# Discovering interpretable models of scientific image data with deep learning

## Supplementary Material

## 1. Datasets

Image datasets are identical to those used by [3, 14, 15].

## 2. Extended methods

### 2.1. Total Correlation VAE

The VAE is a latent variable model that consists of an encoder network parameterized by $\theta$ and a decoder network parameterized by $\phi$. The encoder network transforms an input $\mathbf{x}$ into its latent space representation $\mathbf{z}$ via a two-step process: first, the computation of $\mu$ and $\sigma$ vectors, then the sampling of $\mathbf{z}$ from a multivariate Gaussian distribution: $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu, \sigma)$. Then, the decoder forms a reconstruction based on the latent space representation $\mathbf{z}$.

In the original formulation, the model is trained using the following loss function [10].

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}) = \mathrm{D}_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \tag{1}$$

This loss function is composed of two terms. The first penalizes the Kullback-Leibler divergence ($\mathrm{D}_{\mathrm{KL}}(\cdot \,||\, \cdot)$) between the latent space distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and a prior, chosen as the isotropic unit Gaussian $\mathcal{N}(0, \mathbf{I})$. The second maximizes the marginal likelihood of the original data given this latent space representation. This latter term acts as a reconstruction loss, which penalizes differences between the reconstructed output, and the original input. This is typically calculated in practice using the mean squared difference. Meanwhile, the KL divergence term regularizes the latent space to promote a continuous representation of the underlying input data.

One key contribution of the VAE is its ability to construct disentangled latent spaces, where different latent variables encode semantically distinct features of the data. This is particularly true in the $\beta$-VAE variant, where the two loss terms are differently weighted, with more prominence given to the KL divergence term [4, 8].

The $\beta$-TCVAE operates in a similar vein [5]. Here, each data sample is indexed by a uniform random variable $n$. Hence, the expected value of the KL divergence term over the dataset can be separated into three components:

$$\mathbb{E}_{p(n)}[\mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}))] = \mathrm{D}_{\mathrm{KL}}(q(z,n)||q(z)p(n)) + \mathrm{D}_{\mathrm{KL}}(q(z)||\prod_j q(z_j)) + \sum_j \mathrm{D}_{\mathrm{KL}}(q(z_j)||p(z_j)),$$

where $j$ represents the latent dimension. We can interpret the first two terms as mutual information (MI) terms, and weight the three terms differently using the coefficients $\alpha$, $\beta$ and $\gamma$ [5]:

$$\mathbb{E}_{p(n)}[\mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}))] = \alpha I_q(z;n) + \beta C_q(z_1, z_2, ..., z_L) + \gamma \sum_j \mathrm{D}_{\mathrm{KL}}(q(z_j)||p(z_j)),$$

where $L$ is the dimension of the latent space. The first term $I_q(z;n)$ measures the mutual information between the choice of dataset index $n$ and the corresponding latent vector $\mathbf{z}$, under the empirical data distribution $q(z,n)$. Intuitively, a lower value of this "index-code MI" might represent a higher clustering concentration within latent space of samples throughout the dataset.

The second term $C_q(z_1, z_2, ..., z_L)$ represents a multivariate generalization of mutual information called the total correlation, applied over the latent dimensions. This essentially indicates the shared information between latent dimensions. Penalizing this term (i.e., high $\beta$) would encourage disentangled representations, as information about the value of one latent variable would provide little information about the values of the rest [5].

Lastly, the third term measures the dimension-wise KL divergence of the latent space distribution; this fulfils the regularization objective.

For feature extraction, we used a convolutional $\beta$-TCVAE. The encoder consisted of four convolutional layers, with 8, 16, 32 & 64 filters respectively, followed by a fully-connected layer with 256 units, then the sampling layers, each with 32 units. The 32-dimensional latent space representation is then projected back to image space using a decoder network with the reverse architecture, using upsampling layers. The $\beta$-TCVAE was trained for 5 epochs, using a batch size of 256 and a learning rate of 0.0005, with the Adam optimizer. Flips and 90° rotations were used as augmentation.

### 2.2. Cell state classification

For the Dense CNN we used four convolutional layers with 8, 16, 32 & 64 filters respectively. For the Dense Head, we used four fully-connected layers with 32, 16, 16, 16 and 2 units respectively, with the latter layer providing the final outputs. Scheme 1 & 2 models were trained using a dropout rate of 0.3, while the Sparse Head was trained with no dropout. The Sparse Head contained the same architecture and initial connectivity as the Dense Head. These configurations were chosen such that Scheme 1, 2 & 3 models

all contain a similar number of parameters (∼380,000 for Scheme 1 and ∼316,000 for Schemes 2 & 3).

Models from Schemes 1, 2 & 3 were trained for 100 epochs, using a batch size of 64, with flips and 90° rotations as augmentation. For Scheme 3, we used a dense-training warm up period of $T_{warmup} = 20$ epochs.

### 2.3. Sparsity: RigL

As with other similar algorithms, *RigL* aims to find the optimal topology of non-zero weights, given some fixed, predetermined sparsity value $S$. It achieves this with iterative cycles of pruning and re-growth, implemented every $\Delta T$ training iterations (where $\Delta T$ is a hyperparameter). Between each cycle, the network uses gradient descent to find the optimal weight values, as usual. At the beginning of training, $(1 - S) \times N^l$ weights of each layer are randomly deactivated, where $N^l$ represents the total number of connection weights at layer $l$. When inactive, the connection weights are set to zero and they do not update between training iterations. At every subsequent pruning step, the smallest $(1-S)\times N^l \times \alpha$ weights (in terms of weight magnitude) are deactivated. Then, at every re-growth step, the same number of weights is chosen for re-growth, this time based on the magnitude of their associated loss gradient. Hence, the top $(1-S)\times N^l \times \alpha$ inactive weights, based on gradient magnitude, are re-grown, with their initial weight value set to zero.

