MLMI4: Importance Weighted Autoencoder

Charles Arnal, Jamie Lee, Nicholas Pezzotti

April 1, 2021

Abstract

We compare the performance of the variational autoencoder (VAE) with that of the importance-weighted autoencoder (IWAE) by replicating experiments from [BGS16], and discuss some limitations of the IWAE.

Contents

1 Introduction

Variational autoencoders (VAEs) are a powerful type of probabilistic autoencoders, introduced in [KW14], that can be used as efficient generative models in the same manner as GANs (see [Goo+14]), as well as for a variety of other tasks, such as features extraction or visualisation of high-dimensional data. They have found applications in domains ranging from medical science [Sim+19] to music [REE17], which motivated us to study them further.

In this report, we investigate **importance weighted autoencoders (IWAEs)**, which are based on the same architecture as VAEs, but optimised using a different objective function. They were introduced in [BGS16] by Burda, Grosse and Salakhutdinov - we replicate and comment all experiments from that article. On top of the three datasets used by [BGS16] (MNIST [LCB], Binarise MNIST [doc] and Omniglot [Lak]), we also evaluate a different domain: Fashion MNIST [XRV17]. Finally, going beyond what is discussed in the original paper, we analyse alternative, yet similar, objective functions in an attempt to better understand the strengths and limitations of the IWAE objective, in the same spirit as [Rai+18]. Additionally, we provide effective visualisation in the Appendix that we believe can help understand the phenomena considered.

Our report is organized as follows: we first expose the main notions in Section 2, as well as some preliminary theoretical discussion partially inspired by [Rai+18]. We detail our datasets, model architectures and training procedures in Section 3. In Section 4, we describe our replication of the various experiments from [BGS16] and [Rai+18], further extensions to these experiments of our invention, and conclude with a commentary of the results obtained. Finally, we conclude in Section 5.

The core components of our code are in Appendix D. It can be found in its entirety and run at `https://colab.research.google.com/drive/1SUEvQJlAm9lkCuXnlsSFHilRxOg60Xa1`.

2 Theory

In this section, we introduce the main theoretical notions from the original paper and surrounding literature relevant to understanding the experiments we run and the conclusion we draw throughout the rest of our report.

2.1 Variational autoencoders

VAEs are a type of probabilistic autoencoders introduced in [KW14]. They are composed of a *decoder*, or generative model, which describes the distribution $p(\mathbf{x})$ of a (multidimensional) random variable $\mathbf{x}$ in terms of conditional probabilities $p(\mathbf{x}|\mathbf{h})$ with respect to a (multidimensional) latent variable $\mathbf{h}$ and of the distribution of $\mathbf{h}$ itself, and an *encoder*, or recognition model, which defines a distribution $q(\mathbf{h}|\mathbf{x})$ for $\mathbf{h}$ as a function of $\mathbf{x}$.

A common assumption (see [BGS16]) is that both the decoder and the encoder can be factorized as products

$$q_\phi\left(\mathbf{h} \mid \mathbf{x}\right) = q_\phi\left(\mathbf{h}^1 \mid \mathbf{x}\right) q_\phi\left(\mathbf{h}^2 \mid \mathbf{h}^1\right) \cdot \ldots \cdot q_\phi\left(\mathbf{h}^L \mid \mathbf{h}^{L-1}\right)$$

and

$$p_\theta(\mathbf{x}) = p_\theta\left(\mathbf{h}^L\right) p_\theta\left(\mathbf{h}^{L-1} \mid \mathbf{h}^L\right) \cdot \ldots \cdot p_\theta\left(\mathbf{x} \mid \mathbf{h}^1\right),$$

where ϕ and θ are parameters and $\mathbf{h} = (\mathbf{h}^1, \ldots, \mathbf{h}^L)$, and that if we define $\mathbf{h}^0 := x$, each successive distribution $q_\phi\left(\mathbf{h}^i \mid \mathbf{h}^{i-1}\right)$ (respectively $p_\theta\left(\mathbf{h}^{i-1} \mid \mathbf{h}^i\right)$) is a Gaussian with diagonal covariance, whose mean and covariance parameters are computed as a (deterministic) function of $\mathbf{h}^{i-1}$ (respectively $\mathbf{h}^i$) by a neural network. The parameters ϕ and θ of the model are the weights of the successive neural networks - we sometimes omit them in our notations in what follows. We also typically assume that $\mathbf{h}^1$ follows a normal Gaussian distribution $\mathcal{N}(0, 1)$. We refer to L as the number of *stochastic hidden layers*[1].

[1] We call a stochastic layer the composition of the deterministic neural network and the sampling from the

Given a dataset, one is often interested in creating a generative model that can produce new data that is similar to the data in the dataset, as *e.g.* **generative adversarial networks (GANs)** [Goo+14] do.

One way to turn traditional, deterministic autoencoders into generative models is to randomly pick points in their latent space, and map them using the decoder to new data points. However, this tends to yield poor results, as deterministic autoencoders can map their latent space to data points in a very irregular way: points that are close to each other in the latent space can be mapped to very different decoded points, and points in the latent space that were not the encoded image of any point in the training dataset can be mapped to "unreasonable" points in the decoded space, *e.g.* points that have nothing in common with those in the training dataset.

Classical VAEs are trained by optimising with respect to the parameters of the model an objective function

$$\mathcal{L}^{VAE}(\mathbf{x}) = \mathbb{E}_{q(\mathbf{h}|\mathbf{x})}\left[\log \frac{p(\mathbf{x},\mathbf{h})}{q(\mathbf{h} \mid \mathbf{x})}\right],$$

which can be directly decomposed in two different ways as

$$\begin{aligned}\mathcal{L}^{VAE}(\mathbf{x}) &= \mathbf{E}_{q(\mathbf{h}|\mathbf{x})}[\log(p(\mathbf{x}|\mathbf{h}))] - D_{KL}(q(\mathbf{h}|\mathbf{x})||p(\mathbf{h})) &(1)\\ &= \log(p(\mathbf{x})) - D_{KL}(q(\mathbf{h}|\mathbf{x})||p(\mathbf{h}|\mathbf{x})) &(2)\end{aligned}$$

where all expected values are taken over the distribution $\mathbf{h} = (\mathbf{h}^1, \ldots, \mathbf{h}^L) \sim q(\mathbf{h} \mid \mathbf{x})$ and $D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h}))$ is the Kullblack-Leibler divergence of $q(\mathbf{h} \mid \mathbf{x})$ and $p(\mathbf{h})$.

Looking at Formula (1), we see that optimising $\mathcal{L}^{VAE}$ w.r.t. a dataset means encouraging the output of the decoder to be similar to that dataset when its input follows the distribution $q(\mathbf{h} \mid \mathbf{x})$ (due to the term $\mathbf{E}_{q(\mathbf{h}|\mathbf{x})}[\log(p(\mathbf{x} \mid \mathbf{h}))]$), while constraining $q(\mathbf{h} \mid \mathbf{x})$ to be close to the "regular" distribution $p(\mathbf{h})$ (thanks to the KL divergence) - in particular, remember that $p(\mathbf{h}^1)$ is assumed to be a Gaussian distribution. In other words, it trains the decoder to be an efficient generative model by adding the regularity that a deterministic autoencoder would lack.

From another point of view, looking at Formula (2), we see that the VAE is trained to generate data similar to the dataset (due to the log-likelihood term), while constraining $q(\mathbf{h} \mid \mathbf{x})$ to be an approximation of the true (usually intractable) posterior distribution over the latent variable $p(\mathbf{h} \mid \mathbf{x})$ (thanks to the KL divergence). This justifies the appellation "variational *autoencoder*", as it makes the encoder be an approximate inverse of the decoder, as it would be the case for a deterministic autoencoder.

As a result, VAEs can be used not only as generative models (in which case the encoder is discarded after training), but also for other tasks, such as posterior inference (especially for deep generative models), dimensionality reduction, feature representation and visualisation of high-dimensional data, or structured output prediction (see *e.g.* [SLY15] or [Li+20]). In those cases, the performance of the recognition model is very important.

2.2 Importance weighted autoencoders

IWAEs have the same architecture as classical VAEs, but are optimised using a different objective function, defined as

$$\mathcal{L}_k^{IWAE} = \mathop{\mathbb{E}}_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim q(\mathbf{h}|\mathbf{x})}}\left[\log\left(\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})}\right)\right]$$

for some choice of parameter $k \in \mathbb{N}_{\geq 1}$, and where the latent variables $\mathbf{h}_i = (\mathbf{h}_i^1, \ldots, \mathbf{h}_i^L)$ are independently drawn from $q(\mathbf{h} \mid \mathbf{x})$. Observe that $\mathcal{L}_1^{IWAE} = \mathcal{L}^{VAE}$.

Gaussian distribution that allows us to get $\mathbf{h}^{i+1}$ from $\mathbf{h}^i$ (or vice-versa). Here, we differ slightly from the convention in [BGS16], where "stochastic layer" refers only to the sampling process, and the neural network are not explicitly mentioned.

