Generating and parsing shapes and scenes

D.A. Forsyth, UIUC

Key questions:

- Shapes/scenes are (might be?) composites of primitives
- Generate
  - individual representations
  - composite representations
- Parse into
  - individual representations
  - composite representations
- Couple parser and generator, and learn both
  - ideally, without intermediate annotations

Generating point clouds

- Can construct a feature that represents a point cloud
  - above:
    - embed each point in some hd space,
    - max pool to single feature vector
    - map that
- Why not use to build autoencoder, GAN, etc?
  - Works, but how to evaluate?
  - Standard metrics:
    - EMD
    - Chamfer distance
  - Others

EMD and Chamfer distance

Metrics Two permutation-invariant metrics for comparing unordered point sets have been proposed in the literature (Fan et al., 2016). On the one hand, the *Earth Mover's* distance (EMD) (Rubner et al., 2000) is the solution of a transportation problem which attempts to transform one set to the other. For two equally sized subsets $S_1 \subseteq R^3, S_2 \subseteq R^3$, their EMD is defined by

$$d_{EMD}(S_1, S_2) = \min_{\phi: S_1 \to S_2} \sum_{x \in S_1} \|x - \phi(x)\|_2$$

where ϕ is a bijection. As a loss, EMD is differentiable almost everywhere. On the other hand, the *Chamfer* (pseudo)-distance (CD) measures the squared distance between each point in one set to its nearest neighbor in the other set:

$$d_{CH}(S_1, S_2) = \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2^2 + \sum_{y \in S_2} \min_{x \in S_1} \|x - y\|_2^2.$$

Achlioptas ea 18

You can get EMD by solving a linear program, but it's nasty: see, for example, https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/RUBNER/emd.htm

Building an AE

- Tricks:
  - work with a fixed number of points (2048)
  - encoder follows pointnet trick
  - then decoder is just a conv decoder to 2048x3
  - Main issue:
    - loss: EMD or CD?

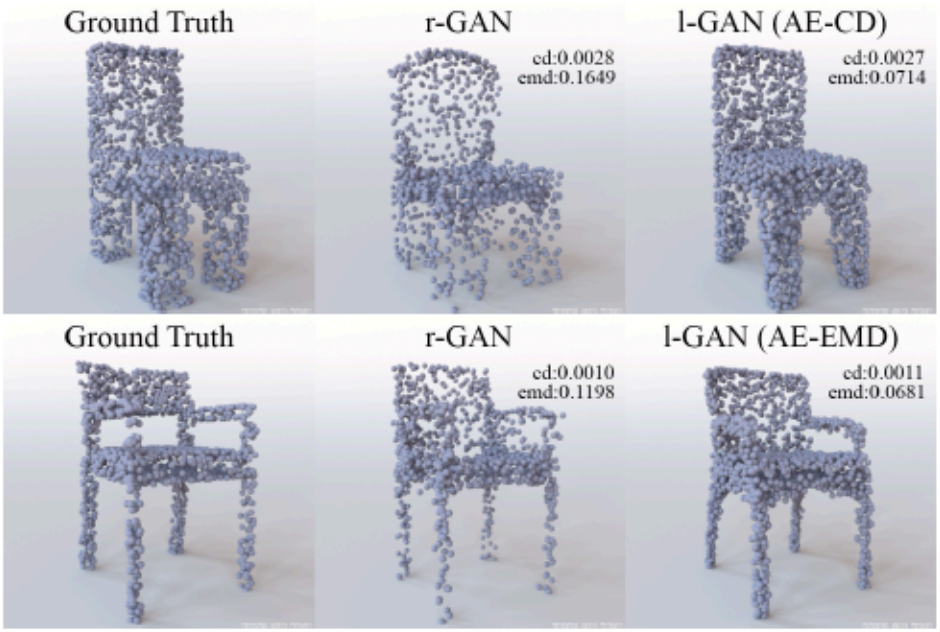


Figure 7. The CD distance is less faithful than EMD to visual quality of synthetic results; here, it favors r-GAN results, due to the overly high density of points in the seat part of the synthesized point sets.

Achlioptas ea 18

AEs can complete (think denoising)

Figure 4. Point cloud *completions* of a network trained with partial and complete (input/output) point clouds and the EMD loss. Each triplet shows the partial input from the test split (left-most), followed by the network's output (middle) and the complete ground-truth (right-most).

Achlioptas ea 18

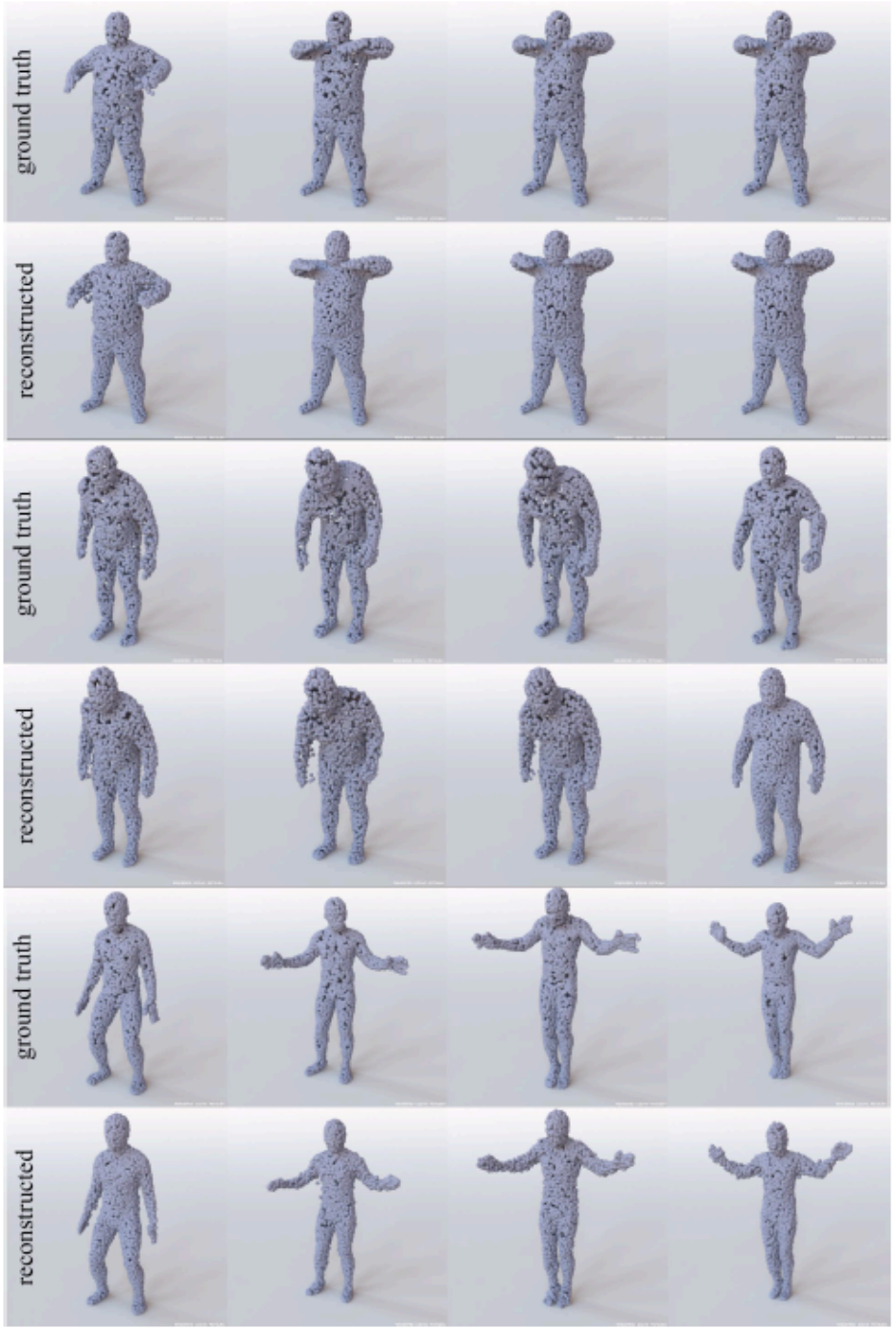


Figure 13. Reconstructions of unseen shapes from the *test* split extracted from the D-FAUST dataset of (Bogo et al., 2017) with an AE-EMD decoding point clouds with 4096 points. In each row the poses depict a motion (left-to-right) as it progress in time.

Achlioptas ea 18

Building a GAN

- Tricks:
  - we know how to encode a point set for discriminator, so OK
  - generator is easy

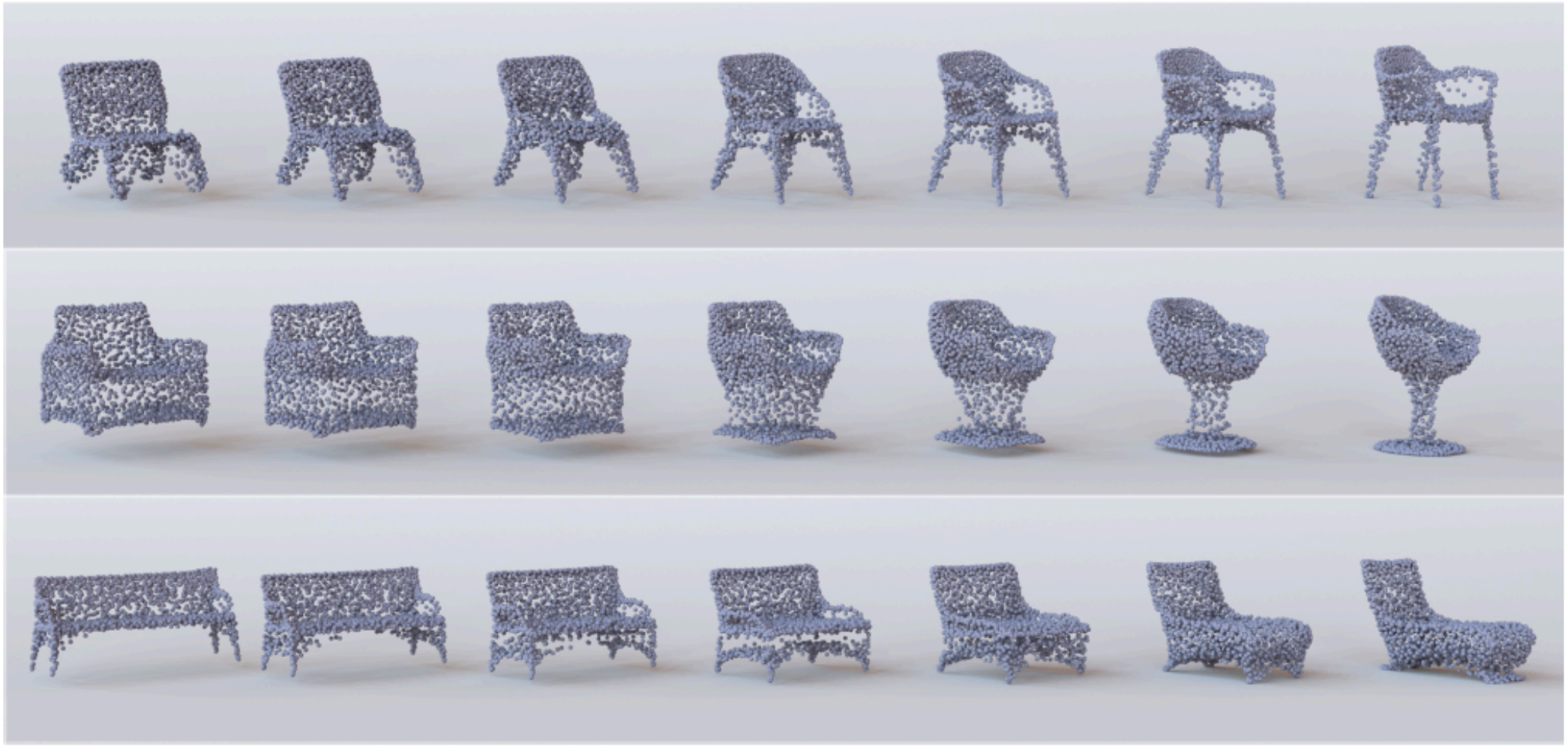


Figure 11. Interpolating between different point clouds (left and right-most of each row), using our latent space representation. Note the interpolation between structurally and topologically different shapes. **Note**: for all our illustrations that portray point clouds we use the Mitsuba renderer (Jakob, 2010).

Achlioptas ea 18

Notes and queries

- Fixed size point cloud is a mild dodge
  - likely easily remedied
- You could apply this to other things
  - think of a CvxNet repn as a point cloud
    - one point per convex, in 4x(# of planes) D
  - what loss would one apply to train autoencoder?
  - how would GAN training work?

Generating indoor scenes

- Goal:
  - generate an overhead layout of objects for an indoor scene
    - we can then place object models from some collection into this layout
    - yielding a synthetic scene
- Issues:
  - keep track of object interactions, counts, etc
    - eg one bed implies likely no more than two nightstands
    - eg objects may touch, but do not intersect

Generating indoor scenes

Figure 1. Synthetic virtual scenes generated by our method. Our model can generate a large variety of such scenes, as well as complete partial scenes, in under two seconds per scene. This performance is enabled by a pipeline of multiple deep convolutional generative models which analyze a top-down representation of the scene.

Ritchie et al 18

Strategy: auto-regressive model

- Construct the layout in steps
- Think of model as a string of unknown length
  - x=(act1, act2, act3, …, stop)
  - where act is
    - choose object category
    - choose location
    - choose orientation
    - choose size (and other attributes)
    - insert into occupancy map
  - Then p(x)=prod_i p(x_il all previous)

Autoregressive models. An *autoregressive* model is an iterative generative model that consumes its own output from one iteration as the input to its next iteration. That is, an autoregressive model examines the output it has generated thus far in order to decide what to generate next. For example, an autoregressive model for generating 3D indoor scenes could synthesize one object at a time, where the decision about what type of object to add and where to place it are based on the partial scene that has been generated so far (Figure 10). Formally, an autoregressive model defines a probability distribution over vectors $\mathbf{x}$ as

$$p(\mathbf{x}) = \prod_{i=1}^{|\mathbf{x}|} p(\mathbf{x}_i | \mathrm{NN}_i(\mathbf{x}_1 \ldots \mathbf{x}_{i-1})),$$

Ritchie et al 18

- Models of this form have several advantages
  - with appropriate choice of NN, can compute likelihood of points
    - normalizing flow
- Not relevant here, but…

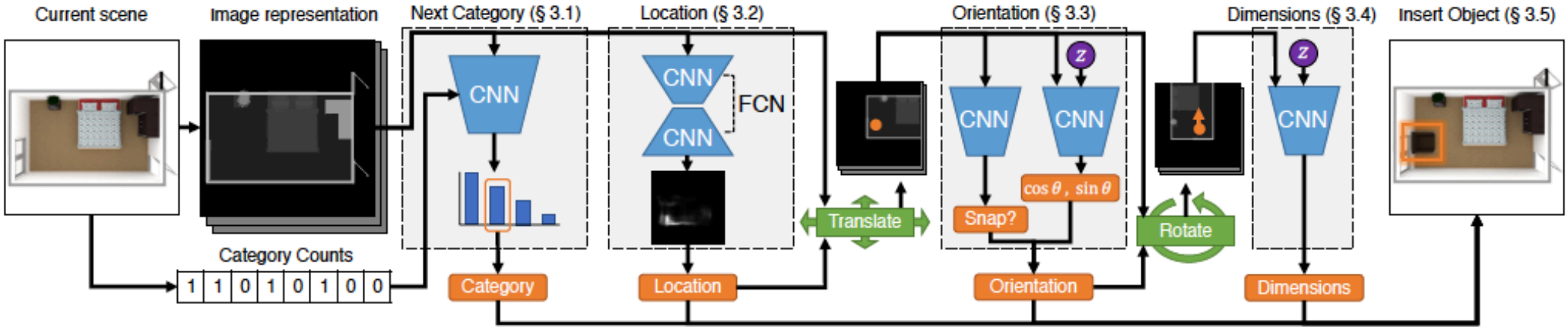


Figure 2. Overview of our automatic object-insertion pipeline. We extract a top-down-image-based representation of the scene, which is fed to four decision modules: which category of object to add (if any), the location, orientation, and dimensions of the object.

