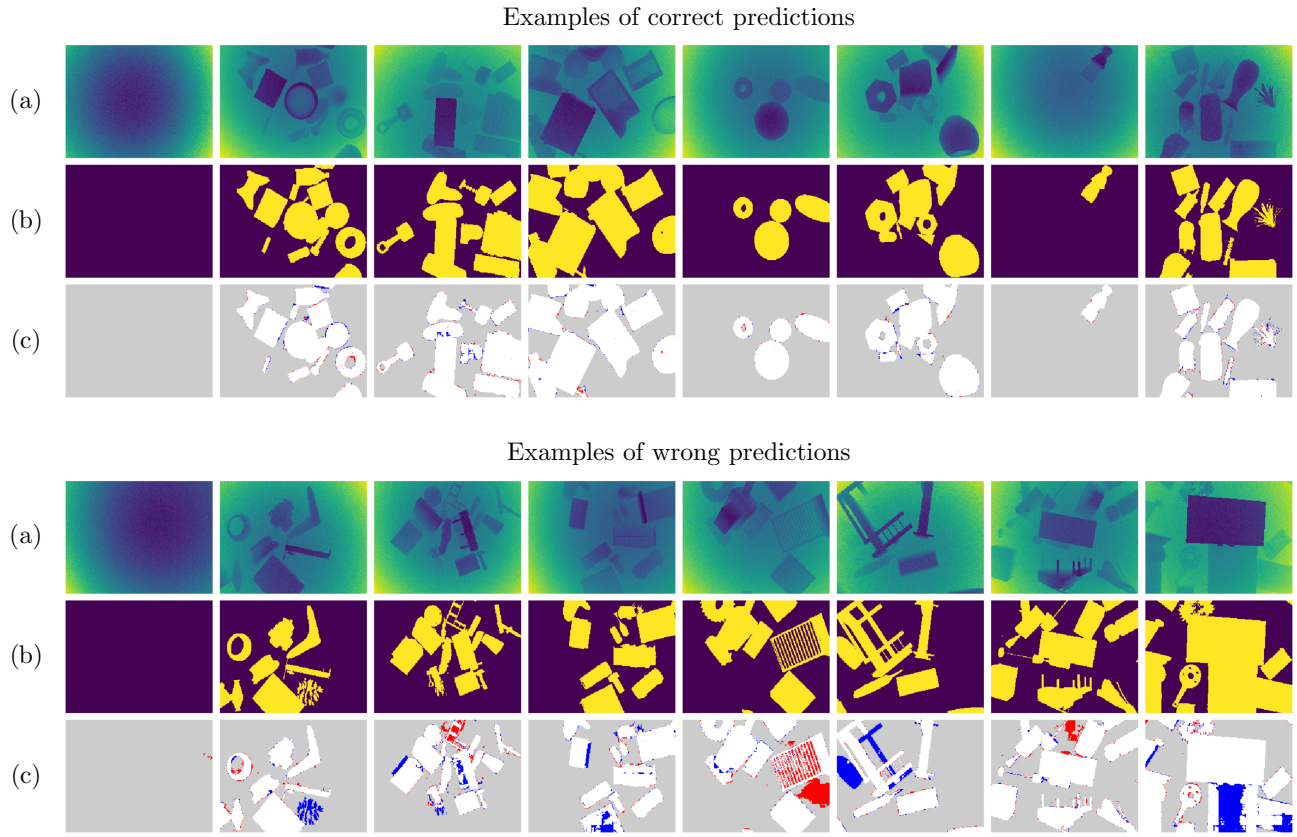


FIG. 5: Examples of correct and wrong predictions of previously unseen simulated depth data with our model (FCN-8s architecture with VGG16 encoder) trained on 70,000 pairs of images: (a) preprocessed input image, (b) ground truth masks, (c) predicted masks where gray pixels indicate a correct background prediction, white pixels a correct object prediction, blue pixels a wrong background prediction and red pixels a wrong object prediction.

in the test data, they show that the robustness of the model is questionable, and that there remains room for improvement. In particular, by visually inspecting various predictions, it was observed that in general the model was less likely to perform well in the following situations:

- the object height is very small, i.e. it can hardly be distinguished from the ground in the input data;
- the object has a large flat part which is parallel or close to parallel to the ground and facing the camera;
- the object shape is very complex;
- The number of object pixels largely dominates the number of background pixels.

