Supplementary Material for OctNet: Learning Deep 3D Representations at High Resolutions

Gernot Riegler[1] Ali Osman Ulusoy[2] Andreas Geiger[2,3]
[1]Institute for Computer Graphics and Vision, Graz University of Technology
[2]Autonomous Vision Group, MPI for Intelligent Systems Tübingen
[3]Computer Vision and Geometry Group, ETH Zürich
riegler@icg.tugraz.at {osman.ulusoy,andreas.geiger}@tue.mpg.de

1. Introduction

In the following supplemental material we present details regarding our operations on the hybrid grid-octree data structure, additional experimental results and all details on the used network architectures. In Section 2 we provide details on the data index used to efficiently access data in the grid-octree data structure. Section 3 yields more insights into the efficient implementation of the convolution operation on octrees. We present further quantitative and qualitative results in Section 4 and in Section 5 we specify all details of the network architectures used in our experiments.

2. Data Index

An important implementation detail of the hybrid grid-octree data structure is the memory alignment for fast data access. We store all data associated with the leaf nodes of a shallow octree in a compact contiguous array. Thus, we need a fast way to compute the offset in this data array for any given voxel. In the main text we presented the following equation:

$$\text{data\_idx}(i) = \underbrace{8 \sum_{j=0}^{\text{pa}(i)-1} \text{bit}(j) + 1}_{\text{\#nodes above i}} - \underbrace{\sum_{j=0}^{i-1} \text{bit}(j)}_{\text{\#split nodes pre i}} + \underbrace{\text{mod}\,(i-1, 8)}_{\text{offset}} . \tag{1}$$

As explained in the main text the whole octree structure is stored as a bit-string and a voxel is uniquely identified by the bit index i, i.e., the index within the bit string. The data is aligned breadth-first and only the leaf nodes have data associated. Consequently, the first part of the equation above counts the number of split and leaf nodes up to the voxel with bit index i. The second term subtracts the number of split nodes before the particular voxel as data is only associated with leaf nodes. Finally, we need to get the offset within the voxel's neighborhood. This is done by the last term of the equation.

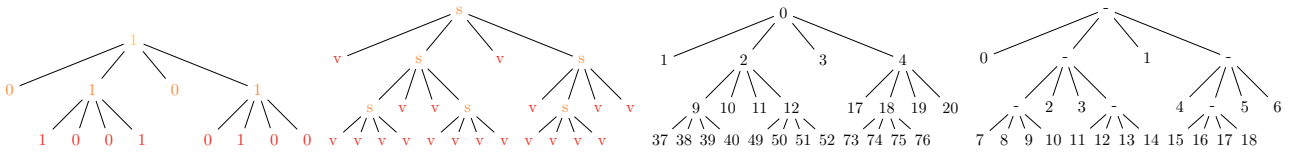
Let us illustrate this with a simple example: For ease of visualization we will consider a quadtree. Hence, each voxel can be split into 4 instead of 8 children. The equation for the offset changes to

$$\text{data\_idx}_4(i) = \underbrace{4 \sum_{j=0}^{\text{pa}_4(i)-1} \text{bit}(j) + 1}_{\text{\#nodes above i}} - \underbrace{\sum_{j=0}^{i-1} \text{bit}(j)}_{\text{\#split nodes pre i}} + \underbrace{\text{mod}\,(i-1, 4)}_{\text{offset}} , \tag{2}$$

with

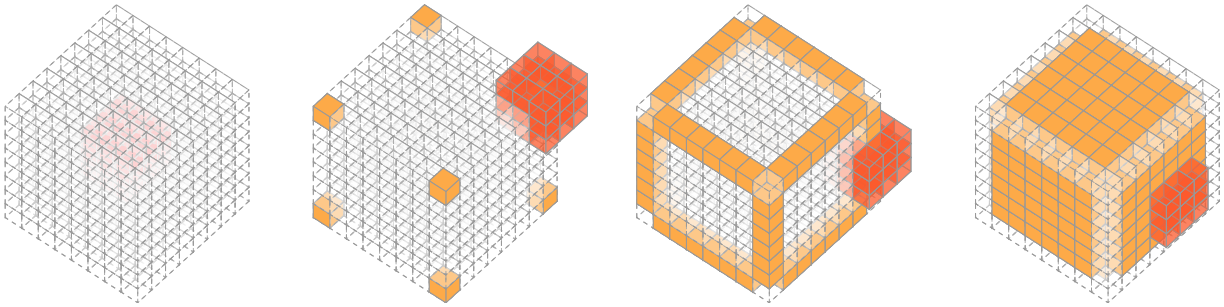
$$\text{pa}_4(i) = \left\lfloor \frac{i-1}{4} \right\rfloor . \tag{3}$$

