LayoutVAE: Stochastic Scene Layout Generation From a Label Set – Supplementary Material

Akash Abdu Jyothi[1,3], Thibaut Durand[1,3], Jiawei He[1,3], Leonid Sigal[2,3], Greg Mori[1,3]
[1]Simon Fraser University [2]University of British Columbia [3]Borealis AI
{aabdujyo, tdurand, jha203}@sfu.ca lsigal@cs.ubc.ca mori@cs.sfu.ca

A. Graphical Model

For each of CountVAE and BBoxVAE, we have a conditional VAE that is autoregressive (Figure 8), where the conditioning variable contains all the information required for each step of generation. By explicitly designing the conditioning variable this way, we forgo using past latent codes for generation.

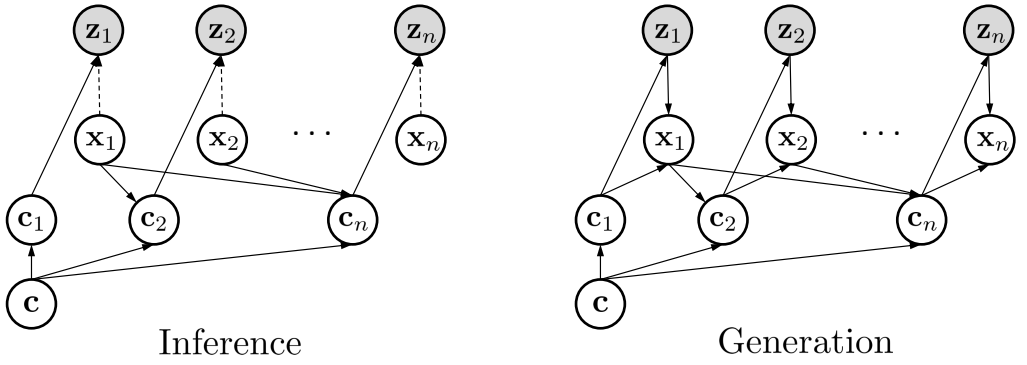

Figure 8: **Graphical model for LayoutVAE.** $\mathbf{c}$ denotes context (label set for CountVAE, label set with counts for BBoxVAE) and $\mathbf{c}_k$ denotes the conditioning information for the CVAE. Dashed arrow denotes inference of the approximate posterior.

B. Model Architecture

B.1. CountVAE

We represent the label set $\mathbb{L}$ by a multi-label vector $\mathbf{s} \in \{0,1\}^M$, where $\mathbf{s}_k = 1$ (resp. 0) means the k-th category is present (resp. absent). For each step of CountVAE, the current label k is represented by a one-hot vector denoted as $\mathbf{l}^k \in \{0,1\}^M$. We represent the count information n_k of category k as a M dimensional one-hot vector $\mathbf{y}^k$ with the non-zero location filled with the count value. This representation of count captures its label information as well. We perform pooling over a set of previously predicted label counts by summing up the vectors. The conditioning input is

$$\mathbf{c}_k^c = \mathrm{FC}\left(\left[\mathrm{MLP}(\mathbf{s}), \mathrm{MLP}(\mathbf{l}^k), \mathrm{MLP}\left(\sum_{m<k} \mathbf{y}^k\right)\right]\right) \tag{14}$$

where $[\cdot,\cdot]$ denotes concatenation, FC a fully connected layer and MLP a generic multi-layer perceptron. Figure 9 shows the architecture in detail.