Ritchie et al 18

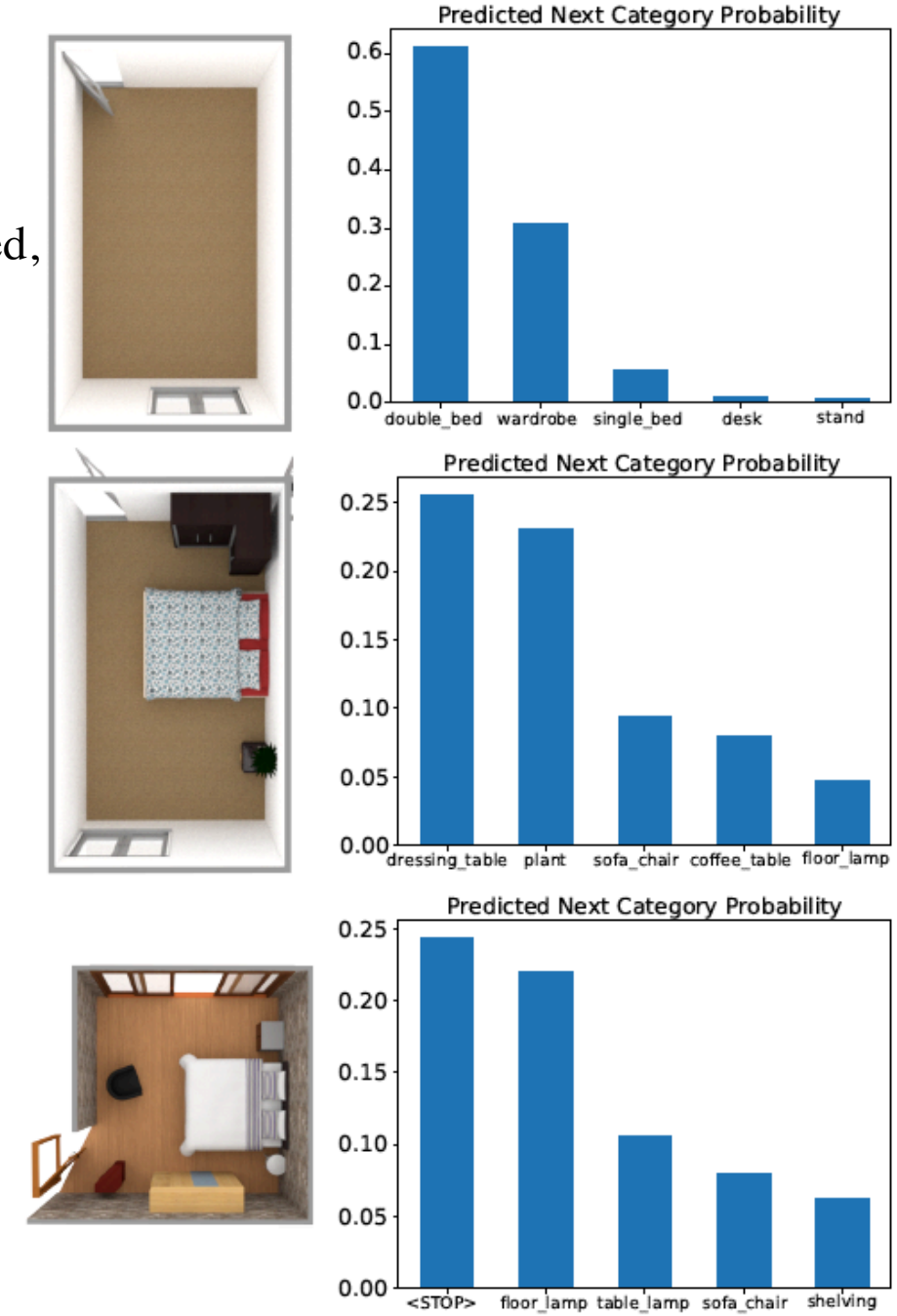
- Note:
  - image representation is basically just a labelled layout map
    - useful - summarizes where objects are
  - Notice the conditioning here:
    - category, location | labelled layout
    - orientation, dimensions| category, location
  - should be fairly obvious you can insert other blocks
    - to predict other attributes
  - there is a p(stop| input), too

I *think*

Training

- Take a set of authored scenes (SUNCG!?!)
  - remove some objects, decide to insert one, train to do so
  - Q: in what order?
    - support before supporting
    - first tier before second tier
    - bigger before smaller
  - Ordering is essential for category, insignificant for attributes, etc.

Overhead summary
(this has been beauty rendered,
which is ridiculous)



Probability of category
of next object

Figure 3. Distributions over the next category of object to add to the scene, as predicted by our model. Empty scenes are dominated by one or two large, frequent object types (*top*), partially populated scenes have a range of possibilities (*middle*), and very full scenes are likely to stop adding objects (*bottom*).

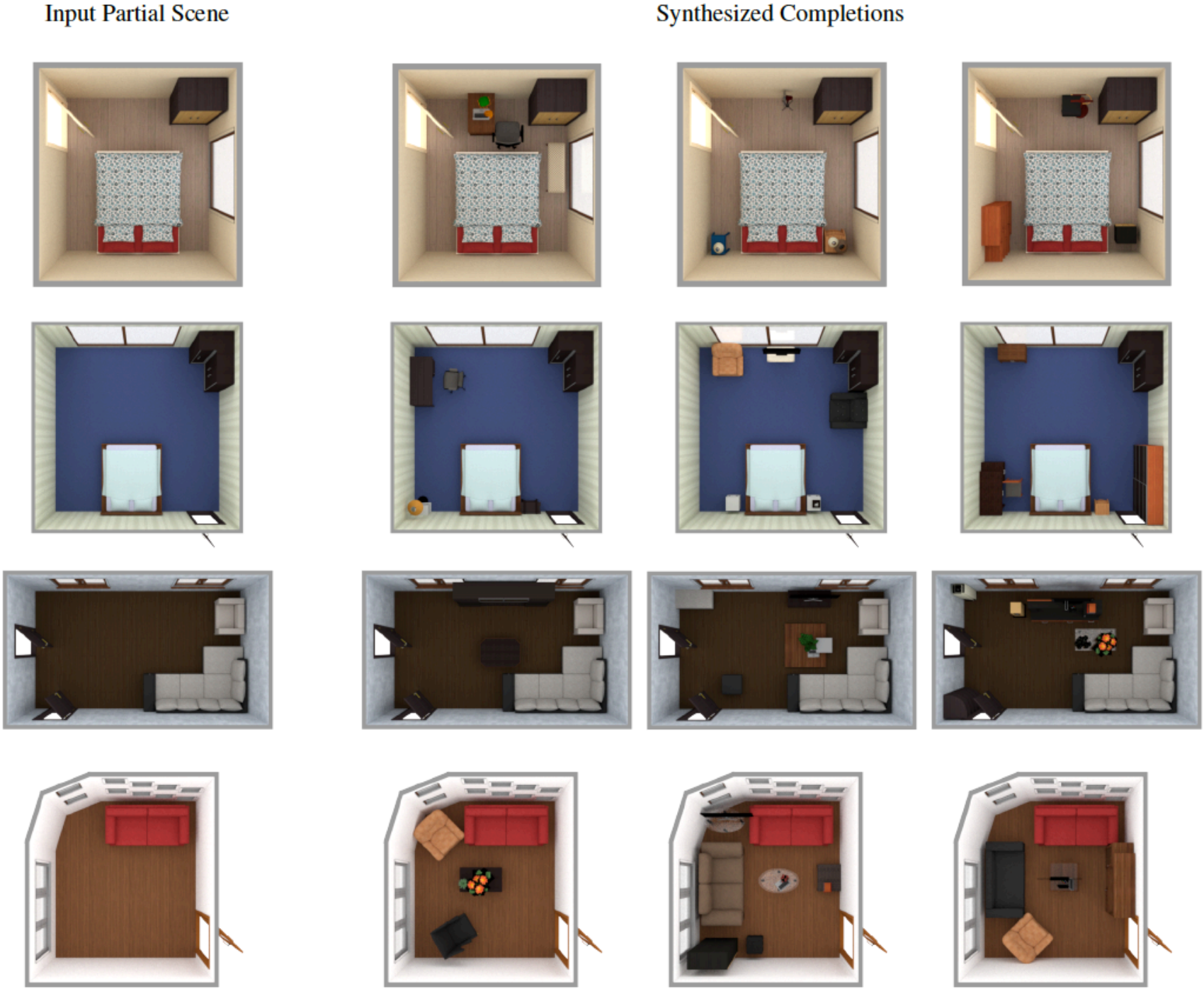


Figure 7. Given an input partial scene (*left column*), our method can generate multiple automatic completions of the scene. This requires no modification to the method's sampling procedure, aside from seeding it with a partial scene instead of an empty one.

Notes and queries

- Autoregressive models have ups and downs
  - Advantage:
    - easy to author
    - “make sense”
  - Disadvantages:
    - can get trapped (an error in early stages of generation persists)
    - can be hard to learn accurately (above)
- What ordering is appropriate?
- Why is this not a point cloud?
  - i.e. each object together with size, location, etc is a “point”
  - and we do pointnet
- Q: is the underlying combinatorial structure meaningful?
  - or just a nuisance?

Parsing and primitives

- Traditional idea, back to at least Binford 71
- Objects are a composite of primitive shapes
- Two issues:
  - What are the primitives?
  - Given some input, parse into the set of primitives?
- Traditional literature:
  - Construct a set of primitives, using
    - geometric insight, guessing, etc
  - Now infer presence of primitive from local image properties, edges, etc.
  - Major problem:
    - objects aren't precisely primitives, so….
  - Could almost be made to work

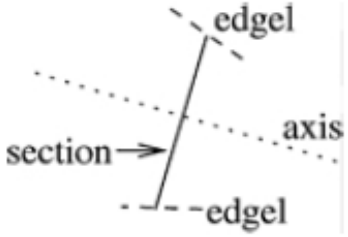


Figure 2. A symmetry: the two edgels (dashed lines) are symmetrical about the symmetry axis (dotted). We represent symmetries by their sections (solid line), which are line segments that connect the midpoints of the two edgels.

Ioffe+Forsyth 01

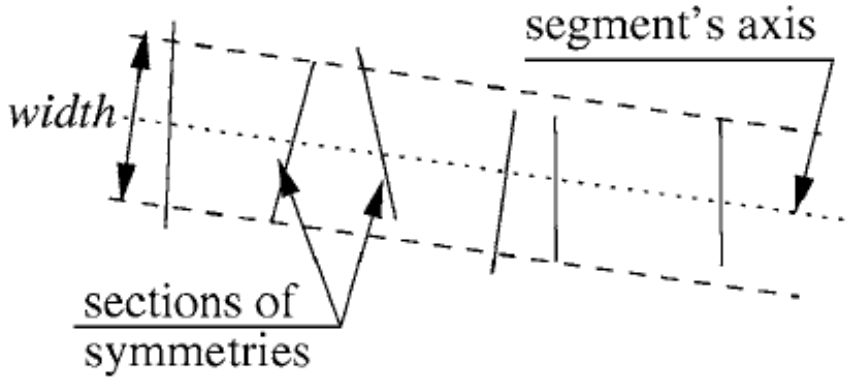


Figure 3. The segment finder groups symmetries into segments. The EM algorithm finds the optimal positions of the segments' axes and the widths, and also estimates the posterior probabilities that a given symmetry is assigned to a particular segment or to noise.

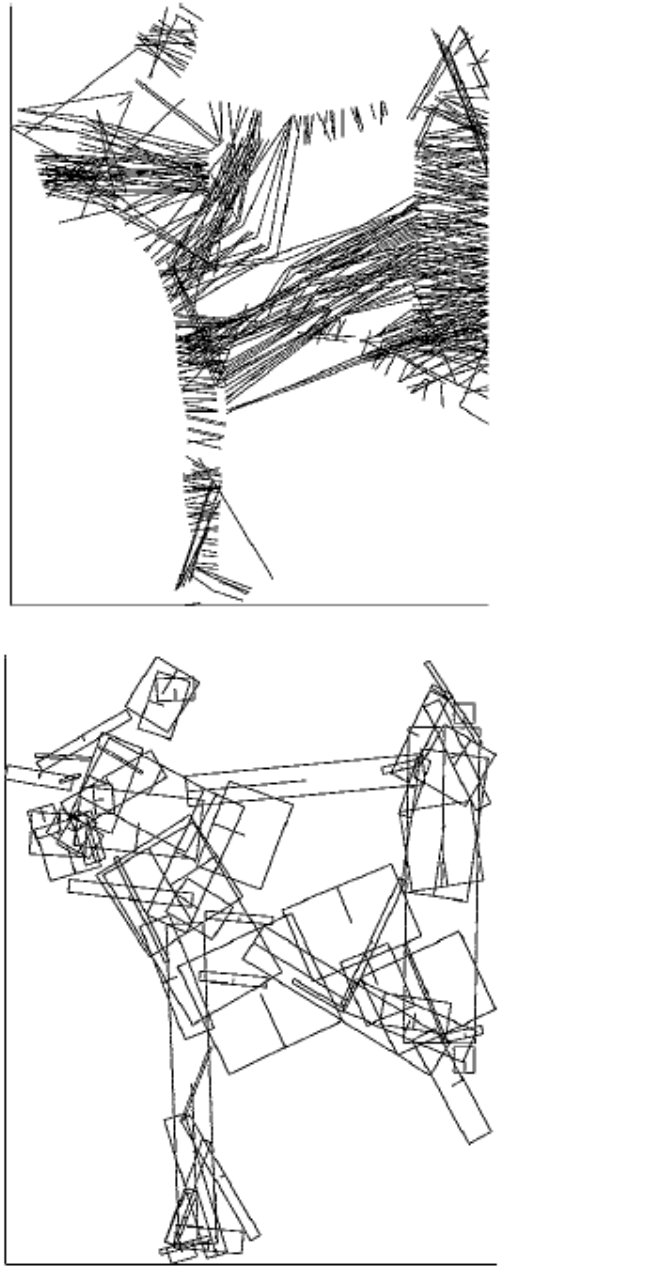
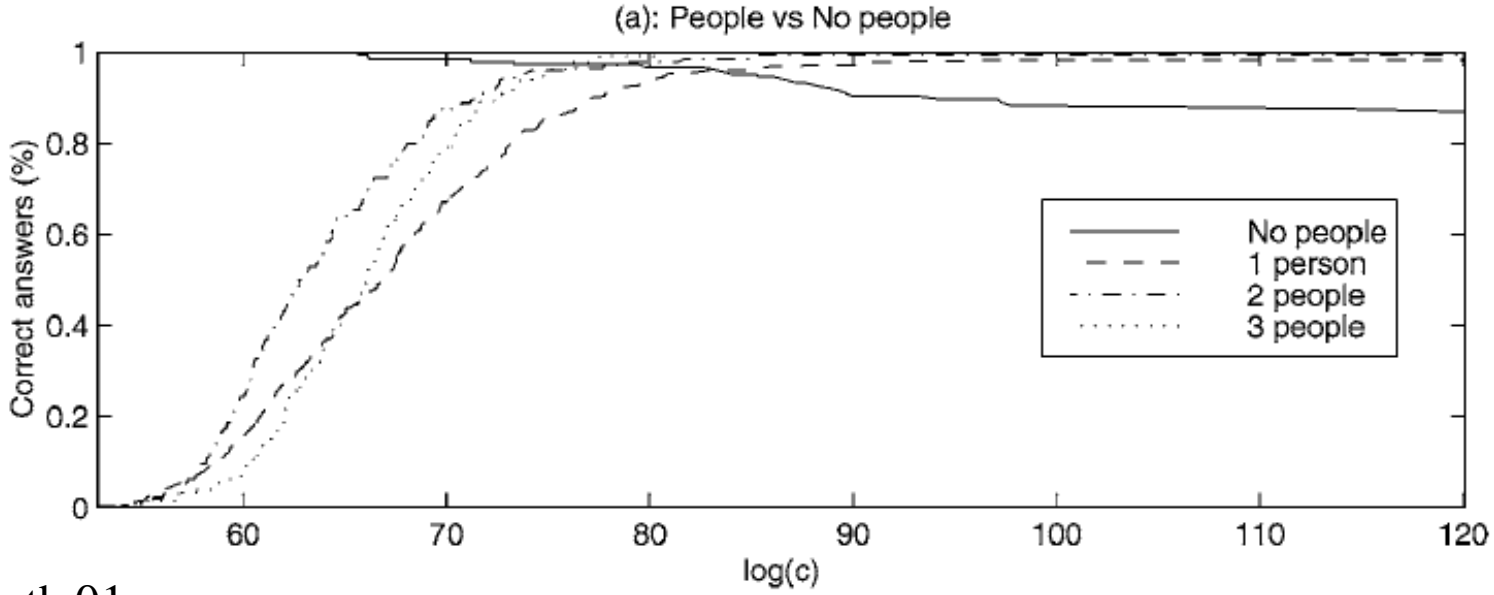