V. DISCUSSION

In the following we provide suggestions for overcoming some of the aforementioned limitations. We also examine the model's ability to segment objects in real images which were captured with the pico flexx under the same assumptions as in the generated dataset.

a. Object shape control In the method suggested in section III B, we chose to scale the objects by setting their largest dimension to a randomized value. However, in doing so, the smallest dimension is not controlled. Hence, the risk of encountering unacceptably flat objects still remains. This leads to the same insconsistencies in the training data than the ones we tried to avoid by removing objects with very flat parts. To avoid them, one can simply add a new parameter to our suggested method, that is, a minimal value for all object dimensions. By using this parameter, the object can be stretched if it is considered too flat. Note that Blender provides tools for such stretching operation. More generally speaking, for a better and total control of object shapes, one should consider generating them from scratch.

b. Review synthetic dataset generation method The parameters of Table I result from arbitrary choices. We did not study whether the performance of our model is sensitive to the values that were assigned. However, we believe that changing some of them could potentially be

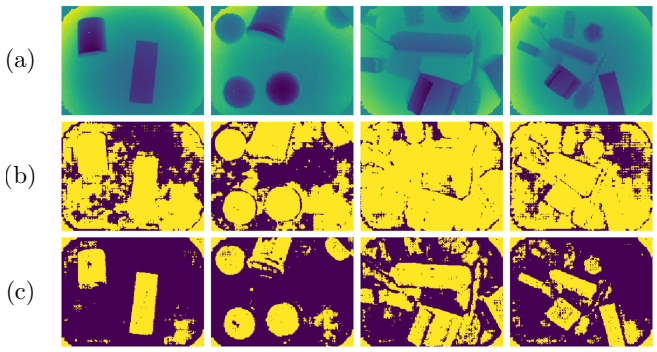


FIG. 6: Illustration of the performance of our model (FCN-8s architecture with VGG16 encoder) on depth images captured with the CamBoard pico flexx from PMD Technologies. (a) preprocessed input where dead pixels are assigned a zero value and all other pixel values are standardized by ignoring the dead ones, (b) prediction with a decision boundary of 0.5, (c) prediction with a decision boundary of 0.9999.

beneficial. For instance, expanding the range of possible positions for each object and changing the proportion of samples containing a heavily cluttered scene are ways to go.

c. Change the decision boundary for prediction The decision boundary was set to 0.5 for pixel-wise object prediction. By increasing or lowering this value, precision can be traded off against recall according to user will.

d. Modify the FCN architecture We used different VGG encoders in this work but there exist other pre-trained models that could be used as feature extractors [18], although they would require to be modified in order to match tensor dimensions for allowing layer additions. As regards to the architecture presented in this work, we thought of two possible improvements. On one hand, it would be interesting to add the first MaxPool layer to the last deconvolution layer. This could possibly help further improve predictions by taking into account finer spatial details from previous layers. On the other hand, by modifying the first layer of the encoder, the three-channel input tensor can be reduced to one. This is possible because the three input channels are identical. The modification consists in taking the sum of the weights of the first VGG16 convolutional layer kernel along the input channel axis and loading them into the new first layer. With this simple trick, the network can be fed with a single channel tensor, i.e. input preprocessing is simplified and the number of required operations for a forward pass is also reduced.

e. Identify real sensor characteristics for a better transfer of the model to a real setting In Figure 6 we illustrate some predictions of our trained model on depth images that were captured with the CamBoard pico flexx from PMD Technologies. Each image consists of objects that were dropped on a parquet floor and captured from a height of $\sim$ 30cm. The objects include small cardboard boxes, tennis balls, toilet paper rolls, and other simple toys. It can be seen that the masks predicted on real data contain large areas of error. Many background pixels are indeed predicted as objects when using a decision boundary of 0.5. However, by increasing the latter as high as 0.9999, it can be observed that the accuracy of the predictions is greatly increased, eventhough noticeable mistakes persist. These relatively good results endorse the validity of our approach for solving a real-world problem by means of simulation. Furthermore, recall that our dataset was generated without prior knowledge of the noise pattern of the data collected by the pico flexx. Indeed default BlenSor settings were used for simulating a TOF camera. A more correct model of the noise that is specific to the pico flexx would help improving performance on real images since in this case the generated dataset would reflect even more accurately what can be expected in real captures. It would also be interesting to add randomness in object material and scene lighting as these components have a significant impact on the error made in TOF-based depth measurements [19]. Furthermore, the introduction of dead pixels should also be considered as they were observed to be commonly encountered in the data collected by the pico flexx. In particular, they are always present in the four corners of the image, as can be seen in Figure 6.