B.2. BBoxVAE

We represent label and count information pair for each category $\{n_m : m \in \mathbb{L}\}$ as $\mathbf{y}^m$ using the same strategy as in CountVAE. Pooled representation of the label set along with counts is obtained by summing up these vectors to obtain a

multi-label vector. For each step of BBoxVAE, the current label k is represented by a one-hot vector denoted as $\mathbf{l}^k$. We use LSTM for pooling previously predicted bounding boxes $\mathbb{B}_{k,j}^{prev}$. We represent each bounding box as a vector of size 4, and we concatenate M dimensional label vector to add label information to it. We pass $M+4$ dimensional vectors of successive bounding boxes through an LSTM and use the final step output as the pooled representation. Figure 11 shows the pooling operation, and Figure 10 shows the detailed architecture for each module in BBoxVAE. The conditioning input is

$$\mathbf{c}_{k,j}^{b} = \text{FC}\left(\left[\text{MLP}\left(\sum_{m\in\mathbb{L}} \mathbf{y}^m\right), \text{MLP}(\mathbf{l}^k), \text{MLP}(\mathbb{B}_{k,j}^{prev})\right]\right). \tag{15}$$

BBoxVAE predicts the mean for the quadrivariate Gaussian (Equation 12), while covariance is assumed to be a diagonal matrix with each value of standard deviation equal to 0.02.