Figure 4. An example run of the segment finder. (top) A set c
symmetries obtained for an image. Each symmetry is represented b
its section. We show every 4th symmetry to avoid clutter. (bottom
The EM algorithm fits a fixed number of rectangular segments to th
symmetries. These are the candidate body segments which becom
the input to our assembly-builder (grouper).

Ioffe+Forsyth 01



Ioffe+Forsyth 01

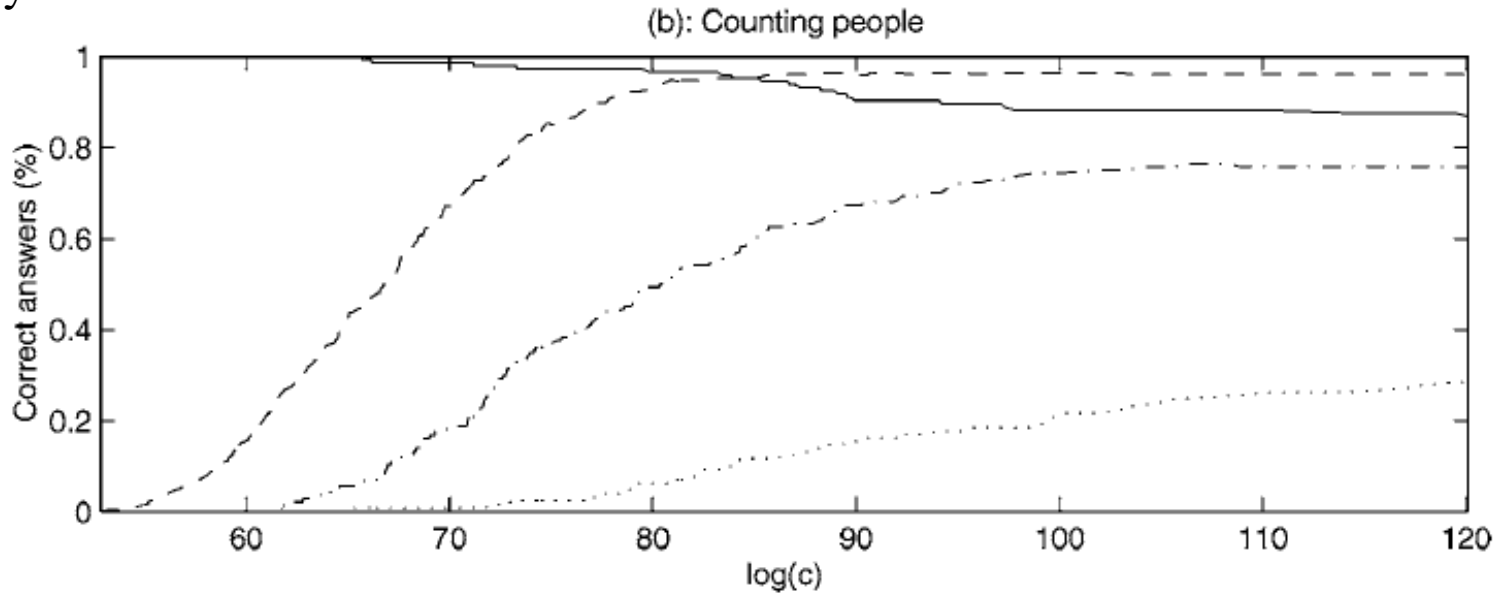


Figure 14. Percentage of correct decisions for Person vs No person classification (a) and Counting (b), as a function of the parameter *c*. Each figure shows the percentages separately for images with 0, 1, 2, and 3 people. We believe that the decrease in the count accuracy as the number of people goes up is due to the segment finder, which fails to extract all the relevant segments.

Ioffe+Forsyth 01

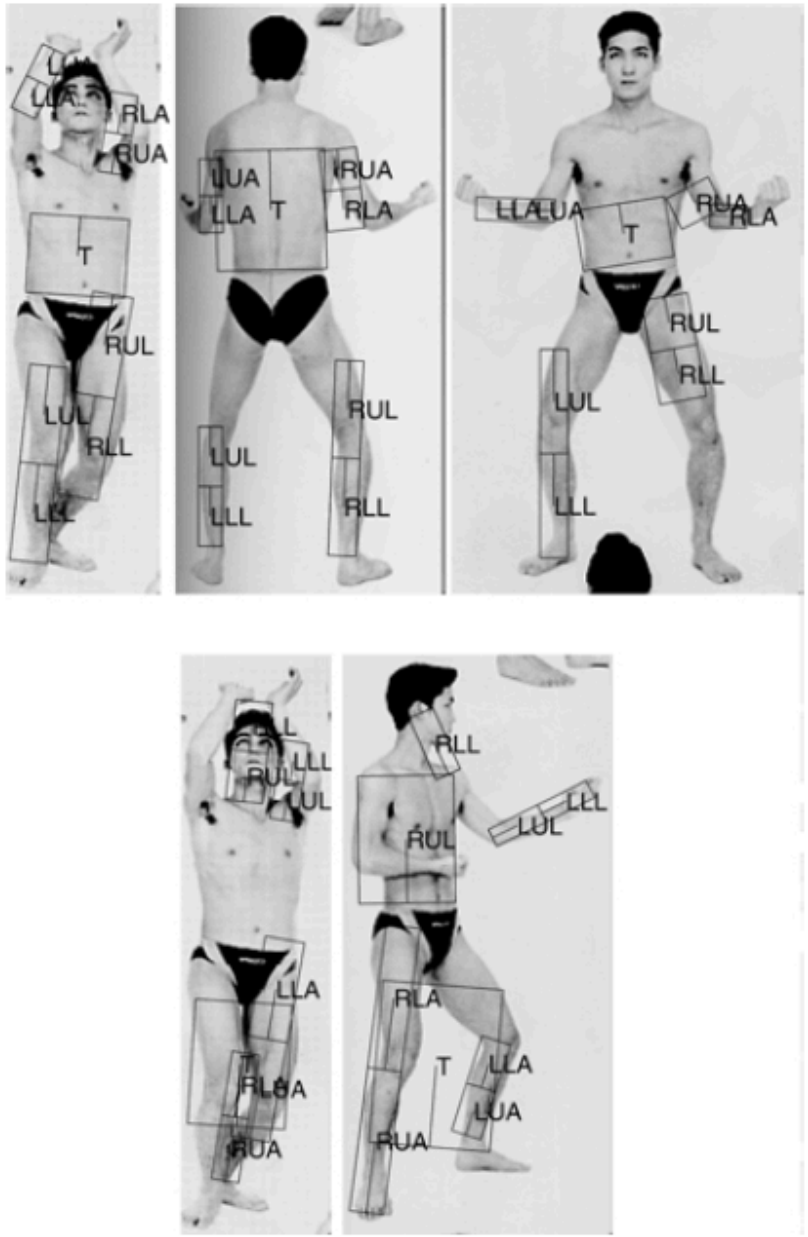


Figure 15. Examples of representative assemblies found for images of people. Each representative assembly is the highest-likelihood sample from a set of samples with overlapping torsos. We use representatives to count people and argue that using representatives does not change the count. Often (top) representatives can also be used to infer configurations of people, although (bottom) that is not always the case.

More modern strategies

- Get a collection of shapes that has been parsed
  - learn to predict segment labels
  - straightforward supervised learning, data presents problems
  - dominant process for (say) people
- But what if you don't have parses?
  - choose a set of possible primitives
  - learn to represent shapes using these
  - Q:
    - what loss?
    - how to handle number of primitives?

We've already seen one of these (CvxNet); but it doesn't handle different numbers of primitives

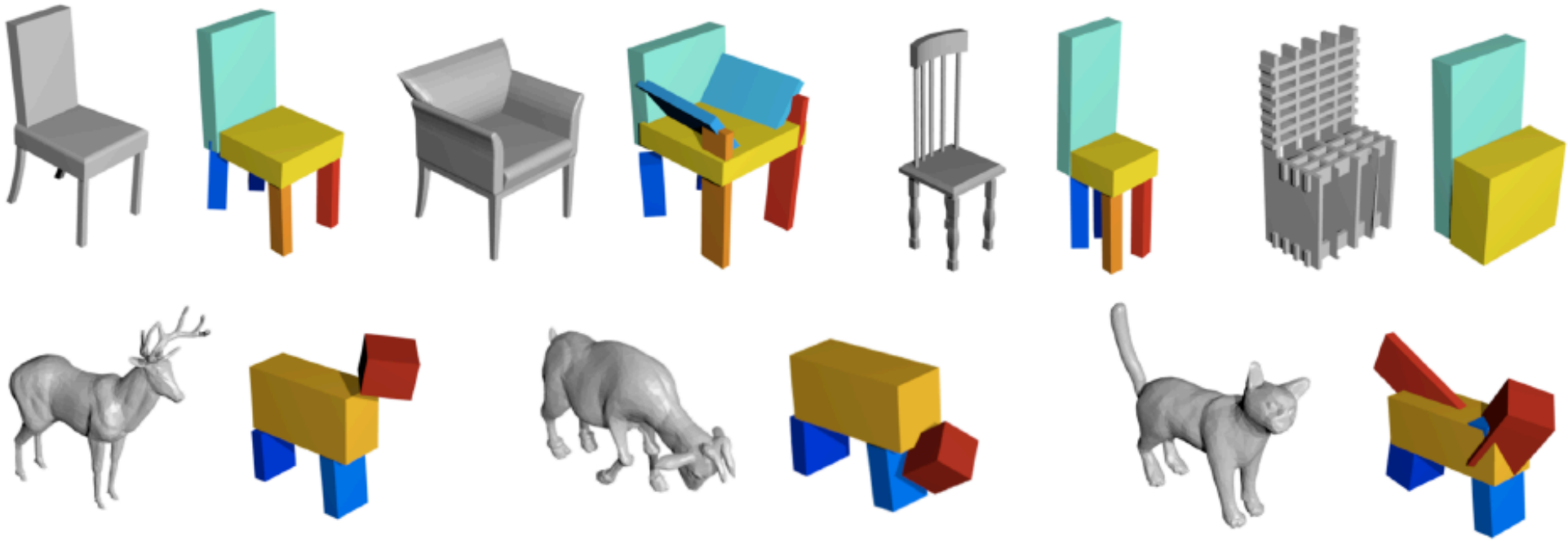


Figure 1: Examples of chair and animal shapes assembled by composing simple volumetric primitives (cuboids). The obtained reconstructions allows an interpretable representation for each object and provides a consistent parsing across shapes *e.g.* chair seats are captured by the same primitive across the category.

Tulsiani et al 17

Notice the varying number of primitives

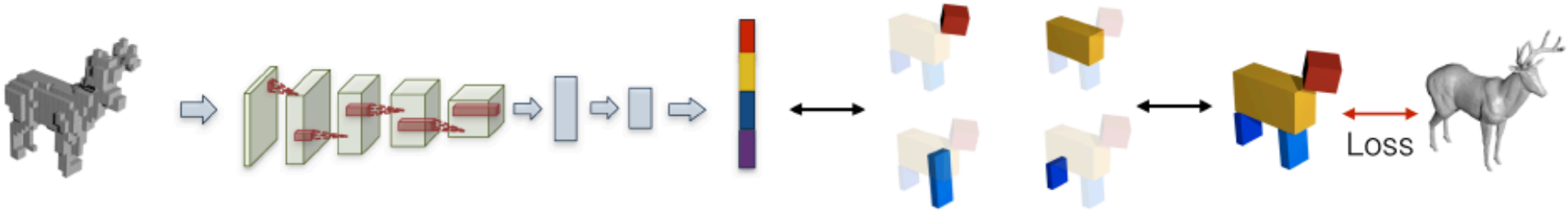


Figure 2: Overview of our approach. Given the input volume corresponding to an object O, we use a CNN to predict primitive shape and transformation parameters $\{(z_m, q_m, t_m)\}$ for each part (Section 3.1). The predicted parameters implicitly define transformed volumetric primitives $\{\bar{P}_m\}$ whose composition induces an assembled shape. We train our system using a loss function which attempts to minimize the discrepancy between the ground-truth mesh for O and the assembled shape which is implicitly defined by the predicted parameters (Section 3.2).

