Implementation of 3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction using PyTorch

Daniil Zverev
zverev@in.tum.de

Baran Deniz Korkmaz
deniz.korkmaz@tum.de

Doğukan Kalkan
dogukan.kalkan@tum.de

Pascual Tejero Cervera
ge32fix@mytum.de

Abstract

Object reconstruction based on multiple images attracted huge interest from researchers all over the world. In particular, the scenario when the input set of images has an arbitrary size, unknown intrinsic/extrinsic camera parameters, and only partially covers the object. In this work, we considered and reimplemented 3D-R^2N^2 model, which was one of the first attempts to solve such a scenario and the model that gained a lot of attention due to its simplicity and power. For the project repository: `https://github.com/Bizilizi/3dr2n2`

1. Introduction

The problem of 3D object reconstruction has been popular in the domain of Machine Learning for 3D Geometry. 3D object reconstruction refers to the reconstruction of shape geometry based on given input representations such as volumetric occupancy grids, implicit representations, etc. The paper 3D-R2N2 proposes the use of multi-views for the reconstruction of shape geometry belonging to an object. The model architecture takes advantage of 3D Convolutional Recurrent Neural Networks for the processing of subsequent RGB images of an object in a recurrent fashion. In the original paper, the authors used the framework called Theano [10] for model implementation and the training was done using the dataset called ShapeNet [3]. The main contributions of our work are listed below:

- The architecture has been re-implemented in PyTorch [8].
- The dataset of 3D-Future [6] has been used in training in order to evaluate the performance on a different dataset.

2. Model Architecture

We use the architecture proposed in the original paper [5]. The main idea of the proposed architecture is to leverage the use of a recurrence module that enables the network to refine the 3D construction over newly seen views. The advantage of a recurrent module is that it enables the network to update the memory cells that are learned by the visible parts of the object fed over different views. The previously occluded or mismatching predictions can be updated as the network sees different views for a given object. The overall architecture is composed of three components: An encoder, a recurrence unit, and a decoder. The encoder extracts a low-dimensional feature embedding for a given 127x127 RGB image which is fed into the 3D-Recurrent Unit. Finally, the decoder takes the final hidden state of the recurrent unit and transforms it into a final volumetric occupancy grid.

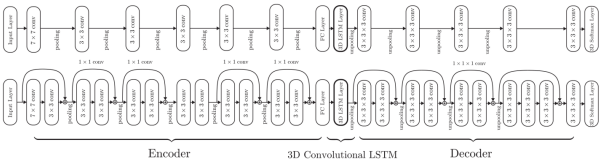


Figure 1. Network architecture.

2.1. Encoder: 2D-CNN

The encoder uses standard 2D convolution layers to encode RGB images into feature embeddings. The final output embedding for an image has a latent size of 1024. Both of the proposed variations of 2D-CNN encoders have been implemented. The first type is a standard feed-forward CNN which stacks convolution layers followed by pooling layers and leaky rectified linear unit activations. The second variation makes use of residual connections in between the standard convolution layers to improve the latent code generation. Both versions use a fully connected layer which

takes the flattened output of the encoder in order to generate a final 1024 dimensional feature embedding.

2.2. Recurrence: 3D Convolutional Recurrent Neural Network

The core part of the proposed architecture aims at refining the 3D reconstruction by recurrently using the information provided by different unseen views for an object. In that sense, the authors propose the use of Gated Recurrent Unit (GRU) [4] and Long Short-Term Memory (LSTM) [7] units as they can accumulate the information over different time steps by updating the memory cells. This module combines the last hidden state h_{t-1} of recurrent unit with the encoded feature embedding of the current step in order to update the hidden state h_t. Each cell of the recurrent unit independently has a fully connected layer and a 3D convolution layer. While the fully connected layer converts the encoder output into another feature vector of the same latent size as the hidden state, the 3D convolution layer processes the previous hidden state by preserving its size. Each cell output has been combined throughout the recurrent unit using the gating mechanisms in order to produce the current hidden state which is a 4D tensor of size 128x4x4x4. The figure below depicts the overall structure of the recurrent units.