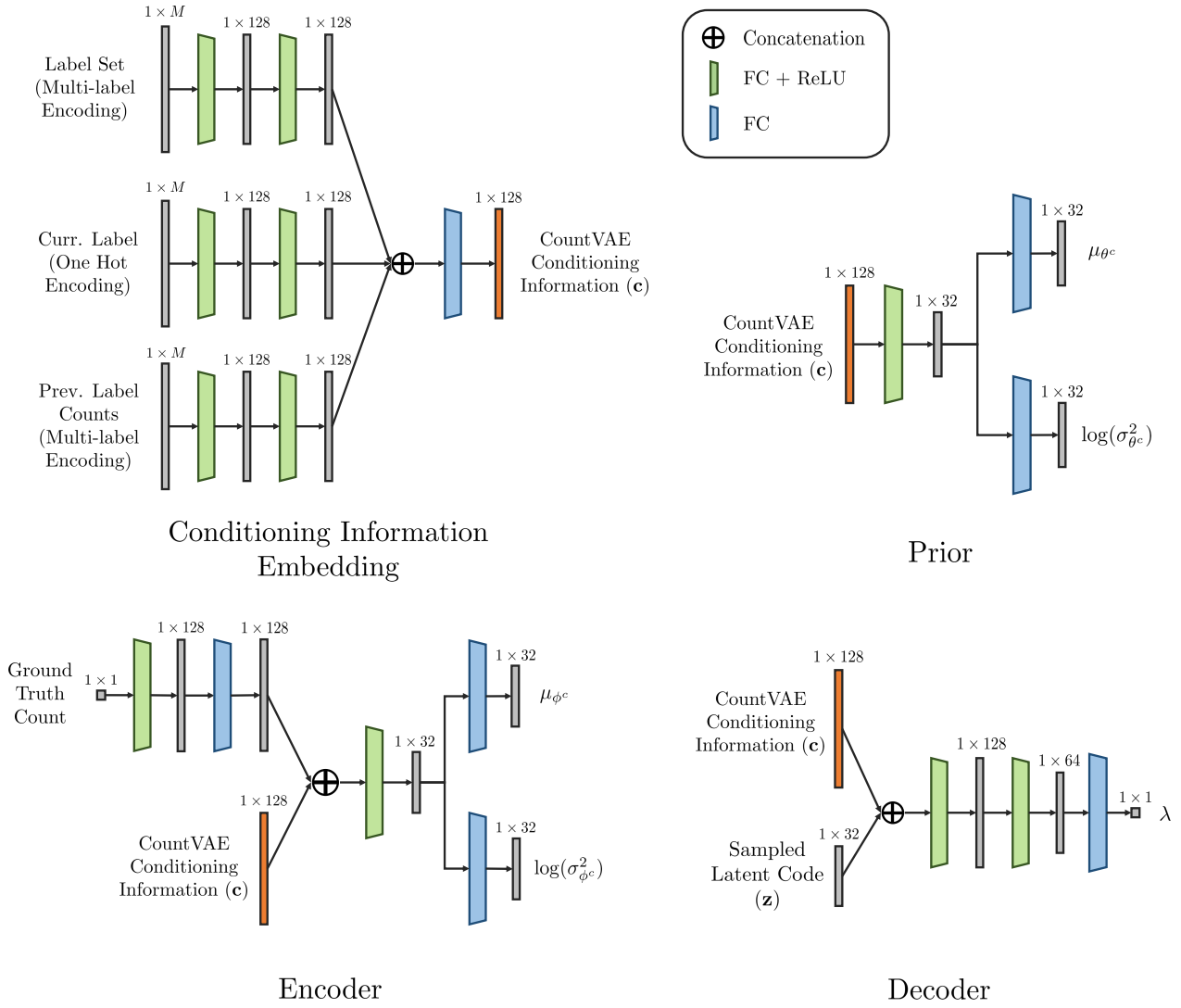

Figure 9: **CountVAE Architecture.**

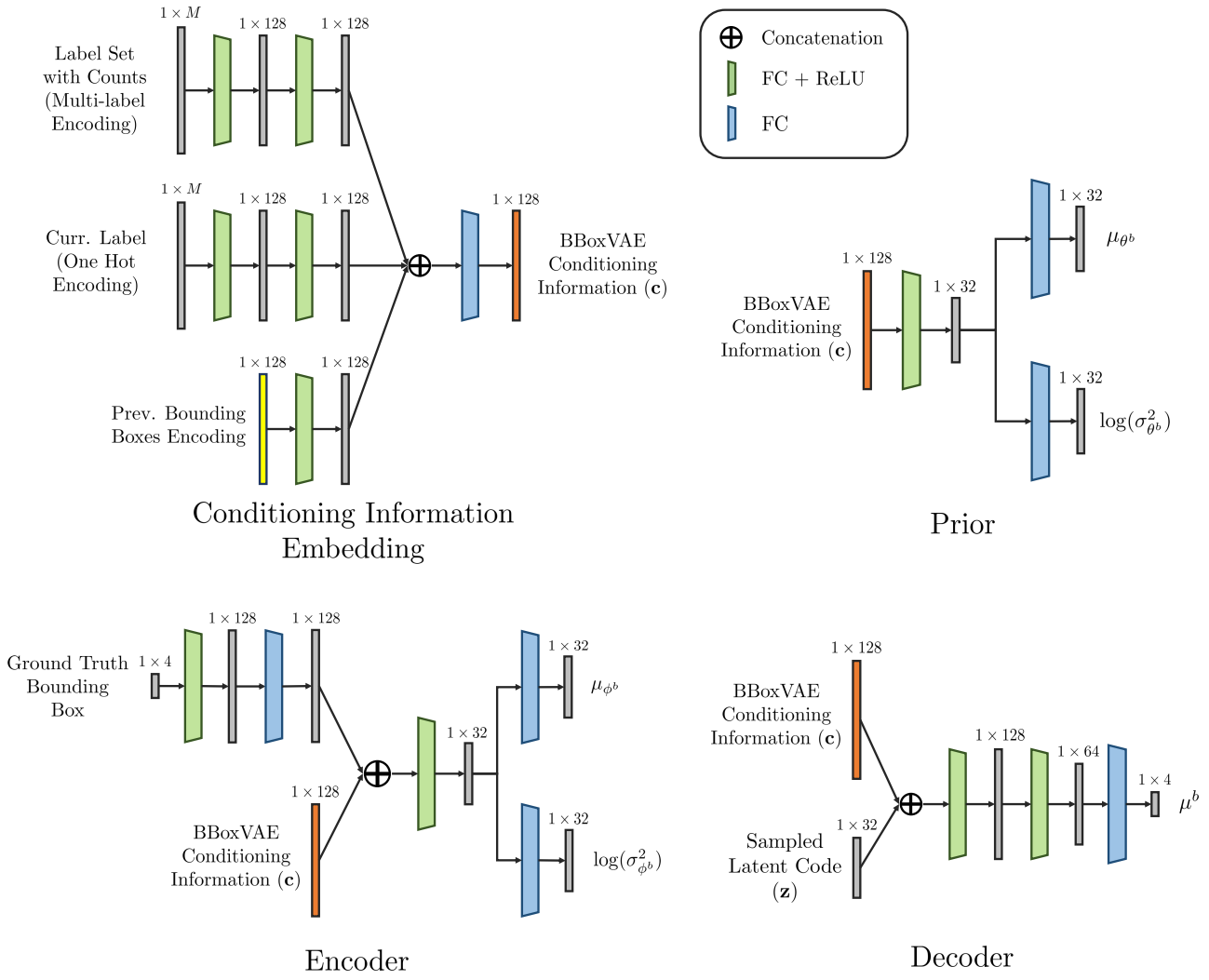

Figure 10: **BBoxVAE Architecture.**

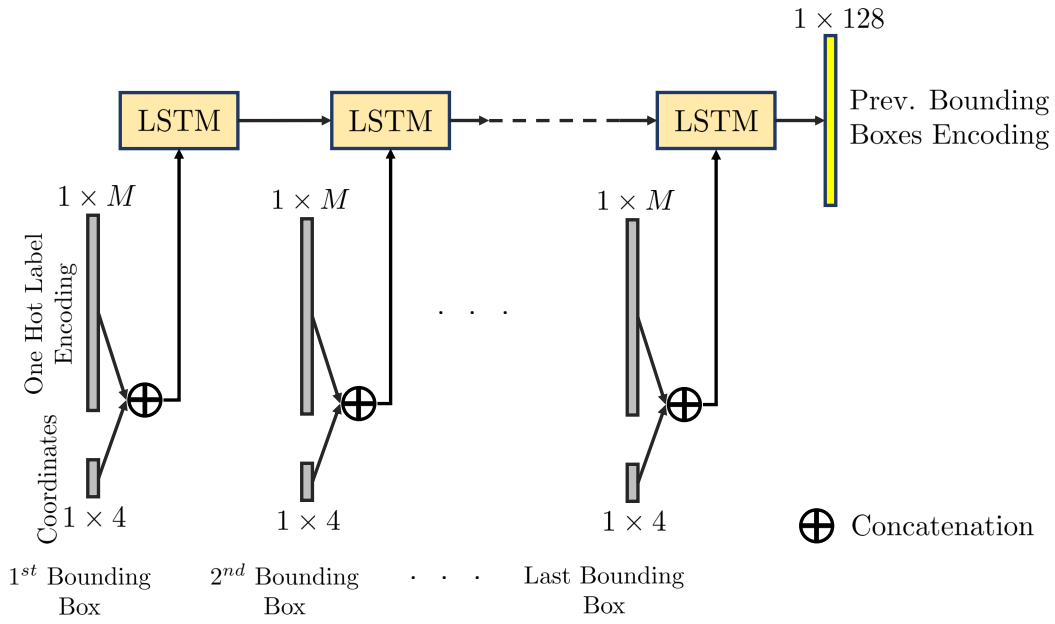

Figure 11: **LSTM pooling for bounding boxes.** We use an LSTM to pool the set of previously predicted bounding boxes to be used in the conditioning information for BBoxVAE.

C. Experiments

C.1. MNIST-Layouts dataset

Table 5 shows the rules used to generate the dataset. To generate the layouts, we adapted the code provided at https://github.com/aakhundov/tf-attend-infer-repeat.

Label	Count	Location	Size
1	3,4	top	medium
2	2,3	middle	large
3	1,2	bottom	small-medium
4 (2 present)	count(2)+3,6	around a 2	small
4 (2 absent)	2	bottom-right	small

Table 5: **Rules for generating MNIST-Layouts dataset.** Given a label set, we use uniform distribution over the possible count values to generate count. We then sample over a uniform distribution over the location and size ranges (precise details skipped in the table for brevity) to generate bounding boxes for each label instance. When label 4 is present in the input label along with label 2 (4^{th} row in the table), we randomly choose an instance of 2 and place all the 4s around that.