Tulsiani et al 17

- Primitives are cuboids
  - attached are shape and transformation parameters
- The number isn't fixed
  - choose M (largest number)
  - give each primitive a "probability of existence"
  - compute expected loss of assembly
    - expectation over probability of existence
  - at test time, threshold this probability, so get k<M primitives

Primitive Representation. As we mentioned above, the primitive representation has an added parameter p_m – the probability of its existence. To incorporate this, we factor the primitive shape z_m into two components – (z_m^s, z_m^e). Here z_m^s represents the primitive's dimensions (*e.g.* cuboid height, width, depth) as before and $z_m^e \sim Bern(p_m)$ is a binary variable which denotes if the primitive actually exists *i.e.* if $z_m^e = 0$ we pretend as if the m^{th} primitive does not exist. The prediction of the CNN in this scenario is as below.

$$\{(z_m^s, q_m, t_m, p_m) | m = 1 \cdots M\} = h_\theta(I) \quad (8)$$

$$\forall_m \; z_m^e \sim Bern(p_m); \; z_m \equiv (z_m^s, z_m^e) \quad (9)$$

Note that the CNN predicts p_m – the parameter of the Bernoulli distribution from which the part existence variable z_m^e is sampled. This representation allows the prediction of a variable number of parts *e.g.* if a chair is best explained using $k < M$ primitives, the network can predict a

Probability of existence

Losses

- Note: primitives are convex, and constructions apply to convex objects

Distance Field. A distance field $\mathcal{C}(\,\cdot\,; O)$ corresponding to an object O is a function $\mathbb{R}^3 \to \mathbb{R}^+$ that computes the distance to the closest point of the object. Note that it evaluates to 0 in the object interior.

$$\mathcal{C}(p; O) = \min_{p' \in O} \|p - p'\|_2 \qquad (2)$$

- For general O, this function is nasty
  - there may be multiple closest points on O
  - for convex O, there is only one

Tulsiani et al 17

Losses

3.2.2 Coverage Loss : $O \subseteq \cup_m \bar{P}_m$.

We want to penalize the CNN prediction if the target object O is not completely covered by the predicted shape $\cup_m \bar{P}_m$. A sufficient condition to ensure this is that the distance field of the assembled shape evaluates to zero for all points on the surface of O.

$$L_1(\{(z_m, q_m, t_m)\}, O) = \mathbb{E}_{p \sim S(O)} \|\mathcal{C}(p; \cup_m \bar{P}_m)\|^2 \quad (3)$$

Computation can be simplified due to a nice property of distance fields. It is easy to show that the distance field of a composed shape equals to the pointwise minimum of the distance fields of all composing shapes:

$$\mathcal{C}(p; \cup_m \bar{P}_m) = \min_m \mathcal{C}(p; \bar{P}_m) \quad (4)$$

This decomposition rule boils the distance field of a whole shape down to the distance field of a primitive. In the following, we show how to efficiently compute $\mathcal{C}$ for primitives as cuboids.

Essentially, the shape lies inside the union of the primitives
easy test with convex primitives

Tulsiani et al 17

3.2.3 Consistency Loss : $\underset{m}{\cup}\bar{P}_m \subseteq O$.

We want to penalize the CNN prediction if the predicted shape $\underset{m}{\cup}\bar{P}_m$ is not completely inside the target object O. A sufficient condition is to ensure this is that the distance field

of the object O shape evaluates to zero for all points on the surface of individual primitives P_m.

$$L_2(\{(z_m, q_m, t_m)\}, O) = \sum_m \mathbb{E}_{p \sim S(P_m)} \|\mathcal{C}(p; O)\|^2 \quad (7)$$

Additionally, we observe that to sample a point p on the surface of $\bar{P}_m$, one can equivalently sample p' on the surface of the untransformed primitive P_m and then rotate, translate p' according to (q_m, z_m).

$$p \sim S(\bar{P}_m) \equiv \mathcal{T}(\mathcal{R}(p', q_m), t_m); p' \sim S(P_m)$$

An aspect for computing gradients for the predicted parameters using this loss is the ability to compute derivatives for z_m given gradients for a sampled point on the canonical untransformed primitive $p' \sim S(P_m)$. We do so by using the *re-parametrization trick* [21] which decouples the parameters from the random sampling. As an example, consider a point being sampled on a rectangle extending from $(-w, -h)$ to (w, h). Instead of sampling the x-coordinate as $x \sim [-w, w]$, one can use $u \sim [-1, 1]$ and $x = uw$. This re-parametrization of sampling allows one to compute $\frac{\partial x}{\partial w}$. We provide the details for applying the re-parametrization trick for a cuboid primitive in the appendix.

Losses

- essentially, sample primitives and check the samples are inside O
- Likely easier to sample O and check outside the primitives

Learning. Under the reformulated representation of primitives, the CNN output does not induce a unique assembled shape – it induces a distribution of possible shapes where the m^{th} primitive stochastically exists with probability p_m. In this scenario, we want to minimize the expected loss across the possible assemblies. The first step is to modify the consistency and coverage losses to incorporate $z_m \equiv (z_m^s, z_m^e)$. Towards this, we note that the untransformed primitive P_m is either a cuboid (if $z_m^e = 1$) or empty (if $z_m^e = 0$). In case it is empty, we can simply skip it the the consistency loss (Section 3.2.3) for this primitive and can incorporate this in the coverage loss (Section 3.2.2) by modifying Eq. 6 as follows -

$$\mathcal{C}(\ \cdot\ ; P_m) = \begin{cases} \infty, & \text{if } z_m^e = 0 \\ \mathcal{C}_{cub}(\ \cdot\ ; z_m^s), & \text{if } z_m^e = 1 \end{cases} \tag{10}$$

We can now define the final loss $L(h_\theta(I), O)$ using the concepts developed. Note that this is simply the expected loss across possible samplings of z_m^e according to p_m.

$$L(\{(z_m, q_m, t_m)\}, O) = L_1(\{(z_m, q_m, t_m)\}, O) + L_2(\{(z_m, q_m, t_m)\}, O) \tag{11}$$

$$L(h_\theta(I), O) = \mathbb{E}_{\forall m\ z_m^e \sim Bern(p_m)} L(\{(z_m, q_m, t_m)\}, O)$$

Under this loss function, the gradients for the continuous variables *i.e.* $\{(z_m^s, q_m, t_m)\}$ can be estimated by averaging their gradients across samples. However, to compute gradients for the distribution parameter p_m, we use the REINFORCE algorithm [37] which basically gives positive feedback if the overall error is low (reward is high) and negative feedback otherwise. To further encourage parsimony, we include a small *parsimony reward* (reward for choosing fewer primitives) when computing gradients for p_m.

Tulsiani et al 17

Losses

- Manage the number of primitives by rewarding rep'ns in terms of few primitives
- Cf CvxNet

Decomposition loss (auxiliary). We seek a parsimonious decomposition of an object akin to Tulsiani et al. [68]. Hence, overlap between elements should be discouraged:

$$\mathcal{L}_{\text{decomp}}(\omega) = \mathbb{E}_{\mathbf{x} \sim \mathbb{R}^3} \|\text{relu}(\text{sum}_k\{\mathcal{C}_k(\mathbf{x})\} - \tau)\|^2, \tag{4}$$

where we use a permissive $\tau = 2$, and note how the ReLU activates the loss only when an overlap occurs.

Deng et al

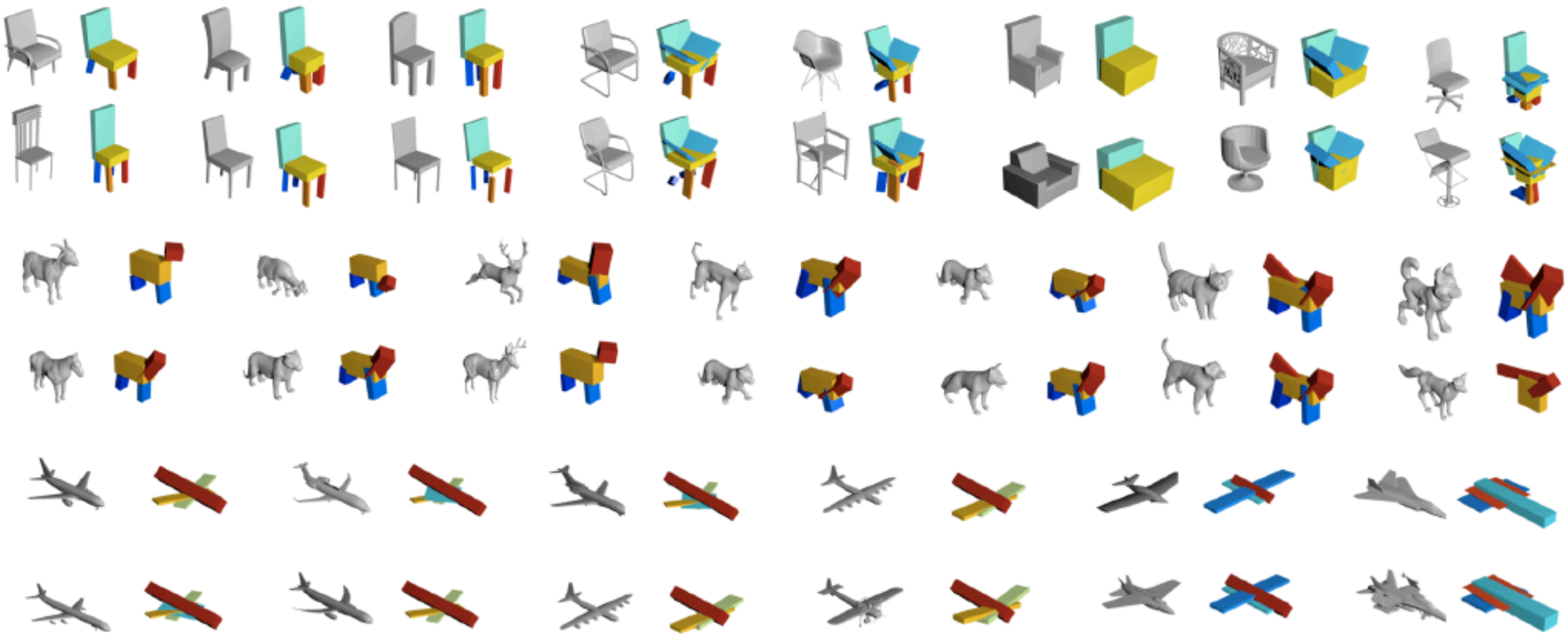


Figure 3: Final predictions of our method on chairs, animals and aeroplanes. We visualize the more commonly occurring modes on the left and progressively towards the right show rarer configurations predicted.

Figure 6: Projection of the predicted primitives onto the original shape. We assign each point p in the original shape to the corresponding primitive with lowest distance field $\mathcal{C}(p, \bar{P}_m)$. We visualize the parsing by coloring each point according to the assigned primitive. We see that similar parts *e.g.* aeroplane wings, chair seat, *etc.* are consistently colored.

- Primitives imply a segmentation of the original shape

Notes and queries

- They're not really learning primitives
  - all boxes
  - But they are really learning to parse
- Q:
  - could/should one do this with a richer vocabulary of primitives?
    - eg large vocab and CSG
    - remove primitives if not used
  - what is the value of using primitives here?
    - likely the implicit learned segmentation
  - could one build a shape generator by:
    - each primitive is a point
    - point set generator to make these primitive repns
    - something to make the residual wrt primitives

More complex primitives

- Superquadrics:

Having specified our network and the loss function, we now provide details about the superquadric representation and its parameterization λ. Note that, in this section, we omit the primitive index m for clarity. Superquadrics define a family of parametric surfaces that can be fully described by a set of 11 parameters [1]. The explicit superquadric equation defines the surface vector $\mathbf{r}$ as

$$\mathbf{r}(\eta,\omega) = \begin{bmatrix} \alpha_1 \cos^{\epsilon_1}\eta \cos^{\epsilon_2}\omega \\ \alpha_2 \cos^{\epsilon_1}\eta \sin^{\epsilon_2}\omega \\ \alpha_3 \sin^{\epsilon_1}\eta \end{bmatrix} \quad \begin{matrix} -\pi/2 \le \eta \le \pi/2 \\ -\pi \le \omega \le \pi \end{matrix} \tag{13}$$

where $\alpha = [\alpha_1, \alpha_2, \alpha_3]$ determine the size and $\epsilon = [\epsilon_1, \epsilon_2]$ determine the global shape of the superquadric, see supplementary material for examples. Following common practice [39], we bound the values ϵ_1 and ϵ_2 to the range $[0.1, 1.9]$ so as to prevent non-convex shapes which are less likely to occur in practice. Eq. 13 produces a superquadric in a canonical pose. In order to allow any position and orientation, we augment the primitive parameter λ with an additional rigid body motion represented by a translation vector $\mathbf{t} = [t_x, t_y, t_z]$ and a quaternion $\mathbf{q} = [q_0, q_1, q_2, q_3]$ which determine the coordinate system transformation $\mathcal{T}(\mathbf{x}) = \mathbf{R}(\lambda)\,\mathbf{x} + \mathbf{t}(\lambda)$ above.

Paschalidou et al, 19

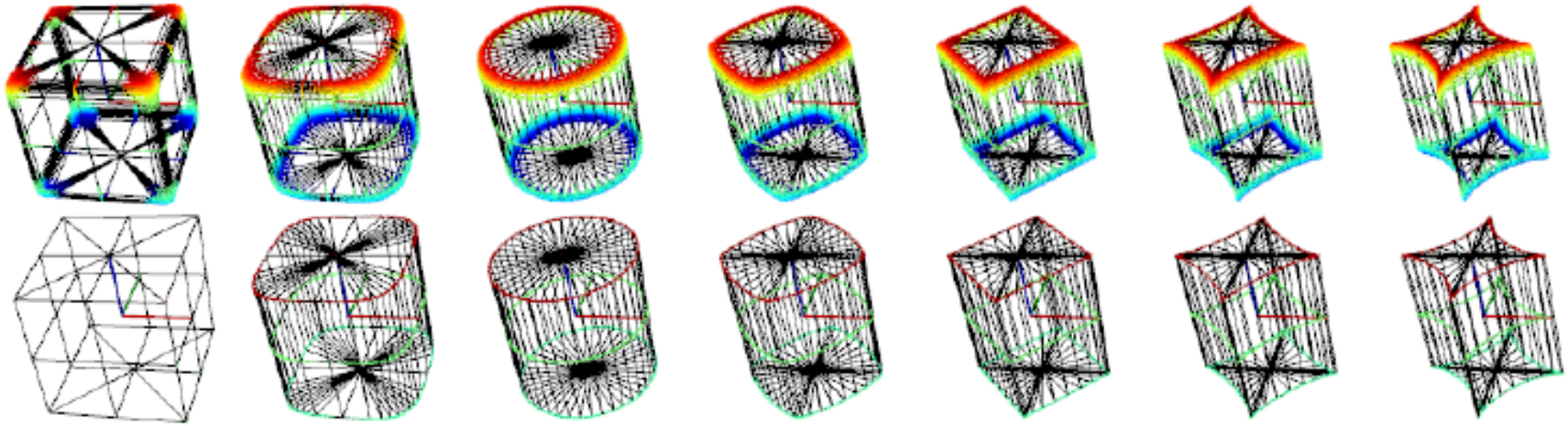


Figure 3: The effect of restricting the exponent parameters to $\varepsilon_1, \varepsilon_2 \geq 0.1$. Top row depicts the restricted superquadrics with $\varepsilon_1 = .1$ and respectively $\varepsilon_2 = 0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0$, whereas the bottom row contains corresponding superquadrics with sharp edges - $\varepsilon_1 = 0.0$ and $\varepsilon_2 = 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0$. Note that colored parts highlight the areas which are supposed to represent a single boundary curve or corner, i.e., the colors indicate the regions where the prominent approximation errors for objects with sharp contours occur.

