Dynamic Narrowing of VAE Bottlenecks Using GECO and L_0 Regularization

Cedric De Boom, Samuel Wauthier, Tim Verbelen, Bart Dhoedt
IDLab – Department of Information Technology, Ghent University – imec
Technologiepark-Zwijnaarde 126, 9052 Ghent, Belgium
firstname.lastname@ugent.be

***Abstract*—When designing variational autoencoders (VAEs) or other types of latent space models, the dimensionality of the latent space is typically defined upfront. In this process, it is possible that the number of dimensions is under- or overprovisioned for the application at hand. In case the dimensionality is not predefined, this parameter is usually determined using time- and resource-consuming cross-validation. For these reasons we have developed a technique to shrink the latent space dimensionality of VAEs automatically and on-the-fly during training using Generalized ELBO with Constrained Optimization (GECO) and the L_0-Augment-REINFORCE-Merge (L_0-ARM) gradient estimator. The GECO optimizer ensures that we are not violating a predefined upper bound on the reconstruction error. This paper presents the algorithmic details of our method along with experimental results on five different datasets. We find that our training procedure is stable and that the latent space can be pruned effectively without violating the GECO constraints.**

***Index Terms*—Variational autoencoder, VAE, latent space reduction, GECO, L_0 regularization, ARM.**

I. Introduction

DEEP neural networks are constructed and trained such that intermediate, latent representations of the input data are learned [1]. For this reason, deep learning and representation learning are two terms that are closely related. The goal of representation learning is to project high-dimensional data points such as images, documents, etc. into a vector space that is typically of low dimensionality compared to the original data points. In this space, similar data points ideally have similar representations, i.e. the points lie close to each other in the vector space in terms of some distance metric [2], [3]. The low number of dimensions and the similarity property pave the way for applications such as recommender systems (detecting similar items), anomaly detection (detecting contrasting items), data generation, data interpolation, etc. In these applications it is highly preferred that the vector space dimensionality is kept to a minimum needed for the task at hand. Not only does it allow for more efficient calculations in the data pipeline, it also lowers the required storage capacity.

Since the arrival of deep learning onto the machine learning scene, we have seen an explosion of models and techniques that are tuned to the task of representation learning and dimensionality reduction. As opposed to PCA and many other classic algorithms that model linear dependencies, deep neural networks have the powerful ability to learn highly non-linear mappings. One powerful class of such models are the autoencoders, where the input data is first projected ('encoded') onto a low-dimensional subspace, after which this projection is used to reconstruct the original input data itself ('auto') as well as possible. Mathematically, if the input of the autoencoder is a d-dimensional vector $\boldsymbol{x}$, the encoder $f_{\text{enc}} : \mathbb{R}^d \to \mathbb{R}^n$ produces a n-dimensional projection $\boldsymbol{z}$, with $n \ll d$. The decoder $g_{\text{dec}} : \mathbb{R}^n \to \mathbb{R}^d$ then takes $\boldsymbol{z}$ as input and produces an d-dimensional output $\boldsymbol{x}'$:

$$\boldsymbol{z} = f_{\text{enc}}(\boldsymbol{x})\,; \quad \boldsymbol{x}' = g_{\text{dec}}(\boldsymbol{z})\,. \tag{1}$$

From these equations, it becomes clear that autoencoders actually consist of two neural networks, which together create a diabolo architecture: the input and output dimensionality is the same, but the intermediate representation has a lower dimensionality, thereby creating the data 'bottleneck'. The projection $\boldsymbol{z}$ has also been called 'latent vector', 'latent code' or 'hidden representation', and 'latent space' is often used to denote the vector space of all possible codes.

To make sure the output $\boldsymbol{x}'$ resembles the input $\boldsymbol{x}$ as closely as possible, the loss function will usually contain a reconstruction error term reflecting either mean squared error (MSE), negative log-likelihood (NLL), a perceptual or adversarial loss, or any other metric or divergence expression (we will elaborate further on loss functions for autoencoders in Section II-A). There are of course many (hyper)parameters that determine the ability of the decoder to reconstruct the original input. One of these parameters is the width of the bottleneck, i.e. the dimensionality of the latent vector $\boldsymbol{z}$. The stronger the bottleneck, the more the input data will be compressed by the encoder, thereby potentially causing information loss, which adheres to the information bottleneck (IB) principle as a "trade-off between compression and prediction" [4], [5]. The amount and nature of the compression depends highly on the model architecture and the reconstruction loss. For example, if we decode an entire image by minimizing a pixel-based L_2 error, it is known that the reconstructions will be an increasingly blurred out version of the original image when the bottleneck gets tightened [6], [7], causing the latent vectors to focus on low-frequency information.

Similar to PCA, the bottleneck dimensionality of autoencoders is predefined in the model architecture. For a given problem statement or use case, one typically provisions an adequate amount of latent dimensions in such a way that the application's requirements are met. Requirements can be quality-centered, e.g. accuracy or reconstruction error; performance-

based, such as memory consumption or processing time; or application-driven, for example if the latent vectors in existing subsystems already have a fixed dimensionality. The general conception reads "the more latent dimensions, the better", at least in order to improve the quality-based metrics. This line of reasoning is indeed backed by the information theoretic result that the conditional entropy of a random variable can only decrease for every additional random variable we condition on. More specifically for autoencoders, increasing the latent space with extra dimensions might reduce the uncertainty about the decoder output. Indeed, it is in general always possible for the decoder to ignore the extra information in the latent vector if it would lead to deteriorated reconstructions.

In this paper we will try to answer the exact opposite question: to what extent can we eliminate dimensions from the state space with a minimal sacrifice on the output quality? Or, more explicitly, given a set of conditions that need to be satisfied—e.g. a maximum classification error—can we effectively prune away latent dimensions without violating these conditions. This can be achieved by often resource- and time-consuming hyperparameter optimization techniques. The core of this work, however, is to devise a learning scheme that allows us to dynamically prune the latent space on-the-fly *during* training, without any additional hyperparameter tuning.