C.2. Analysis of latent code size

In Table 6, we analyze the dimension of the latent space for both CountVAE and BBoxVAE. For each dimension we report the NLL performance on COCO dataset. We observe that both models have good performance when the latent code size is between 32 and 128, and the models are not sensitive to this hyperparameter. Increasing the latent space beyond 128 does not improve the performances.

Latent Code Size	CountVAE	BBoxVAE
2	0.569	4.13
4	0.568	3.11
8	0.565	2.91
16	0.564	2.72
32	**0.562**	2.72
64	0.563	**2.69**
128	**0.562**	2.70

Table 6: **Effect of latent code size.** Average NLL over COCO test set for CountVAE and BBoxVAE while varying the size of the latent code.

C.3. Detecting unlikely layouts

In this section, we present more examples from our experiment on detecting unlikely layouts by flipping the original layout and computing likelihood under LayoutVAE. In Figure 12, we present some typical examples where likelihood under LayoutVAE (BBoxVAE, to be precise, since CountVAE gives the same result for original and flipped layouts as the label counts remain the same) decreases when flipped upside down. This behaviour was observed for 92.58% samples in the test set. In Figure 13, we present some examples of unusual layouts where likelihood under LayoutVAE increases when flipped upside down.

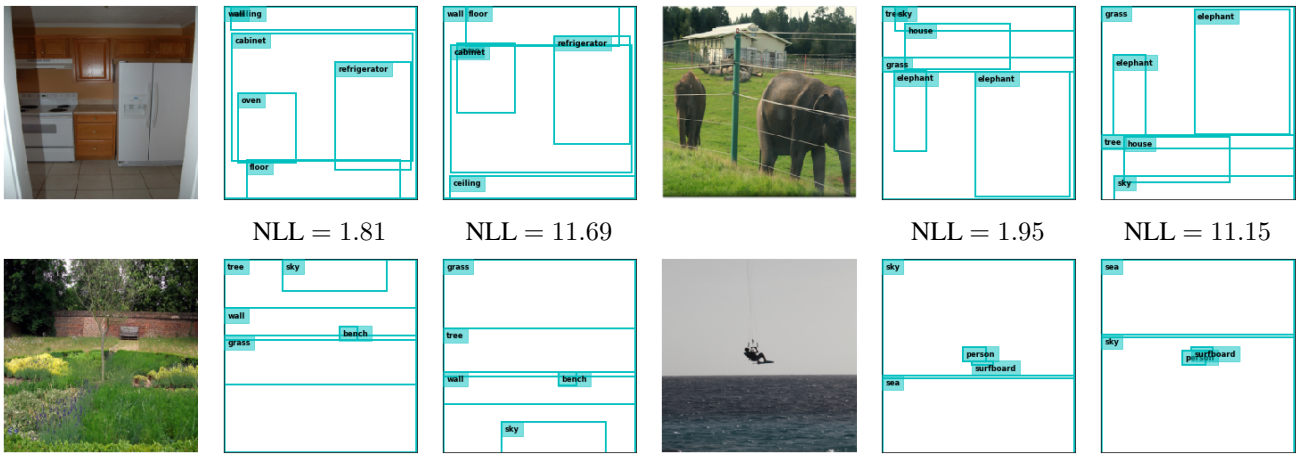

Figure 12: **Some examples where likelihood under BBoxVAE decreases when flipped upside down.** We show the test image, layout for the image and the flipped layout. Negative log likelihood(NLL) of the layout under BBoxVAE is shown along with each layout. We can see that the flipped layout is highly unlikely in these examples.

