Supplementary Material for “Spatially Invariant Unsupervised Object Detection with Convolutional Neural Networks”

A Prior on z_{pres}

Here we provide a derivation for the prior on z_{pres} (a vector of length HW made up of all z^{ij}_{pres}), discussed briefly in Section 4.2. Let C be a random variable giving the number of non-zero entries in z_{pres}. Under the prior, z_{pres} is generated by first sampling C, and then drawing z_{pres} uniformly from all binary vectors that have C non-zero entries:

$$p(z_{\text{pres}}) = p(C = nz(z_{\text{pres}}))p(z_{\text{pres}}|C = nz(z_{\text{pres}}))$$

where $nz(\cdot)$ gives the number of non-zero entries in a vector. There are $\binom{HW}{C}$ binary vectors of length HW with C non-zero entries, so:

$$p(z_{\text{pres}}|C = nz(z_{\text{pres}})) = \binom{HW}{nz(z_{\text{pres}})}^{-1}$$

For the distribution over C we use a Geometric distribution with parameter s. We use the Geometric interpretation that puts C as the number of failures before a success, and s as the success probability. By setting s to a high value, we put most of the probability mass at low values of C, thereby putting pressure on the network to explain images using as few objects as possible.

$$p(C = nz(z_{\text{pres}})) = s(1-s)^{nz(z_{\text{pres}})}$$

We then truncate and normalize to ensure C has support $\{0, 1, \dots, HW\}$:

$$\begin{aligned} p(C = nz(z_{\text{pres}})) &= \frac{s(1-s)^{nz(z_{\text{pres}})}}{\sum_{c=0}^{HW} s(1-s)^c} \\ &= \frac{s(1-s)^{nz(z_{\text{pres}})}}{s\frac{(1-(1-s)^{HW+1})}{1-(1-s)}} \\ &= \frac{s(1-s)^{nz(z_{\text{pres}})}}{1-(1-s)^{HW+1}} \end{aligned}$$

Overall, the prior is:

$$p(z_{\text{pres}}) = \frac{s(1-s)^{nz(z_{\text{pres}})}}{(1-(1-s)^{HW+1})\binom{HW}{nz(z_{\text{pres}})}}$$

B KL Divergence for z_{pres}

Having derived the prior, we now turn to efficient computation of the KL divergence between the distribution over z_{pres} yielded by the network, namely $q_\phi(z_{\text{pres}}|x)$ (embodied in the β^{ij}_{pres} variables from Section 4.3) and $p(z_{\text{pres}})$. Efficient computation of this KL divergence is necessary for computing the second term of the VAE training objective (Equation 3). For convenience, we switch to indexing the entries of z_{pres} using a single index k rather than the grid indices ij. We will use the notation $z^{m:n}_{\text{pres}}$ to mean the sub-vector ranging from index m to n, inclusive.

First, from basic properties of KL divergence we have:

$$D_{KL}(q(z_{\text{pres}}|x) \parallel p(z_{\text{pres}})) = \sum_{k=1}^{n} E\left[D_{KL}(q(z^k_{\text{pres}}|z^{1:k-1}_{\text{pres}}, x) \parallel p(z^k_{\text{pres}}|z^{1:k-1}_{\text{pres}}))\right]$$

where the expectation for index k is taken over variables $z^{1:k-1}_{\text{pres}}$ sampled from the marginal $q(z^{1:k-1}_{\text{pres}}|x)$. We estimate each expectation using a single sample:

$$\approx \sum_{k=1}^{n} D_{KL}(q(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}, x) \parallel p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}))$$

where $\hat{z}$ indicates values that have been sampled. Note that $q(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}, x)$ is just Bernoulli(β^k_{pres}) from Section 4.3.