Now consider the following bit string for instance: 1 0101 0000 1001 0000 0100. According to our definition, this bit string corresponds to the tree structure visualized in Fig. 1a and 1b, where *s* indicates a split node and *v* a leaf node with associated



(a) Bit-String (b) Split and Leaf Nodes (c) Bit Index (d) Data Index

Figure 1: **Data Index.**



(a) Constant (b) Corners (c) Edges (d) Faces

Figure 2: **Efficient Convolution.**

data. In Fig. 1c we show the bit indices for all nodes. Note that the leaf nodes at depth 3 do not need to be stored in the bit string as this information is implicit. Finally, the data index for all leaf nodes is visualized in Fig. 1d. Now we can verify equation (2) using a simple example. Assume the bit index 51: The parent bit index is given by equation (3) as 12. To compute the data index we first count the number of nodes before 49 as it is the first node within its siblings (first term of equation), which is 17. Next, we count the number of split nodes up to 49 (second term of equation), which is 6. Finally, we look up the position 51 within its siblings (last term of equation), which is 2. Combining those three terms yields the data index $17 - 6 + 2 = 13$.

3. Efficient Convolution

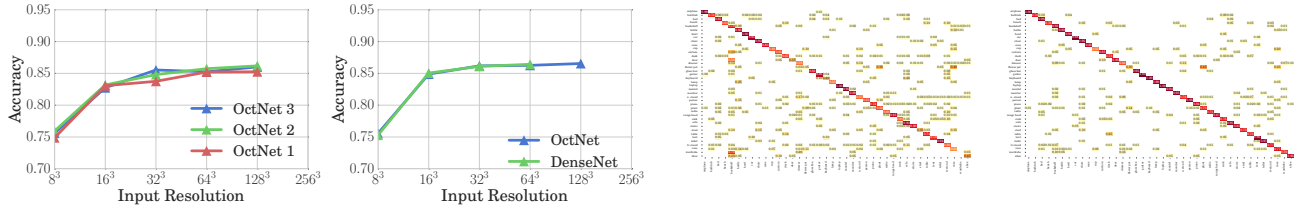
In the main text we discussed that the convolution for larger octree cells and small convolution kernels can be efficiently implemented. A naïve implementation applies the convolution kernel at every location (i, j, k) comprised by the cell $\Omega[i, j, k]$. Therefore, for an octree cell of size 8^3 and a convolution kernel kernel of 3^3 this would require $8^3 \cdot 3^3 = 13,824$ multiplications. However, we can implement this calculation much more efficiently as depicted in Fig. 2. We observe that the value inside the cell of size 8^3 is constant. Thus, we only need to evaluate the convolution once inside this cell and multiply the result with the size of the cell 8^3, see Fig. 2a. Additionally, we only need to evaluate a truncated versions of the kernel on the corners, edges and faces of the voxel, see Fig. 2b-d. This implementation is more efficient, as we need only 27 multiplications for the constant part, $8 \cdot 19$ multiplications for the corners, $12 \cdot 6 \cdot 15$ multiplications for the edges, and $6 \cdot 6^2 \cdot 9$ multiplications for the faces of the voxel. In total this yields 3203 multiplications, or 23.17% of the multiplications required by the naïve implementation.

4. Additional Results

In this Section we show additional quantitative and qualitative results for 3D shape classification, 3D orientation estimation and semantic 3D point labeling.

4.1. 3D Classification

In the main text of our work we analyzed the runtime and memory consumption of OctNet compared with the equivalent dense networks on ModelNet10 [4]. Additionally, we demonstrated that without further data augmentation, ensemble learning, or more sophisticated architectures the accuracy saturates at an input resolution of about 16^3, when keeping the number of network parameters fixed across all resolutions. In this Section we show the same experiment on ModelNet40 [4]. The