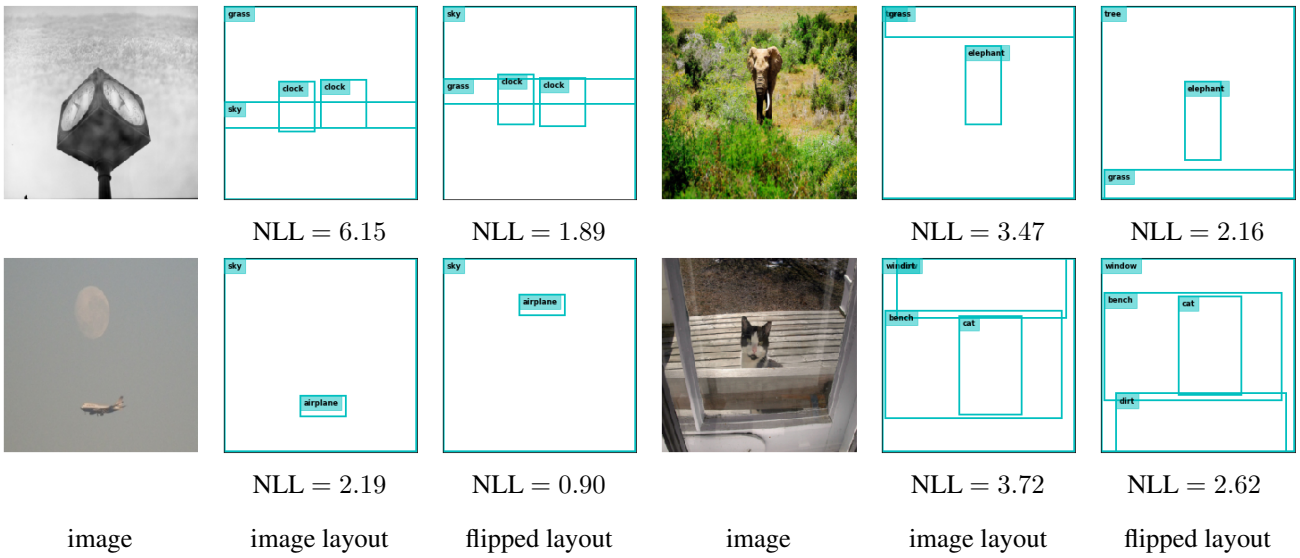

Figure 13: **Some examples where likelihood under BBoxVAE increases when flipped upside down.** We show the test image, layout for the image and the flipped layout. Negative log likelihood(NLL) of the layout under BBoxVAE is shown along with each layout. We can see that the flipped layout is equally or sometimes more plausible in these examples.

C.4. Examples for Layout Generation

Figure 14 shows examples of diverse layouts generated using LayoutVAE.