VI. CONCLUSION

In this work we presented an entire framework for performing binary semantic segmentation of previously unseen 2D depth data using a fully convolutional neural network. The objective was to come up with a model that is able to determine the areas in a depth image that belong to an object. To reach this objective, we first narrowed the scope of the problem to a particular setting by drawing assumptions about the environment. We then presented a method for automatically generating a large dataset of labelled depth images by means of simulation. Using this dataset we trained an encoder-decoder model for performing pixelwise object detection. Our experiments show that great prediction accuracy could be achieved on previously unseen simulated depth images — under the condition that they were generated under the same assumptions. However, after discussing some possible approaches for improvement, we illustrated that overall performance was degraded when transferring the model to a real setting. Nevertheless, relatively satisfying results could be obtained (see Figure 6) which suggest that our approach by simulation is valid but requires more care in correctly modelling the shape of the data collected by the sensor.

VII. ACKNOWLEDGMENTS

This work was made possible thanks to Tom Ewbank and Loïc Sacré who kindly provided both the depth sensor and a remote access to a computer equipped with a Geforce GTX 1080 TI GPU.

[1] Farhana Sultana, A. Sufian, and Paramartha Dutta. Evolution of image segmentation using deep convolutional neural network: A survey. 01 2020.

[2] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. 01 2020.

[3] Muhammad Usman Khalid, Janik Hager, Werner Kraus, Marco Huber, and Marc Toussaint. Deep workpiece region segmentation for bin picking. 08 2019.

[4] Michael Danielczuk, Matthew Matl, Saurabh Gupta, Andrew Li, Andrew Lee, Jeffrey Mahler, and Ken Goldberg. Segmenting unknown 3d objects from real depth images using mask R-CNN trained on synthetic point clouds. *CoRR*, abs/1809.05825, 2018.

[5] Seunghyeok Back, Jongwon Kim, Raeyong Kang, Seungjun Choi, and Kyoobin Lee. Segmenting unseen industrial components in a heavy clutter using rgb-d fusion and synthetic data. *ArXiv*, abs/2002.03501, 2020.

[6] CamBoard pico flexx. `https://pmdtec.com/picofamily/flexx/`. Accessed: 2020-10-04.

[7] BlenSor: Blender Sensor Simulation. `https://www.blensor.org/`. Accessed: 2020-10-04.

[8] Princeton ModelNet. `http://modelnet.cs.princeton.edu/`. Accessed: 2020-16-03.

[9] 3D Models — Procedurally Generated Random Objects. `https://sites.google.com/site/brainrobotdata/home/models`. Accessed: 2020-16-03.

[10] DMU-Net Dataset: A research dataset of Engineering Computer-Aided Design (CAD) Models to enable Computer Vision and Deep Machine Learning Applications in the engineering and manufacturing industry. . `https://www.dmu-net.org/about.php`. Accessed: 2020-17-03.

[11] Blender OFF Addon. `https://github.com/alextsui05/blender-off-addon`. Accessed: 2020-22-03.

[12] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[13] PyTorch documentation. `https://pytorch.org/docs/stable/torch.html`. Accessed: 2020-14-03.

[14] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. *CoRR*, abs/1808.01974, 2018.

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[17] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[18] TORCHVISION.MODELS. `https://pytorch.org/docs/stable/torchvision/models.html`. Accessed: 2020-07-05.

[19] Ying He, Bin Liang, Yu Zou, Jin He, and Jun Yang. Depth errors analysis and correction for time-of-flight (tof) cameras. *Sensors*, 17(1):92, Jan 2017.