Using Jensen's inequality and the concavity of the logarithm, it is easy to show that we have for any $\mathbf{x}$ the following sequence of inequalities:

$$\begin{aligned}\log\left(p(\mathbf{x})\right) \geq \ldots \geq \mathcal{L}_{k+1}^{IWAE}(\mathbf{x}) \geq \mathcal{L}_{k}^{IWAE}(\mathbf{x}) \geq ... \geq & \mathcal{L}_{1}^{IWAE}(\mathbf{x}) = \mathcal{L}^{VAE}(\mathbf{x}) \\ & = \log\left(p(\mathbf{x})\right) - D_{KL}\left(q(\mathbf{h} \mid \mathbf{x}) || p(\mathbf{h} \mid \mathbf{x})\right),\end{aligned} \tag{3}$$

and in fact it can be shown that under reasonable assumptions, $\mathcal{L}_{k}^{IWAE}(\mathbf{x})$ converges to $\log\left(p(\mathbf{x})\right)$ as $k \to \infty$ (see [BGS16, Appendix A] for additional details[2]).

In practice, both the VAE's and the IWAE's objective functions are approximated using Monte Carlo estimations. When comparing them, the authors of [BGS16] draw the same number k of samples for both, *i.e.* they approximate

$$\mathcal{L}^{VAE}(\mathbf{x}) = \mathbb{E}_{q(\mathbf{h}|\mathbf{x})}\left[\log \frac{p(\mathbf{x},\mathbf{h})}{q(\mathbf{h} \mid \mathbf{x})}\right] \cong \sum_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim\, q(\mathbf{h}|\mathbf{x})}} \frac{1}{k} \log\left(\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})}\right)$$

and

$$\mathcal{L}_{k}^{IWAE}(\mathbf{x}) = \mathop{\mathbb{E}}_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim\, q(\mathbf{h}|\mathbf{x})}}\left[\log\left(\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})}\right)\right] \cong \sum_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim\, q(\mathbf{h}|\mathbf{x})}} \log\left(\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})}\right). \tag{4}$$

Note that the right-hand term of Formula 4 is, in fact, a "bad" Monte Carlo estimation (with a single sample) of the expected value of a random variable whose variance is shown in [BGS16, Appendix B] to be satisfyingly small (due to the averaging taking place inside the logarithm) under mild assumptions - hence the estimator itself has small variance as well.

Using the reparameterization trick described in [KW14], we can rewrite each sample $\mathbf{h}_i$ as a deterministic function $\mathbf{h}_i = \mathbf{h}\left(\mathbf{x}, \boldsymbol{\epsilon}_i, \phi\right)$ of $\mathbf{x}$, the recognition model parameter ϕ, and an auxiliary normal Gaussian random variable $\boldsymbol{\epsilon}_i$. This allows us to compute a well-defined estimate of the gradient of a sample $\mathbf{h}$ w.r.t. the model's parameters.

Defining $w_i := \frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ and $\tilde{w}_i := \frac{w_i}{\sum_{j=1}^{k} w_j}$, we then get

$$\sum_{i=1}^{k} \frac{1}{k} \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \log w\left(\mathbf{x}, \mathbf{h}\left(\boldsymbol{\epsilon}_i, \mathbf{x}, \boldsymbol{\phi}\right), \boldsymbol{\theta}, \boldsymbol{\phi}\right)$$

as a gradient estimator for $\mathcal{L}^{VAE}(\mathbf{x})$, and

$$\sum_{i=1}^{k} \tilde{w}_i \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \log w\left(\mathbf{x}, \mathbf{h}\left(\boldsymbol{\epsilon}_i, \mathbf{x}, \boldsymbol{\phi}\right), \boldsymbol{\theta}, \boldsymbol{\phi}\right)$$

for $\mathcal{L}_{k}^{IWAE}(\mathbf{x})$. Note that for fixed $\mathbf{x}$, the coefficients $\tilde{w}_i$ are the normalized importance weights for the unnormalized target distribution $p\left(\mathbf{x}, \mathbf{h}_i\right)$ and the sampling distribution $q\left(\mathbf{h}_i \mid \mathbf{x}\right)$, whence the name *importance weighted* autoencoder.

[2] In fact, the proof of the convergence in [BGS16] seems faulty to us in several ways: the authors ask that $\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ be bounded (for a given $\mathbf{x}$), which means that $\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ is guaranteed to converge almost surely to $\log(p(\mathbf{x}))$ by the Strong Law of Large Numbers. However, the boundedness of $\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ does not seem to be a particularly reasonable assumption (remember that for $L = 1$, the distribution $q(\mathbf{h}_i \mid \mathbf{x})$ is Gaussian - it is easy to construct simple examples where $p(\mathbf{x}, \mathbf{h}_i)$ is also a Gaussian, with greater variance along a certain coordinate than $q(\mathbf{h}_i \mid \mathbf{x})$), which is enough for the quotient not to be bounded). Moreover, it is not actually needed for the Law of Large Number to apply (the quotient only needs to be integrable, which is trivially satisfied).
More concerning is the fact that the convergence almost sure of $\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ does not imply that $\mathop{\mathbb{E}}_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim\, q(\mathbf{h}|\mathbf{x})}}\left[\log\left(\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}\right)\right]$ converges - additional explanations are needed.

2.3 Advantages of IWAEs

We have seen in Formula 3 that the IWAE objective function $\mathcal{L}_k^{IWAE}(\mathbf{x})$ is a tighter lower bound on the log-likelihood of the dataset than $\mathcal{L}^{VAE}(\mathbf{x})$. As explained in [BGS16], it can be seen as a relaxation of the constraint that the approximate posterior distribution $q(\mathbf{h} \mid \mathbf{x})$ be close to the true posterior distribution $p(\mathbf{x}, \mathbf{h})$ which is enforced by the KL divergence term $D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x}))$ in $\mathcal{L}^{VAE}(\mathbf{x})$. In particular, this constraint meant that $p(\mathbf{x}, \mathbf{h})$ had to be well approximable by a succession of feed-forward neural networks and Gaussian sampling processes, which was quite restrictive.

As observed in [BGS16], we show in Section 4 that due to the weakening of that constraint, IWAEs yield higher log-likelihood than VAEs, which suggests they might be more powerful as generative networks. The authors of [BGS16] also state that IWAEs give rise to "richer latent representations" - a claim we dispute in Section 4 as well.

In the same spirit, we came up with the following alternative objective functions, which we also tested[3]:

$$\mathcal{L}_{k,p}^{power}(\mathbf{x}) := \mathop{\mathbb{E}}_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim q(\mathbf{h}|\mathbf{x})}} \left[\log \left(\left[\frac{1}{k} \sum_i \left(\frac{p(\mathbf{x}, \mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})} \right)^p \right]^{\frac{1}{p}} \right) \right],$$

$$\mathcal{L}_k^{median}(\mathbf{x}) := \mathop{\mathbb{E}}_{\substack{\mathbf{h}_1,\ldots,\mathbf{h}_k \\ \sim q(\mathbf{h}|\mathbf{x})}} \left[\log \left(\operatorname*{median}_{i=1,\ldots,k} \left\{ \frac{p(\mathbf{x}, \mathbf{h}_i)}{q(\mathbf{h}_i \mid \mathbf{x})} \right\} \right) \right],$$

and

$$\mathcal{L}_\alpha(\mathbf{x}) := (1-\alpha)\mathbf{E}_{q(\mathbf{h}|\mathbf{x})}[\log(p(\mathbf{x} \mid \mathbf{h}))] + \alpha \mathcal{L}^{VAE}(\mathbf{x}) = \mathbf{E}_{q(\mathbf{h}|\mathbf{x})}[\log(p(\mathbf{x} \mid \mathbf{h}))] - \alpha D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h})).$$

We thought of these new objective functions more as ways to explore the ideas presented in [BGS16] than as potential strict improvements.

Though $\mathcal{L}_k^{median}$ and $\mathcal{L}_{k,p}^{power}$ do not have the same satisfying mathematical properties as $\mathcal{L}_k^{IWAE}$, they are based on the same principle that was exposed in [BGS16]: by applying some averaging process to the weights inside the logarithm, the "bad" samples for which $\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ is very small become less impactful, allowing a recognition model "which places only a small fraction of its samples in the region of high posterior probability" (cited from [BGS16, Section 3]) to achieve a high objective function score. Our reasoning was that by using $\mathcal{L}_{k,p}^{power}$ with $p > 1$, this effect could be strengthened, while $\mathcal{L}_k^{median}$ might be more representative of the performance of the model than $\mathcal{L}_k^{IWAE}$ (as it is less affected by extreme values of $\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$). The function $\mathcal{L}_\alpha$ was simply intended as a direct way to choose the comparative importance of the regularization term $D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h}))$.

2.4 Limitations of IWAEs

Due to the difference between IWAEs and VAEs being essentially a relaxation of the constraint that the approximate posterior distribution should be close to the true posterior distribution, it was our expectation that the performance of the recognition model should suffer: the KL divergence $D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x}))$ should be larger for IWAEs than for VAEs. This, in turn, should make IWAEs less suited to inference and dimensionality reduction tasks. To confirm this hypothesis, we tracked the value of $D_{KL}(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x}))$ for each model, and performed a few data reconstruction experiments.