This way, the sparsity value of $S$ is maintained throughout training, with each layer possessing $S \times N^l$ active weights at all times. Hence, *RigL* belongs to the class of pruning techniques that does *not* redistribute sparsities between layers during training. $S$, $\Delta T$, and the update fraction $\alpha$ are all hyper-parameters to be chosen prior to training.

Following Evci et al. [7], we specify a network-wide value for $S$ but allocate to different layers $l$ a separate sparsity value $s^l$. We use the Erdős-Rényi scheme [12], in which the layer-specific sparsities scale according to $1 - \frac{n^l+n^{l+1}}{n^l \times n^{l+1}}$, where $n^l$ is the number of input neurons at layer $l$. The actual values for $s^l$ are chosen such that $S = \frac{\sum_l s^l N^l}{N}$, where $N$ is the total number of connections in the network. Therefore, the number of inactive connections in each layer is not $S \times N^l$ but $s^l \times N^l$.

We also attenuate the update fraction $\alpha$ as training progresses, using a cosine annealing schedule as the decay function:

$$f_{decay}(t; \alpha, T_{end}) = \frac{\alpha}{2}\left(1 + \cos\left(\frac{t\pi}{T_{end}}\right)\right) \text{ [6]}. \quad (2)$$

With *RigL*, it has been reported that the annealled schedule slightly outperforms a schedule of constant $\alpha$ [7]. The use of this schedule adds an additional hyper-parameter $T_{end}$, after which the weight topology does not update.

In our study, we adapt *RigL* by adding two modifications. Firstly, rather than randomly pruning $(1 - S) \times N^l$ weights at the beginning of training, we train the model as a dense network for a pre-specified "warm up" period $T_{warmup}$. We observed that this method delivered superior performance results and identified a consistent set of relevant inputs between training runs, benefits we did not observe when randomly pruning at the beginning. We hypothesize that prior dense training allows for the development of useful connection paths, which are then reinforced and retained when the first pruning step is implemented after $T_{warmup}$.

Secondly, we implement two post-training pruning steps that eliminating the following types of connection weight:

1. Leaf weight: Connections that feed into a non-output-layer neuron that has no output connections. These connections do not contribute to the output layer at all.
2. Bias weight: Connections that emerge from a non-input-layer neuron that has no input connections. These are called "bias weights" because at each forward pass, they simply add a fixed value to their target neuron, which is the product of the source neuron's bias and the connection weight value. These "bias weights" can be removed and compensated by adjusting the bias of the target neuron[1].

The criteria are implemented in this order. This post-training procedure does not affect the operation of the model at all, but it removes superfluous connection weights, allowing for a more useful assessment of model sparsity. To allow for the innocuous removal of bias weights, we do not use batch normalization when training sparse networks.

#### 2.3.1 Hyper-parameter tuning

As described in Sec. 2.3, *RigL* operates using a small set of hyper-parameters: the sparsity value $S$ (which is the fraction of weights set to zero), update fraction $\alpha$ and update interval $\Delta T$, as well as the end-step $T_{end}$ for the update fraction decay schedule. Hyper-parameter search was conducted on three of these four values - all except $T_{end}$, which was set to $T_{end} = 0.75$.

For this, we used the OPTUNA [1] package, a Python interface that access several optimization algorithms, including grid search and random search, as well as Bayesian methods such as the tree parzen estimator [2]. We ran the OPTUNA search for 200 trials, with each trial querying a unique hyper-parameter configuration. Each trial consisted of five runs, each being run for a reduced duration of 40 epochs. The value ranges we provided reflect our experience from prior experiments: 0.95-0.97 for $S$, 100-200 iterations for $\Delta T$ and 0.7-0.9 for $\alpha$. However, in principle, op-

[1] We did not use layer or batch normalization to train our sparse networks. If this had been used, the extra parameters would have to have been considered when adjusting the bias of the target neuron.

**Algorithm 1** Modified RigL

**Input:** Network $f_\Theta$, dataset $D$, learning rate $\eta$ ▷ $\Theta$ = *un-pruned weights*
Sparsity distribution: $\mathbb{S} = \{s^1, ..., s^L\}$
Update schedule: $\Delta T$, $T_{end}$, $\alpha$, $f_{decay}$
**for** each training step $t$ **do**
  Sample a batch $B_t \sim D$
  $L_t = \sum_{i \in B_t} \mathcal{L}(f_\theta(x_i), y_i)$ ▷ $\mathcal{L}$ *is the loss function*
  **if** $t < T_{warmup}$ **then**
    $\Theta = \Theta - \eta \nabla_\Theta L_t$ ▷ *Update dense weights*
  **else**
    **if** $(t - T_{warmup})(\text{mod } \Delta T) == 0$ and $(t - T_{warmup}) < T_{end}$ **then**
      **for** each layer $l$ **do**
        $k = f_{decay}(t - T_{warmup}; \alpha, T_{end})(1 - s^l)N^l$ ▷ *No. to update*
        $\mathbb{I}_{active} = ArgTopK(|\theta^l|, (1 - s^l)N^l - k)$ ▷ *Connections to keep*
        $\mathbb{I}_{grow} = ArgTopK_{i \notin \mathbb{I}_{active}}(|\nabla_{\Theta^l} L_t|, k)$ ▷ *Connections to grow*
        $\theta \leftarrow \Theta$ connections included in $\mathbb{I}_{active} \cup \mathbb{I}_{grow}$ ▷ *Update topology*
      **end for**
    **else**
      $\theta = \theta - \eta \nabla_\theta L_t$ ▷ *Update sparse weights*
    **end if**
  **end if**
**end for**
$\theta \leftarrow$ Prune leaf and bias weights

timization algorithms such as OPTUNA can output sensible hyper-parameter values in the absence of any prior knowledge (e.g., by setting the range of $S$ to 0.0-1.0). For our objective function, we used the sum of the validation accuracy and the final sparsity (after the post-training pruning steps), with each of these being averaged over the five runs associated with each trial. Hence, the goal is both to maximize classification performance and minimize the number of active weights in the final model.