Vaskevicius and Birk 19

- Superquadrics come with a nasty issue
  - distance, gradient become unstable with small eps
  - standard fix: eps > 0.1 (adopted here)
  - alternative in Vaskevicius and Birk

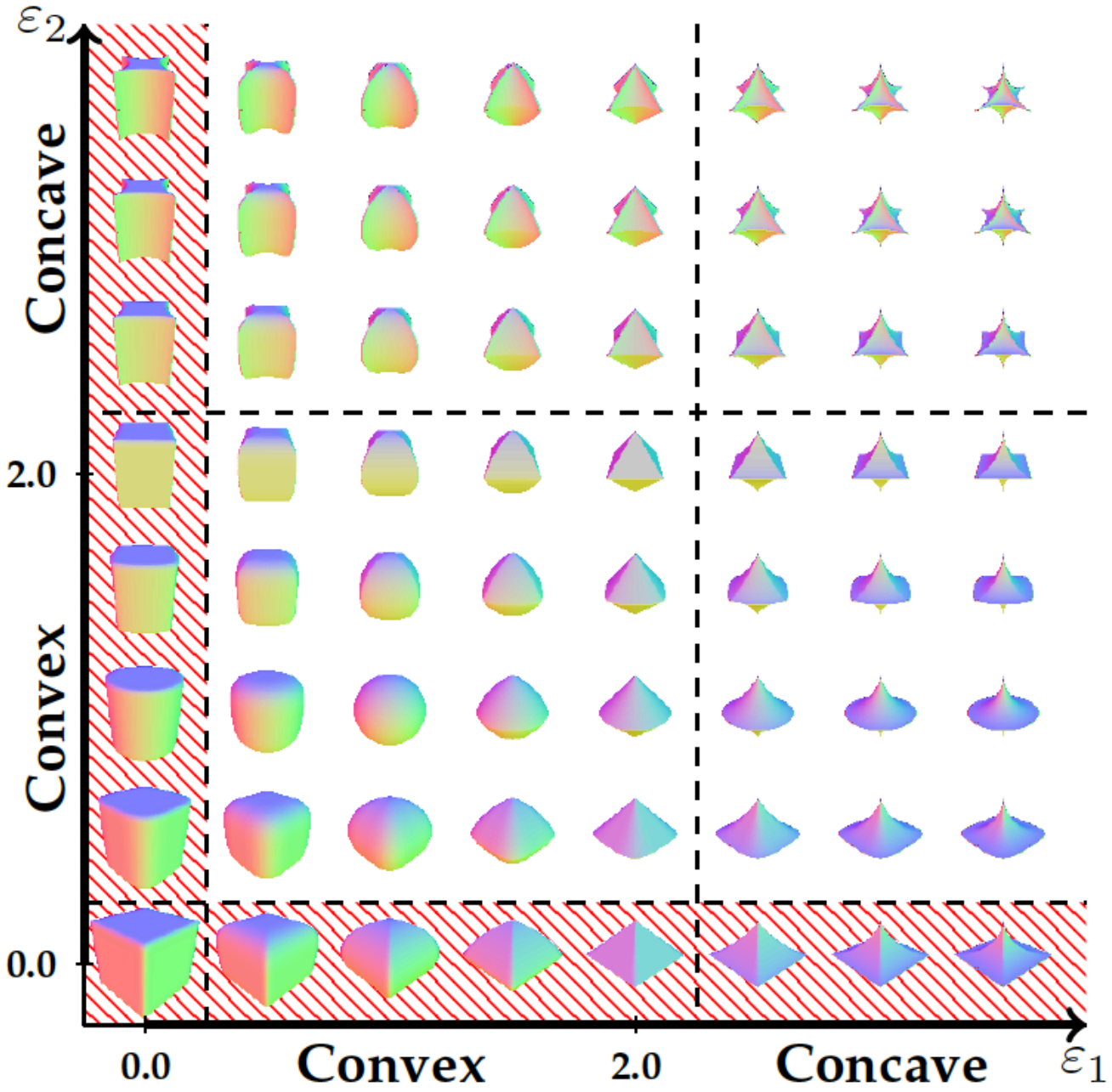


Figure 2: Example superquadrics with respect to the shape parameters ($0 \leq \varepsilon_1, \varepsilon_2 \leq 3.5$ with 0.5 resolution). The surfaces are colored based on RGB encoding of the surface normal vectors with red intensity corresponding to x, green to y, and blue to z components. The superquadrics have a convex shape within the region of $0 \leq \varepsilon_1, \varepsilon_2 \leq 2$.

Vaskevicius and Birk 19

General structure

- Rather like Tulsiani et al (above), BUT
  - Use chamfer distance to sampled primitives
  - Do not require REINFORCE
    - even though primitives could not exist (as in Tulsiani)

in our supplementary material. Thus, we use the Chamfer distance in our experiments

$$\mathcal{L}_D(\mathbf{P}, \mathbf{X}) = \mathcal{L}_{P \to X}(\mathbf{P}, \mathbf{X}) + \mathcal{L}_{X \to P}(\mathbf{X}, \mathbf{P}) \quad (3)$$

where $\mathcal{L}_{P \to X}$ measures the distance from the predicted primitives $\mathbf{P}$ to the point cloud $\mathbf{X}$ and $\mathcal{L}_{X \to P}$ measures the distance from the point cloud $\mathbf{X}$ to the primitives $\mathbf{P}$. We weight the two distance measures in (3) with 1.2 and 0.8, respectively, which empirically led to good results.

    - notice a mild asymmetry, due to existence issue

Paschalidou et al, 19

Primitive to point cloud

Primitive-to-Pointcloud: We represent the target point cloud as a set of 3D points $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^{N}$. Similarly, we approximate the continuous surface of primitive m by a set of points $\mathbf{Y}_m = \{\mathbf{y}_k^m\}_{k=1}^{K}$. Details of our sampling strategy are provided in Section 3.4. This discretization allows us to express the distance between a superquadric and the target point cloud in a convenient form. In particular, for each point on the primitive $\mathbf{y}_k^m$, we compute its closest point on the target point cloud $\mathbf{x}_i$, and average this distance across all points in $\mathbf{Y}_m$ as follows:

$$\mathcal{L}_{P\rightarrow X}^{m}(\mathbf{P},\mathbf{X}) = \frac{1}{K}\sum_{k=1}^{K}\Delta_k^m \tag{4}$$

- Sampling primitives helps a lot
  - closest point on superquadric requires root finding if convex
    - nastier if not

Paschalidou et al, 19

Point cloud to primitive

This allows us to simplify Eq. 8 as follows

$$\mathcal{L}_{X \to P}(\mathbf{X}, \mathbf{P}) = \sum_{\mathbf{x}_i \in \mathbf{X}} \sum_{m=1}^{M} \Delta_i^m \gamma_m \prod_{\bar{m}=1}^{m-1} (1 - \gamma_{\bar{m}}) \quad (11)$$

where $\gamma_{\bar{m}}$ is a shorthand notation which denotes the existence probability of a primitive closer than primitive m. Note that this function requires only M, instead of 2^M, evaluations of the function Δ_i^m which is one of the main results of this paper. For a detailed derivation of (11), we refer the reader to the supplementary material.

1 if m'th primitive exists

Minimum distance from i'th poin to m'th primitive, sorted in ascending order

Paschalidou et al, 19

Parsimony

3.2. Parsimony Loss

Despite the bidirectional loss formulation above, our model suffers from the trivial solution $\mathcal{L}_D(\mathbf{P}, \mathbf{X}) = 0$ which is attained for $\gamma_1 = \cdots = \gamma_m = 0$. Moreover, multiple primitives with identical parameters yield the same loss function as a single primitive by dispersing their existence probability. We thus introduce a regularizer loss on the existence probabilities γ which alleviates both problems:

$$\mathcal{L}_\gamma(\mathbf{P}) = \max\left(\alpha - \alpha \sum_{m=1}^{M} \gamma_m, 0\right) + \beta\sqrt{\sum_{m=1}^{M} \gamma_m} \quad (12)$$

The first term of (12) makes sure that the aggregate existence probability over all primitives is at least one (i.e., we expect at least one primitive to be present) and the second term enforces a parsimonious scene parse by exploiting a loss function sub-linear in $\sum_m \gamma_m$ which encourages sparsity. α and β are weighting factors which are set to 1.0 and 10^{-3} respectively.

Paschalidou et al, 19

It's now straightforward…

- Encoder
  - accept input shape and make code
- Decoder
  - make a fixed number of primitives
    - parameters, rotations, translations, existence prob
  - to minimize loss
- Unsupervised
  - in the sense that it doesn't see ground truth primitives for input

Closer than cuboids

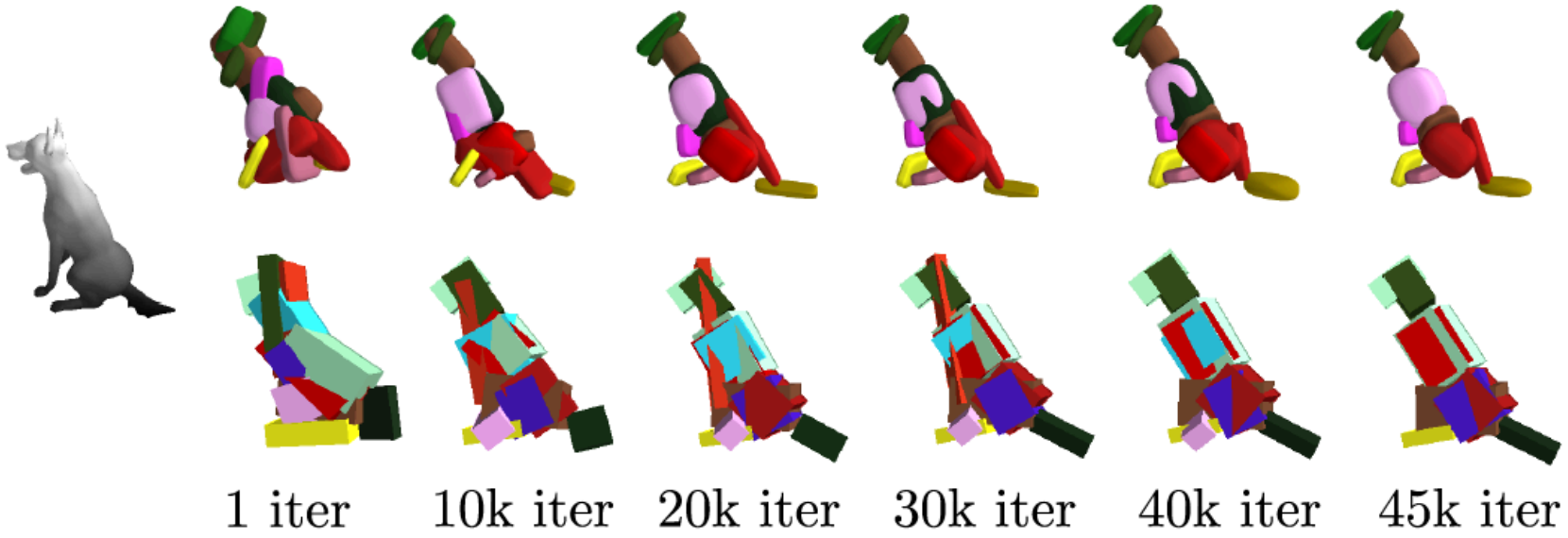


Figure 4: **Training Evolution.** We visualize the qualitative evolution of superquadrics (top) and cuboids (bottom) during training. Superquadrics converge faster to more accurate representations, whereas cuboids cannot capture details such as the open mouth of the dog, even after convergence.

Figure 5: **Qualitative Results on SURREAL.** Our network learns semantic mappings of body parts across different body shapes and articulations. For instance, the network uses the same primitive for the left forearm across instances.

Figure 6: **Qualitative Results on ShapeNet.** We visualize predictions for the object categories *animals*, *aeroplane* and *chairs* from the ShapeNet dataset. The top row illustrates the ground-truth meshes from every object. The middle row depicts the corresponding predictions using the cuboidal primitives estimated by [37]. The bottom row shows the corresponding predictions using our learned superquadric surfaces. Similarly to [37], we observe that the predicted primitive representations are consistent across instances. For example, the primitive depicted in green describes the right wing of the aeroplane, while for the animals class, the yellow primitive describes the front legs of the animal.

Notes and queries

- Closer than boxes
  - but so what?
    - are we trying to encode shape as primitives? or segment? or what?
- Chamfer distance seems clearly the way to go
- You could do this for many kinds of surface primitive
  - eg Gaussian blobs, metaballs,
- Notice important advantage of CvxNet representation
  - Can’t use Farkas’ lemma easily on superquadrics
    - non-convex: obvious
    - convex: the set of supporting planes is infinite, so impractical
      - Q: what happens if you sample?
      - A: why not just use the planes?

CSGNet - parsing into CSG

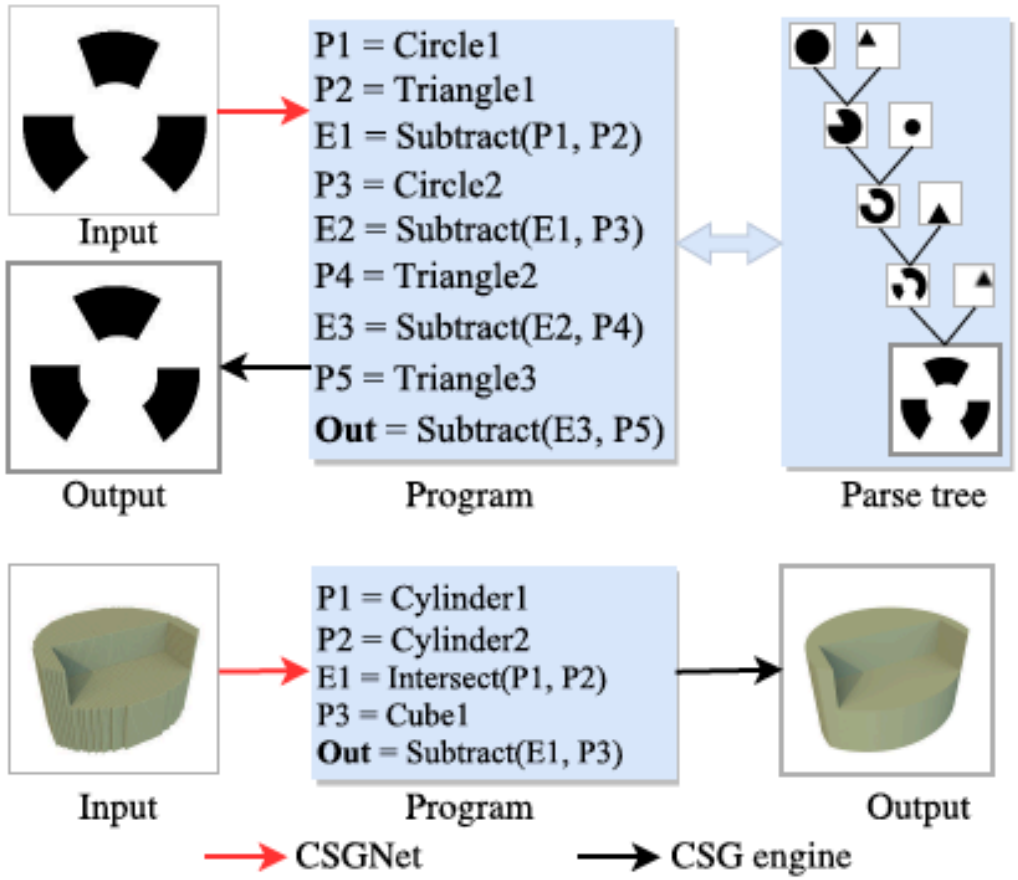


Figure 1. **Our shape parser produces a compact program that generates an input 2D or 3D shape.** On top is an input image of 2D shape, its program and the underlying parse tree where primitives are combined with boolean operations. On the bottom is an input voxelized 3D shape, the induced program, and the resulting shape from its execution.