We will show that VAE bottlenecks can effectively be pruned with our methodology. And even more, we will provide convincing experimental evidence that more dimensions can be eliminated if the constraints are relaxed; that is, if we are satisfied with an overall higher reconstruction error.The other way around, pushing hard on the reconstruction quality will require more latent dimensions to compress the data, which clearly demonstrates the information bottleneck trade-off. The remainder of this paper is structured as follows. First, we will give background information on VAEs and GECO, after which we will provide details on the L_0-ARM (Augment-REINFORCE-Merge) gradient estimator [8], [9]. In Section III our training procedure will be explained. Finally, we will conduct a set of experiments to validate our approach on five different datasets in Section IV.

II. Background Material

In this section we will cover the theoretical building blocks that are used in the algorithm of Section III. We will first cover variational autoencoders and the GECO extension optimization procedure. Afterwards we will explain the details of the L_0-ARM gradient estimator.

A. Variational Autoencoders and GECO

In variational autoencoders (VAEs) the encoder models a distribution $q_\theta(\boldsymbol{z} \mid \boldsymbol{x})$ over latent vectors [10].This distribution is called the approximate posterior and is learned to approximate the true, but unknown posterior distribution $p(\boldsymbol{z} \mid \boldsymbol{x})$. This is typically done by modeling $q_{\boldsymbol{\theta}}$ as a Gaussian distribution with diagonal covariance. In practice, the neural network $f^{\boldsymbol{\theta}}_{\text{enc}} : \mathbb{R}^d \to \mathbb{R}^n \times \mathbb{R}^n$ with parameters $\boldsymbol{\theta}$ produces both the mean and logarithmic standard deviations of this distribution given some input data $\boldsymbol{x}$. A latent vector $\boldsymbol{z}$ is then sampled from the approximate posterior as follows:

$$\begin{aligned}(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) &= f^{\boldsymbol{\theta}}_{\text{enc}}(\boldsymbol{x})\,; \\ \boldsymbol{z} &\sim q_\theta(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}))\,.\end{aligned} \tag{2}$$

The decoder neural network $g^{\boldsymbol{\phi}}_{\text{dec}} : \mathbb{R}^n \to \mathbb{R}^d$ with parameters $\boldsymbol{\phi}$ models the likelihood distribution $p_{\boldsymbol{\phi}}(\boldsymbol{x} \mid \boldsymbol{z})$, and produces a reconstructed data point:

$$\boldsymbol{x}' = g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z})\,. \tag{3}$$

Both the encoder and decoder networks are optimized using the negative ELBO [10]:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = &-\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})}[\log p_{\boldsymbol{\phi}}(\boldsymbol{x} \mid \boldsymbol{z})] \\ &+ D_{\mathrm{KL}}[q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) \parallel p(\boldsymbol{z})]\,.\end{aligned} \tag{4}$$

The first term in this loss function represents the negative log-likelihood of the data given a latent vector sampled from the posterior distribution. Differentiability of this term is ensured by the so-called reparameterization trick, in which sampling $\boldsymbol{z}$ from the posterior is replaced by sampling a noise vector $\boldsymbol{\epsilon}$ from a standard normal distribution and rewriting $\boldsymbol{z}$ as a differentiable function of $\boldsymbol{\epsilon}$:

$$\begin{aligned}(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) &= f^{\boldsymbol{\theta}}_{\text{enc}}(\boldsymbol{x})\,; \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\boldsymbol{z}; \boldsymbol{0}, I)\,; \\ \boldsymbol{z} &= \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}.\end{aligned} \tag{5}$$

Here, $\odot$ denotes an element-wise vector product. In the second term of Equation (4), $p(\boldsymbol{z})$ is the prior distribution over latents, which is often fixed to a standard normal with zero mean and identity covariance matrix: $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{0}, I)$.

The KL divergence in Equation (4) can be regarded as a regularization term, as it tries to pull most of the latent factors close to a standard normal distribution. To make the regularization explicit and tunable, Higgins et al. proposed β-VAE in which β is the regularization coefficient [11]:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}, \beta) = &-\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})}[\log p_{\boldsymbol{\phi}}(\boldsymbol{x} \mid \boldsymbol{z})] \\ &+ \beta\, D_{\mathrm{KL}}[q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) \parallel p(\boldsymbol{z})]\,.\end{aligned} \tag{6}$$

Increasing β puts more weight on the KL term, thereby implicitly tightening the information bottleneck. Indeed, Higgins et al. argue that varying and choosing an appropriate value of β represents trading off reconstruction quality vs. latent channel capacity, and that it is generally advised to set $\beta > 1$ in order to arrive at a disentangled latent space [11].

In contrast, Rezende and Viola propose Generalized ELBO with Constrained Optimization (GECO), which rephrases the negative ELBO loss function in terms of a constrained optimization problem using a Lagrange multiplier λ [12], [13]:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}, \lambda) = &\, D_{\mathrm{KL}}[q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) \parallel p(\boldsymbol{z})] \\ &+ \lambda\, \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})}\Big[\mathcal{C}\Big(\boldsymbol{x}, g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z})\Big)\Big]\,.\end{aligned} \tag{7}$$

This Lagrangian optimizes the KL divergence subject to $\mathbb{E}_{\boldsymbol{z}}[\mathcal{C}(\boldsymbol{x}, g_{\text{dec}}(\boldsymbol{z}))] \leq 0$, for a given constraint function $\mathcal{C}$ for which $\mathcal{C}(\boldsymbol{x}, g_{\text{dec}}(\boldsymbol{z})) \in \mathbb{R}$ (we have left out the parameters for

reading comfort)[1]. A GECO constraint can essentially be any condition or metric the VAE needs to satisfy, e.g. an upper bound on a predefined reconstruction (L_2) error:

$$\mathcal{C}\Big(\boldsymbol{x}, g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z})\Big) = \Big\|\boldsymbol{x} - g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z})\Big\|_2 - \tau, \tag{8}$$

or any other useful divergence. The hyperparameter $\tau \geq 0$ represents a desired tolerance or upper bound on the reconstruction error, such that $\mathbb{E}_{\boldsymbol{z}}[\mathcal{C}(\boldsymbol{x}, g_{\text{dec}}(\boldsymbol{z}))]$ indeed becomes negative if the upper bound is satisfied. One might argue that tuning τ is not easier than picking an appropriate β in β-VAE. However, tuning parameters of latent spaces is an abstract operation, while tuning quality metrics in the data space is much more tangible and practical since its effect can be observed directly in the data reconstruction quality.

