1. Equivalence of specification inputs in encoding head's output space vs. latent space

As explained in Section 4, the specification input set Z_{des} for VAEs is defined using the encoding head's outputs $\hat{z} = [\mu, \sigma]$ and not the latent vector $z = \mu + \epsilon\sigma, \epsilon \sim \mathcal{N}(0, 1)$, that is fed to the decoder after reparameterisation. Therefore, valid correspondence between the verification outcomes and the reconstructions, especially counterexamples, should require that a Z_{des} defined using $\hat{z}$ corresponds to the Z_{des} defined using the corresponding z.

To assess the same for *Segment* queries, consider Z_{des} as the line segment joining v_1, v_2, i.e., $\overline{v_1 v_2}$. It is straightforward to show that every point on $\overline{\hat{z}_1 \hat{z}_2}$ lies on $\overline{z_1 z_2}$ as follows:

$$\overline{\hat{z}_1 \hat{z}_2} = (1-\alpha)\begin{bmatrix}\mu_1 \\ \sigma_1\end{bmatrix} + \alpha\begin{bmatrix}\mu_2 \\ \sigma_2\end{bmatrix}, \alpha \in [0,1] \tag{1}$$

$$= \begin{bmatrix}(1-\alpha)\mu_1 + \alpha\mu_2 \\ (1-\alpha)\sigma_1 + \alpha\sigma_2\end{bmatrix} \tag{2}$$

$$= (1-\alpha)\mu_1 + \alpha\mu_2 + \epsilon((1-\alpha)\sigma_1 + \alpha\sigma_2), \epsilon \sim \mathcal{N}(0,1) \tag{3}$$

$$= (1-\alpha)(\mu_1 + \epsilon\sigma_1) + \alpha(\mu_2 + \epsilon\sigma_2) \tag{4}$$

$$= (1-\alpha)z_1 + \alpha z_2 = \overline{z_1 z_2} \tag{5}$$

Therefore, $\overline{\hat{z}_1 \hat{z}_2}$ maps to $\overline{z_1 z_2}$ when using a constant ϵ for reparameterisation (step 3 above) in the verification phase. The converse is not true as multiple $\overline{\hat{z}_1 \hat{z}_2}$ can generate the same $\overline{z_1 z_2}$ under reparameterisation.

Similarly for *Axis* and *Region* queries, where Z_{des} is a uniform or per-dimension epsilon ball, we have for a non-negative reparameterisation constant c:

$$\begin{aligned} Z_{\text{des}}(\hat{z}) &= \left[\begin{bmatrix}\mu \\ \sigma\end{bmatrix} - \epsilon, \begin{bmatrix}\mu \\ \sigma\end{bmatrix} + \epsilon\right] \\ &= [\mu + c\sigma + (1-c)\epsilon, \mu + c\sigma + (1+c)\epsilon] \\ &\supseteq [z - \epsilon, z + \epsilon] = Z_{\text{des}}(z). \end{aligned}$$

Therefore, under certain conditions for reparameterisation during the verification phase, the proposed specification input sets based on the encoding head's outputs $\hat{z}$ are the same or a superset of the input sets based on the corresponding latent vectors z.

2. Verifying Affine-Coupling layer

As mentioned in Section 4, our pipelines can use an encoding head consisting of alternating affine-coupling layers [1]. We refer to an alternating affine-coupling layer as the bijective mapping $f : v \to y \to z$ such that

$$\begin{aligned} y_u &= v_u & v_u &= y_u \\ y_l &= v_l \odot \mathrm{e}^{s_1(v_u)} + t_1(v_u) & v_l &= (y_l - t_1(y_u)) \odot \mathrm{e}^{-s_1(v_u)} \\ z_u &= y_u \odot \mathrm{e}^{s_2(y_l)} + t_2(y_l) & y_u &= (z_u - t_2(z_l)) \odot \mathrm{e}^{-s_2(z_l)} \\ z_l &= y_l & y_l &= z_l, \end{aligned}$$

where for a vector x, $\{x_u, x_l\}$ represent its upper and lower halves, i.e., $\{x[:\text{len}(x)/2], x[\text{len}(x)/2:]\}$. With input dependent scaling (s_*) and translation (t_*) units, which are typically nonlinear networks, this seemingly affine mapping can learn the nonlinearities in its inputs. However, most standard NN verifiers do not support this layer, due to its input splitting, and input dependent scaling and translation units.

Since the focus of our work is the effective use of *existing* verification backends for verifying deep networks, we replace these units to an input-independent scaling ($s_*(x) \to s_*$) and an affine translation ($t_*(x) \to W_* x + b$) units respectively. This change makes the affine-coupling-layer linear with respect to input and thereby, allows standard NN verifiers to verify it with minimal modifications, as explained next.