Transformations are absent - just cause they make the figure too busy?

Idea:
you can learn to parse into a CSG rep by minimizing loss of generated parse against true volume; you may not need any actual CSG reps

Sharma et al 18

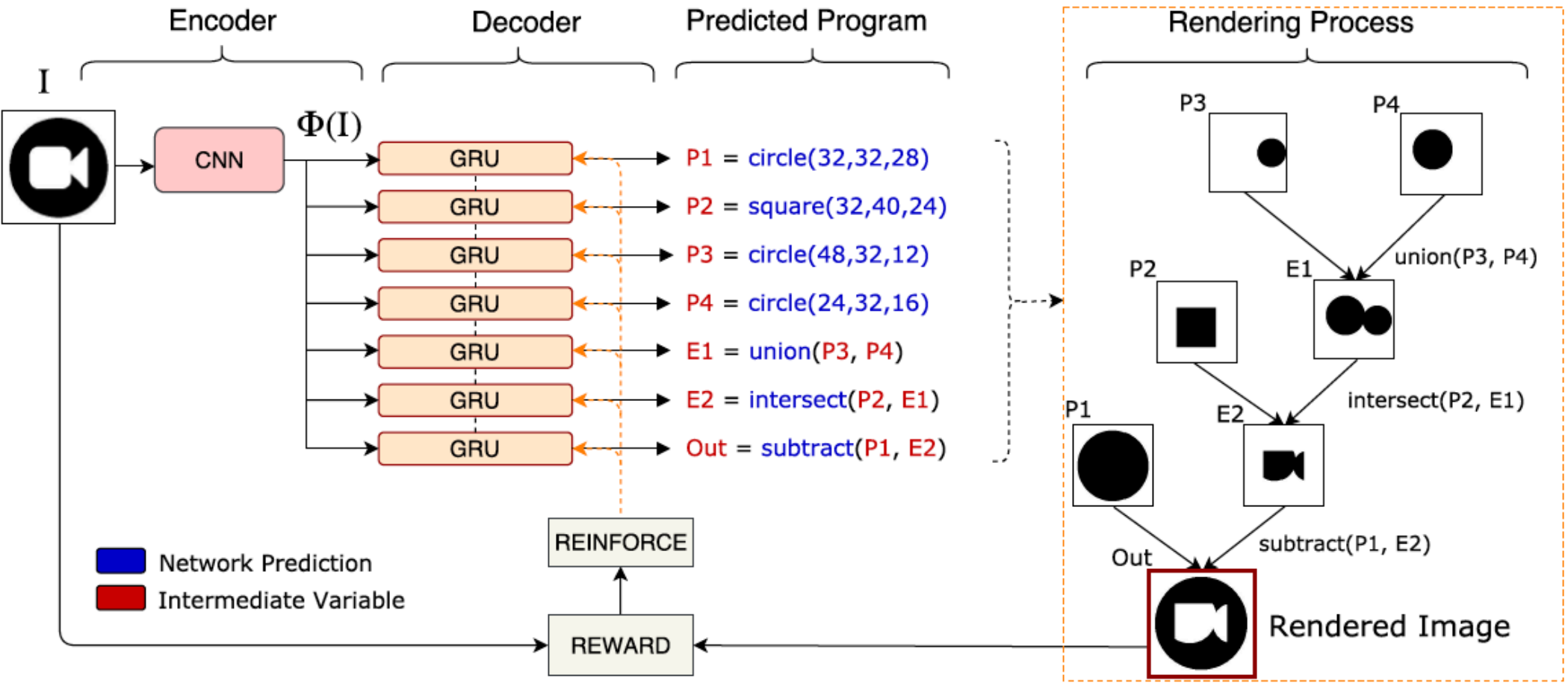


Figure 2. **Architecture of our neural shape parser (CSGNet).** CSGNet consists of three parts, first an encoder takes a shape (2D or 3D) as input and outputs a feature vector through a CNN. Second, a decoder maps these features to a sequence of modeling instructions yielding a visual program. Third, the rendering engine processes the program and outputs the final shape. The primitives annotated as $P1, P2, P3, P4$ are predicted by the network, while $E1, E2$ are the outputs of boolean modeling operations acting on intermediate shapes.

Sharma et al 18

The form of the parses

For example, in constructive solid geometry the instructions consist of drawing primitives (*e.g.*, spheres, cubes, cylinders, *etc.*) and performing boolean operations described as a grammar with the following production rules:

$$S \rightarrow E$$
$$E \rightarrow E\ E\ T \mid P$$
$$T \rightarrow \texttt{OP}_1 | \texttt{OP}_2 | \ldots | \texttt{OP}_m$$
$$P \rightarrow \texttt{SHAPE}_1 | \texttt{SHAPE}_2 | \ldots | \texttt{SHAPE}_n$$

Each rule indicates possible derivations of a non-terminal symbol separated by the | symbol. Here S is the start symbol, $\texttt{OP}_i$ is chosen from a set of defined modeling operations and the $\texttt{SHAPE}_i$ is a primitive chosen from a set of basic shapes at different positions, scales, orientations, etc. Instructions can be written in a standard post-fix notation, *e.g.* $\texttt{SHAPE}_1\texttt{SHAPE}_2\texttt{OP}_1\texttt{SHAPE}_3\texttt{OP}_2$. Figure 2 also gives an example of a program predicted by the network, that follows the grammar described above.

All CSG trees are binary (fairly usual), but aren't required to be unbalanced

so

S1 S2 O1 S3 S4 O2 O3

is OK by this grammar.

Note also:

string==tree==program

Sharma et al 18

Parsing

- Parser output is a “categorical distribution over instructions at each time step”
  - I *think* this means
    - we model string S1 S2 O1 etc with autoregressive model
      - derived from network, which gives

$$p(s_i | s_1 \dots s_{i-1}, \text{input})$$

- Learning
  - supervised data: (bad idea, for reasons to follow)
    - ie shape + program
    - maximize likelihood of program
  - unsupervised data:
    - produce program that best encodes shape

$$\mathbb{E}_{I \sim \mathcal{D}} \left[J_\theta(I) \right] = \mathbb{E}_{I \sim \mathcal{D}} \mathbb{E}_{y \sim \pi_\theta(I))} \left[R \right].$$

Encoding shape

- Network produces probability distribution over programs
  - conditioned on input shape
- Reward:
  - similarity between input shape and predicted shape

?? We want a large reward, but we don't want a large chamfer distance?

Reward. The rewards should be primarily designed to encourage visual similarity of the generated program with the target. Visual similarity between two shapes is measured using the Chamfer distance (CD) between points on the edges of each shape. The CD is between two point sets, $\mathbf{x}$ and $\mathbf{y}$, is defined as follows:

$$Ch(\mathbf{x},\mathbf{y}) = \frac{1}{2|\mathbf{x}|}\sum_{x\in\mathbf{x}}\min_{y\in\mathbf{y}}\|x-y\|_2 + \frac{1}{2|\mathbf{y}|}\sum_{y\in\mathbf{y}}\min_{x\in\mathbf{x}}\|x-y\|_2$$

The points are scaled by the image diagonal, thus $Ch(\mathbf{x},\mathbf{y}) \in [0,1]\ \forall \mathbf{x},\mathbf{y}$. The distance can be efficiently computed using distance transforms. In our implementation, we also set a maximum length T for the induced programs to avoid having too long or redundant programs (e.g., repeating the same modeling instructions over and over again). We then define the reward as:

$$R = \begin{cases} f\left(Ch(\texttt{Edge}(I), \texttt{Edge}(Z(y)\right), & y \text{ is valid} \\ 0, & y \text{ is invalid.} \end{cases}$$

Sharma et al 18

Shaping function

where f is a shaping function and Z is the CSG rendering engine. Since invalid programs get zero reward, the maximum length constraint on the programs encourages the network to produce shorter programs with high rewards. We use maximum length $T = 13$ in all of our RL experiments. The function f shapes the CD as $f(x) = (1 - x)^{\gamma}$ with an exponent $\gamma > 0$. Higher values of γ encourages CD close to zero. We found that $\gamma = 20$ provides a good trade-off between program length and visual similarity.

Sharma et al 18

Minimizing loss

- Assert:
  - can't use gradient descent, because
    - "output space is discrete"
    - "execution engines are typically not differentiable"
- I have trouble buying this
- Consequence:
  - use reinforcement learning, reward as above
- Inference
  - beam search (MAP program is usually intractable)

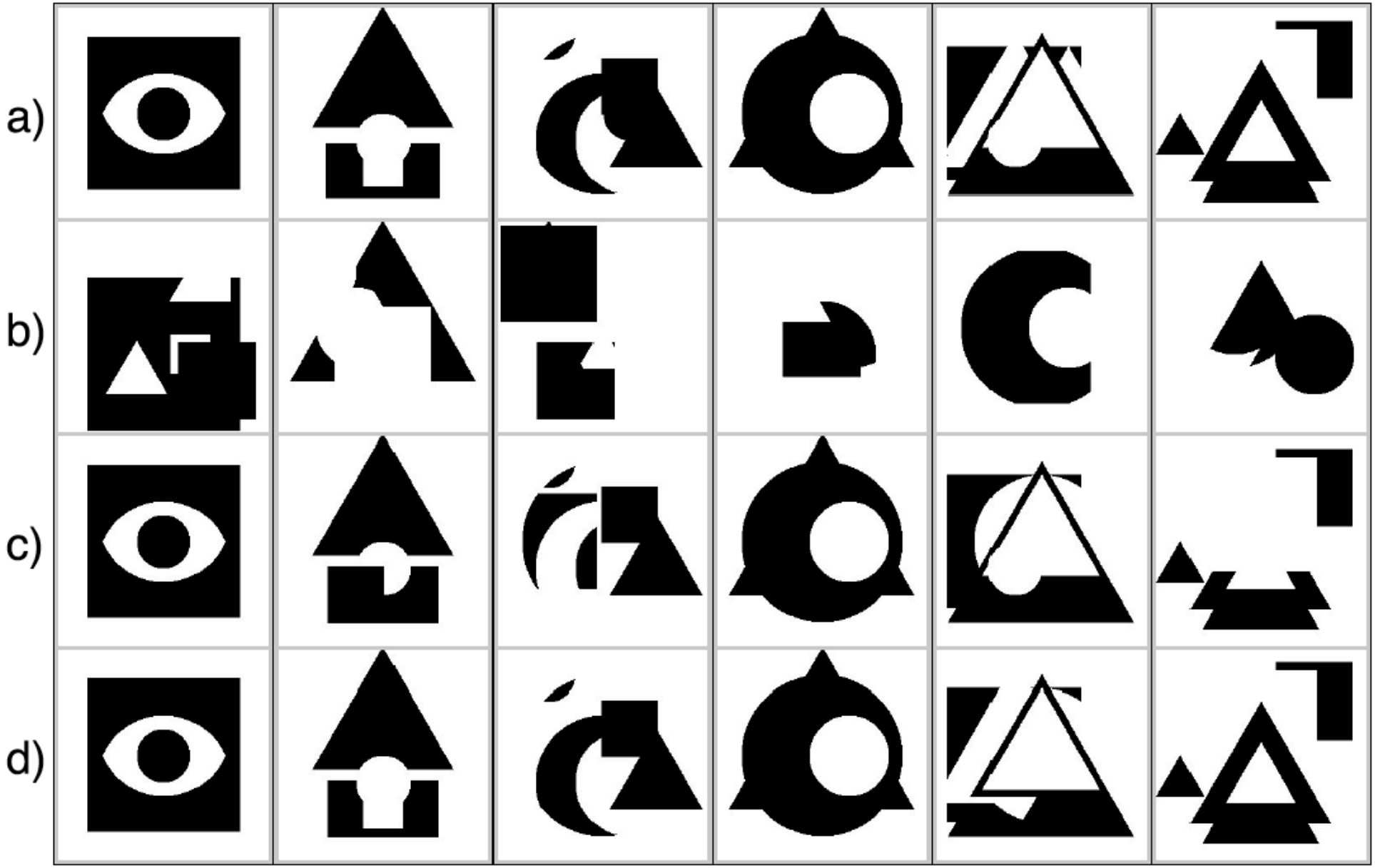


Figure 4. **Comparison of performance on synthetic 2D dataset.** a) Input image, b) NN-retrieved image, c) top-1 prediction, and d) best result from top-10 beam search predictions of CSGNet.

Sharma et al 18

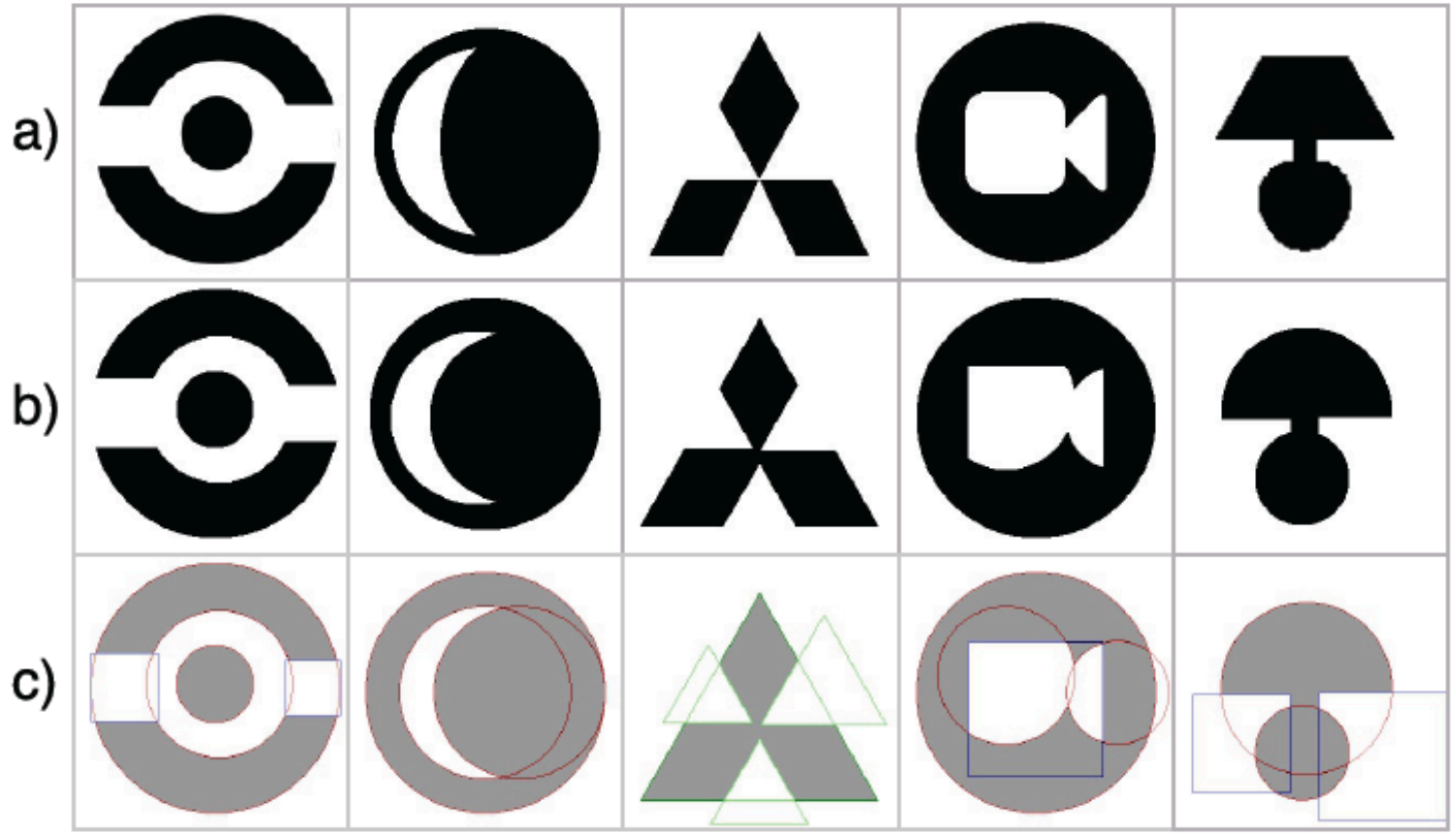


Figure 6. **Results for our logo dataset**. a) Target logos, b) output shapes from CSGNet and c) inferred primitives from output program. Circle primitives are shown with red outlines, triangles with green and squares with blue.