While this hyper-parameter search was done in order to generally optimize classification performance, the search for an optimal $S$ has particular significance, for it represents the maximum sparsity appropriate to a particular task. This reflects both the complexity of the inputs required and the complexity of the target function itself. Hence, our objective function - the sum of the accuracy and the final sparsity - reflects the principle of Occam's razor.

The values yielded were $S = 0.951$, $\Delta T = 115$ and $\alpha = 0.758$.

### 2.4. PySR implementation

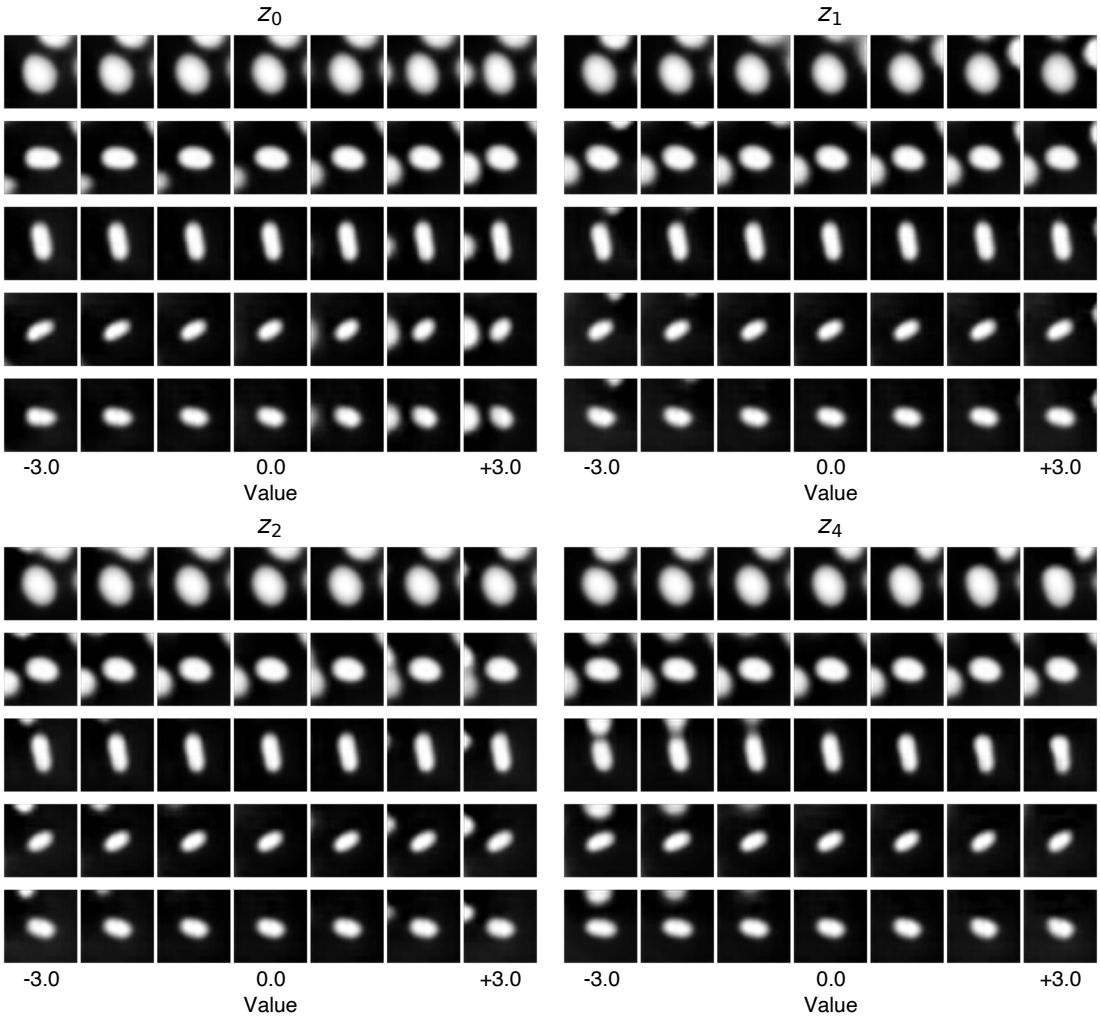
We configured our regression models such that they could choose between four binary operators ($+, \times, -$ and $\div$) and six unary operators ($x^2$, $e^x$, $\log(x)$, $\sin(x)$, $\sqrt{x}$ and $|x|$) in the expression trees. We penalized all operators, constants and variables in the tree with a complexity score of 1, except for the $\sin(x)$ (score of 3), $e^x$ (score of 2) and $\log(x)$ (score of 2) functions. We used a parsimony value of 0.001 to scale the complexity score in the overall loss function, and we set 20 as a hard limit on the complexity of our expressions.