With the specified modifications, the inverse mapping of the linear version of the affine-coupling layer, f^{-1}_{linear}, gets simplified to the following:

$$\begin{aligned} y_u &= (z_u - W_2 z_l) \odot \mathrm{e}^{-s_2} \\ y_l &= z_l \\ v_u &= z_u \odot \mathrm{e}^{-s_2} - W_2 z_l \odot \mathrm{e}^{-s_2} \\ v_l &= (y_l - W_1 y_u) \odot \mathrm{e}^{-s_1} \end{aligned} \tag{6}$$

The f^{-1}_{linear} mapping above is implementable using the standard fully-connected layer and vector addition operations supported by most available verifiers. When propagating a latent space segment,

$$\overline{z_1 z_2} = z_1 + \alpha(z_2 - z_1) = z + \alpha d, \quad \alpha \in [0,1],$$

where $z = z_1, d = z_2 - z_1$, the inverse mapping f^{-1}_{linear} becomes:

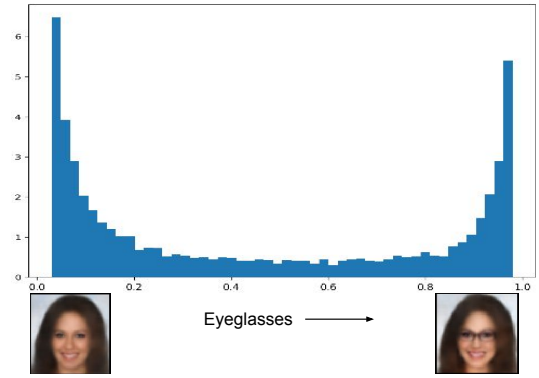
$$\begin{aligned} y_u &= (z_u + \alpha d_u - W_2(z_l + \alpha d_l)) \odot \mathrm{e}^{-s_2}, \\ &= \underbrace{(z_u - W_2 z_l) \odot \mathrm{e}^{-s_2}}_{c_1} + \alpha \underbrace{(d_u - W_2 d_l) \odot \mathrm{e}^{-s_2}}_{c_2} \\ y_l &= z_l + \alpha d_l \\ v_u &= c_1 + \alpha c_2 \\ v_l &= (z_l + \alpha d_l - W_1(c_1 + \alpha c_2)) \odot \mathrm{e}^{-s_1} \\ &= \underbrace{(z_l - W_1 c_1) \odot \mathrm{e}^{-s_1}}_{c_3} + \alpha \underbrace{(d_l - W_1 c_2) \odot \mathrm{e}^{-s_1}}_{c_4}. \end{aligned} \tag{7}$$

Thus, propagating *back* a segment through the last affine-coupling layer of the encoding head is tight and outputs the following line segment,

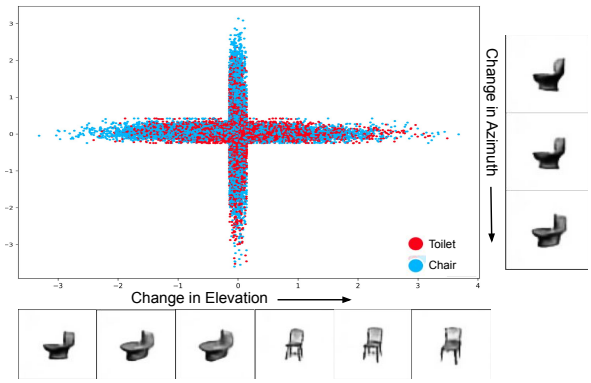
$$\overline{v_1 v_2} = \begin{bmatrix} c_1(W_2, s_2, z) \\ c_3(W_1, W_2, s_1, s_2, z) \end{bmatrix} + \alpha \begin{bmatrix} c_2(W_2, s_2, d) \\ c_3(W_1, W_2, s_1, s_2, d) \end{bmatrix},$$

which is the exact inverse of the input segment $\overline{z_1 z_2}$ under f_{linear}. Also notice that the discussed modifications make the affine-coupling layer equivalent to a fully-connected layer in terms of expressiveness.