(a) Accuracy (b) Accuracy (c) Confusion Matrix 8^3 (d) Confusion Matrix 32^3

Figure 3: **ModelNet40 results.**

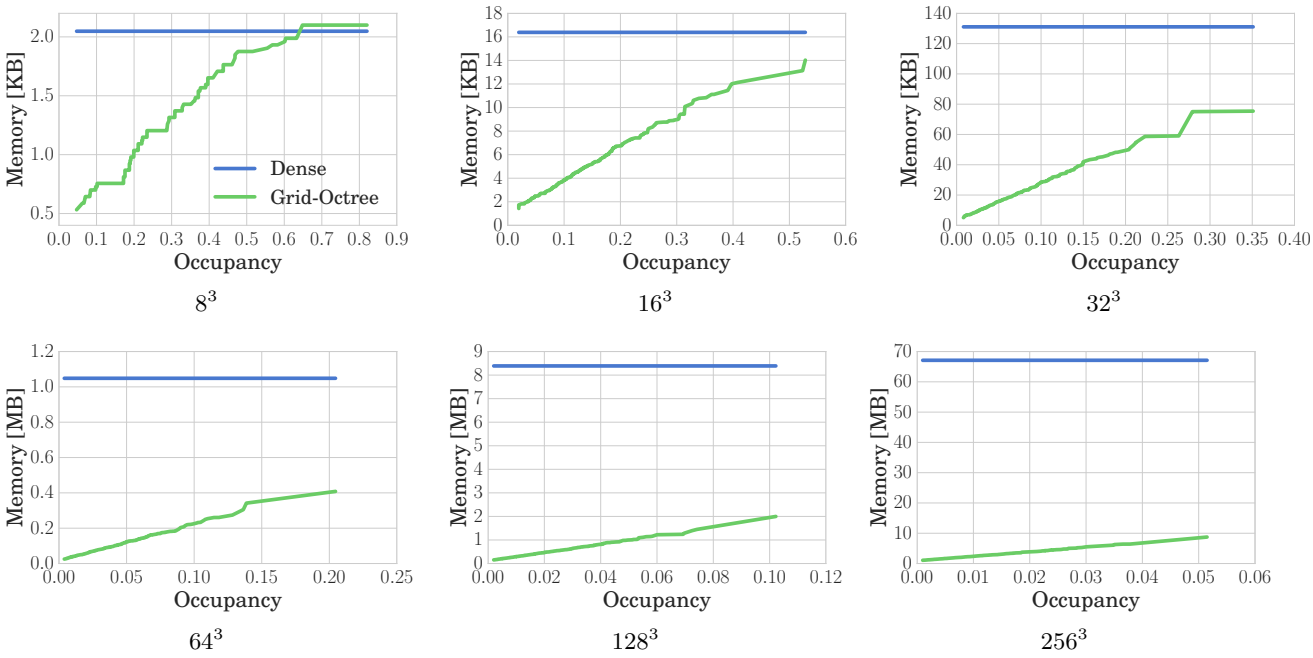


Figure 4: **Memory consumption vs. occupancy.**

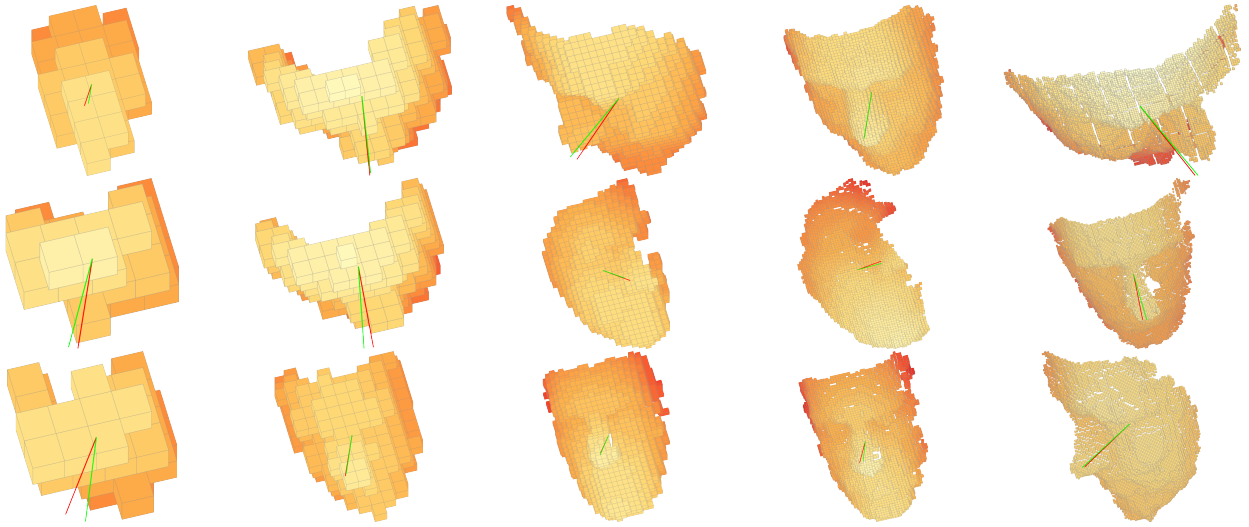
results are summarized in Fig. 3. In contrast to ModelNet10, we see an increase in accuracy up to an input resolution of 32^3. Beyond this resolution the classification performance does not further improve. Note that the only form of data augmentation we used in this experiment was rotation around the up-vector as the 3D models in this dataset vary in pose. We conclude that object classification on the ModelNet40 dataset is more challenging than on the ModelNet10 dataset, but both datasets are relatively easy in the sense that details do not matter as much as in the datasets used for our other experiments.