## 3. Neighborhood features

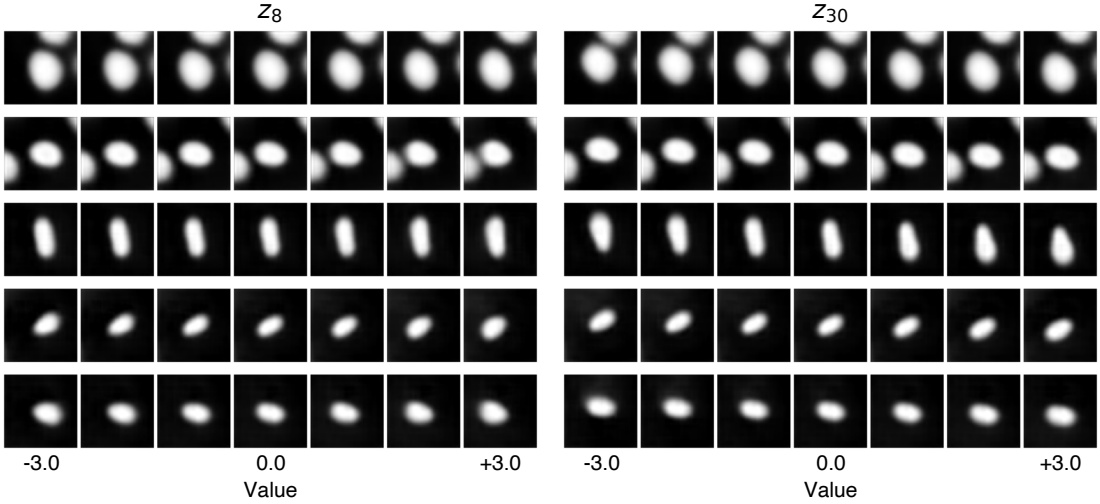
Latent-space traversals for neighborhood and positional features are shown in Fig. 1.

## 4. Sparse model topologies

Topologies for all ten Scheme 3 models are shown in Fig. 2.




(a) Some latent variables that encode neighborhood features.




(b) Latent variables that encode central cell position: $z_8$ ($x$-position) & $z_{30}$ ($y$-position)

Figure 1. Latent variables that encode neighborhood features and central cell position.

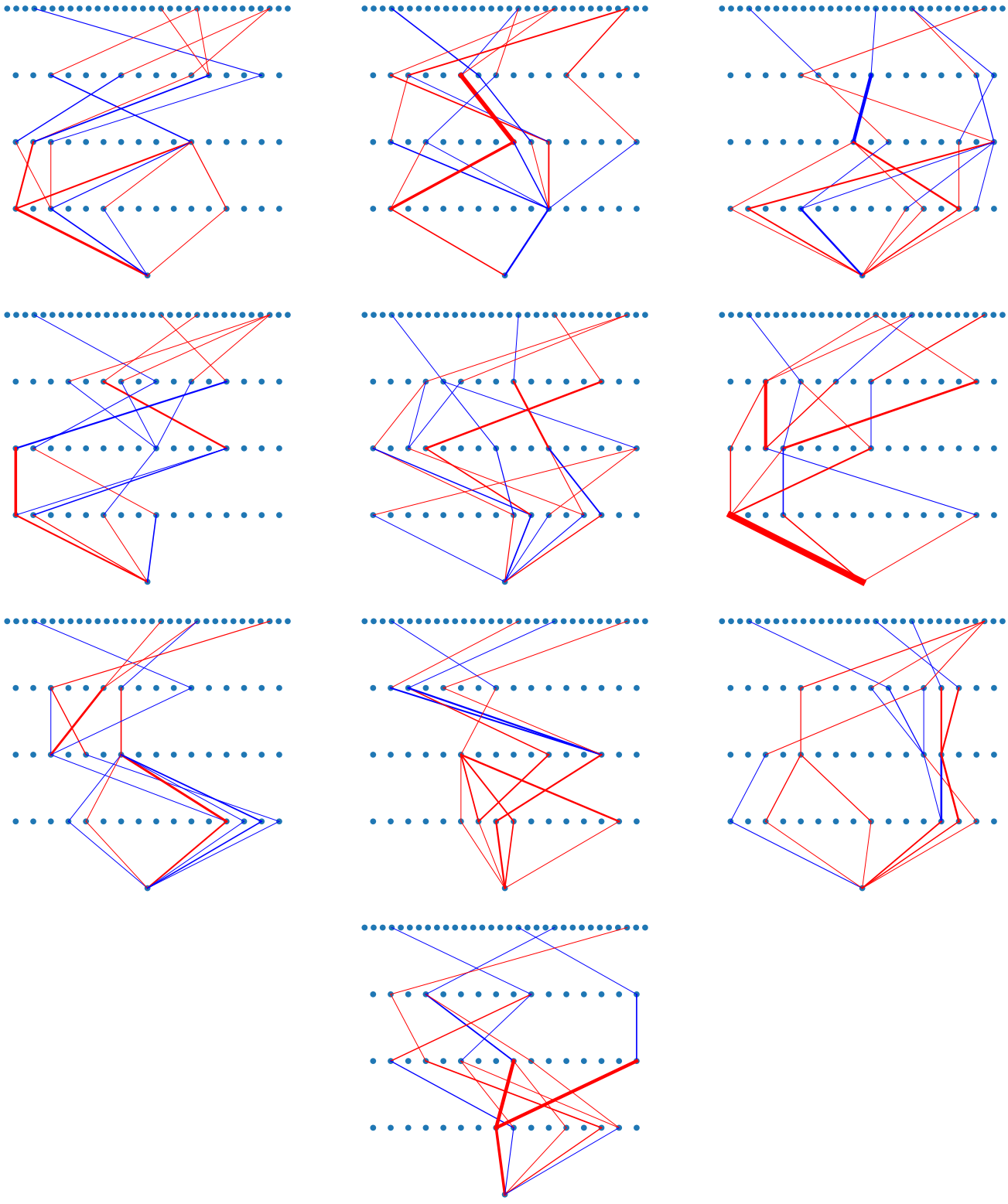


Figure 2. Scheme 3 model topologies. Blue connections are positively weighted, red connections are negatively weighted. The thickness of the connection line is proportional to its weight magnitude. The network flow is up to down, so the top layer is the input layer and the bottom layer is the output layer.