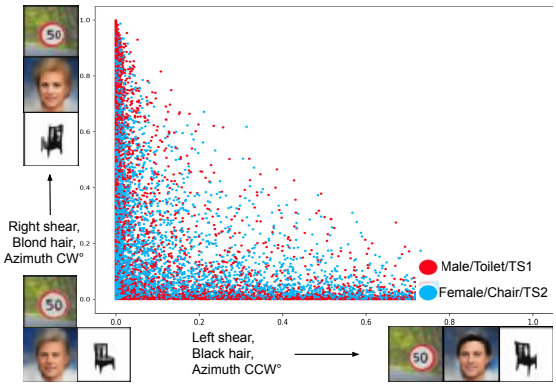
In terms of learning, the presence of more trainable parameters than in a fully-connected layer and the enforcing of invertibility through specifically structured layers seems to make the training of this encoding head variant more difficult and slightly unstable as compared to the training of the encoding head using fully-connected layers. With the nonlinear units replaced, the nonlinearity in encoding head essential to transform the FDN output is introduced by adding invertible activation layers in between the affine-coupling layer(s). In turn, the propagation of Z_{des} through the inverse of the encoding head is no longer tight beyond the last coupling layer.



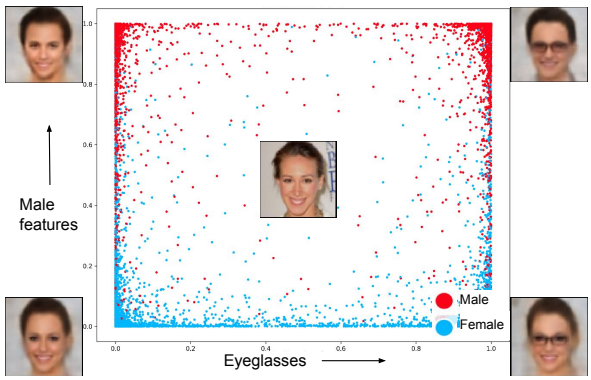
(a) Histogram of latent dimension values
(conditionally trained with BCE loss for a single attribute)



(b) Latent vectors for Azimuth vs Elevation change
(conditionally trained as a graphics code)



(c) Latent vector dimensions for multi-value attributes
(conditionally trained with CE loss for discrete values of an attribute)



(d) Latent vectors for Glasses dim. vs Gender dim.
(conditionally trained for two orthogonal attributes with BCE loss in both dimensions)

Figure 1. Encodings for conditional dimensions when trained with different loss functions.

3. Comparison of the conditional loss functions

As mentioned in Section 4, we use the following two conditional training approaches:

- For mapping *discrete attributes*, we use (Binary) Cross Entropy ((B)CE) loss to learn a bimodal encoding for a dimension,
- For mapping *continuous transformations*, we train conditional dimensions as *graphics code* as per [2] to learn disentangled and more uniformaly distributions along the dimensions.

This section elaborates on these choices of the conditional loss functions. The representative conditional dimension embeddings learnt with these loss functions are shown in Figure 1. The (B)CE loss skews the conditional latent distribution to be bimodel, which is a natural fit for encoding discrete attributes, with modes at $0, 1$ indicating attribute absence and presence. However, since the resulting distribution has much less, albeit non-zero mass in the middle, it may not be ideal for encoding continuous transformations. If the mass in between the distribution modes is too less, then the decoder may produce vague, blurry or out-of-distribution images for samples from these regions, owing to less familiarity. Therefore, when using CE loss for conditional training, there is a trade-off between training the modes to have sufficiently high mass that they allow making precise conditional queries, and having enough diversity in data that sufficient mass gets placed in between the modes. In our experiments, we used CE loss for encoding planar transformations in TRDS and (F)MNIST datasets, obtained fairly continuous transformations, and the encoding plots did not show gaps along the conditional dimensions.

Training conditional dimensions as *graphics code* encourages them to capture all variations corresponding to only a desired attribute. In terms of implementation, it involves the following steps:

1. Training the LVM with mini-batches such that each batch has variations in only one of the attributes,
2. Replacing the output of the conditional dimensions that are not meant to capture the current attribute, with their mean values for the batch,
3. For each batch, updating only the intended conditional dimension using the same loss function as used for the non-conditional dimensions, i.e., the ELBO or GAN loss.

The resulting encoding is shown in Figure 1b with each conditional dimension resembling an independent normal distribution as desired. However, since there is no explicit incentive to organise the different extents of an attribute variation in any specific manner within the dimension, it can be difficult to refine the ranges for different extents of a variation and form precise queries when using this loss.