We further use this experiment to investigate the question at which level of sparsity OctNet becomes useful. To answer this, we visualize the memory consumption of a dense representation vs. our data structure at different resolutions and occupancies by sampling 500 shapes from the ModelNet40 dataset (see Fig. 4). It can be observed that even at very low resolutions (8^3), our data structure is superior compared to the dense representation, even up to an occupancy level of 50%. As the voxel resolution increases, the occupancy levels (x-axis) decreases since the data becomes sparser. Our OctNet exploits this sparsity to achieve a significant reduction in memory consumption.

4.2. 3D Orientation Estimation

To demonstrate that OctNet can handle input resolutions larger than 256^3 we added results for the 3D orientation estimation experiment with an input resolution of 512^3. The results are presented in Fig. 5. As for the orientation experiment in the main paper, we can observe the trend that performance increases with increasing input resolution.

We evaluated our OctNet also on the Biwi Kinect Head Pose Database [2] as an additional experiment on 3D pose



(a) 8^3 (b) 16^3 (c) 32^3 (d) 64^3 (e) 128^3

Figure 7: **Qualitative Results for Head Pose Estimation.**

estimation. The dataset consists of 24 sequences of 20 individuals sitting in front of a Kinect depth sensor. For each frame the head center and the head pose in terms of its 3D rotation is annotated. We split the dataset into a training set of 18 individuals for training and 2 individuals for testing and project the depth map to 3D points with the given camera parameters. We then create the hybrid grid-octree structure from the 3D points that belong to the head (In this experiment we are only interested in 3D orientation estimation, as the head can be reliable detected in the color images). As in the previous experiment we parameterize the orientation with unit quaternions and train our OctNet using the same settings as in the previous 3D orientation estimation experiment. Fig. 6 shows the quantitative results over varying input resolutions. We see a reasonable improvement of accuracy from 8^3 up to 64^3. Beyond this input resolution the octree resolution becomes finer than the resolution of the 3D point cloud. Thus, further improvements can not be expected. In Fig. 7 we show some qualitative results.

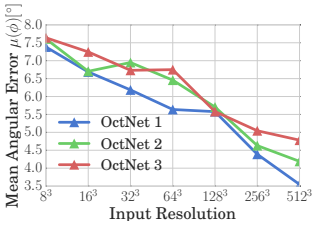


Figure 5: **Additional Chair Orientation Results.**

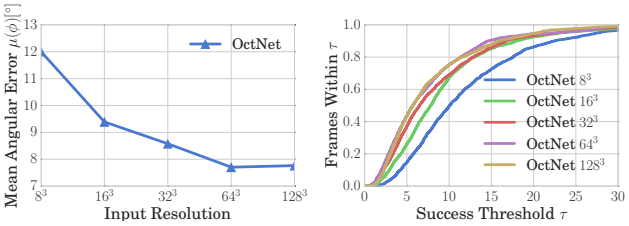


Figure 6: **Head Pose Results.**

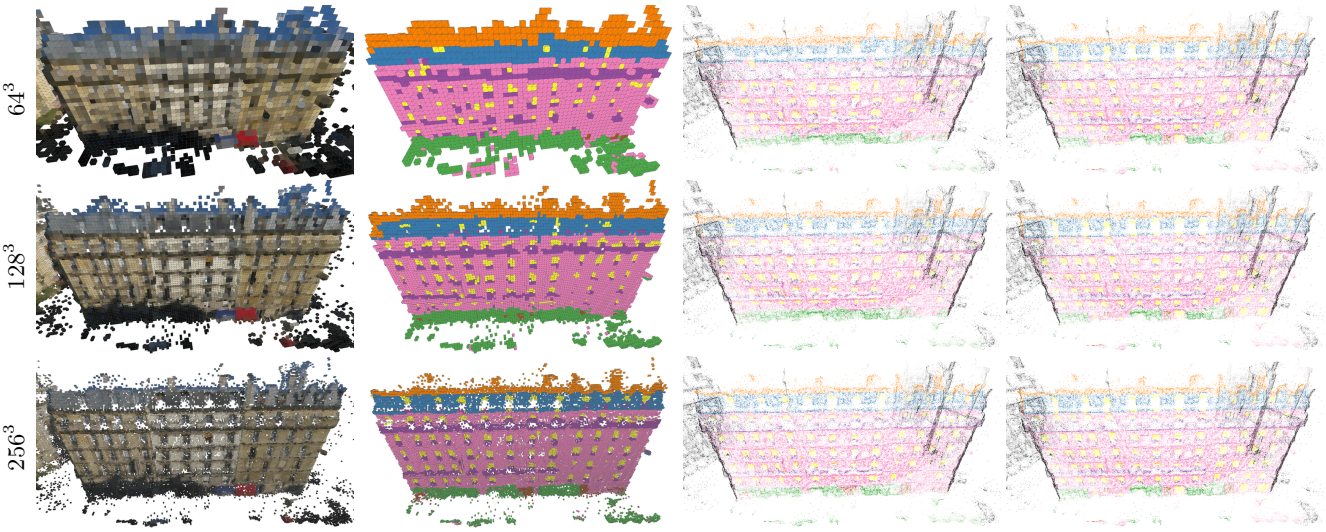
4.3. 3D Semantic Segmentation

In this Section we present additional qualitative results of the semantic 3D point labeling task in Fig. 8 to 10. In these visualizations we show the color part of the voxelized input, the result of the labeling in the voxel representation, and the result back-projected to the 3D point cloud for different houses in the test set of [3].