## 5. Calculation of head expression size

The goal here is to transform a neural network into a symbolic expression. When a neuron at layer $i$ accepts inputs $x$ from connected neurons $m$ at layer $i-1$, the output $y$ of the neuron will be:

$$y = f\left(b + \sum_m x_m w_{mn}\right), \tag{3}$$

where $f$ is the activation function and $w_m$ is the weight connecting neuron $m$ at layer $i-1$ to the present neuron at layer $i$. We use the activation function *Mish* [11], which can be written as

$$f(x) = x \cdot \tanh(\log(1 + e^x)). \tag{4}$$

The expression tree representation of this function is shown in Fig. 3a. The expression inside $f(\cdot)$ in Eq. Eq. (3) can likewise be represented by the tree shown in Fig. 3b.

It is possible to see that for any given number of source neurons $M$, the expression size of the tree in Fig. 3b is $4M+1$. Therefore, when one substitutes the tree in Fig. 3b for each $x$ term in the *Mish* tree in Fig. 3a, one can calculate the total expression size as $8(M+1)$. Therefore, a pair of consecutive layers fully connected to each other, with $M$ and $N$ numbers of neurons respectively, can be represented by an expression tree with size $8N(M+1)$. This is in the absence of normalization.

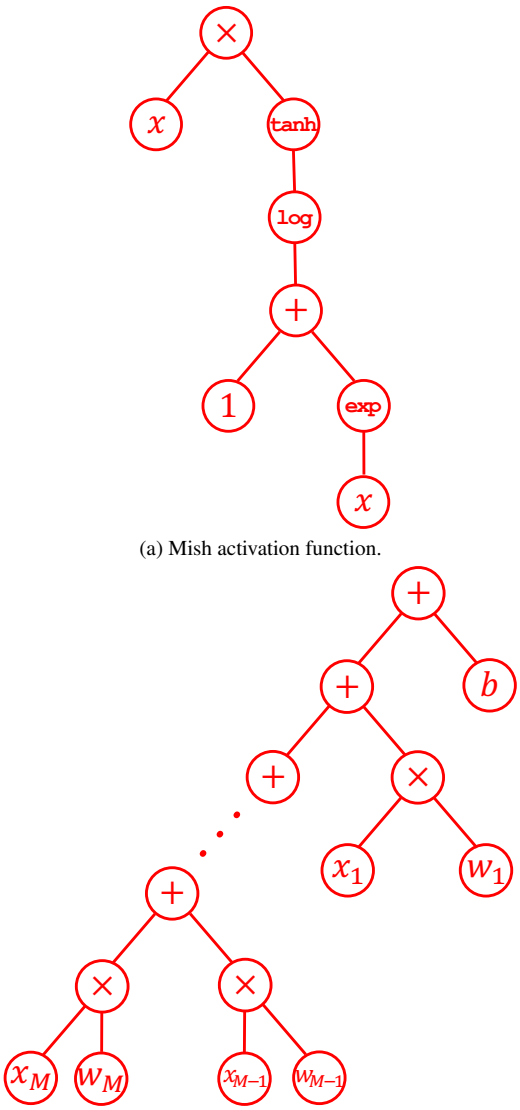
For all our classification heads, no activation was used in the final layer; therefore, for this layer, the expression size is simply $N(4M+1)$. For Scheme 2 models with $I$ layers and layer-wise neuron counts $n_1, n_2, ..., n_I$, the total expression size $E$ is therefore

$$E = n_I(4n_{I-1}+1) + \sum_{i=1}^{I-2} 8n_{i+1}(n_i+1). \tag{5}$$

For Scheme 1 models, this calculation is complicated by the fact that batch normalization [9] is used. Batch normalization reduces internal covariate shift by normalizing layer-wise inputs according to the batch-wide mean and variance. The output $y$ of this normalization is

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \cdot \gamma + \beta, \tag{6}$$

where $\mathrm{E}[x]$ and $\mathrm{Var}[x]$ represent the batch-wise mean and variance respectively, $\epsilon$ is a small numerical stabilizer and $\gamma$ and $\beta$ are trainable parameters to scale and shift the final output. This is done on a per-neuron basis, and before any activation. Furthermore, the mean and variance terms are calculated during training, but for our models, they are frozen during evaluation, so they do not involve any calculation during the forward pass.




(a) Mish activation function.

(b) Addition of neuron bias and input terms.

Figure 3. Expression trees. $M$ is the total number of source neurons.