Optimizing the Lagrangian in Equation (7) is done through min-max optimization: it is minimized w.r.t. parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, and maximized w.r.t. λ. The details on how to optimize the multiplier λ are explained in the original work by Rezende and Viola, and will be further clarified in Section III.

B. L_0 Regularization and ARM

There has been considerable effort in the past to sparsify neural networks by pruning redundant or low-magnitude weights [14], [15], [16]. Louizos et al. introduced a learning method that employs L_0 regularization [17]. The L_0 norm of a numerical vector represents the number of non-zero components, so that minimizing the L_0 norm comes down to setting as much vector components to zero as possible. This is useful for sparsification of neural networks as follows. Consider the vector $\boldsymbol{w}$ of all layer weights in a network, and a binary vector $\boldsymbol{\nu}$ (containing only 0s and 1s) with the same dimensionality as $\boldsymbol{w}$. By performing the element-wise vector product $\boldsymbol{w} \odot \boldsymbol{\nu}$, the vector $\boldsymbol{\nu}$ acts as an on–off gating mechanism: for every 0 in $\boldsymbol{\nu}$ the corresponding weight in the neural network is essentially eliminated from the computational graph. Therefore, L_0 regularization on $\boldsymbol{\nu}$ comes down to eliminating weights from the neural network.

Minimizing the L_0 norm $\|\boldsymbol{\nu}\|_0$ in a gradient descent context is, however, not straightforward, since it is not differentiable w.r.t. $\boldsymbol{\nu}$. More specifically, because $\|\boldsymbol{\nu}\|_0$ takes discrete values, it has zero gradients everywhere in its domain. Gradient estimators such as REINFORCE [18], Straight-Through estimation [19], and Hard Concrete estimation [17] can overcome this issue. More recently, Augment-REINFORCE-Merge or ARM was presented as an unbiased and low-variance gradient estimator that outperforms previously mentioned estimators on a variety of tasks [9], and it has been applied successfully to L_0-based network sparsification by Li and Ji shortly after its introduction [8]. It works as follows, given an arbitrary neural network f with n parameters $\boldsymbol{w}$, a gating vector $\boldsymbol{\nu}$ with dimensionality n, and a loss function $\mathcal{L}_{\boldsymbol{w},\boldsymbol{\nu}}$ for which:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{w},\boldsymbol{\nu}}(\boldsymbol{x}, \beta) &= \mathcal{E}(f(\boldsymbol{x}; \boldsymbol{w} \odot \boldsymbol{\nu}), y) + \beta \|\boldsymbol{\nu}\|_0 \\ &= \mathcal{E}(f(\boldsymbol{x}; \boldsymbol{w} \odot \boldsymbol{\nu}), y) + \beta \sum_{j=1}^{n} \mathbb{1}_{[\nu_j \neq 0]}. \end{aligned} \tag{9}$$

In the above expression, $\mathcal{E}$ represents any error metric between the output of f and a label y, $\mathbb{1}$ is the indicator function, and β is a regularizer. Both terms in Equation (9) are not differentiable w.r.t. $\boldsymbol{\nu}$. To overcome this issue, we model the components ν_j as samples from a Bernoulli distribution with parameters π_j: $\nu_j \sim \text{Ber}(\nu; \pi_j)$. An upper bound of $\min_{\boldsymbol{\nu}} \mathcal{L}_{\boldsymbol{w},\boldsymbol{\nu}}$ can then be calculated [20]:

$$\begin{aligned}&\min_{\boldsymbol{\nu}} \mathcal{L}_{\boldsymbol{w},\boldsymbol{\nu}}(\boldsymbol{x}, \beta) \\ &\leq \mathbb{E}_{\boldsymbol{\nu} \sim \prod_{j=1}^{n} \text{Ber}(\nu;\pi_j)}[\mathcal{E}(f(x; \boldsymbol{w} \odot \boldsymbol{\nu}), y)] + \beta \sum_{j=1}^{n} \pi_j. \end{aligned} \tag{10}$$

We write the right-hand side of (10) as $\hat{\mathcal{L}}_{\boldsymbol{w},\boldsymbol{\gamma}}$. The regularizing term is now differentiable w.r.t. $\boldsymbol{\pi}$, but the first term is still problematic. We will use ARM to estimate the gradients of this term, after writing the Bernoulli parameters $\boldsymbol{\pi}$ as a function of logit parameters $\boldsymbol{\gamma}$: $\pi_j = \sigma(\gamma_j)$. The function $\sigma : \mathbb{R} \to [0, 1]$ is ideally antithetic and symmetric around its inflection point, e.g. the sigmoid function: $\sigma(\gamma) = 1/(1+\exp(-\gamma))$. With this reparameterization, and the shorthand notation $\mathcal{F}(\boldsymbol{\nu}) = \mathcal{E}(f(x; \boldsymbol{w} \odot \boldsymbol{\nu}), y)$, Equation (10) can be written as [9]

$$\begin{aligned}&\hat{\mathcal{L}}_{\boldsymbol{w},\boldsymbol{\gamma}}(\boldsymbol{x}, \beta) \\ &= \mathbb{E}_{\boldsymbol{u} \sim \prod_{j=1}^{n} \text{Unif}(u;0,1)}\big[\mathcal{F}(\mathbb{1}_{[\boldsymbol{u} < \sigma(\boldsymbol{\gamma})]})\big] + \beta \sum_{j=1}^{n} \sigma(\gamma_j). \end{aligned} \tag{11}$$

Here, sampling gates from a Bernoulli distribution is replaced by sampling from a uniform distribution between 0 and 1. The gradient of $\hat{\mathcal{L}}_{\boldsymbol{w},\boldsymbol{\gamma}}$ w.r.t. $\boldsymbol{\gamma}$ can now be estimated—unbiased and with low variance—as [8], [9]:

$$\begin{aligned}&\nabla_{\boldsymbol{\gamma}} \hat{\mathcal{L}}_{\boldsymbol{w},\boldsymbol{\gamma}}(\boldsymbol{x}, \beta) \\ &\approx \mathbb{E}_{\boldsymbol{u} \sim \prod_{j=1}^{n} \text{Unif}(u;0,1)}\Bigg[\\ &\quad \Big(\mathcal{F}(\mathbb{1}_{[\boldsymbol{u} > \sigma(-\boldsymbol{\gamma})]}) - \mathcal{F}(\mathbb{1}_{[\boldsymbol{u} < \sigma(\boldsymbol{\gamma})]})\Big)\left(\boldsymbol{u} - \frac{1}{2}\right)\Bigg] \\ &\quad + \beta \sum_{j=1}^{n} \nabla_{\boldsymbol{\gamma}} \sigma(\gamma_j). \end{aligned} \tag{12}$$

This simple estimator has but one disadvantage, which is that two forward passes through the network are required. We also have to point out that σ is smooth and not guaranteed to be exactly zero for closed gates. However, in line with the original research by Li and Ji, we observe that ν_j is either close to 0 or close to 1 after training, which means that after sampling the gates are almost always entirely closed or entirely open [8]. For this reason, at inference time, we decide to explicitly close gates for which $\sigma(\gamma_j) \leq 0.5$, i.e. ν_j is set to 0, such that the corresponding latent dimensions are effectively eliminated.