5. Network Architecture Details

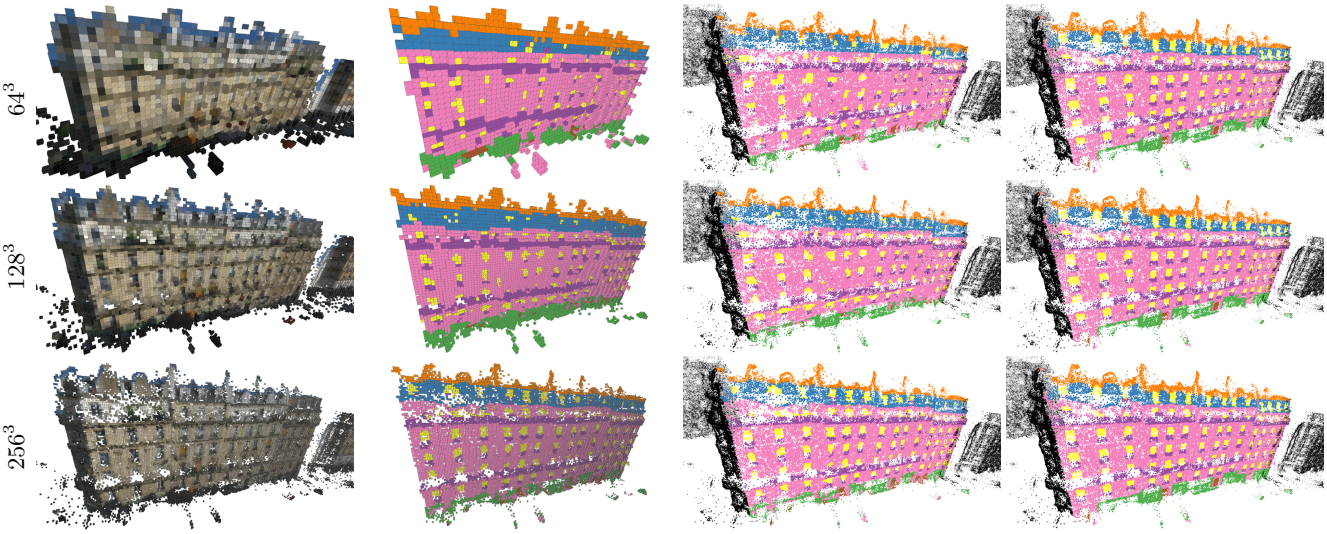
In this Section we detail the network architectures used throughout our experimental evaluations. We use the following notation for brevity: conv(x, y) denotes a 3^3 convolutional layer with x input feature maps and y output feature maps. Similarly, maxpool(f) is a max-pooling operation that decreases dimensionality by a factor of f along each axis. All convolutional and fully-connected layers, except the very last one, are followed by a ReLU as activation function.

In the first two experiments, 3D classification and 3D orientation estimation, we show two different classes of architectures.



(a) Voxelized Input (b) Voxel Estimates (c) Estimated Point Cloud (d) Ground Truth Point Cloud

Figure 8: **Facade Labeling Results.** Zoom in for details.