The expression tree for batch normalization is shown in Fig. 4. The size of this tree is 12, therefore the size of the input to activation is $4M+12$, so the total expression size of the neuron is $8M+30$. The total expression size for the feed-forward network, again omitting normalization and activation from the final layer, is then

$$E = n_I(4n_{I-1}+1) + \sum_{i=1}^{I-2} n_{i+1}(8n_i+30). \tag{7}$$

Finally, for Scheme 3 models, the expression size can be calculated by summing $8M_j+1$ (or $4M_j+1$ for the final layer) over the target neurons with active connections, where $M_j$ is the number of source neurons connected to

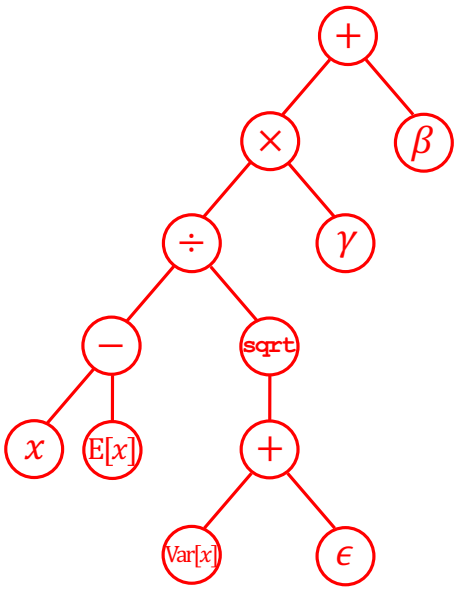



Figure 4. Expression tree for batch normalization.

each target neuron $j$. More precisely, for layers $i$ with layer-wise active neuron count $J_i$, the whole network expression size is

$$E = \left(\sum_{j=1}^{J_I} 4M_j + 1\right) + \left(\sum_{i=1}^{I-1}\sum_{j=1}^{J_i} 8M_j + 1\right). \quad (8)$$

## 6. Symbolic expressions obtained

We assessed two strategies for choosing the loss function and target output values:

1. **Hinge loss**: Our target values $y'$ are the ground-truth labels of the cell states, expressed as $-1$ (for interphase) or $+1$ (for metaphase). Our loss function is the hinge loss [13], widely used in support vector machines, which is defined as:

$$\mathcal{L}(y, y') = \max(0, 1 - y'y). \quad (9)$$

   Hence, $y$ is encouraged to share the same sign with $y'$. In this strategy, the neural network outputs are not used; however, we still use only the four latent variables identified as relevant by our Scheme 3 models.
2. **MSE loss**: Our target values $y'$ are the outputs of a neural network - in this case, we use our Scheme 3 models. Our loss function is mean-squared error:

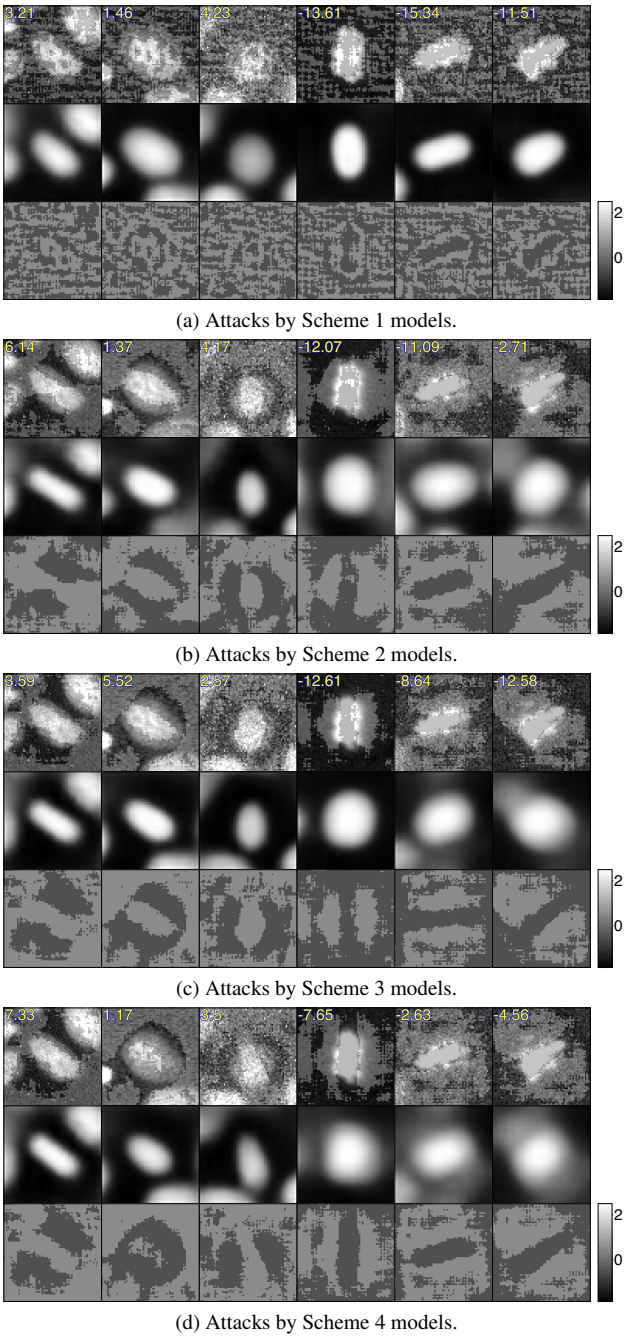
$$\mathcal{L}(y, y') = (y' - y)^2. \quad (10)$$

   Hence, our symbolic expressions are encouraged to adhere as closely as possible to the function learnt by the neural network.

   Tabs. 1 and 2 show the expressions obtained for each.

## 7. Adversarial attack results

Shown are image-based attacks (Fig. 5) and latent-based attacks (Fig. 6).




(a) Attacks by Scheme 1 models.

(b) Attacks by Scheme 2 models.

(c) Attacks by Scheme 3 models.

(d) Attacks by Scheme 4 models.