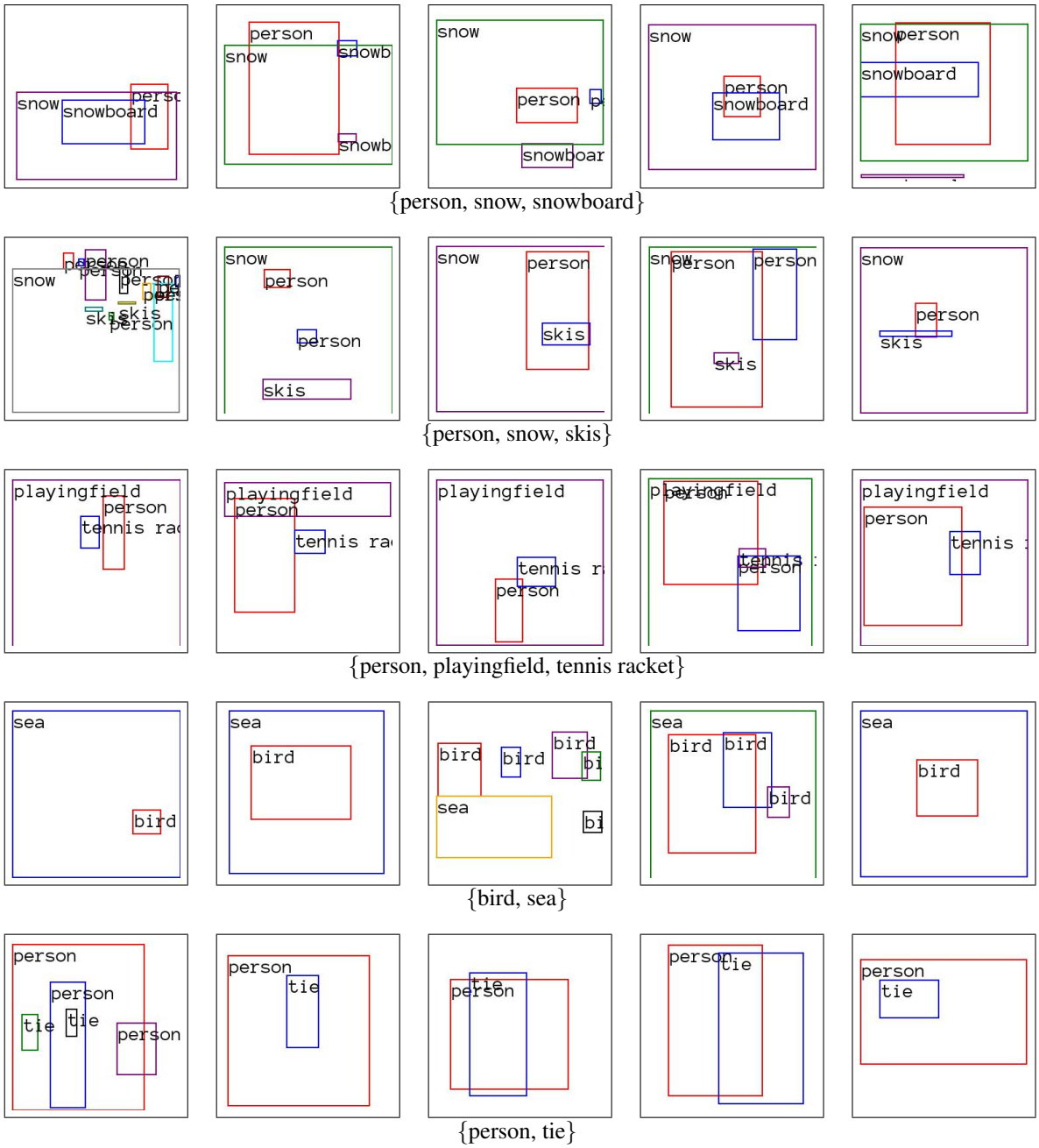

Figure 14: **Layout generation using LayoutVAE.** We show 5 randomly sampled layouts for each input label set.

C.5. Examples for Bounding Box Generation

Figure 15 presents examples that showcase the ability of LayoutVAE to use conditioning information to predict plausible bounding boxes. We present additional examples of stochastic bounding box generation for test samples that have labels *person*, *surfboard* and *sea* in Figure 16 and Figure 17.