This initial intuition of ours was validated by our reading of [Rai+18], which further explores the impact of optimising the IWAE objective function on the recognition model by considering the

[3] We are aware that performing stochastic gradient optimization on $\mathcal{L}_k^{median}$ could prove to be a risky (and in fact somewhat poorly defined) proposition, due to the median operator, but it can be shown to be a sensible operation as long as the step size is small enough, and we observed no difficulty in practice.

gradient of $\mathcal{L}_k^{IWAE}$ w.r.t. the parameters ϕ of the encoder (see Subsection 2.1). Though we refer to [Rai+18] for all technical details, the core idea is simple: studying Formula (4), we see that $\frac{1}{k}\sum_{i=1}^{k}\frac{p(\mathbf{x},\mathbf{h}_i)}{q(\mathbf{h}_i|\mathbf{x})}$ is a Monte Carlo estimation of $\log p(\mathbf{x})$ computed with importance sampling (where $q(\mathbf{h} \mid \mathbf{x})$ is the sampling distribution). This means that as k grows larger, this quantity converges to $\log p(\mathbf{x})$ independently from the parameters of the recognition model[4] - hence the value $\mathcal{L}_k^{IWAE}(\mathbf{x})$ depends less and less on ϕ and the norm of (the average of) its gradient $\nabla_\phi \mathcal{L}_k^{IWAE}(\mathbf{x})$ w.r.t. ϕ decreases as k increases. Though the standard deviation of $\nabla_\phi \mathcal{L}_k^{IWAE}(\mathbf{x})$ decreases as well (in $O(1/\sqrt{k})$), the norm of its expected value diminishes faster (in $O(1/k)$), so that the signal to noise ratio of the gradient converges to 0 as k grows larger. Consequently, the encoder cannot be properly trained anymore.

The authors of [Rai+18] came up with several new objective functions which, according to their claims, train the models to have the same level of generative power as the ones trained with the IWAE objective function, but increase their performance as a recognition network (similar improvements have been proposed in [Tuc+19] and [FT19]). As another extension, we implemented and tested two of these functions. The first one, referred to as **CIWAE**, is

$$\mathcal{L}_{k,\beta}^{CIWAE}(\mathbf{x}) := \beta\mathcal{L}^{VAE}(\mathbf{x}) + (1-\beta)\mathcal{L}_k^{IWAE}(\mathbf{x}),$$

where $\mathcal{L}_k^{IWAE}(\mathbf{x})$ is estimated using k samples $\mathbf{h}$ - the idea being that even for large k, the presence of the $\beta\mathcal{L}^{VAE}$ should keep the gradient with respect to ϕ from becoming too small. The other objective function, referred to as **MIWAE**, is in fact a different estimation of $\mathcal{L}_k^{IWAE}$: instead of averaging over k samples inside the logarithm, then taking a single sample of that quantity (see Subsection 2.2), we average over k_1 samples inside the logarithm, then compute a Monte Carlo estimation of that quantity with k_2 samples. Committing a small abuse of notations, we write

$$\mathcal{L}_{k_1,k_2}^{MIWAE}(\mathbf{x}) := \frac{1}{k_2}\sum_{j=1}^{k_2}\sum_{\substack{\mathbf{h}_{1,j},\dots,\mathbf{h}_{k_1,j}\\ \sim\, q(\mathbf{h}|\mathbf{x})}} \log\left(\frac{1}{k_1}\sum_{i=1}^{k_1}\frac{p(\mathbf{x},\mathbf{h}_{i,j})}{q(\mathbf{h}_{i,j} \mid \mathbf{x})}\right),$$

where samples $\mathbf{h}_{i,j}$ are independently drawn. The averaging process outside of the logarithm leads to a decreased variance for the gradient, which improves its signal to noise ratio. When comparing $\mathcal{L}_{k_1,k_2}^{MIWAE}$ to another objective function that uses k samples, we let $k_1 \cdot k_2 = k$.

3 Datasets, models and training procedures

In this section, we describe in detail the datasets used and the experimental configurations in terms of model structure and training parameters.

3.1 Datasets

We attempted to reproduce the original paper's results across all datasets. The original paper uses two density estimation benchmark datasets to compare the performance of the VAE and IWAE models in terms of held-out log-likelihoods: MNIST, a dataset of images of handwritten digits, and OMNIGLOT, a dataset of handwritten characters of various alphabets. In addition to these datasets, we also experiment with Fashion MNIST, which is described below.

3.1.1 MNIST

We used the official MNIST dataset as provided by [LCB]. It is a dataset of 60,000 train + 10,000 test 28×28 pixel grayscale images of handwritten digits. Each pixel has an intensity value between 0 and 255, which we rescaled to be between 0 and 1. We then binarised the dataset at the start of each training by sampling each pixel from a Bernoulli with probability equal to the intensity of that pixel.

[4] Under reasonable assumptions on $q(\mathbf{h} \mid \mathbf{x})$.

3.1.2 Fixed binarisation MNIST

The second dataset we used was the fixed binarisation MNIST [doc]. This dataset is a version of MNIST that was pre-binarised according to the same procedure described above. It is divided into a train/dev/test split of 50,000/10,000/10,000, as opposed to 60,000/0/10,000 for the original MNIST. Following the procedure adopted by the original authors, we ignore the development set, training with just 50,000 samples. Here on out, we refer to this dataset as "fixed binarisation MNIST", to distinguish this dataset from the stochastically binarised MNIST detailed above.

3.1.3 Omniglot

Omniglot is a dataset of 1623 different handwritten characters from 50 different alphabets with 20 examples for each character [Lak]. Each element is of size 105x105; the authors of [BGS16] rescaled the images and binarised them, and we downloaded the data provided on their GitHub. We use the same training split as them: 24,345 examples for training and 8,070 for testing.

3.1.4 Fashion MNIST

The Fashion MNIST dataset was developed by [XRV17] to serve as a more challenging replacement dataset for the MNIST dataset. It consists of 60,000 train + 10,000 test 28×28 pixel grayscale images of items of 10 types of clothing such as shoes, t-shirts, dresses, etc. We chose to utilize this dataset as to verify the experimental results on a domain that is not hand-drawn characters and more closely resembles natural images.

3.2 Codebase

We decided to rebuild the whole codebase from scratch. While the authors of [BGS16] wrote their code in Theano, we opted for the much more popular TensorFlow within Google Colab; this way the experiments can be run more easily and the configurations changed more quickly.

Our code (which can be read in Appendix D) is highly configurable to other experimental setups. For example, while the original code is designed to be able to reproduce only the experiments in the paper, we make it easy to use a different dataset, increase the number of layers on the fly, or switch the objective function the model is being optimised on. We also fully integrated our code with TensorBoard to allow viewing of the training and test statistics.

3.3 Model Architecture

As in [BGS16], we evaluated the performance of two different model architectures, with either one or two stochastic layers. A diagram as well as a description of both architectures, including layers, hyperparameters and activation functions, is presented in Appendix A.

3.4 Training Configurations

All models were initialized with the heuristic of [GB10]. While this initialisation is done by default on TensorFlow for most layers, it is not the case for Bernoulli sampling layers. While not explicitly mentioned in [BGS16] (though present in their code), we found it essential that this heuristic be used even for the Bernoulli layers.

For optimization, we used Adam (see [KB17]) with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-4}$ and a batch size of 100. Training was conducted for 3^i passes over the data with a learning rate equal to $0.0001 \times 10^{-i/7}$ for $i = 0, ..., 7$, for a total of $\sum_{i=0}^{7} 3^i$ or 3280 passes over the data.

4 Experiments

4.1 Log-likelihood and KL divergence

Replicating the main experiment from [BGS16], we trained our models with 1 and 2 stochastic layers using the VAE and IWAE objective functions on the MNIST dataset (with both fixed and stochastic binarisation) for $k \in \{1, 5, 50\}$ and on the Omniglot dataset for $k \in \{1, 50\}$. We also train single-layer models on the Fashion MNIST dataset (which was not used in [BGS16]) for $k \in \{1, 50\}$.

Additionally, the one stochastic layer model is trained with the objective functions $\mathcal{L}_\alpha$ (for $\alpha \in \{0, 0.25, 0.5\}$), $\mathcal{L}_{k,p}^{power}$ (for $p \in \{0.5, 2, 3, 5\}$) and $\mathcal{L}_k^{median}$ on the fixed binarisation MNIST dataset with $k = 50$ samples.

Finally, reproducing some of the experiments performed in [Rai+18], we trained the one stochastic layer model with the objective functions $\mathcal{L}_{k=50,\beta}^{CIWAE}$ (for $\beta \in \{0.5, 0.25, 0.05\}$) and $\mathcal{L}_{k_1,k_2}^{MIWAE}$ (for $(k_1, k_2) \in \{(5, 10), (10, 5)\}$) on the MNIST dataset.

For each model, we recorded the final value of the empirical negative log-likelihood (to remain consistent with [BGS16]) measured on the test set which is denoted as "NLL", as well as that of the KL divergence $D_{KL}\left(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x})\right)$, denoted as "$D_{KL}$". Following [BGS16], we use $\mathcal{L}_{k=5000}^{IWAE}(\mathbf{x})$ as a very precise estimation of $\log(p(\mathbf{x}))$.

4.1.1 Results

Our results in Tables 1, 2 and 3 are in line with those from Tables 1 and 3 from [BGS16], provided for reference in Appendix C.

Both objective functions lead to similar performances for $k = 1$ (as they coincide for a single sample). While increasing k has no significant effect on the log-likelihood for the VAE objective function (as we are simply estimating more precisely the same quantity), we see that it improves the log-likelihood for the IWAE objective function, which suggests more generative power. Models with two stochastic layers are more flexible, and hence yield greater log-likelihood. The same tendencies can be observed on the Fashion MNIST dataset in Table 4; we see that the log-likelihood is lower than for the other datasets, suggesting that the generative model has a harder time modeling this more complex dataset.

Our models perform slightly worse on the stochastically binarised datasets MNIST and Omniglot than reported in [BGS16]; the reason for this is that unlike what is suggested in the article, we discovered that the authors' code re-binarises stochastically the training data at the beginning of each epoch, which leads to less overfitting and better generalization.