[1]The original, more general formulation allows for multiple constraints and Lagrange multipliers, but in this work we will focus on a single constraint.

Why we need L_0 regularization: With L_0 regularization we effectively achieve the desired on-off behavior: gates are completely open or closed. In the past, L_1 and L_2 regularization have been used to prune weights and filters from (convolutional) neural networks [21]. This ensures that the components become smaller, but not exact zero [8]. And this is needed for the purpose of pruning, since a very tiny but non-zero component is still able to carry information, and is therefore not eliminated effectively. Put differently, it is possible to make the L_1 norm of all gates arbitrarily small since the corresponding weights can be made arbitrarily large. This can be countered by applying additional L_2 regularization on the weights themselves, but it would pose an extra optimization trade-off. Earlier work has therefore often proceeded in a multi-stage fashion: first train the network using L_1 regularization, then prune all small weights below a certain threshold, and finally finetune the (smaller) network [21], [22]. In this work, on the other hand, we only need a single stage in which the network is both trained and pruned at the same time. Next to this, we do not need an additional threshold hyperparameter to determine the fraction of weights that will be pruned. Instead we use GECO to make this decision entirely transparent, as will be explained in the next section.

III. Shrinking VAE bottlenecks

In this section we will describe how to combine GECO optimization and the L_0-ARM gradient estimator in order to narrow VAE bottlenecks. The result of our methodology will be a novel VAE optimization scheme that effectively learns to project data points in a latent space while *at the same time* reducing that latent space's dimensionality and ensuring high-quality reconstructions. Before diving into the details, we first take a look at a couple of issues with the original L_0-ARM weight pruning technique. First, as Li and Ji state, sparsity strength—i.e. a measure for the number of weights that can be deleted—needs to be tuned explicitly by the regularization parameter β [8]. Apart from this regularization parameter, gates are encouraged to close themselves deliberately without any explicitly imposed constraints. Even more, in the original text the authors show that most of the gates are already settled to open or closed after ca. 50 training epochs. That is, often well before convergence of the main metric, i.e. the classification or regression error, and this causes a "drift" between pruning and performance optimization. Another concern is that the prune rate highly depends on the value of the regularization parameter, and setting this parameter precisely according to one's needs is not transparent and requires hyperparameter optimization. Finally, since the regularization parameter is our only proxy to tune the prune rate, we have little control over the final performance of the network. For example, if we are satisfied with a slightly lower predictive accuracy, we can decide to increase the prune rate. This operation will indeed come at a performance cost, but it is unclear by how much the performance will deteriorate. Therefore, we have to resort to a complete hyperparameter sweep. With GECO, however, it is possible to set an upper bound on the reconstruction error during training. So, instead of tuning the prune rate implicitly with a regularization parameter, we will *explicitly* set a maximum on the reconstruction error and prune as many dimensions as possible without violating this maximum.

Input : Dataset $\mathcal{D}$
Parameters : τ, α, k, $\lambda_{\min}$, $\lambda_{\max}$
Results : Learned parameters $\boldsymbol{\theta}$, $\boldsymbol{\phi}$, $\boldsymbol{\gamma}$ and λ

hit $\leftarrow$ False
foreach $\boldsymbol{x}_i \in \mathcal{D}$ **do**
  if *hit* **then**
    Sample $\boldsymbol{u} \sim \prod_{j=1}^{n} \mathrm{Unif}(u; 0, 1)$
    $\boldsymbol{\nu}_1 \leftarrow \mathbb{1}_{[\boldsymbol{u} > \sigma(-k\boldsymbol{\gamma})]}$
    $\boldsymbol{\nu}_2 \leftarrow \mathbb{1}_{[\boldsymbol{u} < \sigma(k\boldsymbol{\gamma})]}$
  else
    $\boldsymbol{\nu}_1 \leftarrow \mathbf{1}$
    $\boldsymbol{\nu}_2 \leftarrow \mathbf{1}$
  end
  Sample $\boldsymbol{z} \sim f_{\text{enc}}^{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ with reparameterization trick
  $\mathcal{E}_1 \leftarrow \left\| g_{\text{dec}}^{\boldsymbol{\phi}}(\boldsymbol{z} \odot \boldsymbol{\nu}_1) - \boldsymbol{x}_i \right\|_2$
  $\mathcal{E}_2 \leftarrow \left\| g_{\text{dec}}^{\boldsymbol{\phi}}(\boldsymbol{z} \odot \boldsymbol{\nu}_2) - \boldsymbol{x}_i \right\|_2$
  $\mathcal{C} \leftarrow \mathcal{E}_2 - \tau$
  if $i == 0$ **then**
    $\mathcal{C}_{\text{ma}} \leftarrow \mathcal{C}$
  else
    $\mathcal{C}_{\text{ma}} \leftarrow \alpha \cdot \mathcal{C}_{\text{ma}} + (1 - \alpha) \cdot \mathcal{C}$
  end
  $\mathcal{C}' \leftarrow \mathcal{C} + \mathrm{StopGradient}(\mathcal{C}_{\text{ma}} - \mathcal{C})$
  $\lambda' \leftarrow \max(\min(\mathrm{softplus}(\lambda)^2, \lambda_{\max}), \lambda_{\min})$
  $\mathcal{L} \leftarrow \lambda' \cdot \mathcal{C}' + \frac{1}{\|\boldsymbol{\nu}_2\|_0} \left\| \mathrm{KL}_{\text{elem}}\left[f_{\text{enc}}^{\boldsymbol{\theta}}(\boldsymbol{x}_i) \,\middle\|\, p(\boldsymbol{z}) \right] \odot \boldsymbol{\nu}_2 \right\|_1$
  if $\mathcal{C} \leq 0$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} + \sum_j \sigma(k\gamma_j)$
    hit $\leftarrow$ True
  end
  $\nabla_{\boldsymbol{\phi},\boldsymbol{\theta},\boldsymbol{\gamma},\lambda} \mathcal{L} \leftarrow \mathrm{Backpropagation}(\mathcal{L})$
  if *hit* **then**
    $\nabla_{\boldsymbol{\gamma}} \mathcal{L} \leftarrow \nabla_{\boldsymbol{\gamma}} \mathcal{L} + \lambda' \cdot k \cdot (\mathcal{E}_1 - \mathcal{E}_2) \cdot \left(\boldsymbol{u} - \frac{1}{2}\right)$
  end
  Optimize parameters $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, $\boldsymbol{\gamma}$ and λ with Adam
end