Networks	#Activations	Architecture
		Classifiers
ConvSmall	6k	$x \rightarrow \text{conv}(4, 4, 2) \rightarrow \text{relu} \rightarrow \text{conv}(4, 4, 2) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(784, 384) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(384, 128) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(128, 64) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(64, \#\text{class})$
ConvBig	72k	$x \rightarrow \text{conv}(32, 4, 2) \rightarrow \text{relu} \rightarrow \text{conv}(16, 3, 1) \rightarrow \text{relu} \rightarrow \text{conv}(8, 3, 1) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(5832, 1024) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(1024, 384) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(384, 128) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(128, 64) \xrightarrow{*} \text{relu} \rightarrow \text{linear}(64, \#\text{class})$
ResNet18	189k	$x \xrightarrow[2048]{\text{initial\_layers }*} \rightarrow \text{avg\_pool2d} \xrightarrow{*} \text{relu} \rightarrow \text{linear}(512, \#\text{class})$
MobileNetv2	122k	$x \xrightarrow[5120]{\text{initial\_layers}} \text{relu} \rightarrow \text{avg\_pool2d} \rightarrow \text{dropout} \xrightarrow{*} \text{relu} \rightarrow \text{linear}(1280, \#\text{class})$
		Decoders
DecConvTiny	14k	$z \rightarrow \text{linear}(\#\text{latent\_ dims}, 200) \rightarrow \text{leaky\_relu} \rightarrow \text{linear}(200, 1922) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(3, 3, 2) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(3, 2, 1) \rightarrow x$
DecConvSmall	21.2k	$z \rightarrow \text{linear}(\#\text{latent\_ dims}, 512) \rightarrow \text{leaky\_relu} \rightarrow \text{linear}(512, 4805) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(4, 3, 2) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(3, 2, 1) \rightarrow x$
DecConvDeep	36.2k	$z \rightarrow \text{linear}(\#\text{latent\_ dims}, 1024) \rightarrow \text{leaky\_relu} \rightarrow \text{linear}(1024, 2700) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(9, 3, 2) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(6, 3, 2) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(3, 2, 1) \rightarrow x$
DecResNet18	380.5k	$z \rightarrow \text{linear}(\#\text{latent\_ dims}, 512) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(512, 3, 2) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(256, 3, 2, 1) \rightarrow \text{batchnorm} \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(128, 3, 2, 1) \rightarrow \text{batchnorm} \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(64, 3, 2, 1) \rightarrow \text{batchnorm} \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(64, 3, 2, 1) \rightarrow \text{batchnorm} \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(64, 3, 2, 1) \rightarrow \text{leaky\_relu} \rightarrow \text{conv\_trans}(3, 6, 1, 3) \rightarrow x$
		Encoders
EncConvBig		$x \rightarrow \text{conv}(4, 4, 2) \rightarrow \text{relu} \rightarrow \text{conv}(4, 4, 2) \rightarrow \text{relu} \rightarrow \text{linear}(784, 384) \rightarrow \text{relu} \rightarrow \text{linear}(384, 128) \rightarrow \text{relu} \rightarrow \text{linear}(128, 2\#\text{latent\_dims})$
		Encoding heads
Lin{#layers}-{#ldims}		$x \rightarrow \text{linear}_1(2\#\text{ldims}, 2\#\text{ldims}) \rightarrow \text{leaky\_relu} \rightarrow \ldots \rightarrow \text{linear}_{\#\text{layers}}(2\#\text{ldims}, 2\#\text{ldims})$
AC{#layers}-{#ldims}		$x \rightarrow \text{affine\_coupling}_1\left(\text{linear(2\#ldims, 2\#ldims), linear(1, 2\#ldims)}\right) \rightarrow \text{leaky\_relu} \rightarrow \ldots \rightarrow \text{affine\_coupling}_{\#\text{layers}}\left(\text{linear(2\#ldims, 2\#ldims), linear(1, 2\#ldims)}\right)$

• The $\xrightarrow{*}$ indicate the positions where an encoding head was added for the SRVP pipelines reported in Table 1 in the main paper. These reported SRVP pipelines corresponding to $*$s in the order they appear above are ConvDeep {SRVP Lin-392, SRVP Lin-192, SRVP Lin-64, SRVP Lin-32}, ResNet18 {SRVP Lin-1024, SRVP Lin-256} and MobileNetv2 SRVP Lin-640.

• Operations in red are not supported by the verification backend, therefore the network must be split such that these operations are not included in the network head. Note that with the EDN approach, the lack of support for a network operation by the verification backend implies that the network cannot be verified at all.

• conv(NF, K, S) and conv_trans(NF, K, S, P) represent convolution and transposed convolution layers with NF output filters, uniform kernel size of K, stride of S and padding of P pixels. The number of activations (#Activations) reported are for 64x64 input.

Table 1. Description of the networks constituting the reported SRVP pipelines.

References

[1] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real NVP. In *Proceedings of the 5th International Conference on Learning Representations (ICLR17)*. OpenReview.net, 2017. 1

[2] T.D. Kulkarni, W.F. Whitney, P. Kohli, and J. Tenenbaum. Deep convolutional inverse graphics network. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. 2