As seen in Tables 5 and 6, our new objective functions $\mathcal{L}_\alpha$ and $\mathcal{L}_k^{median}$ led to slightly lower log-likelihoods than the $\mathcal{L}^{VAE}$ and $\mathcal{L}^{IWAE}$, and do not seem to be a particularly promising idea.

Conversely and as illustrated in Table 7, using $\mathcal{L}_{k,p}^{power}$ seems to yield a very slightly better log-likelihood than with $\mathcal{L}^{IWAE}$ for $p > 1$ (with the best log-likelihood obtained for $p = 3$), and for the same computing costs. As differences are small and as results are affected by some random variability, further testing would be required to reach a definitive conclusion.

We observe on all datasets that as predicted in Subsection 2.4 and as reported in [Rai+18], using the IWAE objective function returns greater values for the KL divergence $D_{KL}\left(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x})\right)$ as k grows larger; as explained earlier, this means that the encoder performs more poorly (or more exactly that it is less of an inverse for the decoder[5]).

As reported in [Rai+18], using the objective functions $\mathcal{L}^{CIWAE}$ and $\mathcal{L}^{MIWAE}$ allowed us to obtain advantageous trade-offs between generative power and performance of the recognition network (see

[5]We confirmed this by encoding, then decoding the elements of the dataset, and measuring the average crossentropy between the (binary) input and the reconstructed (continuous) output - we observed that IWAEs with large k generate greater reconstruction error.

Tables 8 and 9), with log-likelihoods about as high as those obtained with the IWAE objective function (for example with $\mathcal{L}^{CIWAE}_{50,0.25}$), and KL divergences much closer to those from VAEs systems.

4.2 Latent space representation

Given a number of stochastic layers L and a coordinate $h^{i,n}$ of the latent variable $\mathbf{h}^i$ (for some $n \in \{1, \dots, N_i\}$ and $i \in \{1, \dots, L\}$, where $\mathbf{h} = (\mathbf{h}^1, \dots, \mathbf{h}^L)$), authors of [BGS16] define the *empirical level of activity* of $h^{i,n}$ as

$$A(h^{i,n}) = \mathrm{Cov}_{\mathbf{x}} \left(\mathop{\mathbb{E}}_{q(h^{i,n}|\mathbf{x})} [h^{i,n}] \right),$$

where the empirical covariance is computed over the entire dataset. The level of activity of a coordinate measures how dependent it is on the input - a very low level of activity indicates that the coordinate carries almost no encoded information about the input $\mathbf{x}$. The authors consider that a coordinate, which they call *unit*, is active if its level of activity is greater than 10^{-2}, and use the number of active coordinates to measure the effective dimension (*i.e.* ignoring the noise) of the image of the encoder over the dataset.

We compute the levels of activity for each of the experiments detailed in Subsection 4.1, with a few differences. The first is that we measure the levels of activity on the test data set, which we thought would be a more reasonable choice than the training set that they use. We also use the entire set of $10,000$ points, whereas they restrict themselves to a subset of 500 points: we observed that this can lead to a change in the number of active units. Finally, we do not think that simply counting the number of active units is a good measure of the true "dimension" of the image of the encoder. Instead, we decided to apply Principal component analysis (PCA) on the means $\mathop{\mathbb{E}}_{q(h^{i,n}|\mathbf{x})}[h^{i,n}]$, and count the number of eigenvalues that are greater than 10^{-2}, which we refer to as the number of PCA active units[6].

We also replicated another experiment from [BGS16], in which we first train a model with one stochastic layer using the VAE objective function with $k = 1$ sample, then keep training it for one epoch of $3^7 = 2,187$ passes with a stepsize of 10^{-4} using the IWAE objective function with $k = 50$ samples. We did the same using $\mathcal{L}^{IWAE}_{k=50}$ first, then $\mathcal{L}^{VAE}$ with $k = 1$. Additionally, we performed the same experiments using $k = 50$ samples when evaluating $\mathcal{L}^{VAE}$, which seemed a fairer comparison to us.

4.2.1 Results

As in [BGS16], we observe in Tables 1, 2 and 3, as well as for other objective functions, that IWAEs tend to have more active units than VAEs, especially for large numbers of samples k, and that the number of active units is consistently smaller than the dimension of the latent space. Unsurprisingly, the number of active units is smaller in the second stochastic layer than in the first, as the inputs are further encoded. We also notice that applying PCA does make an important difference whenever there are two layers. The details of our results differ more from those of [BGS16] than when considering the log-likelihood; we assume that it must be due to the differences exposed above.

In particular, we do see in Table 10 (as in Table 2 in [BGS16]) an augmentation in the number of active units when further training as an IWAE a model initially trained as a VAE, which suggests that IWAEs really tend to have a higher number of active units (as opposed to it being due to some training artifact). However, and unlike the authors of [BGS16], we observe no such phenomenon when going in the other direction. We also observe larger decreases (of about 1 nat) in log-likelihood when removing the inactive units from trained models and testing them on the test set than reported in [BGS16], where no decrease of more than 0.06 nats was observed.

[6] To be even more rigorous, more advanced dimensionality reduction techniques, suited to non-linear submanifolds, could be applied.

		MNIST (Fixed Binarisation)							
		VAE				IWAE			
L	k	NLL	Active Units	PCA AU	D_{KL}	NLL	Active Units	PCA AU	D_{KL}
1	1	89.87	14	14	5.30	89.87	14	14	5.30
	5	89.85	14	14	5.76	87.61	23	23	7.60
	50	90.78	13	13	5.90	87.13	25	25	12.87
2	1	86.81	$36 + 7$	$18 + 7$	5.07	86.75	$37 + 8$	$18 + 8$	5.00
	5	86.39	$27 + 9$	$19 + 9$	4.77	85.58	$59 + 9$	$23 + 9$	6.77
	50	86.43	$40 + 10$	$19 + 10$	4.63	84.77	$43 + 11$	$26 + 11$	9.89

Table 1: Summary of the results obtained on the fixed binarisation MNIST dataset. L is the number of stochastic layers of the model, "NLL" is the negative log-likelihood, "PCA AU" is the number of active units measured by applying PCA, and "D_{KL}" is $D_{KL}\left(q(\mathbf{h} \mid \mathbf{x})||p(\mathbf{h} \mid \mathbf{x})\right)$. For models with two latent layers, "$k_1 + k_2$" denotes k_1 active units in the first layer and k_2 in the second layer. The generative performance of IWAEs improves with increasing k, unlike that of VAEs. Two-layer models achieve better generative performance than one-layer models. For comparison, the original article's [BGS16] results are presented in Appendix C (Figure 7).

		MNIST (Stochastic Binarisation)							
		VAE				IWAE			
L	k	NLL	Active Units	PCA AU	D_{KL}	NLL	Active Units	PCA AU	D_{KL}
1	1	87.96	20	20	4.87	87.67	21	21	4.82
	5	87.44	21	21	4.78	86.97	23	23	7.27
	50	87.54	21	21	4.70	86.34	25	25	12.1
2	1	85.59	$21 + 8$	$20 + 8$	4.68	86.19	$25 + 8$	$19 + 8$	4.61
	5	85.47	$22 + 8$	$19 + 8$	4.41	85.01	$34 + 10$	$22 + 10$	6.25
	50	85.57	$21 + 10$	$19 + 10$	4.30	84.37	$65 + 11$	$27 + 11$	9.27

Table 2: Summary of the results obtained on the stochastically binarised MNIST dataset. Explanation of abbreviations are given in the caption in Table 1. Findings are similar to those in Table 1. For comparison, the original paper's [BGS16] results are presented in Appendix C (Figure 6).

		Omniglot							
		VAE				IWAE			
L	k	NLL	Active Units	PCA AU	D_{KL}	NLL	Active Units	PCA AU	D_{KL}
1	1	113.17	29	29	6.51	113.20	28	28	8.45
	50	113.50	27	27	7.51	109.90	36	36	14.11
2	1	111.40	$95 + 8$	$25 + 8$	7.98	111.19	$92 + 9$	$30 + 9$	8.32
	50	111.52	$62 + 10$	$28 + 10$	8.01	108.48	$99 + 10$	$39 + 10$	15.11

Table 3: Summary of the results obtained on the Omniglot dataset. Explanation of abbreviations are given in the caption in Table 1. For comparison, the original paper's [BGS16] results are presented in Appendix C (Figure 6).

	Fashion MNIST							
	VAE				IWAE			
k	NLL	Active Units	PCA AU	D_{KL}	NLL	Active Units	PCA AU	D_{KL}
1	235.01	7	7	4.46	234.68	8	8	3.49
50	235.73	7	7	5.68	231.95	14	14	11.57

Table 4: Summary of the results obtained on the Fashion MNIST dataset for single-layer models. Findings are similar to those from Table 1. Explanation of abbreviations are given in the caption in Table 1.

	$\mathcal{L}_\alpha$ **Objective Function**			
α	NLL	Active units	PCA active units	D_{KL}
0	88.27	20	20	5.12
0.25	88.45	19	19	5.24
0.5	88.01	21	21	5.27

Table 5: Summary of the results obtained for the $\mathcal{L}_\alpha$ objective function using a VAE model with 1 stochastic layer and $k = 50$ trained on the fixed binarised MNIST dataset with varying values of α. Explanation of abbreviations are given in the caption in Table 1. The results did not change significantly for varying values of α.

	$\mathcal{L}_k^{median}$ **Objective Function**			
k	NLL	Active units	PCA active units	D_{KL}
50	88.05	20	20	5.62

Table 6: Summary of the results obtained for the $\mathcal{L}_k^{median}$ objective function using a model with 1 stochastic layer and $k = 50$ trained on the fixed binarised MNIST dataset. Explanation of abbreviations are given in the caption in Table 1.