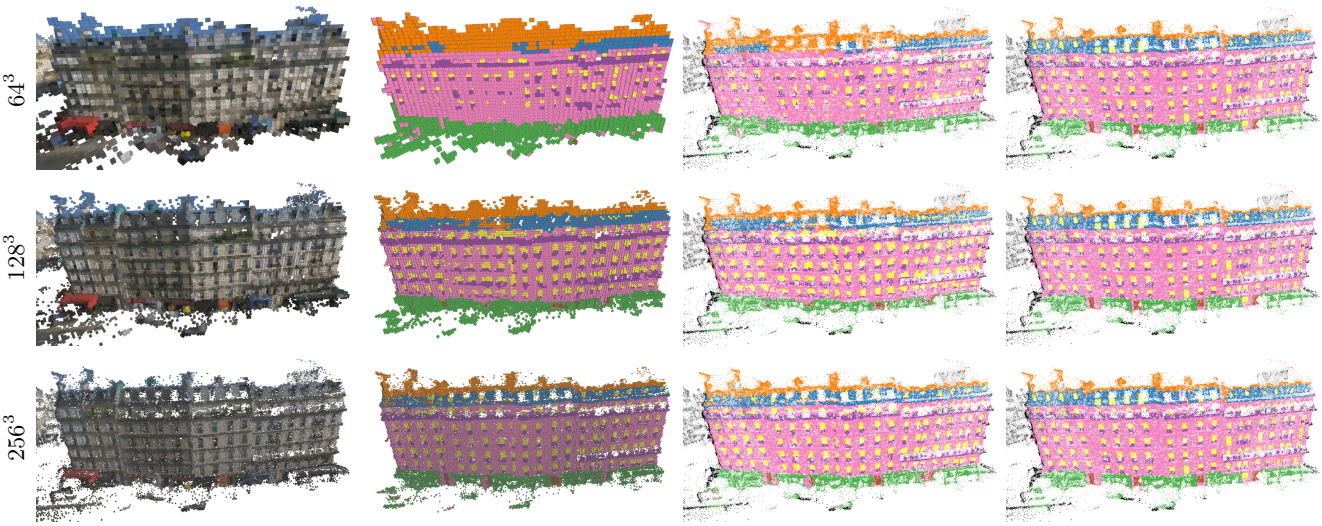


(a) Voxelized Input (b) Voxel Estimates (c) Estimated Point Cloud (d) Ground Truth Point Cloud

Figure 9: **Facade Labeling Results.** Zoom in for details.

In the first one we keep the number of convolution layers per block fixed and add blocks depending on the input resolution of the network. We call those networks OctNet1, OctNet2, and OctNet3, depending on the number of convolution layers per block. Therefore, the number of parameters increases along with the input resolution. The detailed architectures are depicted in Table 1, 2, and 3 for the classification task and in Table 5, 6, and 7 for the orientation estimation tasks, respectively. Second, we trained network architectures where we keep the number of parameters fixed, independently of the input resolution. The detailed network architectures for those experiments are presented in Table 4 and 8.

Finally, for semantic 3D point labeling, we use the U-Net type architecture [1,5] shown in Table 9. We use a concatenation layer concat$(\cdot,\cdot)$ to combine the outputs from the decoder and encoder parts of the networks to preserve details.



(a) Voxelized Input (b) Voxel Estimates (c) Estimated Point Cloud (d) Ground Truth Point Cloud

Figure 10: **Facade Labeling Results.** Zoom in for details.

$\mathbf{8^3}$	$\mathbf{16^3}$	$\mathbf{32^3}$	$\mathbf{64^3}$	$\mathbf{128^3}$	$\mathbf{256^3}$
conv$(1, 8)$	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)
	conv$(8, 16)$	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)
		conv$(16, 24)$	conv$(16, 24)$ maxpool(2)	conv$(16, 24)$ maxpool(2)	conv$(16, 24)$ maxpool(2)
			conv$(24, 32)$	conv$(24, 32)$ maxpool(2)	conv$(24, 32)$ maxpool(2)
				conv$(32, 40)$	conv$(32, 40)$ maxpool(2)
					conv$(40, 48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(10)					
SoftMax					

Table 1: **Network Architectures ModelNet10 Classification: OctNet1**

8^3	16^3	32^3	64^3	128^3	256^3
conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$
conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$
	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$
	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$
		maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
		conv$(16,24)$	conv$(16,24)$	conv$(16,24)$	conv$(16,24)$
		conv$(24,24)$	conv$(24,24)$	conv$(24,24)$	conv$(24,24)$
			maxpool(2)	maxpool(2)	maxpool(2)
			conv$(24,32)$	conv$(24,32)$	conv$(24,32)$
			conv$(32,32)$	conv$(32,32)$	conv$(32,32)$
				maxpool(2)	maxpool(2)
				conv$(32,40)$	conv$(32,40)$
				conv$(40,40)$	conv$(40,40)$
					maxpool(2)
					conv$(40,48)$
					conv$(48,48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(10)					
SoftMax					

Table 2: **Network Architectures ModelNet10 Classification: OctNet2**

8^3	16^3	32^3	64^3	128^3	256^3
conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$	conv$(1,8)$
conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$
conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$	conv$(8,8)$
	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$	conv$(8,16)$
	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$
	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$	conv$(16,16)$
		maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
		conv$(16,24)$	conv$(16,24)$	conv$(16,24)$	conv$(16,24)$
		conv$(24,24)$	conv$(24,24)$	conv$(24,24)$	conv$(24,24)$
		conv$(24,24)$	conv$(24,24)$	conv$(24,24)$	conv$(24,24)$
			maxpool(2)	maxpool(2)	maxpool(2)
			conv$(24,32)$	conv$(24,32)$	conv$(24,32)$
			conv$(32,32)$	conv$(32,32)$	conv$(32,32)$
			conv$(32,32)$	conv$(32,32)$	conv$(32,32)$
				maxpool(2)	maxpool(2)
				conv$(32,40)$	conv$(32,40)$
				conv$(40,40)$	conv$(40,40)$
				conv$(40,40)$	conv$(40,40)$
					maxpool(2)
					conv$(40,48)$
					conv$(48,48)$
					conv$(48,48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(10)					
SoftMax					