We now turn to computation of $p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}})$. Recalling that C is a random variable giving the number of non-zero entries in z_{pres}, we have:

$$p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}) = \sum_{c=0}^{HW} p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}, C = c)p(C = c|\hat{z}^{1:k-1}_{\text{pres}}) \tag{B1}$$

We focus first on the factor $p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}, C = c)$. Given that we have sampled $\hat{z}^{1:k-1}_{\text{pres}}$, we still need $c - nz(\hat{z}^{1:k-1}_{\text{pres}})$ non-zeros, and we have $HW - (k-1)$ “slots” to get them with. Recalling that given $C = c$, we are assuming uniform sampling over all binary vectors with c non-zeros, we have:

$$p(z^k_{\text{pres}}|\hat{z}^{1:k-1}_{\text{pres}}, C = c) = \frac{c - nz(\hat{z}^{1:k-1}_{\text{pres}})}{HW - (k-1)}$$

The final piece that we need is $p(C = c|\hat{z}^{1:k-1}_{\text{pres}})$. This can be decomposed recursively as:

$$p(C = c|\hat{z}^{1:k-1}_{\text{pres}}) \propto p(\hat{z}^{k-1}_{\text{pres}}|C = c, \hat{z}^{1:k-2}_{\text{pres}})p(C = c|\hat{z}^{1:k-2}_{\text{pres}}) \tag{B2}$$

Starting from $k = 1$, we first evaluate Equation (B1), then sample z^k from $q_\phi(z^k_{\text{pres}} | \hat{z}^{1:k-1}_{\text{pres}})$, and finally update the conditional count distribution $p(C = c | \hat{z}^{1:k}_{\text{pres}})$ via Equation (B2).

C Object Rendering in the Decoder Network

Here we describe in detail the differentiable rendering algorithm implemented in the decoder network. For each object we have the following values:

1. Object RGB map o^{ij} with shape $(H_{obj}, W_{obj}, 3)$
2. Object transparency map α^{ij} with shape $(H_{obj}, W_{obj}, 1)$
3. Object bounding box b^{ij} derived from z^{ij}_{where} (see Section 4.1 and Figure 1).
4. Relative depth value z^{ij}_{depth}
5. Presence value z^{ij}_{pres}

We also have a background image x_{bg} with shape $(H_{img}, W_{img}, 3)$, which may have been predicted from the input image by a neural network, or may be a fixed image calculated from the dataset in some other way.

We first perform two initial computations for each object:

$$\tilde{\alpha}^{ij} = \alpha^{ij} z^{ij}_{\text{pres}}$$
$$\gamma^{ij} = \alpha^{ij} z^{ij}_{\text{pres}} \sigma(-z^{ij}_{\text{depth}})$$

where σ is the sigmoid function. $\tilde{\alpha}^{ij}$ is a transparency map that takes into account whether the object exists, γ^{ij} is an *importance* map which is used to implement a differentiable approximation of relative depth.

For each pixel in the output image, we iterate over all objects, checking whether each object affects the current pixel (i.e. whether the pixel is inside the bounding box for the object). For all objects that affect the pixel, three values are extracted from the object: RGB values o', a transparency value α' and an importance value γ'. These values are extracted by finding the position of the pixel with respect to the object's coordinate frame, and then using bilinear interpolation to extract a value from the relevant map. The RGB value o' is then mixed with the background using α'. The mixed values from all affecting objects are then mixed with one another using a convex combination consisting of the γ' values normalized to sum to 1 (within the pixel). This mixing step implements the differentiable approximation of relative depth; objects with higher importance values (lower depth) get a larger "share" of the pixels that they affect, and appear on top of objects with lower importance (higher depth). Pseudo-code for this process is given in Algorithm 1.

Algorithm 1 Differentiable rendering algorithm

```
 1: function CONTAINS($b$, $(y, x)$)
      Return True iff pixel $(y, x)$ is inside bounding box $b$
 2: end function

 3: function INTERPOLATE($b$, $v$, $(y, x)$)
      Assuming pixel $(y, x)$ is inside bounding box $b$, extract
      a value from map $v$ for pixel $(y, x)$ using bilinear inter-
      polation.
 4: end function

 5: function DIFFERENTIABLERENDERING
 6:     $x_{out} \leftarrow$ COPY($x_{bg}$)
 7:     for pixel with position $(y, x)$ do
 8:         n-writes $\leftarrow 0$
 9:         weighted-sum $\leftarrow 0$
10:         normalizer $\leftarrow 0$
11:         for object with index $(i, j)$ do
12:             if CONTAINS($b^{ij}$, $(y, x)$) then
13:                 $\alpha' \leftarrow$ INTERPOLATE($b^{ij}, \tilde{\alpha}^{ij}, (y, x)$)
14:                 $\gamma' \leftarrow$ INTERPOLATE($b^{ij}, \gamma^{ij}, (y, x)$)
15:                 $o' \leftarrow$ INTERPOLATE($b^{ij}, o^{ij}, (y, x)$)
16:                 $o' \leftarrow \alpha' o' + (1 - \alpha') x_{bg}[y, x]$
17:                 n-writes $\leftarrow$ n-writes+1
18:                 weighted-sum $\leftarrow$ weighted-sum + $\gamma' o'$
19:                 normalizer $\leftarrow$ normalizer + $\gamma'$
20:             end if
21:         end for
22:         if n-writes $> 0$ then
23:             $x_{out}[y, x] \leftarrow \frac{\text{weighted-sum}}{\text{normalizer}}$
24:         end if
25:     end for
26:     return $x_{out}$
27: end function
```