	$\mathcal{L}_{k,p}^{power}$ **Objective Function**			
p	NLL	Active units	PCA active units	D_{KL}
0.5	87.88	25	25	12.61
2	86.75	25	25	12.52
3	86.70	26	26	13.01
5	86.80	25	25	12.45

Table 7: Summary of the results obtained for the $\mathcal{L}_{k,p}^{power}$ objective function using a model with 1 stochastic layer and $k = 50$ trained on the fixed binarised MNIST dataset with varying values of p. Explanation of abbreviations are given in the caption in Table 1.

	CIWAE			
β	NLL	Active units	PCA active units	D_{KL}
0.05	87.69	24	24	5.66
0.25	86.27	25	25	7.20
0.5	86.59	23	23	5.35

Table 8: Summary of the results obtained for the CIWAE objective function using a model with 1 stochastic layer and $k = 50$ trained on the stochastically binarised MNIST dataset with varying values of β. Explanation of abbreviations are given in the caption in Table 1.

		MIWAE			
k_1	k_2	NLL	Active units	PCA active units	D_{KL}
1	50	87.54	21	21	4.70
5	10	86.92	23	23	6.80
10	5	86.58	24	24	8.19
50	1	86.34	25	25	12.10

Table 9: Summary of the results obtained for the MIWAE objective function using a model with 1 stochastic layer trained on the stochastically binarised MNIST dataset with two different settings of k_1 and k_2. Observe that the case $k_1 = 1, k_2 = 50$ is simply the VAE objective function, while the case $k_1 = 50, k_2 = 1$ corresponds to the IWAE objective function. Explanation of abbreviations are given in the caption in Table 1.

Changing Objective Function									
First Stage					Second Stage				
trained as	NLL	AU	PCA AU	D_{KL}	trained as	NLL	AU	PCA AU	D_{KL}
IWAE k=50	87.13	24	24	12.76	VAE k=50	87.50	24	24	5.10
IWAE k=50	87.13	24	24	12.76	VAE k=1	87.69	24	24	5.36
VAE k=50	90.81	13	13	5.98	IWAE k=50	88.25	19	19	12.70
VAE k=1	89.88	14	14	5.40	IWAE k=50	88.70	23	20	12.17

Table 10: Summary of the results obtained by continuing to train a single-layer VAE model with the IWAE objective, and vice versa. Further training the VAE with the IWAE objective increased the number of active units and test log-likelihood - training the IWAE with the VAE objective seemed to have a weaker (opposite) effect on the log-likelihood, and no effect on the number of active units. Explanation of abbreviations are given in the caption in Table 1.

More generally, we question the notion that a greater number of active units for IWAEs (with or without applying PCA) necessarily corresponds to a "richer latent representation", as is claimed in [BGS16]. To us, the expression "richer latent representation" suggests the idea of a better encoding of the input data $\mathbf{x}$ (which could then be approximately recovered using the decoder). A greater number of active units does mean that the approximate posterior distribution $q(\mathbf{h} \mid \mathbf{x})$ is more spread-out (this is illustrated in Figure 2 in Appendix B). However, we have observed earlier than it tends to be a less faithful approximation of the true posterior distribution $p(\mathbf{h} \mid \mathbf{x})$ than with VAEs, which can translate to poorer encoding-decoding performance. In other words, though $\mathbf{h}$ takes more diverse values, it does not necessarily mean that it is a better encoding of the input data (see Figure 3 in Appendix B for an illustration).

5 Conclusion

We confirmed most of the conclusions from [BGS16] regarding the increased generative power of models trained using the IWAE objective function, though we were less in agreement with their analysis of the number of active units. We also observed a decrease in the performance of the recognition network, as predicted by [Rai+18], which can be mitigated using more subtle objective functions. The authors of [BGS16] did not comment on this phenomenon - in fact, they made no claim as to the impact on the recognition network, and seemed mostly interested in the generative power of the model. As mentioned in Section 2, there are applications for which the performance of the encoder are paramount, while it is of no importance for others. The best objective function surely depends on the task considered.

While we believe we fulfilled our initial objectives of replicating the original paper from scratch and performing an in-depth analysis of the strengths and weaknesses of the IWAE, we do see room for improvement. Specifically, we worked exclusively on binarised grayscale images; we would be curious to study how the IWAE's strengths and weaknesses are amplified in different domains such as the RGB natural images of CIFAR10 or text-data. Similarly, we only evaluated two model architectures, both of which are shallow feed-forward neural networks. There is no reason not to apply the techniques studied here to any other machine learning model, such as convolutional neural network and recurrent neural networks.

On a smaller note, there were minor inconsistencies between our experimental results and those of the original article, as we binarised our datasets upon loading them, while the original code binarises them at the beginning of each epoch (unlike what is suggested in their text). Given more time, we would have rerun the experiments using both procedures for a more precise comparison. We also would have run several iterations of each experiment in order to mitigate the effects of random variability and reach more definitive conclusions, and experimented with a wider range of parameters choices.

References

[BGS16] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. *Importance Weighted Autoencoders*. 2016. arXiv: 1509.00519 [cs.LG].

[doc] MLPython 0.1 documentation. *datasets.binarized_mnist*. http://www.dmi.usherb.ca/~larocheh/mlpython/_modules/datasets/binarized_mnist.html. (Accessed on 03/30/2021).

[FT19] Axel Finke and Alexandre Thiery. *On importance-weighted autoencoders*. arXivpreprintarXiv:1907.10477. 2019.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[Goo+14] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[KW14] Diederik Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *Proceedings of the 2nd International Conference on Learning Representations (ICLR)* (2014).

[Lak] Brenden Lake. *brendenlake/omniglot: Omniglot data set for one-shot learning*. https://github.com/brendenlake/omniglot/. (Accessed on 03/31/2021).

[LCB] Yann LeCun, Corinna Cortes, and Chris Burges. *MNIST handwritten digit database*. http://yann.lecun.com/exdb/mnist/. (Accessed on 03/31/2021).

[Li+20] Chunyuan Li et al. "Optimus: Organizing Sentences via Pre-trained Modeling of a Latent Space". In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020).

[Rai+18] Tom Rainforth et al. "Tighter variational bounds are not necessarily better". In: *International Conference on Machine Learning* (2018).

[REE17] Adam Roberts, Jesse Engel, and D. Eck. "Hierarchical Variational Autoencoders for Music". In: 2017.

[Sim+19] Nikola Simidjievski et al. "Variational Autoencoders for Cancer Data Integration: Design Principles and Computational Practice". In: *Frontiers in Genetics* 10 (Dec. 2019), p. 1205. DOI: 10.3389/fgene.2019.01205.

[SLY15] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. "Learning Structured Output Representation using Deep Conditional Generative Models". In: *Part of Advances in Neural Information Processing Systems 28 (NIPS 28)* (2015).

[Tuc+19] George Tucker et al. "Doubly Reparameterized Gradient Estimators for Monte Carlo Objectives". In: *International Conference on Learning Representations (ICLR)* (2019).