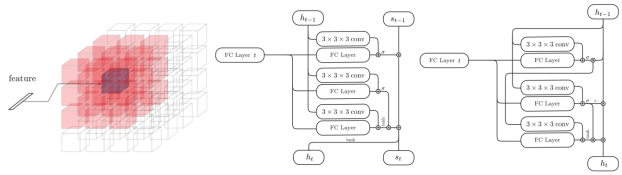


Figure 2. (On the left) Inputs for each recurrent unit. (In the middle) 3D Convolutional LSTM. (On the right) 3D Convolutional GRU

2.3. Decoder: 3D Deconvolutional Neural Network

After processing a sequence of RGB images for an object, the 3D Convolutional Recurrent Neural Network module passes the last hidden state to the decoder. The decoder unit applies 3D convolutions, unpooling operations and leaky rectified linear unit activations until the target output resolution has been reached. Similar to encoder, the decoder unit has two variations which are designed in similar fashion as in the encoder. 3D Softmax activation has been applied in order to convert the decoder output into 2x32x32x32 volumetric occupancy grids. The final output at each voxel (i, j, k) denotes the occupancy probability $p_{(i,j,k)}$.

2.4. Loss: 3D Voxel-wise Cross-Entropy Loss

The loss function is defined as the average of voxel-wise cross-entropy which can be formulated as below where the final output at each voxel (i, j, k) is designed to represent Bernoulli distributions [1-$p_{(i,j,k)}$,$p_{(i,j,k)}$].

$$L(\mathcal{X}, y) = \sum_{i,j,k} y_{(i,j,k)} log(p_{(i,j,k)}) + (1 - y_{(i,j,k)}) log(1 - p_{(i,j,k)}) \tag{1}$$

3. Implementation

In this block, we describe our implementation pipeline, configurations and all the other metrics necessary to reproduce our results.

Framework: In contrast to the original implementation [5] we used Pytorch [8] and Pytorch Lightning [1] to implement our model along with data modules. We saved all our metrics, reconstructions, model weights, and datasets with Weight and Biases [2].

Computational resources: We mainly used Google Colab with one Nvidia V-100 GPU per configuration training, that allowed us to train one model for 24 hours until early signs of overfitting.

Experiments: We run 8 different experiments each of them corresponding to one of the possible Encoder/Decoder configurations (Simple, Residual), intermediate layer for image feature fusion (LSTM/GRU) and kernel size for convolutions ($3 \times 3 \times 3$, $1 \times 1 \times 1$).

Optimization: For the optimization of all mentioned models we use the following configuration. We use 3D Voxel-Wise Cross-Entropy Loss. The learning rate is 1×10^{-10} and the optimizer is ADAM. We apply a weight decay of 5×10^{-5}. The batch size is 64 and we sample a random number of views up to 20 from random positions for each batch.

Random sampling: One of the crucial parts of the training pipeline, without which the model will not generalize or train at all, is random sampling number of views per each training batch as well as random sampling of positions of views for a given number of views. We exploit this both for training and validation procedures. Precisely, we train all the models on max 20 views, for each new batch, before constructing it, we sample a random number of views n within the region [1, 20] and then sample a random n position for each object.

4. Experiments

In this section, we present the details about our dataset and the performance of our models.

4.1. Dataset

For our experiments, we use 3D-Future [6] dataset. This choice was made due to the fact 3D-Future contains 20,240 photo-realistic synthetic objects captured in 5000 different scenes and 16,563 furniture shapes with high-quality texture. Unlike the other datasets, which contain poor textures and self-intersecting meshes, 3D-Future was well designed and contains rich color and shape information. It also provides additional attributes about objects such as styles and themes. However, they are irrelevant to our task formulation and were not used.

Figure 3. 3D-Future [6] shapes example

During our experiments, we had to work with limited resource constraints, so we decided to reduce the original size of the dataset to 3000 objects, which were uniformly sampled from the full dataset. This allowed us to run multiple experiments and reduce one train run to 24 hours.

4.1.1 Rendering

The model takes multiple images as input and produces a dense voxels grid with a probability of being occupied. To this end, we need to transform the dataset consisting of OBJ files and corresponding meshes into a set of renders and target voxels grids. In order to create renders, we create 50 view points uniformly around the object and render them through the PyTorch3D library [9]. Following the original architecture [5] rendered images have a size of 127×127 pixels. The training pipeline later uses those renders for random uniform sampling of object views, meaning whenever the data loader requests renders, we randomly sample a fixed number of views from those 50 images and return them as output. Here is an example of such sampling:

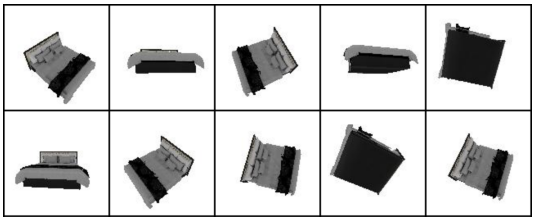


Figure 4. Sampled views of bed object. Views were randomly sampled from the set of 50 images.