Table 3: **Network Architectures ModelNet10 Classification: OctNet3.**

$\mathbf{8^3}$	$\mathbf{16^3}$	$\mathbf{32^3}$	$\mathbf{64^3}$	$\mathbf{128^3}$	$\mathbf{256^3}$
conv$(1, 8)$ conv$(8, 14)$	conv$(1, 8)$ conv$(8, 14)$	conv$(1, 8)$ conv$(8, 14)$	conv$(1, 8)$ conv$(8, 14)$	conv$(1, 8)$ conv$(8, 14)$	conv$(1, 8)$ conv$(8, 14)$ maxpool(2)
conv$(14, 14)$ conv$(14, 20)$	conv$(14, 14)$ conv$(14, 20)$	conv$(14, 14)$ conv$(14, 20)$	conv$(14, 14)$ conv$(14, 20)$	conv$(14, 14)$ conv$(14, 20)$ maxpool(2)	conv$(14, 14)$ conv$(14, 20)$ maxpool(2)
conv$(20, 20)$ conv$(20, 26)$	conv$(20, 20)$ conv$(20, 26)$	conv$(20, 20)$ conv$(20, 26)$	conv$(20, 20)$ conv$(20, 26)$ maxpool(2)	conv$(20, 20)$ conv$(20, 26)$ maxpool(2)	conv$(20, 20)$ conv$(20, 26)$ maxpool(2)
conv$(26, 26)$ conv$(26, 32)$	conv$(26, 26)$ conv$(26, 32)$	conv$(26, 26)$ conv$(26, 32)$ maxpool(2)	conv$(26, 26)$ conv$(26, 32)$ maxpool(2)	conv$(26, 26)$ conv$(26, 32)$ maxpool(2)	conv$(26, 26)$ conv$(26, 32)$ maxpool(2)
conv$(32, 32)$ conv$(32, 32)$	conv$(32, 32)$ conv$(32, 32)$ maxpool(2)	conv$(32, 32)$ conv$(32, 32)$ maxpool(2)	conv$(32, 32)$ conv$(32, 32)$ maxpool(2)	conv$(32, 32)$ conv$(32, 32)$ maxpool(2)	conv$(32, 32)$ conv$(32, 32)$ maxpool(2)
Dropout(0.5)					
fully-connected(512)					
fully-connected(10)					
SoftMax					

Table 4: **Network Architectures ModelNet10 Classification.**

$\mathbf{8^3}$	$\mathbf{16^3}$	$\mathbf{32^3}$	$\mathbf{64^3}$	$\mathbf{128^3}$	$\mathbf{256^3}$
conv$(1, 8)$	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)	conv$(1, 8)$ maxpool(2)
	conv$(8, 16)$	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)	conv$(8, 16)$ maxpool(2)
		conv$(16, 24)$	conv$(16, 24)$ maxpool(2)	conv$(16, 24)$ maxpool(2)	conv$(16, 24)$ maxpool(2)
			conv$(24, 32)$	conv$(24, 32)$ maxpool(2)	conv$(24, 32)$ maxpool(2)
				conv$(32, 40)$	conv$(32, 40)$ maxpool(2)
					conv$(40, 48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(4)					
Normalize					

Table 5: **Network Architectures Orientation Estimation: OctNet1**

8^3	16^3	32^3	64^3	128^3	256^3
conv$(1,8)$ conv$(8,8)$	conv$(1,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ maxpool(2)
	conv$(8,16)$ conv$(16,16)$	conv$(8,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ maxpool(2)
		conv$(16,24)$ conv$(24,24)$	conv$(16,24)$ conv$(24,24)$ maxpool(2)	conv$(16,24)$ conv$(24,24)$ maxpool(2)	conv$(16,24)$ conv$(24,24)$ maxpool(2)
			conv$(24,32)$ conv$(32,32)$	conv$(24,32)$ conv$(32,32)$ maxpool(2)	conv$(24,32)$ conv$(32,32)$ maxpool(2)
				conv$(32,40)$ conv$(40,40)$	conv$(32,40)$ conv$(40,40)$ maxpool(2)
					conv$(40,48)$ conv$(48,48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(4)					
Normalize					