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

A Model Architectures

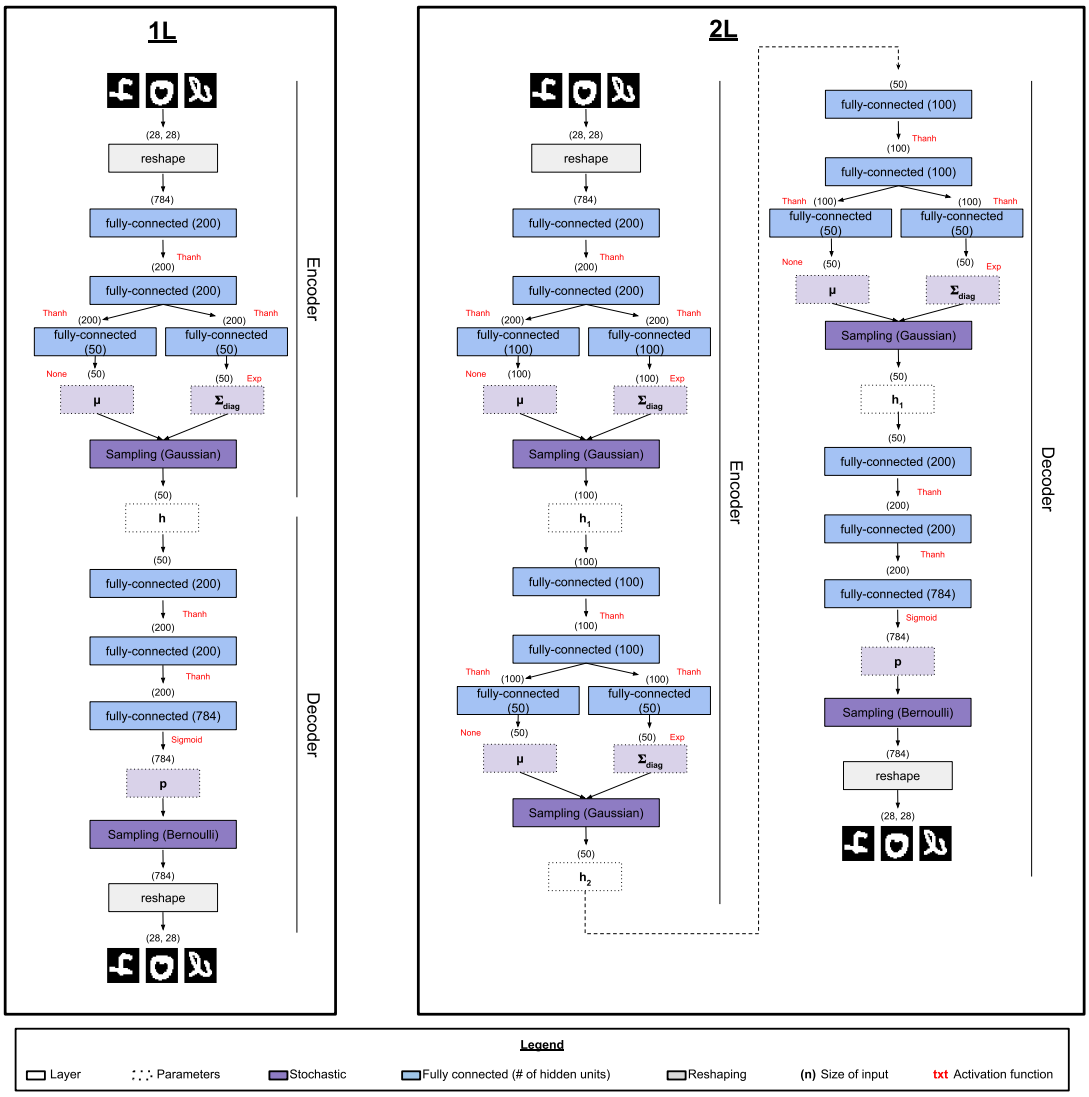

Figure 1: The architecture corresponding to the 1L (left) and 2L (right) experiments. For both model architectures the input, originally in 28×28 format are flattened to a vector of 754 units. Symmetrically, the output is reshaped to a 28×28. **1L:** The encoder is formed of one stochastic layer (see Subsection 2.1) with 50 latent units, which itself is composed of two sequential deterministic feed-forward fully-connected layers with 200 hidden units (*tanh* activations), two parallel fully-connected layers of 50 hidden units that compute the mean vector and the diagonal covariance matrix from which the final Gaussian sampling layer outputs the latent representation h_1. The decoder's structure mirrors that of the encoder, but the output is passed through a sigmoid function that computes the parameters $\mathbf{p}$ of a Bernoulli distribution used to re-binarise the image. **2L:** The encoder is built up of two stochastic layers: the first with 100 and the second with 50 units; their structure is analogous to that of the 1L architecture. Again, the structure of the decoder mimics that of the encoder.

B Additional Visualisation

B.1 The IWAE Learns a More Spread Out Posterior

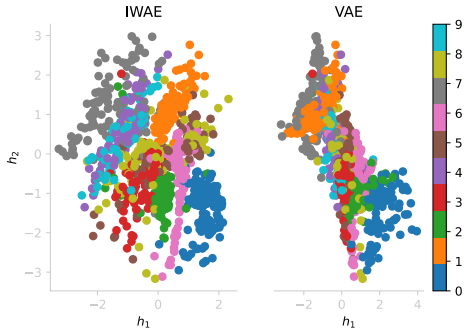

Figure 2: Latent space representations for the IWAE (left) and VAE (right) generated from the MNIST dataset for two dimensions, in which h_1 is plotted on the x-axis and h_2 on the y-axis. The encoder maps the image of a digit to a location in latent space; these points form clusters which are identified by the legend. The latent space representation learned by the IWAE is more spread-out.

B.2 A Qualitative Analysis of the Generative Power of an IWAE compared to a VAE

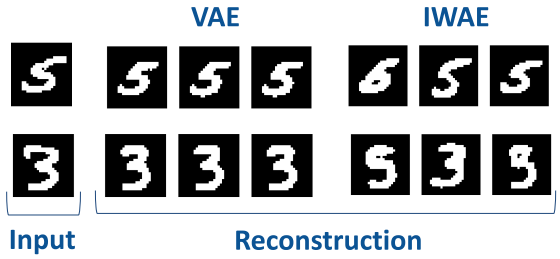

Figure 3: A demonstration of the encoding-decoding process of VAE and IWAE models using sample images from the MNIST dataset. IWAEs tend to make more reconstruction errors.

B.3 Example Reconstructions of Fashion MNIST from an IWAE

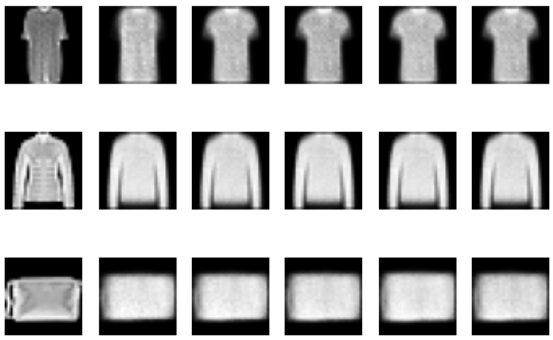

Figure 4: The left-most image is the input image fed into the IWAE. Each of the others are the results of encoding and decoding the first image. Usually, we binarise the input and output to stay consistent with previous literature; here we do not as the Fashion MNIST is hard to visualise when binarised.

B.4 Visualisation of the Latent Space of an IWAE

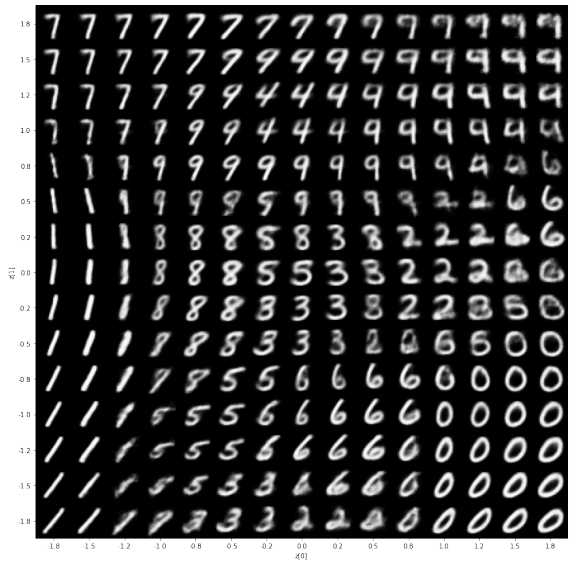

Figure 5: A visualisation of the latent space of an IWAE's with latent dimension of 2. The image is obtained by sampling the latent space uniformly, and passing each representation through the decoder. This visualisation shows how the model learns to represent each digit in this 2D latent space.

C Original article's results

		MNIST				OMNIGLOT			
		VAE		IWAE		VAE		IWAE	
# stoch. layers	k	NLL	active units	NLL	active units	NLL	active units	NLL	active units
1	1	86.76	19	86.76	19	108.11	28	108.11	28
	5	86.47	20	85.54	22	107.62	28	106.12	34
	50	86.35	20	84.78	25	107.80	28	104.67	41
2	1	85.33	16+5	85.33	16+5	107.58	28+4	107.56	30+5
	5	85.01	17+5	83.89	21+5	106.31	30+5	104.79	38+6
	50	84.78	17+5	82.90	26+7	106.30	30+5	103.38	44+7

Figure 6: The results obtained for MNIST (stochastically binarised) and Omniglot by the original article [BGS16] with the same experiment conditions.

# stoch layers	k	VAE NLL	VAE active units	IWAE NLL	IWAE active units
1	1	88 71	19	88 71	19
	5	88 83	19	87 63	22
	50	89 05	20	87 10	24
2	1	88 08	16+5	88 08	16+5
	5	87 63	17+5	86 17	21+5
	50	87 86	17+6	85 32	24+7

Figure 7: The results obtained for Binarised MNIST by the original paper [BGS16] with the same experiment conditions.

	First stage			Second stage		
	trained as	NLL	active units	trained as	NLL	active units
Experiment 1	VAE	86.76	19	IWAE, $k = 50$	84.88	22
Experiment 2	IWAE, $k = 50$	84.78	25	VAE	86.02	23

Figure 8: The results obtained from continuing to train a VAE model with the IWAE objective, and vice versa, by the original paper [BGS16] with the same experiment conditions.

D Code

D.1 Stochastic layer

```
# Stochastic layer composed of two deterministic layers and a gaussian distribution
class Stochastic_layer(tf.keras.Model):
    def __init__(self, n_hidden, n_latent, **kwargs):
        super(Stochastic_layer, self).__init__(**kwargs)

        self.l1 = tf.keras.layers.Dense(n_hidden, activation=tf.nn.tanh)
        self.l2 = tf.keras.layers.Dense(n_hidden, activation=tf.nn.tanh)
        self.lmu = tf.keras.layers.Dense(n_latent, activation=None)
        self.lstd = tf.keras.layers.Dense(n_latent, activation=tf.exp)

    # outputs a distribution (from which we can then sample or compute likelihoods)
    def call(self, input):
        y1 = self.l1(input)
        y2 = self.l2(y1)
        q_mu = self.lmu(y2)
        q_std = self.lstd(y2)
        qh_given_input = tfd.Normal(q_mu, q_std + 1e-6)
        return qh_given_input
```

Code 1: The 'Stochastic_layer' class.