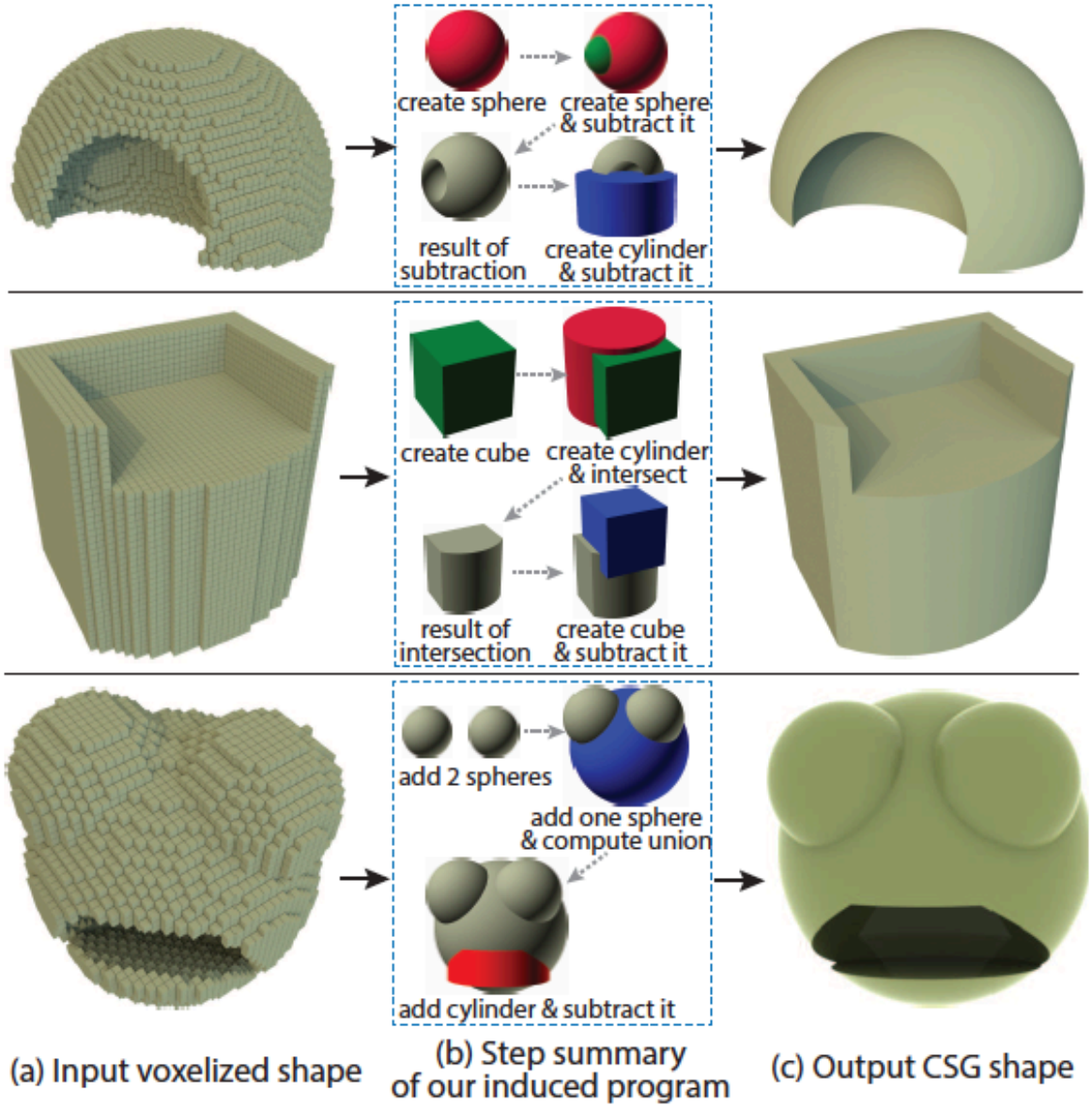


Figure 7. **Qualitative performance of 3D-CSGNet.** a) Input voxelized shape, b) Summarization of the steps of the program induced by CSGNet in the form of intermediate shapes, c) Final output created by executing induced program.

Sharma et al 18

Notes and queries

- Supervised learning is dangerous
  - same shape can have two parse trees implies
  - same input to network must have high probability at two distinct outputs
    - asking for trouble
- A CSG parse more important than the right CSG parse
- What is this for?
  - very useful for reverse engineering (perhaps)
  - do we need the combinatorial structure for (say) shape matching?
  - CF Sharma et al with Paschalidou et al; Tulsiani et al
- Could one do shape completion like this?
- What about a pure shape generator?
  - ie build an unconditional model using conditional examples

One form of shape generator

- Generating indoor scenes (as above, Ritchie et al)
- BUT
  - parse examples to make codes
  - build generator to produce new codes
  - decode these into parse trees
    - then indoor scenes

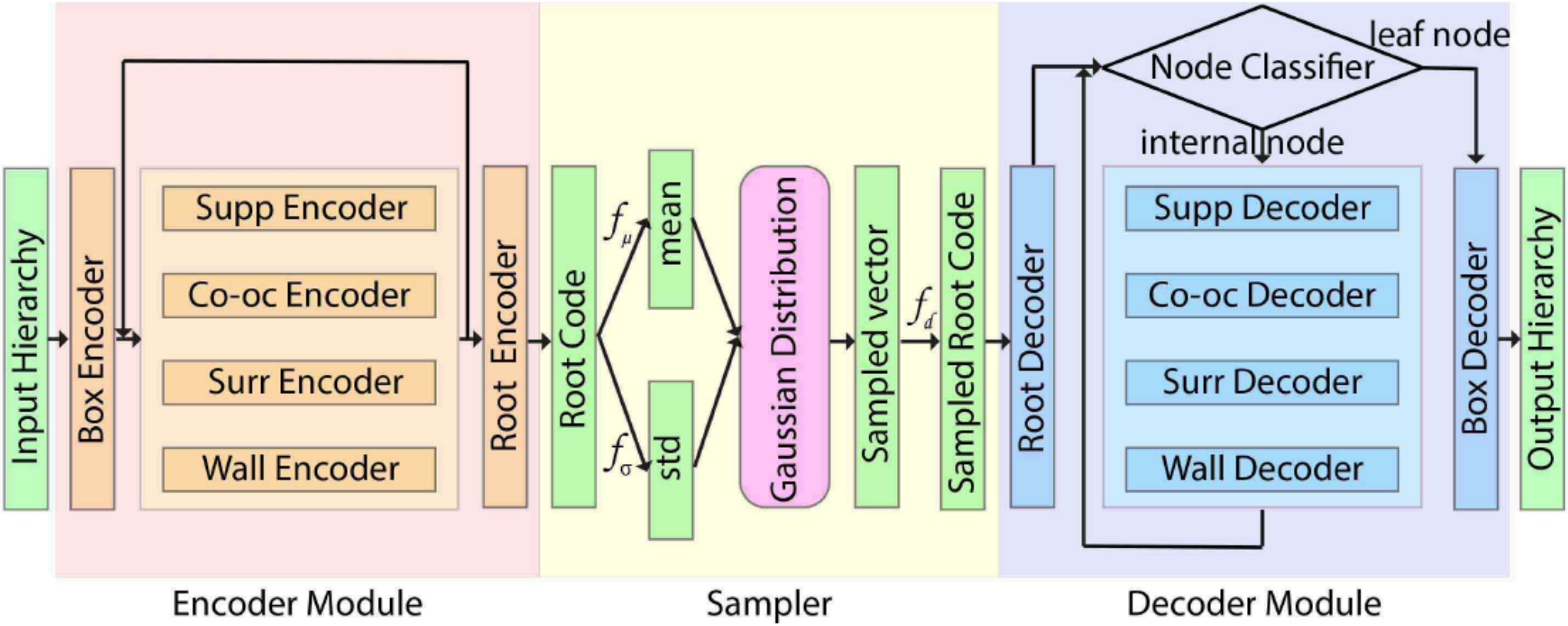


Fig. 2. Architecture of our RvNN-VAE, which is trained to learn a generative model of indoor scene structures. Input to the network is a scene hierarchy composed of labeled OBBs enclosing 3D scene objects. The boxes are recursively grouped and codified by a set of encoders, resulting in a root code. The root code is approximated by a Gaussian distribution, from which a random vector is drawn and fed to the decoder. The recursive decoder produces an output hierarchy to minimize a reconstruction+VAE loss.

Li et al 19

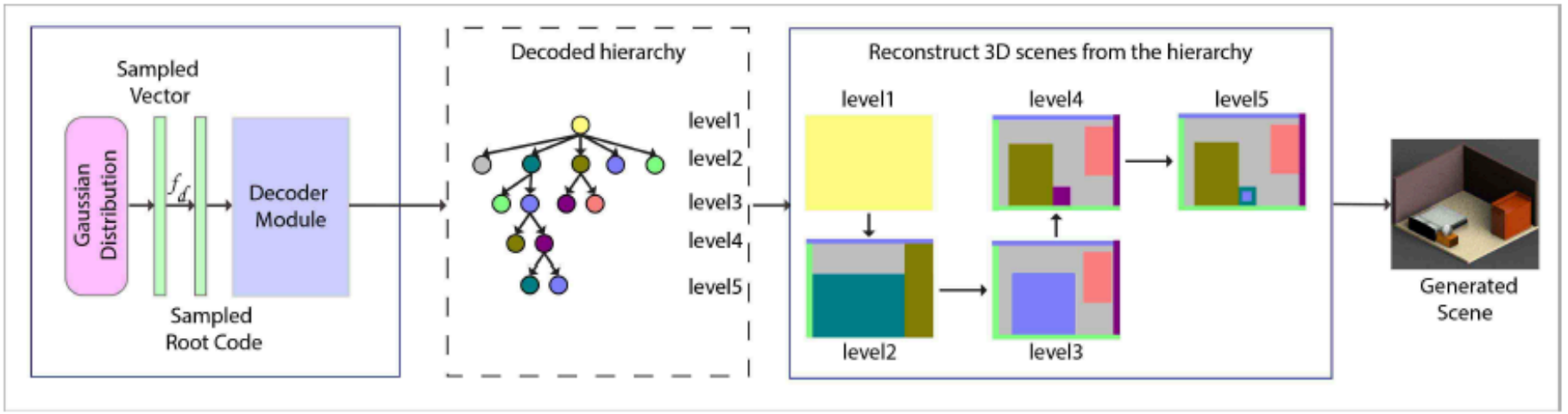


Fig. 3. Overall pipeline of our scene generation. The decoder of our trained RvNN-VAE turns a randomly sampled code from the learned distribution into a plausible indoor scene hierarchy composed of OBBs with semantic labels. The labeled OBBs are used to retrieve 3D objects to form the final 3D scene.

Li et al 19

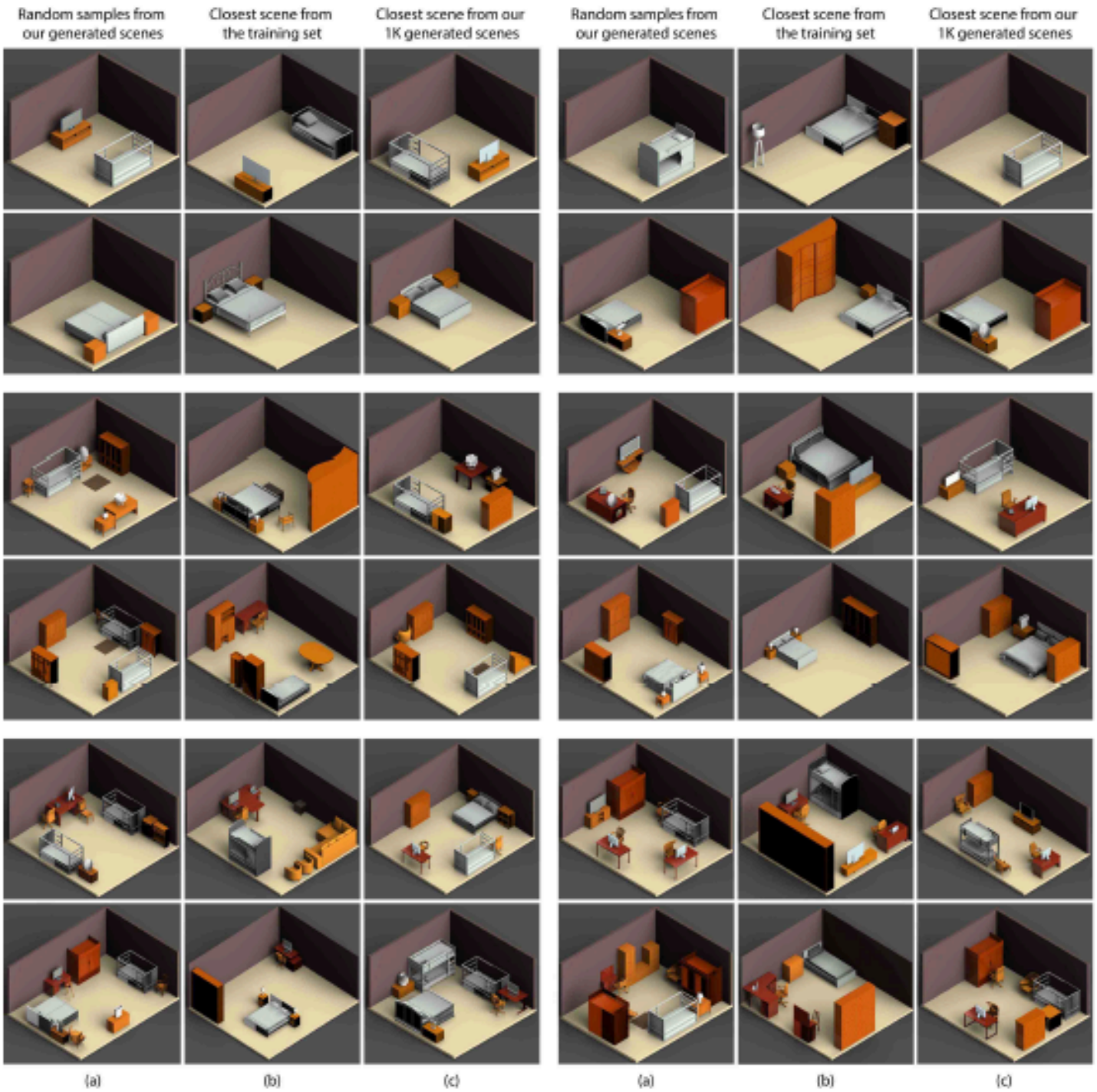


Fig. 11. Bedrooms generated by our method (a), in comparison to (b) closest scene from the training set, to show novelty, and to (c) closest scene from among 1,000 generated results, to show diversity. Different rows show generated scenes at varying complexity, i.e., object counts.