Figure 5. Image-based attacks at $\epsilon = 0.5$. Exp. H1 was used for **(c)**.

| No. | Expression | Expression size | Accuracy |
|---|---|---|---|
| 1 | $z_{29}(z_{17}^2 + z_{21}^2) - e^{e^{z_3}}$ | 11 | 97.6% |
| 2 | $0.83(z_{29} - 1.37z_3 - \lvert z_3\rvert)(z_{17}^2 + z_{21}^2 - 0.19) - 1.74$ | 20 | 97.8% |
| 3 | $(z_{29} - z_3)(\lvert z_{17}\rvert + z_{21}^2) - 2.88$ | 11 | 97.4% |
| 4 | $0.74(z_{29} - 0.62z_3)(z_{17}^2 + z_{21}^2) - 2.11$ | 15 | 97.5% |
| 5 | $(z_{17}^2 + z_{21}^2)(z_{29} - \sin(z_3)) - 2.86$ | 12 | 97.3% |
| 6 | $0.74(z_{17}^2 + z_{21}^2)(z_{29} - \sin(z_3 + 0.18)) - 1.99$ | 16 | 97.4% |
| 7 | $0.71(z_{17}^2 + z_{21}^2)(z_{29} - \frac{z_3}{1.4\sqrt{\lvert z_3\rvert}}) - 2.11$ | 19 | 97.6% |
| 8 | $(z_{17}^2 + z_{21}^2)(z_{29} - e^{z_3} + 0.71) - 2.03$ | 14 | 97.4% |
| 9 | $0.70(z_{17}^2 + z_{21}^2)(z_{29} - \frac{z_3}{z_3^2 + 0.66}) - 2.11$ | 19 | 97.5% |
| 10 | $(z_{29} - 0.44z_3)(z_{17}^2 + z_{21}^2) - 2.56$ | 13 | 97.4% |

Table 1. Hinge loss models. Symbolic expression, testing accuracy and complexity across ten Scheme 4 models trained using hinge loss. "Expression size" is the number of nodes in the expression tree. It differs from "complexity" since the latter is calculated in accordance with the additional penalty placed on $\sin(x)$, $e^x$ & $\log(x)$.

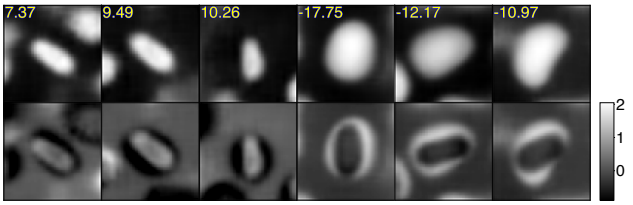
| No. | Expression | Expression size | Accuracy |
|---|---|---|---|
| 1 | $2.32\lvert z_{17}\rvert + z_{21}^2 + 4.46z_{29} - 3.16(0.56z_3 + 1)^2 - 6.15$ | 18 | 97.4% |
| 2 | $z_{21}^2 + 4.22(\sqrt{\lvert z_{17}\rvert} + \sin(z_{29}) - \sin z_3)) - 10.54$ | 16 | 96.9% |
| 3 | $z_{17}^2 + z_{21}^2 + z_{29} - 6.00e^{\sin(z_3)}$ | 13 | 97.0% |
| 4 | $z_{17}^2 + z_{21}^2 + z_{29} - z_3^2 - 8.20(0.35z_3 + 1)^2 + 2.13$ | 17 | 96.9% |
| 5 | $2\lvert z_{17}\rvert + z_{21}^2 + 3.72z_{29} - 3.72(0.52z_3 + 1)^2 - 4.94$ | 18 | 97.3% |
| 6 | $3.79z_{29} - \lvert z_{17}^2 + z_{21}^2 - 3.93(0.50x_3 + 1)^2 - 5.49\rvert + 1.45$ | 19 | 97.0% |
| 7 | $z_{17}^2 + z_{21}^2 + (z_3 + 3.99)(z_{29} - z_3) - 6.85$ | 15 | 97.4% |
| 8 | $z_{17}^2 + (\lvert z_{21}\rvert - z_3 + 2.55)(e^{\sin(z_{29})}) - 10.00$ | 15 | 97.0% |
| 9 | $z_{17}^2 + z_{21}^2 + 4.32(\sin(z_{29}) - \sin(z_3)) - 8.77$ | 18 | 97.1% |
| 10 | $(z_{17} - 0.25)^2 + z_{21}^2 + 3.76z_{29} - 3.16e^{z_3} - 3.53$ | 18 | 97.1% |

Table 2. MSE loss models. Symbolic expression, testing accuracy and complexity across ten Scheme 4 models trained using MSE loss.

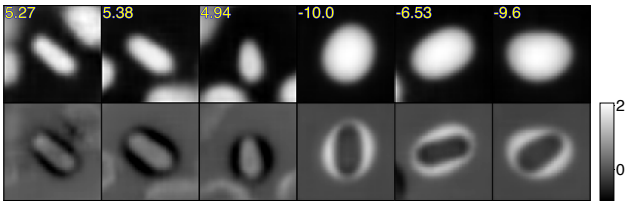
## 8. Failure modes

One significant benefit of interpretability is that it allows us to understand how and why our models fail when faced with out-of-distribution (OOD) data. Such situations can occur when the data input system produces erroneous outputs, or when the training data insufficiently represents the full range of possible inputs. One common example in automated microscopy is the collection of blank images, where all the pixels are zero-valued. This occurrence can arise from asynchrony between LED illumination and camera capture during experimental data collection.