Table 6: **Network Architectures Orientation Estimation: OctNet2**

8^3	16^3	32^3	64^3	128^3	256^3
conv$(1,8)$ conv$(8,8)$ conv$(8,8)$	conv$(1,8)$ conv$(8,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ conv$(8,8)$ maxpool(2)	conv$(1,8)$ conv$(8,8)$ conv$(8,8)$ maxpool(2)
	conv$(8,16)$ conv$(16,16)$ conv$(16,16)$	conv$(8,16)$ conv$(16,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ conv$(16,16)$ maxpool(2)	conv$(8,16)$ conv$(16,16)$ conv$(16,16)$ maxpool(2)
		conv$(16,24)$ conv$(24,24)$ conv$(24,24)$	conv$(16,24)$ conv$(24,24)$ conv$(24,24)$ maxpool(2)	conv$(16,24)$ conv$(24,24)$ conv$(24,24)$ maxpool(2)	conv$(16,24)$ conv$(24,24)$ conv$(24,24)$ maxpool(2)
			conv$(24,32)$ conv$(32,32)$ conv$(32,32)$	conv$(24,32)$ conv$(32,32)$ conv$(32,32)$ maxpool(2)	conv$(24,32)$ conv$(32,32)$ conv$(32,32)$ maxpool(2)
				conv$(32,40)$ conv$(40,40)$ conv$(40,40)$	conv$(32,40)$ conv$(40,40)$ conv$(40,40)$ maxpool(2)
					conv$(40,48)$ conv$(48,48)$ conv$(48,48)$
Dropout(0.5)					
fully-connected(1024)					
fully-connected(4)					
Normalize					

Table 7: **Network Architectures Orientation Estimation: OctNet3.**

$\mathbf{8^3}$	$\mathbf{16^3}$	$\mathbf{32^3}$	$\mathbf{64^3}$	$\mathbf{128^3}$	$\mathbf{256^3}$
conv$(1, 8)$	conv$(1, 8)$	conv$(1, 8)$	conv$(1, 8)$	conv$(1, 8)$	conv$(1, 8)$
conv$(8, 14)$	conv$(8, 14)$	conv$(8, 14)$	conv$(8, 14)$	conv$(8, 14)$	conv$(8, 14)$
					maxpool(2)
conv$(14, 14)$	conv$(14, 14)$	conv$(14, 14)$	conv$(14, 14)$	conv$(14, 14)$	conv$(14, 14)$
conv$(14, 20)$	conv$(14, 20)$	conv$(14, 20)$	conv$(14, 20)$	conv$(14, 20)$	conv$(14, 20)$
				maxpool(2)	maxpool(2)
conv$(20, 20)$	conv$(20, 20)$	conv$(20, 20)$	conv$(20, 20)$	conv$(20, 20)$	conv$(20, 20)$
conv$(20, 26)$	conv$(20, 26)$	conv$(20, 26)$	conv$(20, 26)$	conv$(20, 26)$	conv$(20, 26)$
			maxpool(2)	maxpool(2)	maxpool(2)
conv$(26, 26)$	conv$(26, 26)$	conv$(26, 26)$	conv$(26, 26)$	conv$(26, 26)$	conv$(26, 26)$
conv$(26, 32)$	conv$(26, 32)$	conv$(26, 32)$	conv$(26, 32)$	conv$(26, 32)$	conv$(26, 32)$
		maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$
conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$	conv$(32, 32)$
	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)	maxpool(2)
Dropout(0.5)					
fully-connected(512)					
fully-connected(4)					
Normalize					

Table 8: **Network Architectures Orientation Estimation.**

Output name	Operation
Enc1	conv$(8, 8)$ conv$(8, 16)$ maxpool(2)
Enc2	conv$(16, 16)$ conv$(16, 32)$ maxpool(2)
Enc3	conv$(32, 32)$ conv$(32, 64)$ maxpool(2)
Enc4	conv$(64, 64)$ conv$(64, 128)$ maxpool(2)
Dec4	conv$(128, 128)$ conv$(128, 128)$ conv$(128, 128)$ unpool(2)
Dec3	concat(Enc4,Dec4) conv$(256, 128)$ conv$(128, 64)$ unpool(2)
Dec2	concat(Enc3,Dec3) conv$(128, 64)$ conv$(64, 32)$ unpool(2)
Dec1	concat(Enc2,Dec2) conv$(64, 32)$ conv$(32, 16)$ unpool(2)
Output	concat(Enc1,Dec1) conv$(32, 32)$ conv$(32, 8)$ SoftMax

Table 9: **Network Architecture Semantic 3D Point Cloud Labeling.**

References

[1] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv.org*, 1511.00561, 2015. 5

[2] G. Fanelli, J. Gall, and L. Van Gool. Real time head pose estimation with random regression forests. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2011. 3

[3] H. Riemenschneider, A. Bódis-Szomorú, J. Weissenberg, and L. V. Gool. Learning where to classify in multi-view semantic segmentation. In *Proc. of the European Conf. on Computer Vision (ECCV)*, 2014. 4

[4] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015. 2

[5] Özgün Cicek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger. 3d u-net: Learning dense volumetric segmentation from sparse annotation. *arXiv.org*, 1606.06650, 2016. 5