Algorithm 1: Optimization procedure to shrink the bottleneck dimensionality of a VAE using GECO and L_0-ARM gradient estimation.

Our procedure, as shown in Algorithm 1, is simple yet effective at shrinking the bottleneck dimensionality of VAEs. At the core is a basic VAE architecture consisting of an encoder $f_{\text{enc}}^{\boldsymbol{\theta}}$ and decoder $g_{\text{dec}}^{\boldsymbol{\phi}}$ neural network, as described in Section II-A, and a global gating vector $\boldsymbol{\nu}$ as explained in Section II-B. The vector $\boldsymbol{\nu}$ has the same number of dimensions as the VAE latent space. For a given data point $\boldsymbol{x}$, we determine its latent vector $\boldsymbol{z}$ by sampling from the output of the encoder using the reparameterization trick: $\boldsymbol{z} \sim f_{\text{enc}}^{\boldsymbol{\theta}}(\boldsymbol{x})$

(line 11 in Algorithm 1). Next, the sampled vector $\boldsymbol{z}$ is gated through vector $\boldsymbol{\nu}$ using the operation $\boldsymbol{z} \odot \boldsymbol{\nu}$, thereby deactivating some of the components. The resulting vector is used by the decoder to reconstruct the original data point: $\boldsymbol{x}' = g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z} \odot \boldsymbol{\nu})$. To measure the reconstruction quality, we have chosen to use the squared error between the original and reconstructed data points, as shown in lines 12 and 13. By analogy with Equation (8), we can now define an upper-bound constraint as the difference between the reconstruction error and a pre-defined threshold τ (line 14). At this point we are able to define the constrained optimization problem that we are solving, in which the Lagrangian incorporates an additional L_0 regularization term on the gates:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi},\boldsymbol{\nu}}(\boldsymbol{x},\lambda) = D_{\text{KL}}\big[f^{\boldsymbol{\theta}}_{\text{enc}}(\boldsymbol{x}) \,\big\|\, p(\boldsymbol{z})\big] + \|\boldsymbol{\nu}\|_0 \\ + \lambda\, \mathbb{E}_{\boldsymbol{z}}\Big[\mathcal{C}\Big(\boldsymbol{x}, g^{\boldsymbol{\phi}}_{\text{dec}}(\boldsymbol{z} \odot \boldsymbol{\nu})\Big)\Big]\,.\end{aligned} \tag{13}$$

Lines 14 to 26 in Algorithm 1 comprise the GECO optimization details, as implemented by Rezende and Viola [13]. In line 22 we calculate both the constraint term and the KL regularization. For the latter we have introduced the notation KL_{elem} to denote the vector of element-wise KL components. More specifically, given that $(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = f^{\boldsymbol{\theta}}_{\text{enc}}(\boldsymbol{x})$, parameterizing a multidimensional Gaussian distribution with mean $\boldsymbol{\mu}$ and diagonal covariance $\text{diag}(\boldsymbol{\sigma})$ as explained in Equation (2), and $p(\boldsymbol{z})$ a standard normal distribution, the j'th component of the element-wise KL is calculated as:

$$\begin{aligned}&\text{KL}_{\text{elem}}\big[f^{\boldsymbol{\theta}}_{\text{enc}}(\boldsymbol{x}) \,\big\|\, p(\boldsymbol{z})\big]_j \\ &\quad = -\frac{1}{2}\left(1 + \log \boldsymbol{\sigma}_j - \boldsymbol{\sigma}_j - \boldsymbol{\mu}_j^2\right).\end{aligned} \tag{14}$$

Since the right-hand side is always positive, line 22 is just a compact way of writing the average KL divergence, thereby taking into account the pruned components. After all, once a dimension is eliminated, its KL divergence should not weigh on the optimization loss anymore. Regarding the constraint term, we entertain a moving average which simulates the expectation of the constraint w.r.t. $\boldsymbol{z}$, as defined in Equation (13). The parameter α is set to control the rate of the moving average, with a value typically just slightly lower than 1. For efficiency reasons, we stop the gradients from flowing through this moving average, i.e. only the current constraint (at step i) is involved in backpropagation. We also perform a monotonous squared softplus operation on λ to make sure that the multiplier is positive and is able to 'move fast', i.e. to decrease quickly once the constraint is met, and vice versa. We also clamp the values of λ' such that they cannot become arbitrarily small or large. This is needed to prevent λ' from growing exponentially. In line 24, we add the L_0 regularization term to the overall loss when the constraint is satisfied for the current data point, i.e. we only allow gates to close themselves when we have some wiggle room and are not violating constraints. This is needed to stabilize the optimization procedure: if we are in the process of eliminating a dimension by gradually decreasing the corresponding $\boldsymbol{\gamma}$ component, the number one priority is maintaining the required reconstruction loss. If this loss would increase too much, we should not proceed, but instead focus on improving the model parameters.