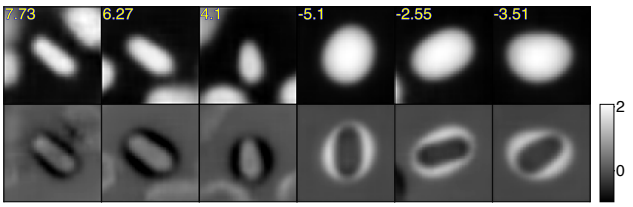
Blank images are not part of the training dataset of the $\beta$-TCVAE, therefore we would expect their reconstructions to be poor (Fig. 7). However, we are also interested in studying the output of our classification models when applied on them. All ten of our sparse models and all four of the symbolic expressions we analyzed classified these blank images as interphase. A cursory glance at the latent space encoding of the blank image reveals why (Fig. 8). In the eccentricity sub-space (consisting of variables $z_{17}$ & $z_{21}$). In the size sub-space ($z_3$ & $z_{29}$), the value of $z_{29}$ is negative; however,




(a) Attacks by Scheme 2 models.




(b) Attacks by Scheme 3 models.




(c) Attacks by Scheme 4 models.

Figure 6. Latent-based attacks at $\epsilon = 1.0$. Exp. H1 was used for **(c)**.

the value of $z_3$ is even lower. With these statements, it is possible to rationalize how we would obtain a negatively-valued output from our symbolic expressions, as well as the one sparse network we analyzed.

Therefore, even though this point in latent space does not encode any sensible cell image, it is possible to identify characteristics that explain the interphase classifications produced by our models. The only un-interpretable aspect of this classification is in the latent space encoding itself; since our $\beta$-TCVAE is un-interpretable with respect to OOD inputs, we can only speculate why the blank image was encoded to this specific point in the latent space. Nevertheless, the interpretability of our downstream models allow us to explain why they produce interphase classifications given this particular failure mode. We suggest that similar analyses could be used to study other failure modes in a wide range of contexts.

## References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework, 2019. arXiv:1907.10902 [cs, stat]. 2

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2011. 2

[3] Anna Bove, Daniel Gradeci, Yasuyuki Fujita, Shiladitya Banerjee, Guillaume Charras, and Alan R. Lowe. Local cellular neighborhood controls proliferation in cell competition. *Molecular Biology of the Cell*, 28(23):3215–3228, 2017. 1

[4] Christopher P. Burgess, Irina Higgins, Arka Pal, Loic Matthey, Nick Watters, Guillaume Desjardins, and Alexander Lerchner. Understanding disentangling in $\beta$-VAE, 2018. arXiv:1804.03599 [cs, stat]. 1

[5] Ricky T. Q. Chen, Xuechen Li, Roger Grosse, and David Duvenaud. Isolating Sources of Disentanglement in Variational Autoencoders, 2019. arXiv:1802.04942 [cs, stat]. 1

[6] Tim Dettmers and Luke Zettlemoyer. Sparse Networks from Scratch: Faster Training without Losing Performance, 2019. arXiv:1907.04840 [cs, stat]. 2

[7] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the Lottery: Making All Tickets Winners, 2019. 2

[8] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. 2016. 1

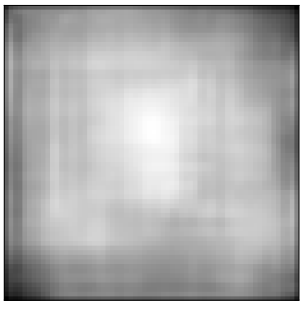
[9] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015. arXiv:1502.03167 [cs]. 6

[10] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, 2022. arXiv:1312.6114 [cs, stat]. 1

[11] Diganta Misra. Mish: A Self Regularized Non-Monotonic Activation Function, 2020. arXiv:1908.08681 [cs, stat]. 6

[12] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive

(a) Original.



(b) $\beta$-TCVAE reconstruction.

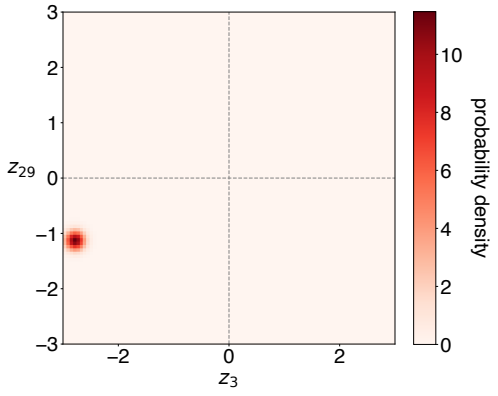
Figure 7. Blank image.

sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, 2018. Number: 1 Publisher: Nature Publishing Group. 2

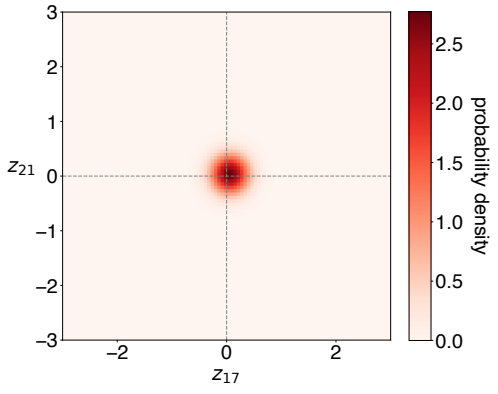
[13] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004. 7

[14] Christopher J. Soelistyo, Giulia Vallardi, Guillaume Charras, and Alan R. Lowe. Learning biophysical determinants of cell fate with deep neural networks. *Nature Machine Intelligence*, 4(7):636–644, 2022. Number: 7 Publisher: Nature Publishing Group. 1

[15] Kristina Ulicna, Giulia Vallardi, Guillaume Charras, and Alan R. Lowe. Automated Deep Lineage Tree Analysis Using a Bayesian Single Cell Tracking Approach. *Frontiers in Computer Science*, 3, 2021. 1




(a) Size sub-space




(b) Eccentricity sub-space

Figure 8. Blank image encoding. Gaussian distribution shown reflects the mean and variance of the encoding.