D.2 Encoder

```
# contains n_stochastic layers. n_hidden and n_latent are vectors of integers of
# length n_stochastics, each coordinate specifying the way to build the corresponding
class Encoder(tf.keras.Model):
    def __init__(self, n_hidden, n_latent, n_stochastic, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.stochastic_layers=[]
        self.n_stochastic = n_stochastic
        for i in range(n_stochastic):
          self.stochastic_layers.append(Stochastic_layer(n_hidden[i], n_latent[i]))

    # Returns a list h = [h1,h2,...,hn_stochastics]),
    # where each hi is of the shape [n_samples,batch_size,n_latent[i-1]]
    # Returns the probability q(h|x) as well, which will be a n_samples x batch_size tensor
    # Doesn't return any distribution at the moment
    # Data x needs to be preemptively flattened
    def call(self, x, n_samples):
      assert(x.shape[1] == 28*28)
      qh1Ix = self.stochastic_layers[0](x)
      h1 = qh1Ix.sample(n_samples)
      log_qh1Ix = tf.reduce_sum(qh1Ix.log_prob(h1),axis = -1)
      h = [h1]
      distributions=[qh1Ix]
      log_qhip1Ihi = [log_qh1Ix]
      for i in range(1,self.n_stochastic):
        # the distribution q(hi+1|hi), with h0 := x
        qhip1Ihi = self.stochastic_layers[i](h[-1])
        # the corresponding samples
        h.append(qhip1Ihi.sample())
        # the scalar log(q(hi+1|hi))
        log_qhip1Ihi.append(tf.reduce_sum(qhip1Ihi.log_prob(h[-1]), axis=-1))
        distributions.append(qhip1Ihi)
```

```
      log_qhIx = tf.reduce_sum(log_qhip1Ihi,axis = 0)

      return h, log_qhIx, distributions[-1]
```

Code 2: The 'Encoder' class.

D.3 Decoder

```
class Decoder(tf.keras.Model):
    def __init__(self, n_hidden, n_latent, n_stochastic, dataset_bias="Binarized_MNIST", ...
        **kwargs):
        super(Decoder, self).__init__(**kwargs)

        self.stochastic_layers=[]
        self.n_stochastic = n_stochastic
        # the first layer goes from hL to hL-1, second layer from hL-1 to hL-2,
        # ..., last layer from h1 to x  -> stochastic_layers[i] goes from
        # hL-i to hL-1-i (with h0=x)
        for i in range(n_stochastic-1):
          self.stochastic_layers.append(Stochastic_layer(n_hidden[i], n_latent[i]))

        self.stochastic_layers.append(
            tf.keras.Sequential(
              [
                  tf.keras.layers.Dense(n_hidden[-1], activation=tf.nn.tanh),
                  tf.keras.layers.Dense(n_hidden[-1], activation=tf.nn.tanh),
                  tf.keras.layers.Dense(28*28, activation="sigmoid", ...
                        bias_initializer=self.get_bias(dataset_bias))
              ]
        ) )

    # Outputs the probabilities of being black for each pixel given h=[h1,...,h_n_stochastic]
    # (i.e., in fact, given h1), as well as the corresponding Bernoulli distribution p(x|h)
    def call(self, h):
        probs = self.stochastic_layers[-1](h[0])
        probs = probs*(1-10**(-6)) + 10**(-7)
        pxIh = tfd.Bernoulli(probs=probs)

        return probs, pxIh

    def generate_x(self,h_n_stochastic):
      # h_L of the shape [n_samples,batch_size,D]
      # (in most usages, n_samples should be 1)
      reversed_h = [h_n_stochastic]
      # unlike h in the rest of this code, reversed_h = [h_L,h_L-1,...,h_1]
      for i in range(self.n_stochastic-1):
        phLmim1IhLmi = self.stochastic_layers[i](reversed_h[-1])
        reversed_h.append(phLmim1IhLmi.sample())
      # this turns reversed_h into h
      reversed_h.reverse()
      probs_x_reconstructed, _ = self.call(reversed_h)
      return probs_x_reconstructed


    # log p(x|h): outputs log likelihood of x knowing h=[h1,...,h_n_stochastic]
    # (simply a Bernoulli)
    def get_log_pxIh(self, x, h):
      # probs is a deterministic function of h1 = h[0]
      probs = self.stochastic_layers[-1](h[0])
```

```
      probs = probs*(1-10**(-6)) + 10**(-7)
      pxIh = tfd.Bernoulli(probs=probs)
      log_pxIh = tf.reduce_sum(pxIh.log_prob(x), axis=-1)
      return log_pxIh

    # log p(h): outputs log likelihood of h=[h1,h2,...,h_n_stochastic]
    # which is equal to log p(h_n_stochastic) + p(h_n_stochastic-1|h_n_stochastic)...
    # + log p(h1|h2)
    def get_log_ph(self, h):
      ph_n_stochastic = tfd.Normal(0, 1)
      log_phn_stochastic = tf.reduce_sum(ph_n_stochastic.log_prob(h[-1]), axis=-1)
      log_phiIhip1 = [log_phn_stochastic]
      for i in range(self.n_stochastic-1):
        phiIhip1 = self.stochastic_layers[i](h[self.n_stochastic-1-i])
        log_phiIhip1.append(tf.reduce_sum(phiIhip1.log_prob(h[self.n_stochastic-2-i]),...
            axis=-1))
      get_log_ph = tf.reduce_sum(log_phiIhip1,0)
      return get_log_ph

    def get_log_pxh(self, x, h):
        return self.get_log_pxIh(x, h) + self.get_log_ph(h)

    def get_bias(self, dataset="Binarized_MNIST"):
      if "binarized_mnist" in dataset.lower():
        tf.print("Setting the bias to the Fixed Binarisation MNIST")
        Xtrain, _ = tfds.load("mnist",
                              split=['train', 'test'],
                              shuffle_files=True,
                              batch_size=-1)

        Xtrain = Xtrain["image"].numpy()

      elif "mnist" in dataset.lower():
        tf.print("Setting the bias to the Stochastic Binarisation MNIST")
        # For initializing the bias in the final Bernoulli layer for p(x|z)
        (Xtrain, _), (_, _) = keras.datasets.mnist.load_data()

      elif "omniglot" in dataset.lower():
        tf.print("Setting the bias to the OMNIGLOT")
        d = sio.loadmat("chardata.mat")
        Xtrain = d["data"].transpose().reshape((-1, 28*28))
      else:
        raise Exception("Trying to set the initialisation for the bias, \
                        but the dataset is not recognized")

      Ntrain = Xtrain.shape[0]
      # reshape to vectors
      Xtrain = Xtrain.reshape(Ntrain, -1) / 255
      train_mean = np.mean(Xtrain, axis=0)
      bias = -np.log(1. / np.clip(train_mean, 0.001, 0.999) - 1.)
      return tf.constant_initializer(bias)
```

Code 3: The 'Decoder' class.