The remaining lines of Algorithm 1 constitute the L_0-ARM gradient estimation of the gating vector. First, notice the *hit* variable which is set to True once the constraint is satisfied. Therefore, at the start of training, the gates are set wide open, and only when the constraints are achieved for the first time, we will start the L_0-ARM optimization. The parameter k in lines 8 and 9 is introduced in order to scale the sigmoid function; a typical value of $k > 1$ will make the sigmoid shape steeper, thereby allowing faster transitions between open and closed gates. Most operations are self-explanatory and are in line with Equation (12). We do want to point out that we only need to sample a single latent vector from the encoder, but that we need two forward passes through the decoder in lines 12 and 13. The resulting reconstruction errors are then used to update the gradients for $\boldsymbol{\gamma}$ in line 29. In this equation, according to the chain rule, we need to multiply by k and by λ'. Finally, in the last line, all parameters are optimized using Adam—or any other gradient descent flavor.

IV. Experiments

We will evaluate our methodology in a variety of settings on five different, standard image datasets. We consider five image datasets, of which the first three are typically used to evaluate VAE performance and latent space disentanglement. The last two datasets contain realistic images, are not artificially constructed, and have been extensively used in the past to benchmark different varieties of machine learning models.

- dSprites (Shapes2D) [23]: 64×64 images of white 2D shapes on a black background. There are five degrees of freedom: shape (square, ellipse or heart), scale, rotation, x-position, y-position.
- Shapes3D [24]: 64×64 images of 3D shapes in a 3D environment. There are six degrees of freedom: shape, scale, rotation, floor hue, wall hue, object hue.
- Cars3D [25]: 64×64 images of 3D car models on a white background. There are three degrees of freedom: object type, elevation, azimuth.
- MNIST: 28×28 images of handwritten digits.
- CIFAR-100: 32×32 images of real-life pictures of objects; we upscale the images to 64×64 pixels.

We use a standard convolutional neural network as architecture for the encoder $f^{\boldsymbol{\theta}}_{\text{enc}}$. For input color images of $64 \times 64 \times 3$ pixels, we use four convolutional layers with stride 2 and kernel sizes of 4×4 pixels for the first layer and 3×3 pixels for the subsequent layers. Each of these layers is fed through a leaky ReLU activation with leakiness 0.02 and a BatchNorm layer with momentum 0.8. The output of the final convolutional layer is flattened and used as input for a linear layer with 128 output neurons. A final linear layer is applied which calculates both a mean and variance prediction; this layer therefore contains $2 \cdot n$ output neurons, with n the dimensionality of the latent space. For MNIST images of $28 \times 28 \times 1$ pixels, there are only three convolutional layers with resp. 4×4, 3×3 and 2×2 filters. The decoder

architecture is a mirrored version of the encoder network. Two linear layers transform the latent vector into the input of four consecutive convolutional layers. Each of the convolutions has a 3×3 kernel with stride 1, and is preceded by a BatchNorm layer and an upsampling operation which increases the image size by 2 through nearest neighbor interpolation. Between each layer we apply a leaky ReLU function, and the final activation is a regular sigmoid to obtain pixel values between 0 and 1. For MNIST we have again three convolutional layers with resp. kernel sizes 2×2 and two times 3×3.

In all experiments $k = 7$, $\alpha = 0.99$, $\lambda_{\min} = 10^{-5}$, $\lambda_{\max} = 5$, batch size is 64, learning rate is 10^{-3}, and we use Adam as optimizer. We initialize $\lambda = 1$ and all components of $\boldsymbol{\gamma}$ are initialized to 0.42 such that $\sigma(k \cdot 0.42) \approx 0.95$, which means that a gate is open with probability 95%. This is done to ensure training stability.

A. *Visualizing the training procedure*

In a first experiment we will observe the training behavior by observing four different quantities over time: the mean squared reconstruction error (MSE), the number of gates that are open or closed, the Lagrange multiplier λ, and the fraction of data points within a batch that satisfy the constraint. These graphs are shown in Figure 1 in which we have trained a VAE on dSprites for 50K batches, with $n = 10$, i.e. the original—and therefore maximum—dimensionality of the VAE. At the start of training we see that λ first increases until the MSE dives below the threshold τ, after which it drops to a small value. This has the effect that the regularization terms gain more importance, and after ca. 10K batches we notice that the gates are gradually starting to close themselves, which is visible in the descending staircase pattern in the second plot. This plot shows the L_1 norm of the smoothed gate vector $\sigma(k\boldsymbol{\gamma})$, which is essentially the sum of the vector components. The y-axis of the graph therefore represents a continuous measure of the number of open gates. Each plateau in the graph corresponds to an integer amount of opened gates, so we see that after 10K batches there are 9 open gates, which drops to 6 gates at 20K batches, and to 5 gates at 25K batches. Of course, when more gates are closed, the reconstruction errors are expected to increase, which can be observed in the first graph, and the GECO tolerance hit/miss ratio will drop. Whenever the MSE is above τ, the multiplier λ will increase in order to temporarily attribute more weight to the reconstruction error. We can visually distinguish two such increases around 25K and 35K batches in the third plot. As a final remark, we want to bring to attention the oscillatory behavior of the gates after 40K batches. These oscillations come from the effect that in the process of closing an additional gate, the constraint is violated severely, which causes λ to increase in order to lower the MSE in favor of the regularization terms, thereby opening the gate again. In other words, the model is constantly trying to close a gate, but it is obstructed by the GECO mechanism not allowing any heavy constraint violations.