For the creation of voxelized representation of an object, we used Open3D library [11], with a voxel grid size equal to 32. Open3D allows the creation of voxelized representation out of box, however, it works only with CPU, so we have to precalculate target representations in advance and store them for later use.

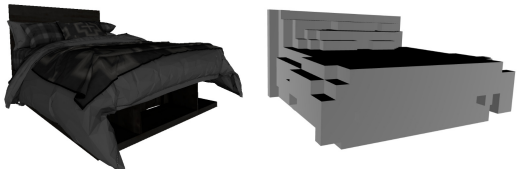


Figure 5. Original and voxel representation of bed object. The dense grid has a size of $32 \times 32 \times 32$ voxels.

4.2. Evaluation

For the evaluation, we choose two metrics: LogLoss and IoU. While the first one is our objective, the second one has a high correlation with the qualitative performance of the model. LogLoss was logged to Weight and Biases during the training time every 20 epochs, along with validation reconstructions. IoU in contrast was calculated after the model was trained based on saved checkpoints and uses the same setting with a random number of views per validation batch as during the training. It allowed us to build richer plots and investigate reconstructions in an interactive fashion. An example of such plots can be found here 6. You can find similar plots for all the configurations that we trained in the Appendix part.

4.3. Network Structures Comparison

This section demonstrates the performances obtained by different variations of network components as can be seen in Fig. 1. As we presented in section 2, the proposed architecture introduces 8 different variations in total which can be obtained by different types of encoder and decoder, recurrent unit, and the kernel size of the 3D convolution used in recurrent unit.

The results suggest that there is a large deviation in terms of IoU loss for training and test results. This means that, in most of the cases, our models overfitted the training data since we could reach into very high IoU levels of 84.3%. However, the models fail to generalize on the unseen test set. This problem can be resolved by increasing the size of the training data for better generalization regarding the fact that we use $\sim$ 20% of the entire 3D-Future dataset. We make similar observations as in the paper that the use of GRUs improves the performance. However, we see that we cannot fully leverage the power of residual encoder and decoder structures. It is highly likely that we have to further tune our training configuration for training those structures accurately.

			Ours				3D-R2N2	
Model Parameters			Training		Test		Test	
Encoder Decoder	3DConv RNN	Kernel	Loss	IoU	Loss	IoU	Loss	IoU
Simple	GRU	1	0.008	0.843	0.123	0.317	0.105	0.540
Simple	GRU	3	0.012	0.740	0.118	0.316	0.091	0.592
Simple	LSTM	1	0.027	0.482	0.067	0.277	0.116	0.499
Residual	GRU	1	0.016	0.692	**0.085**	**0.359**		
Residual	GRU	3	0.023	0.583	0.068	0.354	**0.080**	**0.634**
Residual	LSTM	1	0.048	0.197	0.059	0.294		
Residual	LSTM	3	0.038	0.310	0.054	0.245		

Table 1. Results for different architectural variations.

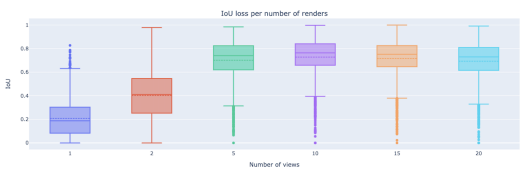
4.4. Multi-view Reconstruction Evaluation

In this section, we report a quantitative evaluation of our network's performance in multi-view reconstruction on the 3D-Future validation set.

Experiment setup: We performed 7 different configuration on validation dataset consisted of 600 objects. For each object we sample random number of views, following the training setup and then calculate IoU (Intersection over Union) with a voxelization threshold of 0.4 between the predicted and the ground truth voxel models.