D Model Details

D.1 AIR / DAIR

For our implementation of AIR/DAIR, we adapted code from `github.com/aakhundov/tf-attend-infer-repeat`. For backpropagating through discrete latent variables, we make use of the Concrete trick rather than the REINFORCE-style gradient estimator used in the original paper (the same is also done for SPAIR). For the majority of AIR's parameters the defaults were used, and these are expected not to have a significant impact on performance. As stated previously, AIR and DAIR were always provided with the true number of objects in the image (imparted to the network by hard-coding the number of recurrent steps), even at test time, which allowed us to avoid setting AIR hyperparameters that control the prior distribution over number of objects (i.e. parameters analogous to the s hyperparameter from SPAIR). For all datasets on which AIR was used, we performed random hyperparameter searches over AIR-analogs of $\mu_{y/x}$, $\sigma_{h/w}$, and $\sigma_{y/x}$.

Description	Variable	Value
Base bbox size	(a_h, a_w)	(48, 48)
Batch size		32
Bbox location bounds	$(b_{y/x}^{min}, b_{y/x}^{max})$	(-0.5, 1.5)
Cell size	(c_h, c_w)	(12, 12)
Dim. of z_{what}^{ij}	A	50
Rendered object size	(H_{obj}, W_{obj})	(14, 14)
Learning rate		0.0001
Max gradient norm		1.0
Optimizer		Adam
Prior on z_h, z_w	$(\mu_{h/w}, \sigma_{h/w})$	(-2.2, 0.5)
Prior on z_y, z_x	$(\mu_{y/x}, \sigma_{y/x})$	(0, 1)
Prior on z_{depth}	$(\mu_{\text{depth}}, \sigma_{\text{depth}})$	(0, 1)
Prior on z_{what}	$(\mu_{\text{what}}, \sigma_{\text{what}})$	(0, 1)
Prior on z_{pres}	s	See Sec D.2

Table 1: Base hyperparameter values for SPAIR.

D.2 SPAIR

The base set of hyperparameters for SPAIR is given in Table D.2. To facilitate stability early on in training, we use a schedule, rather than a fixed value, for s (which controls the prior over z_{pres}). Early in training we use a value near 0, which does not penalize the network for using many objects. Over the first few thousand updates, we anneal this to a value of ≈0.99, which encourage network to use few objects. Without an initial period where the network is not penalized for using many objects, we found the network would become stuck in local minima where all objects are turned off. This is likely because early in training the network is not good at reconstructing objects, and can get a lower reconstruction loss by simply turning all objects off (i.e. setting all $z_{\text{pres}}^{ij} = 0$). This trick was adapted from a similar trick used for AIR in the cited repository.

For the convolutional encoder network, we used 2 layers with (stride=2, kernel-size=4), followed by a layer with (stride=3, kernel-size=4), followed by 3 layers with (stride=1, kernel-size=1). All convolutional layers used 128 filters, except the final layer which used 100. Additionally, for the first 3 convolutional layers, we employed a padding style which always pads on the right/bottom side of the volume. The helps to ensure a consistent relationship between the receptive field of neurons in the final layer of the convolutional network and the location of the corresponding cells in the input image, which is necessary for outputting accurate object locations for differently sized images.

D.3 ConnComp

See Section 5.1 of the main paper for a description of the ConnComp algorithm. One required step is finding the connected components of a binary image; this was performed using the function `tf.contrib.image.connected_components` from tensorflow 1.8. ConnComp has a single parameter τ. A pixel is considered non-background if and only if the absolute difference between the pixel color and the background color is at least τ. τ was set by doing a grid search, using the value that yielded the best performance on the training set.