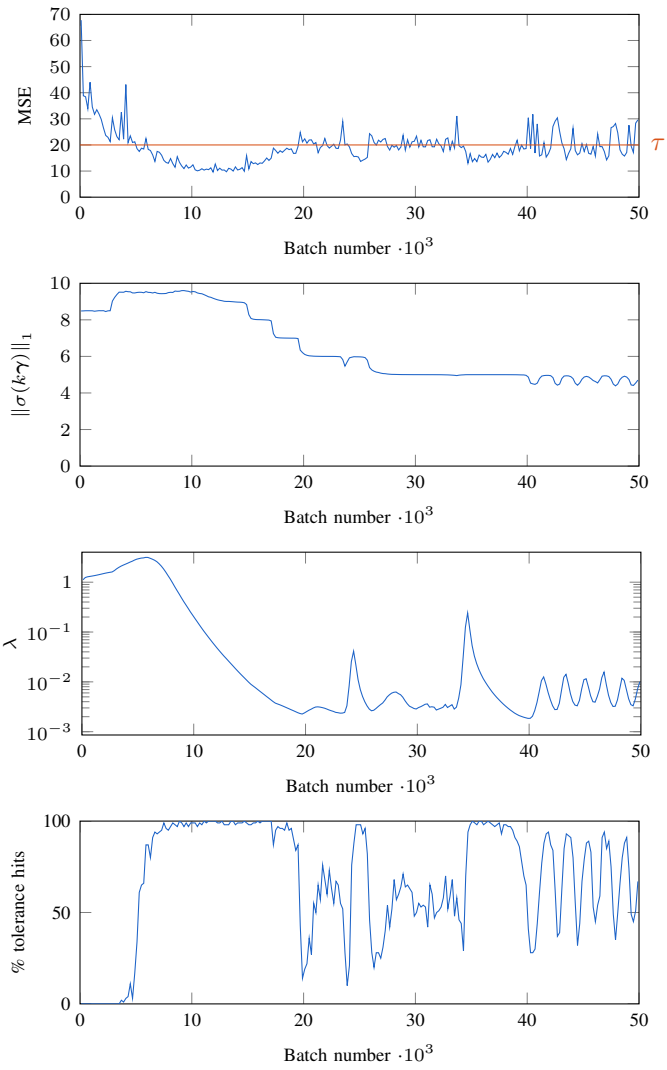

Fig. 1: Overview of four quantities during a typical training run: MSE with the GECO threshold τ indicated by a red line, the L_1 norm of the continuous gate vector (i.e. the sum of vector components), the Lagrange multiplier λ and a measure for the percentage of GECO tolerance hits within a batch.

B. *Visualizing the information bottleneck*

We will now vary the GECO tolerance and observe the final bottleneck dimensionality after training. This allows us to test whether an increased tolerance will lead to a tighter bottleneck and vice versa. We perform tests on the MNIST dataset, for which we set the initial dimensionality $n = 32$. The GECO tolerance is set to different MSEs of 3, 4, 6, 8, 10, 14, 18, 22, 26 and 30, and train for 2000 epochs. The resulting bottleneck pareto front is shown in Figure 3, and we can clearly observe the inverse relationship between the threshold τ and the number of retained latent dimensions $\|\boldsymbol{\nu}\|_0$. For $\tau = 3$ the model learns to withhold 27 out of 32 dimensions, while for $\tau = 30$ we only need 2 dimensions. For this specific dataset, neural architecture and training procedure, Figure 3

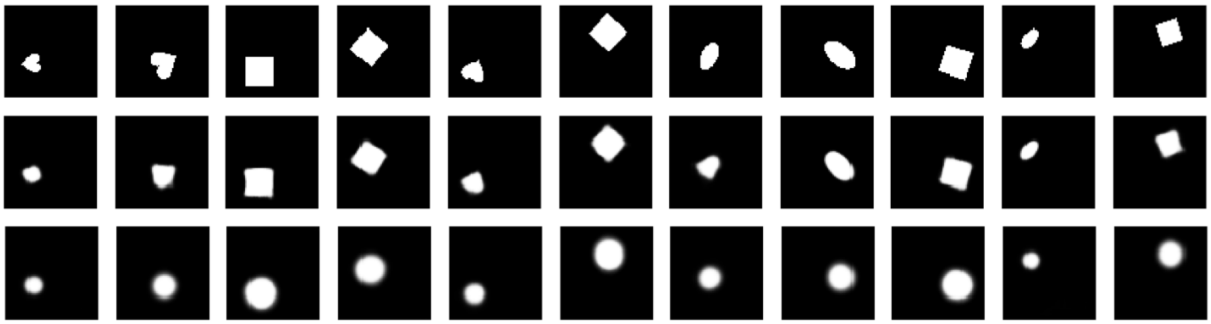

Fig. 2: Original dSprites images (top row) are reconstructed by a VAE trained with $\tau = 25$ resulting in 5 dimensions (middle row), and $\tau = 35$ resulting in 3 dimensions (bottom row).

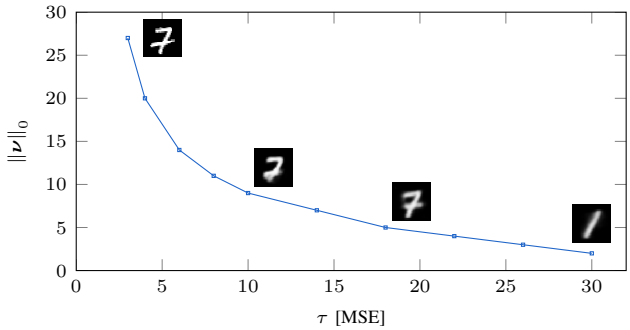

Fig. 3: This plot represents the information bottleneck pareto front for a convolutional VAE trained on MNIST. It shows the number of retained dimensions in the latent space $\|\nu\|_0$ as a function of the GECO threshold τ.

makes the information bottleneck pareto front very visual and tangible, since it shows to what extent it is possible to tighten the bottleneck for a given GECO threshold. Next to the plotted bottleneck line, we also show reconstructions of a given image of the number 7 for a few selected points on the graph. We can clearly see that the quality of the reconstructions deteriorates as we increase τ—for $\tau = 30$ the construction resembles a 1 rather than a 7—and that they also become blurrier. Based on visual samples, it is relatively easy to select the GECO threshold one is comfortable with for the application at hand.

C. Comparing reconstruction qualities

In the next experiment we train two VAEs on the dSprites dataset with different settings for the GECO threshold: $\tau = 20$ and $\tau = 35$ (in MSE). For $\tau = 20$ this results in 5 latent dimensions, while $\tau = 35$ gives 3 dimensions. As mentioned before, the dSprites dataset is constructed with 5 degrees of freedom (shape, scale, rotation, x- and y-position). In Figure 2 we can observe that in the middle row all five variables are used to reconstruct the original image. In the bottom row, however, we only see blurred circles at the positions where the original shapes are located. This means that both shape and rotational information is discarded, which indeed corresponds to the 3 retained latent dimensions instead of 5. While this line of reasoning may sound logical, it is far from a rigorous proof that the 3 dimensions indeed each correspond with the remaining degrees of freedom (scale, x- and y-position). Since we are not explicitly striving for disentangled latent spaces in this work, we will not go deeper into this matter.