Overall results: From the plots in Appendix 12, 11, 10, 9, 8, 7, 6, we can observe that the model successfully learns to reconstruct unobserved models, over the different classes. However we can also see that at some point LogLoss and IoU start to decorrelate, e.g. while LogLoss started to grow, IoU converged to a specific value. Our intuition and qualitative research shows the mode at the end tends to converge to predict boxed-like objects that are poorly performed on LogLoss, however, it allows to maintain IoU.

Number of views: Following the experiments in the original paper [cite] we performed quantitative and qualitative results for scenarios with a changing number of views for reconstruction.



We achieve similar model behavior when the reconstruction improves with the number of views. For a qualitative example look at the following plot 13

5. Challenges

Finding an Accurate Training Configuration: As there were many hyperparameters, it was not easy to obtain an appropriate configuration that would give a good result. The key feature is the randomization of the number of views and positions per render to train the model.

Extracting Useful Renders: Getting good renders that give meaningful information for the reconstruction of the 3D geometry is also challenging because of the whole underlying process of creation.

Training Multiple Model Variants: Training 8 different variations proposed for the network architecture was challenging due to the enormous required amount of time.

6. Conclusion

In our work, we successfully re-implemented the multi-view 3D reconstruction architecture called 3D-R2N2 using a different framework, PyTorch [8]. We trained all possible variations suggested by the network architecture on 3D-FUTURE [6] dataset in order to evaluate the performance of the model on different to ShapeNet [3] dataset. We were able to obtain satisfactory training results for some variations of the architecture. However, the models failed to generalize for the unseen test data, especially for some particular classes with thin parts. Our suggestion is that the bottleneck was using a smaller subset of the entire dataset due to limited computational resources.

References

[1] Pytorch lightning framework. `https://pytorchlightning.ai`.

[2] Weights and biases. `https://wandb.ai/site`.

[3] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository, 2015. cite arxiv:1512.03012.

[4] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[5] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[6] Huan Fu, Rongfei Jia, Lin Gao, Mingming Gong, Binqiang Zhao, Steve Maybank, and Dacheng Tao. 3d-future: 3d furniture shape with texture. *International Journal of Computer Vision*, pages 1–25, 2021.

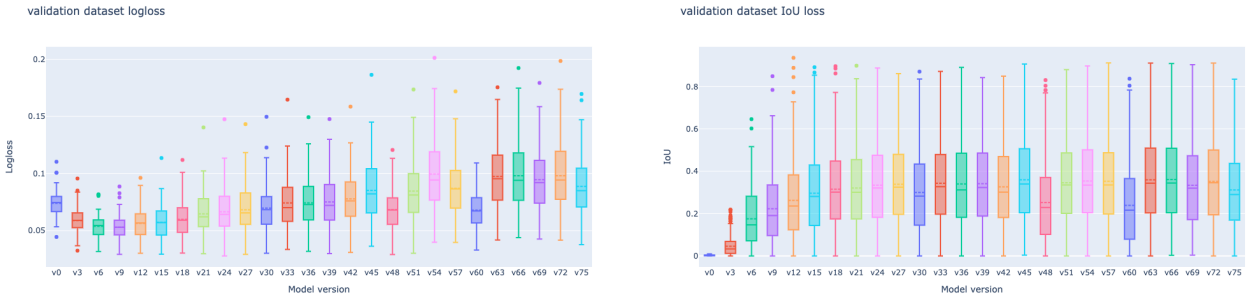
[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020.

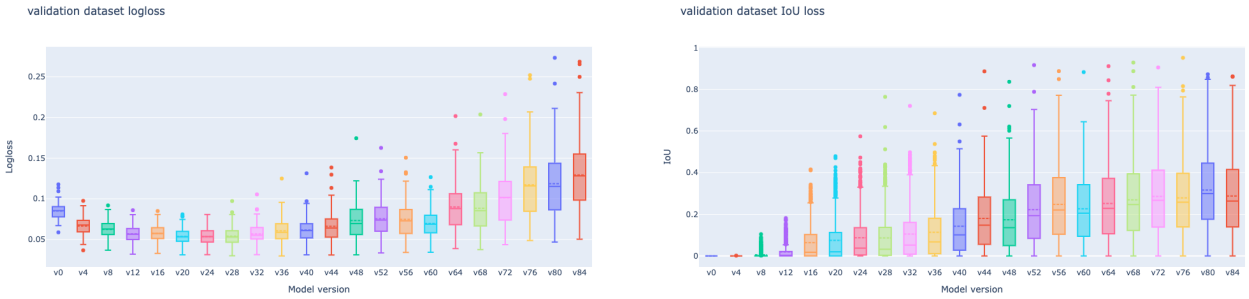
[10] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[11] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.