Li et al 19

End to end shape programs

- Idea:
  - build an “execution engine” that is differentiable
  - train together with parser
- Parser:
  - LSTMs
    - block LSTM to identify blocks of program
    - step LSTM - contents of blocks
- Execution engine
  - LSTM+CNN

Program	$\rightarrow$	Statement; Program
Statement	$\rightarrow$	`Draw`(Semantics, Shape, Position_Params, Geometry_Params)
Statement	$\rightarrow$	`For`(For_Params); Program; `EndFor`
Semantics	$\rightarrow$	semantics 1 | semantics 2 | semantics 3 | ...
Shape	$\rightarrow$	Cuboid | Cylinder | Rectangle | Circle | Line | ...
Position_Params	$\rightarrow$	(x, y, z)
Geometry_Params	$\rightarrow$	$(g_1, g_2, g_3, g_4, ...)$
For_Params	$\rightarrow$	Translation_Params | Rotation_Params
Translation_Params	$\rightarrow$	(times i, orientation u)
Rotation_Params	$\rightarrow$	(times i, angle θ, axis a)

Table 1: The domain specific language (DSL) for 3D shapes. Semantics depends on the types of objects that are modeled, i.e., semantics for *vehicle* and *furniture* should be different. For details of DSL in our experimental setting, please refer to supplementary.

Tian et al 19

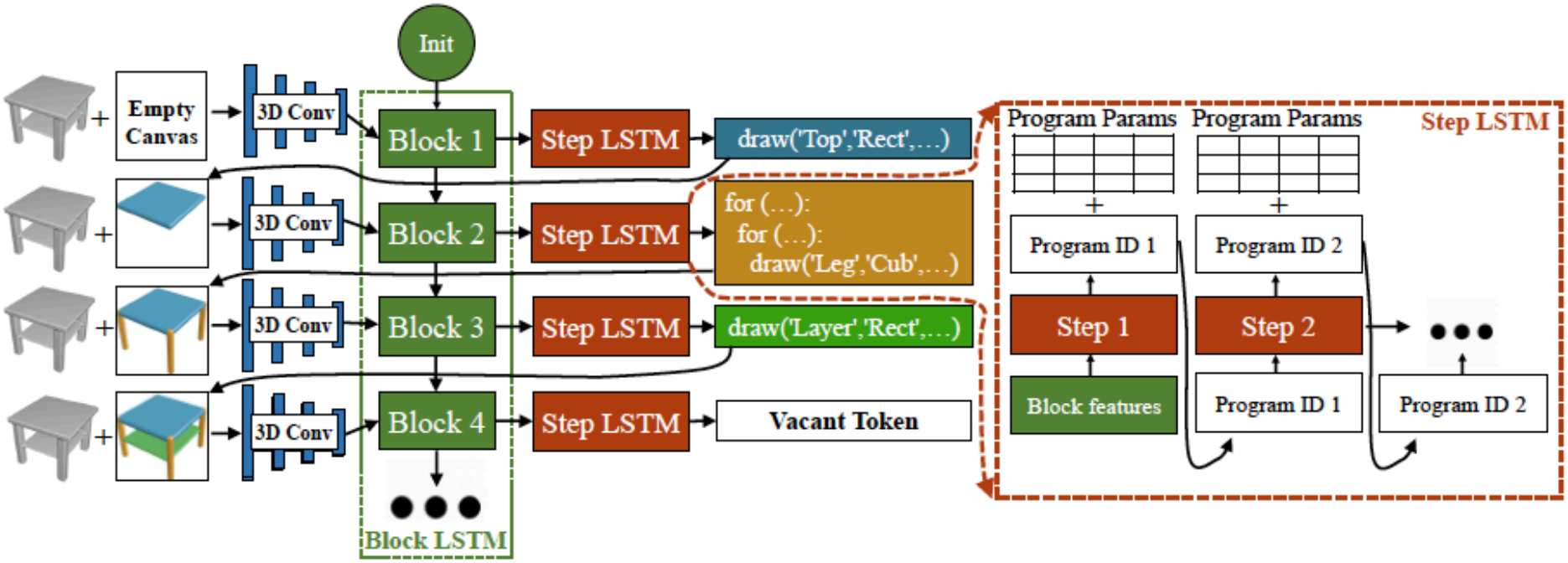


Figure 2: The core of our 3D shape program generator are two LSTMs. The Block LSTM emits features for each program block. The Step LSTM takes these features as input and outputs programs inside each block, which includes either a single drawing statement or compound statements.

Tian et al 19

4.2 NEURAL PROGRAM EXECUTOR

We propose to learn a neural program executor, an approximate but differentiable graphics engine, which generates a shape from a program. The program executor can then be used for training the program generator by back-propagating gradients. An alternative is to design a graphics engine that explicitly executes a symbolic program to produce a voxelized 3D shape. Certain high-level program commands, such as `For` statements, will make the executor non-differentiable. Our use of a neural, differentiable executor enables gradient-based fine-tuning of the program synthesizer on unannotated shapes, which allows the model to generalize effectively to novel shapes outside training categories.

Learning to execute a long sequence of programs is difficult, since an executor has to learn to interpret not only single statements but also complex combinations of multiple statements. We decompose the problem by learning an executor that executes programs at the block level, e.g., either a single drawing statement or a compound statements. Afterwards, we integrate these block-level shapes by max-pooling to form the shape corresponding to a long sequence of programs. Our neural program executor includes an LSTM followed by a deconv CNN, as shown in Figure 3. The LSTM aggregates the block-level program into a fixed-length representation. The following deconv CNN takes this representation and generates the desired shape.

To train the program executor, we synthesize large amounts of block-level programs and their corresponding shapes. During training, we minimize the sum of the weighted binary cross-entropy losses over all voxels via

$$\mathcal{L} = \sum_{v \in V} -w_1 y_v \log \hat{y}_v - w_0 (1 - y_v) \log(1 - \hat{y}_v), \tag{1}$$

where v is a single voxel of the whole voxel space V, y_v and $\hat{y}_v$ are the ground truth and prediction, respectively, while w_0 and w_1 balance the losses between vacant and occupied voxels. This training leverages only synthetic data, not annotated shape and program pairs, which is a blessing of our disentangled representation.

Tian et al 19

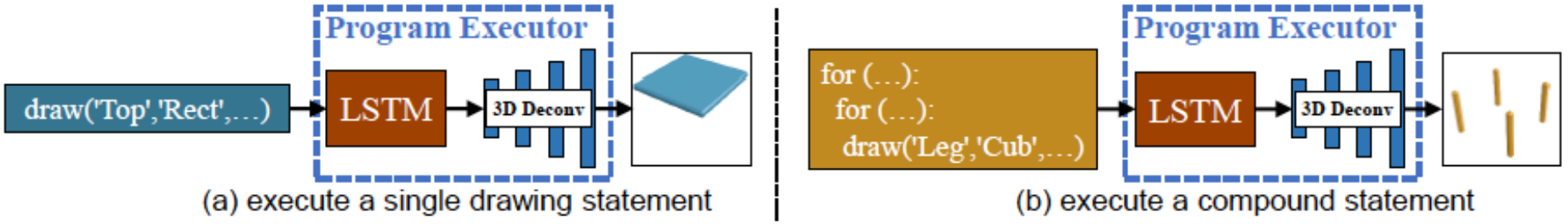


Figure 3: The learned program executor consits of an LSTM, which encodes multiple steps of programs, and a subsequent 3D DeconvNet which decodes the features to a 3D shape.

This is what the blocks are about?

Tian et al 19

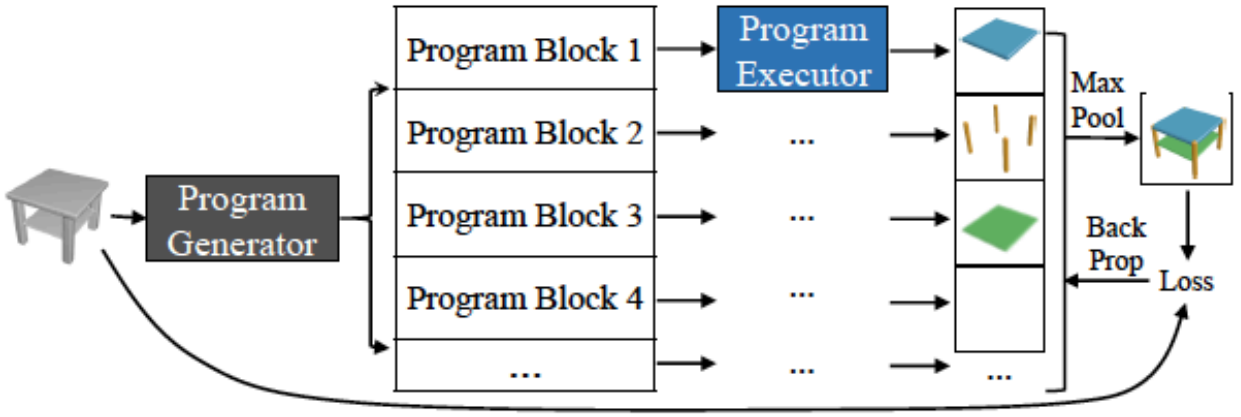


Figure 4: Given an input 3D shape, the neural program executor executes the generated programs. Errors between the rendered shape and the raw input are back-propagated.

Learn executor, freeze, learn parser

4.3 GUIDED ADAPTATION

A program generator trained only on a synthetic dataset does not generalize well to real-world datasets. With the learned differentiable neural program executor, we can adapt our model to other datasets such as ShapeNet, where program-level supervision is not available. We execute the predicted program by the learned neural program executor and compute the reconstruction loss between the generated shape and the input. Afterwards, the program generator is updated by the gradient back-propagated from the learned program executor, whose weights are frozen.

Tian et al 19

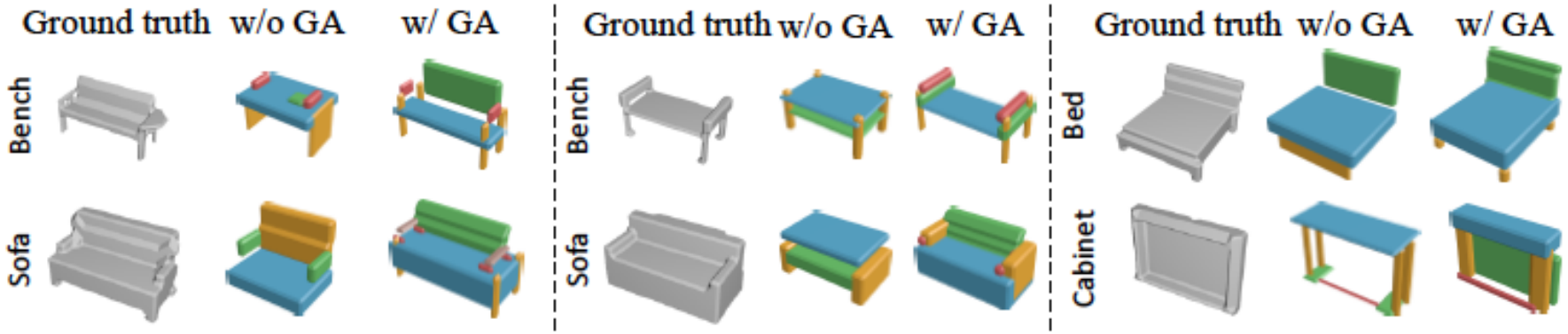


Figure 6: ShapeNet objects from unseen categories reconstructed with shape programs before and after *guided adaptation*. Shape Programs can learn to adapt and explain objects from novel classes.

Primitives yield powerful smoothers

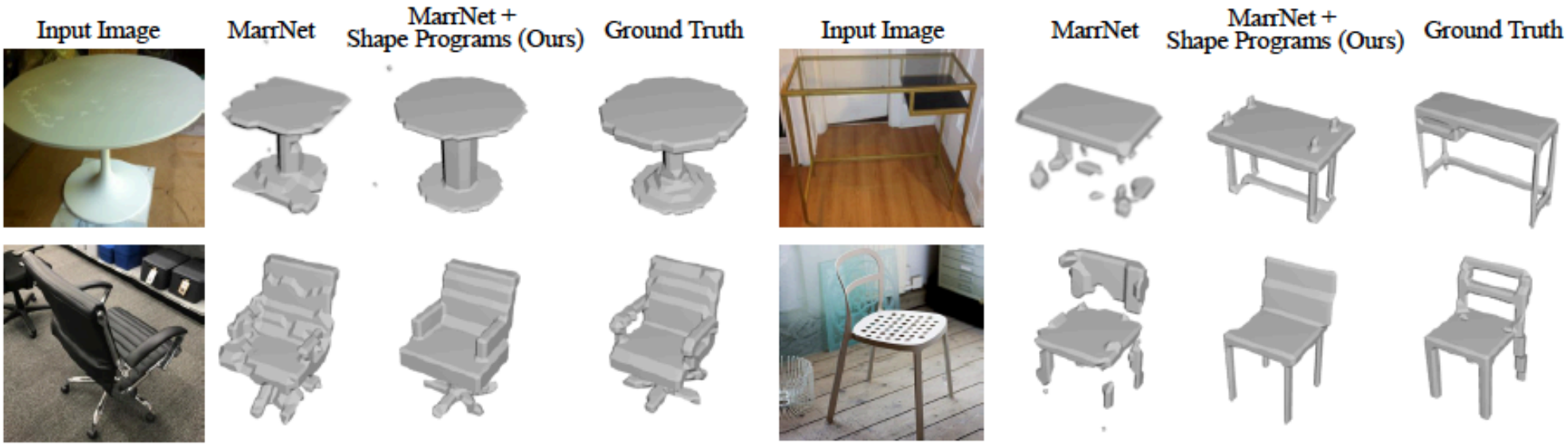


Figure 7: 3D reconstruction results on Pix3D dataset. MarrNet generates fragmentary shapes and our model further smooths and completes such shapes.

Tian et al 19

Notes and queries

- Power of primitives:
  - smoothing
  - compact representation
- Combinatorial structures are different but comparable
  - Tian vs Sharma vs Li
  - All use recurrent models (in different ways)
- What is the combinatorial structure for?
  - does it simplify representation of distributions?
  - why not regard as pointnet?