D. Evaluating compression efficiency

In this final experiment we want to find out how efficient our method is at pruning latent dimensions. For this purpose, we devise an approach in which we first train a baseline regular VAE with a predefined number of latent dimensions n and with a GECO tolerance of 0 MSE, which means that we want to squeeze out every bit of reconstruction quality there is to gain, thereby sacrificing KL regularization. We give the model a fixed training budget of 200K batches, after which we write down the moving average (with factor 0.95) of the final reconstruction error. Then, in a second step, we train a new VAE with L_0 regularization and GECO that we provision with $2n$ latent dimensions, and we observe to what extent this number can be lowered to n given the obtained reconstruction error in step one as GECO tolerance. We do this for all five datasets, for values $n = 5, 10, 30$, and for fixed training budgets of 200K, 300K and 400K batches. This will allow us to measure the efficiency of our approach in terms of how fast we can reduce the latent dimensionality compared to the original VAE. The resulting numbers are presented in Table I. For most configurations, our approach is able to prune the bottleneck very effectively, coming close to the original value of n, sometimes achieving the same value or even surpassing it. This is often true for $n = 30$ (except for CIFAR-100), which shows that the original VAE was probably overprovisioned. We also observe that we are coming close to the optimal number of dimensions already after a training budget of 200K batches, i.e. the same budget with which we have trained the original VAE. This is probably thanks to the higher initial dimensionality of $2n$, which allows the reconstruction error to drop more quickly below the GECO treshold at the start of training, after which we can start pruning the bottleneck. Only in a select number of cases—again, mostly for $n = 30$—are we able to further reduce the dimensionality significantly if we continue training for 400K batches. In summary, Table I

	n	MSE (200K)	$\|\nu\|_0$ (200K)	$\|\nu\|_0$ (300K)	$\|\nu\|_0$ (400K)
dSprites	5	9.48	7	6	6
	10	8.83	7	6	6
	30	6.14	13	12	10
Cars3D	5	94.42	6	5	5
	10	65.67	14	13	12
	30	60.97	21	17	16
Shapes3D	5	36.73	6	6	6
	10	19.53	9	9	9
	30	18.74	20	15	11
MNIST	5	17.07	5	5	5
	10	9.49	10	9	9
	30	7.16	14	14	14
CIFAR-100	5	325.87	6	6	6
	10	249.1	10	10	10
	30	147.8	31	30	30

TABLE I: For five different datasets, we train a regular VAE with bottleneck dimensionality n and GECO threshold $\tau = 0$. The obtained MSE is shown after a training budget of 200K batches. We train VAEs with L_0 regularization and initial bottleneck dimensionality $2n$, we set the GECO threshold τ to the previously obtained MSE, and the resulting 0-norm of the gating vector after 200K, 300K and 400K batches is shown.

shows that our method is widely applicable in different settings and that it can provide near-optimal dimensionality reductions without incurring a large additional training budget.

V. Conclusion

We have devised an algorithm to reduce the bottleneck dimensionality of VAEs on-the-fly during training. For this purpose we have used L_0 regularization and the L_0-ARM gradient estimator to train a gating mechanism that either blocks or passes information for each latent factor independently. To decide how many factors are needed, we employ GECO to define constraints as upper bounds on the reconstruction error. In the experiments we show that our algorithm is effective at reducing the latent dimensionality on five different datasets. It is also a useful tool to assess whether a VAE bottleneck is over- or underprovisioned. The only downside of the algorithm is the (small) computational overhead that comes from a second forward pass through the decoder network, as required by L_0-ARM. In future work one can look at applying the method to all intermediate representations [16], and not only to the encoder output. It would also be interesting to see if it can be applied to recommender systems or VAE-based models for control tasks, e.g. in robotics.

Acknowledgments

This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] Y. Bengio, A. Courville, P. V. I. t. o. pattern, and 2013, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[3] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *Proceedings of Workshop at ICLR*, 2013.

[4] N. Tishby and N. Zaslavsky, "Deep Learning and the Information Bottleneck Principle," *arXiv.org*, 2015.

[5] A. A. Alemi, I. Fischer, J. V. Dillon, and K. Murphy, "Deep Variational Information Bottleneck," *arXiv.org*, 2016.

[6] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context Encoders: Feature Learning by Inpainting," *arXiv.org*, 2016.

[7] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," *arXiv.org*, 2016.

[8] Y. Li and S. Ji, "L0-ARM: Network Sparsification via Stochastic Binary Optimization," in *ECML*, 2019.

[9] M. Yin and M. Zhou, "ARM - Augment-REINFORCE-Merge Gradient for Stochastic Binary Networks." in *ICLR*, 2019.

[10] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *ICLR*, 2014.

[11] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "beta-VAE - Learning Basic Visual Concepts with a Constrained Variational Framework." *ICLR*, 2017.

[12] D. J. Rezende and F. Viola, "Generalized ELBO with Constrained Optimization, GECO," *bayesiandeeplearning.org*, 2018.

[13] ——, "Taming VAEs," *arXiv.org*, 2018.

[14] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational Dropout Sparsifies Deep Neural Networks," in *ICML*, 2017.

[15] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," in *ICLR*, 2017.

[16] B. Dai, C. Zhu, and D. Wipf, "Compressing Neural Networks using the Variational Information Bottleneck," in *ICML*, 2018.

[17] C. Louizos, M. Welling, and D. P. Kingma, "Learning Sparse Neural Networks through L0 Regularization," in *ICLR*, 2018.

[18] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, 1992.

[19] Y. Bengio, N. Léonard, and A. Courville, "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation," *arXiv.org*, 2013.

[20] T. Bird, J. Kunze, and D. Barber, "Stochastic Variational Optimization," *arXiv.org*, 2018.

[21] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," in *ICLR*, 2017.

[22] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the Value of Network Pruning," in *ICLR*, 2019.

[23] L. Matthey, I. Higgins, D. Hassabis, and A. Lerchner, "dsprites: Disentanglement testing sprites dataset," https://github.com/deepmind/dsprites-dataset, 2017.

[24] C. Burgess and H. Kim, "3d shapes dataset," https://github.com/deepmind/3dshapes-dataset, 2018.

[25] S. E. Reed, Y. Zhang, Y. Zhang, Y. Zhang, and H. Lee, "Deep Visual Analogy-Making," in *NIPS*, 2015.