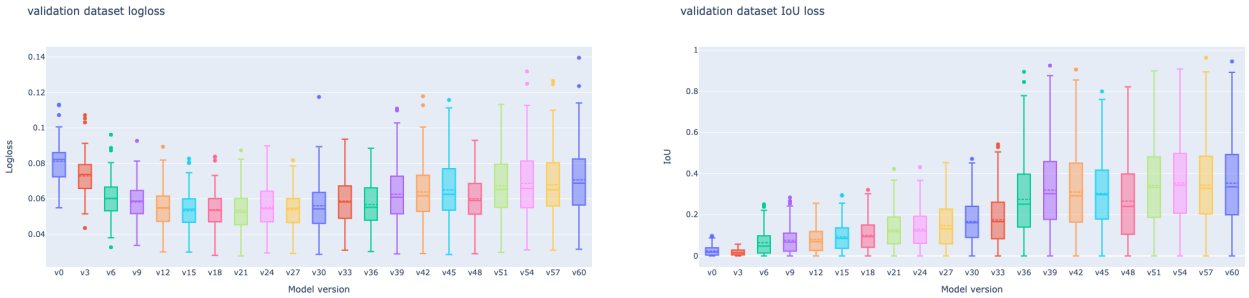
(a) Log loss (b) IoU loss

Figure 6. Validation losses performed for model with residual encoder/decoder backbone and GRU with kernel size 1



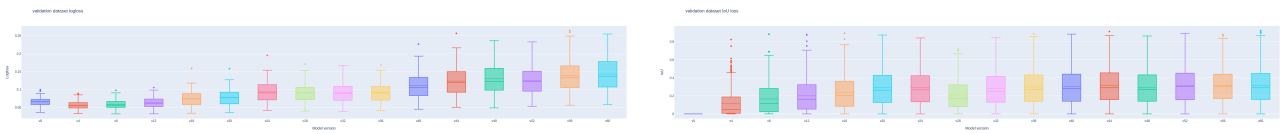
(a) Log loss (b) IoU loss

Figure 7. Validation losses performed for model with simple encoder/decoder backbone and GRU with kernel size 3



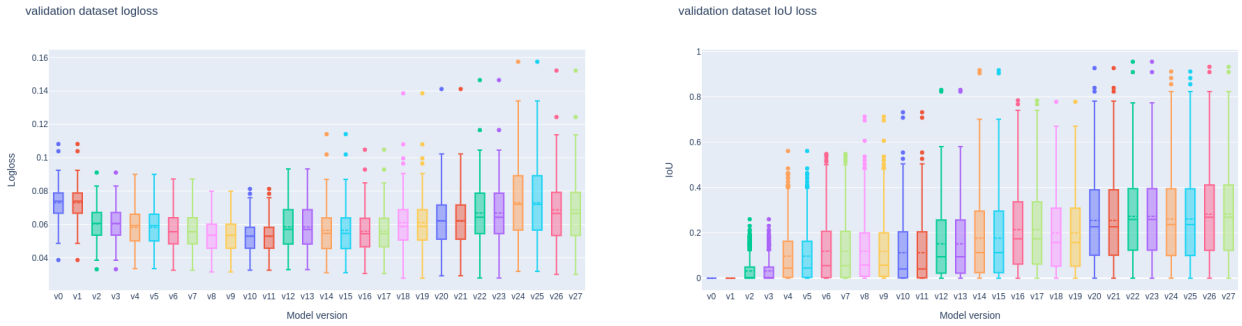
(a) Log loss (b) IoU loss

Figure 8. Validation losses performed for model with residual encoder/decoder backbone and GRU with kernel size 3