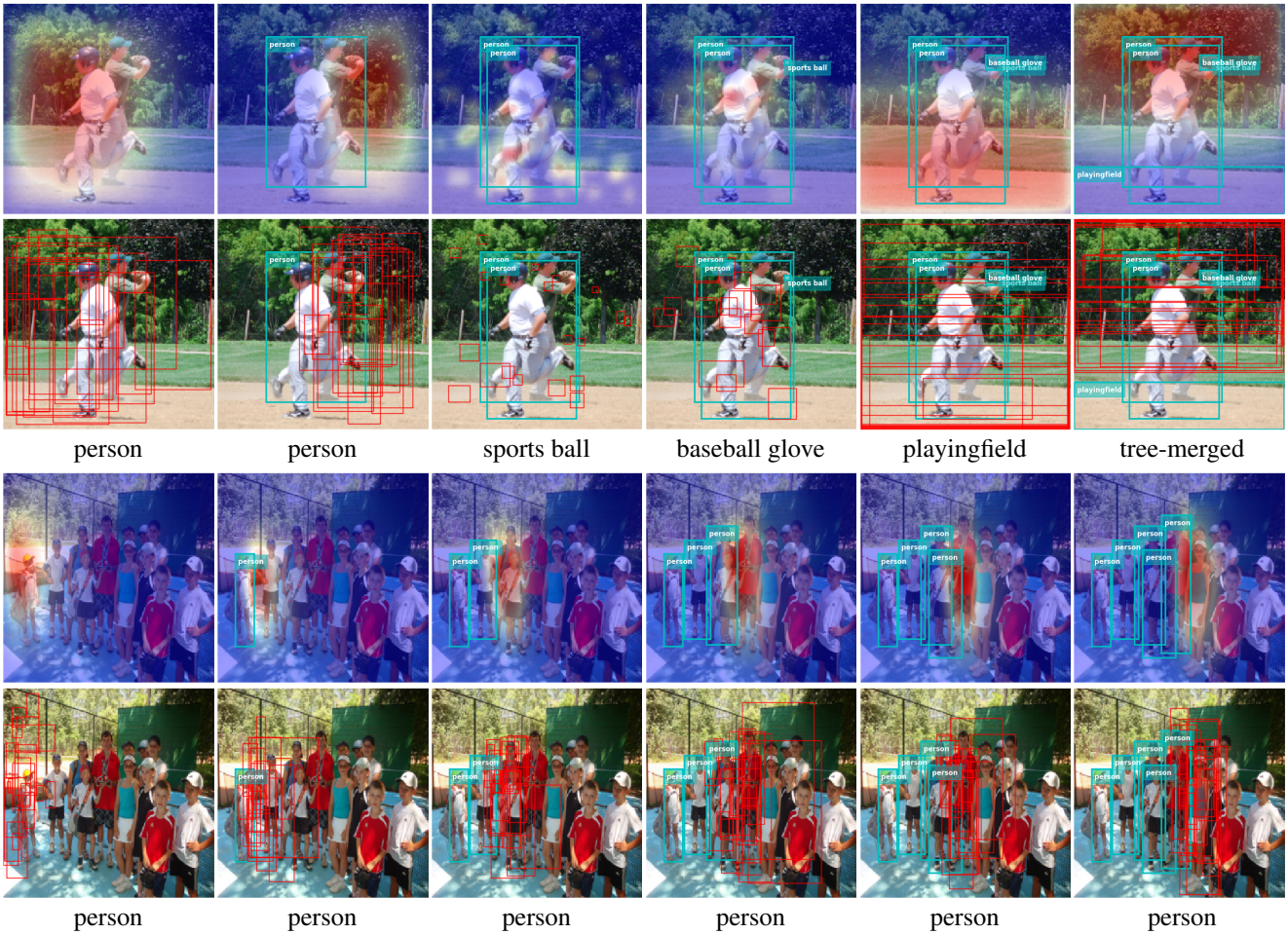

Figure 15: **Importance of conditioning information.** We present two examples here both of which have *person* as their first label. We see that bounding boxes for the first label *person* (first column) are close to the center in the first example, whereas they are smaller and close to the left side of the scene in the second example. This is because BBoxVAE gets count of *persons* (2 for the first example, and 12 for the second examples) in its conditioning information, which prompts the model to predict bounding boxes appropriately.

Figure 16: **Test examples with labels person, surfboard, sea.** We show steps of bounding box generation for test set samples in the same manner as in Figure 7 from the main paper.

Figure 17: **An example with more objects in the scene.** We show steps of bounding box generation for test set samples in the same manner as in Figure 7 from the main paper.