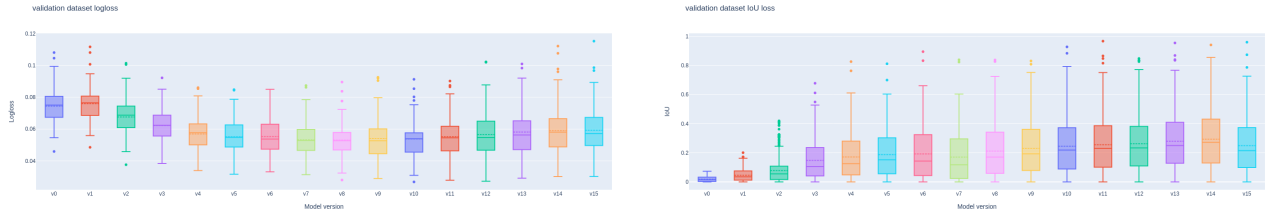
(a) Log loss (b) IoU loss

Figure 9. Validation losses performed for model with simple encoder/decoder backbone and GRU with kernel size 1



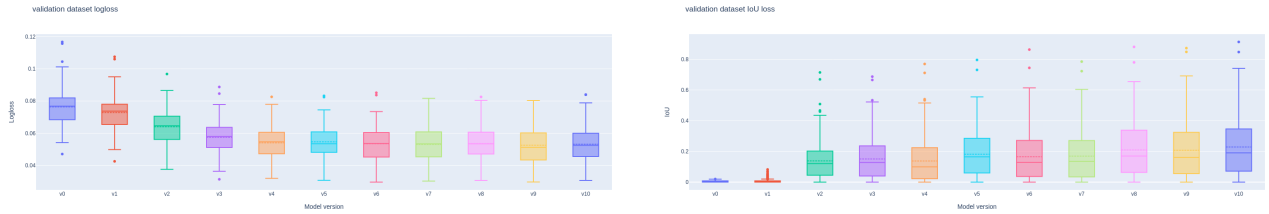
(a) Log loss (b) IoU loss

Figure 10. Validation losses performed for model with simple encoder/decoder backbone and LSTM with kernel size 1



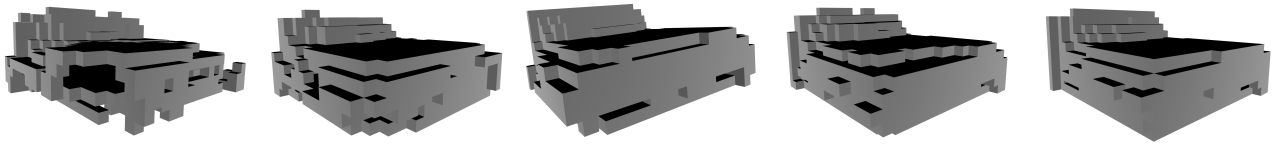
(a) Log loss (b) IoU loss

Figure 11. Validation losses performed for model with residual encoder/decoder backbone and LSTM with kernel size 3



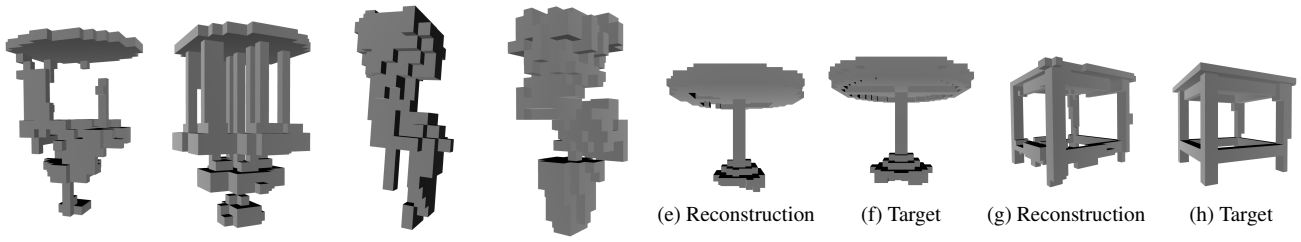
(a) Log loss (b) IoU loss

Figure 12. Validation losses performed for model with residual encoder/decoder backbone and LSTM with kernel size 1



(a) 3 views (b) 5 views (c) 10 views (d) 15 views (e) 20 views

Figure 13. Reconstruction improvements as models observes more views



(a) Reconstruction (b) Target (c) Reconstruction (d) Target (e) Reconstruction (f) Target (g) Reconstruction (h) Target

Figure 14. Examples of Reconstructions from the Training Set

(a) Reconstruction (b) Target (c) Reconstruction (d) Target (e) Reconstruction (f) Target

Figure 15. Examples of Reconstructions from the Validation Set