D.4 Flexible Model

```
class Flexible_Model(keras.Model):
    def __init__(self, n_hidden_encoder, n_hidden_decoder, n_latent_encoder,
                 n_latent_decoder, dataset_bias="Binarized_MNIST",
```

```
            loss_function = "VAE", k = 50, p = 1, alpha = 1, beta=0.5, **kwargs):
    """
    Params
    ------
    n_hidden_encoder: list indicating the dimensions of the hidden layers of
                      each stochastic layer in the encoder. E.g. [100, 200],
                      creates two stochastic layers in the encoder with
                      hidden layers of shape 100 and 200, respectively
    n_hidden_decoder: analogous to n_hidden_encoder, but for the decoder.
                      Note: the layers need to be fed in the order in which
                      the input must be fed. That is, to maintain symmetry,
                      it should be the inverse of n_hidden_encoder
    n_latent_encoder: list indicating the dimensions of the outputs of
                      each stochastic layer in the encoder. E.g. [100, 50],
                      creates two stochastic layers in the encoder with
                      outputs of shape 100 and 50, respectively
    n_latent_decoder: analogous to n_latent_encoder, but for the decoder.
                      Note: the layers need to be fed in the order in which
                      the input must be fed. That is, to maintain symmetry,
                      it should be the inverse of n_latent_encoder. The last
                      layer needs to be 28*28 (i.e. the size of the input/
                      output)
    """

    super(Flexible_Model, self).__init__(**kwargs)

    n_stochastic_encoder = len(n_hidden_encoder)
    n_stochastic_decoder = len(n_hidden_decoder)
    self.encoder = Encoder(n_hidden_encoder, n_latent_encoder, n_stochastic_encoder)
    self.decoder = Decoder(n_hidden_decoder, n_latent_decoder, n_stochastic_decoder,...
                       dataset_bias)
    self.n_stochastic_encoder = n_stochastic_encoder
    self.n_stochastic_decoder = n_stochastic_decoder
    self.dataset_bias = dataset_bias
    self.loss_function = loss_function
    self.k = k
    self.p = p
    self.alpha = alpha
    self.beta= beta
    self.epoch = 0

#@tf.function
def train_step(self, x):
  # We expect x to be of the shape [batch_size,28,28,1]
    k = self.k
    p = self.p
    alpha = self.alpha
    beta= self.beta
    with tf.GradientTape() as tape:
      if self.loss_function == "VAE":
         loss = - self.get_L(x,k)
      elif self.loss_function == "IWAE":
         loss = - self.get_L_k(x,k)
      elif self.loss_function == "VAE_V1":
        loss = - self.get_L_V1(x,k)
      elif self.loss_function == "L_alpha":
        loss = - self.get_L_alpha(x,k,alpha)
      elif self.loss_function == "L_power_p":
        loss = - self.get_L_power_p(x,k,p)
      elif self.loss_function == "L_median":
```

```
        loss = - self.get_L_median(x,k)
      elif self.loss_function == "CIWAE":
        loss = -self.get_L_CIWAE(x,k,beta)
    res = {self.loss_function: loss}
    grads = tape.gradient(loss, self.trainable_weights)
    self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
    self.epoch += 1

    return res

def reconstructed_x_probs(self,x):
  x = keras.layers.Flatten()(x)
  h, _ , _ = self.encoder(x,1)
  reconstructed_x = self.decoder.generate_x(h[-1])

  return reconstructed_x

def get_reconstruction_loss(self,x):
  probs = self.reconstructed_x_probs(x)
  y = tf.expand_dims(keras.layers.Flatten()(x), axis=0)
  y = tf.expand_dims(y, axis=-1)
  probs = tf.expand_dims(probs, axis=-1)
  reconstruction_loss = tf.reduce_mean(tf.reduce_sum...
                        (keras.losses.binary_crossentropy(y, probs),axis=-1))
  return reconstruction_loss

def get_levels_of_units_activity(self,x,n_samples):
  # of the shape [batch_size,784]
  x = keras.layers.Flatten()(x)
  variances = []
  eigen_values=[]
  h_means, _ , _ = self.encoder(x,1)
  for i in range(n_samples - 1):
    h , _ , _ = self.encoder(x,1)
    for index, hi in enumerate(h):
      h_means[index] += hi
  for hi_mean in h_means:
    # take the mean over q(h|x)
    # should now be of dimension [batch_size,n_latent_encoder[i-1]]
    hi_mean = tf.squeeze(hi_mean/n_samples, axis= 0)
    # should now be of dimension [n_latent_encoder[i-1]]
    variances.append(tf.math.reduce_variance(hi_mean, axis = 0))
    eigen_values.append(self.get_eigenvalues_PCA(hi_mean))
  return variances, eigen_values

# data meant to be of dimension [batch_size,D]
def get_eigenvalues_PCA(self,data):
    # normalize
    normalized_data = data - tf.reduce_mean(data, axis=0)
    # Empirical covariance matrix
    cov = tf.tensordot(tf.transpose(normalized_data), normalized_data, axes=1) / ...
                normalized_data.get_shape()[0]
    # Find eigen_values
    eigen_values, _ = tf.linalg.eigh(cov)
    return eigen_values


def get_active_units(self,variances,eigen_values, threshold=0.01):
  active_units = []
  number_active_units = []
```

```
    number_active_units_PCA = []
    for index, variance in enumerate(variances):
      active_units.append([1 if coordinate_var>threshold ...
                      else 0 for coordinate_var in variance])
      number_active_units.append(tf.reduce_sum(active_units[-1]))
      number_active_units_PCA.append(tf.reduce_sum([1 if eig>threshold ...
                      else 0 for eig in eigen_values[index]]))
    return active_units, number_active_units, number_active_units_PCA

  def get_E_qhIx_log_pxIh(self,x,n_samples):

    x = keras.layers.Flatten()(x)
    # encoding
    # h = [h1,...,h_n_stochastic_encoder]
    # h_i of the shape [n_samples,batch_size,n_latent_encoder[i-1]]
    # log_qhIx of the shape [n_samples, batch_size]
    h,log_qhIx, qhLIx = self.encoder(x,n_samples)
    # decoding
    # probs of the shape [n_samples, batch_size,784]
    # pxIh a distribution
    probs, pxIh = self.decoder(h)

    y = tf.expand_dims(x, axis=0)
    y = tf.repeat(y, n_samples, axis=0)
    # y of dim [n_samples,batch_size,784]
    # E_q(h|x)[log(p(x|h))]
    y = tf.expand_dims(y, axis=-1)
    probs = tf.expand_dims(probs, axis=-1)
    E_qhIx_log_pxIh = -1*tf.reduce_mean(tf.reduce_sum ...
                      (keras.losses.binary_crossentropy(y, probs),axis=-1))

    return E_qhIx_log_pxIh

  def get_log_weights(self,x,n_samples):
    # of the shape [batch_size,784]
    x = keras.layers.Flatten()(x)

    # encoding
    # h = [h1,...,h_n_stochastic_encoder]
    # h_i of the shape [n_samples,batch_size,n_latent_encoder[i-1]]
    # log_qhIx of the shape [n_samples, batch_size]
    h,log_qhIx, _ = self.encoder(x,n_samples)

    # decoding
    # probs of the shape [n_samples, batch_size,784]
    # pxIh a distribution
    probs, pxIh = self.decoder(h)

    # all of the shape [n_samples, batch_size]
    log_pxIh = self.decoder.get_log_pxIh(x,h)
    log_ph = self.decoder.get_log_ph(h)
    log_pxh = log_ph + log_pxIh

    # Computing the weights wi
    # of the shape [n_samples,batch_size] (due to the sum along the batch_size axis)
    log_weights=log_pxh-log_qhIx

    return log_weights

    # L_k = - total loss IWAE
```

```
def get_L_k(self, x, k):
  # log_weights of the shape [n_samples,batch_size]
  log_weights = self.get_log_weights(x,k)
  # L_k(x) computed as in the article (k= n_samples), i.e. MC approximation
  # of E_q(h|x)[log(mean(w1,...,wk))]
  L_k = self.L_k_from_weights(log_weights)

  return L_k

def L_k_from_weights(self,log_weights):
  # log_weights of the shape [n_samples,batch_size]
  # L_k(x) computed as in the article (k= n_samples), i.e. MC approximation
  # of E_q(h|x)[log(mean(w1,...,wk))]
  # max is of shape [batch_size]
  max = tf.reduce_max(log_weights, axis = 0)
  L_k = tf.reduce_mean(tf.math.log(tf.reduce_mean(tf.exp(log_weights - max), axis=0)) + max)
  return L_k

# computes L_median := E_qhIx[log(median(w1,...,wk))] =~ E_qhIx[median(log(w1,...,wk))]
def get_L_median(self,x,k):
  # log_weights of the shape [n_samples,batch_size]
  log_weights = self.get_log_weights(x,k)
  # of the shape [batch_size]
  median_log_weights = tfp.stats.percentile(log_weights, 50.0, interpolation='midpoint', ..
                  axis=0)
  L_power_p = tf.reduce_mean(median_log_weights)
  return L_power_p

# Computes L_CIWAE = beta * LVAE + (1-beta) * LIWAE
def get_L_CIWAE(self,x,n_samples,beta):
  return beta*self.get_L(x, n_samples) + (1-beta)*self.get_L_k(x,n_samples)

# computes L_alpha := E_qhIx_log_pxIh - alpha*Dkl_qhIx_ph
def get_L_alpha(self,x,n_samples,alpha):
  x = keras.layers.Flatten()(x)
  h,log_qhIx, _ = self.encoder(x,n_samples)
  probs, pxIh = self.decoder(h)
  log_pxIh = self.decoder.get_log_pxIh(x,h)
  log_ph = self.decoder.get_log_ph(h)
  log_pxh = log_ph + log_pxIh
  log_weights=log_pxh-log_qhIx
  L = self.L_from_weights(log_weights)
  #---
  y = tf.expand_dims(x, axis=0)
  y = tf.repeat(y, n_samples, axis=0)
  y = tf.expand_dims(y, axis=-1)
  probs = tf.expand_dims(probs, axis=-1)
  E_qhIx_log_pxIh = -1*tf.reduce_mean(tf.reduce_sum ...
                (keras.losses.binary_crossentropy(y, probs),axis=-1))
  L_alpha=(1-alpha)*E_qhIx_log_pxIh + alpha*L
  return L_alpha

# computes L_power_p = E_q(h|x)[log(mean(w1^p,...,wk^p)^1/p)]
def get_L_power_p(self,x,k,p):
  log_weights = self.get_log_weights(x,k)
  max = tf.reduce_max(log_weights, axis = 0)
  L_power_p = tf.reduce_mean(tf.math.log(tf.reduce_mean ...
                (tf.exp((log_weights-max)*p), axis=0))/p + max)
  return L_power_p
```

```
def get_Dkl_qhIx_phIx(self,x,k):
  return  -1*(self.get_L(x,k) + self.get_NLL(x))

def get_Dkl_qhIx_ph(self,x,k):
  return  self.get_E_qhIx_log_pxIh(x,k) -self.get_L(x,k)

# k measures the degree of precision of the MC approximation
# L = - total loss VAE
def get_L(self,x,k=5000):

  # log_weights of the shape [n_samples,batch_size]
  log_weights = self.get_log_weights(x,k)
  # -L(x), where L(x) is computed as in the article,
  # i.e. MC approximation of E_q(h|x)[log(w)]
  # Should be equal to total_loss_VAE_1 up to MC approximation error
  L = self.L_from_weights(log_weights)
  return L

def L_from_weights(self,log_weights):
  return tf.reduce_mean(log_weights)

# alternative way of computing L (for comparison)
# only works when there is a single stochastic layer
def get_L_V1(self,x,n_samples):
  x = keras.layers.Flatten()(x)
  # encoding
  # h = [h1,...,h_n_stochastic_encoder]
  # h_i of the shape [n_samples,batch_size,n_latent_encoder[i-1]]
  # log_qhIx of the shape [n_samples, batch_size]

  # distribution qhLIx is only used to compute total_loss_VAE_1 in the case
  # where there is only one stochastic layer
  h,log_qhIx, qhLIx = self.encoder(x,n_samples)

  # decoding
  # probs of the shape [n_samples, batch_size,784]
  # pxIh a distribution
  probs, pxIh = self.decoder(h)

  y = tf.expand_dims(x, axis=0)
  y = tf.repeat(y, n_samples, axis=0)
  # y of dim [n_samples,batch_size,784]
  # E_q(h|x)[log(p(x|h))]
  y = tf.expand_dims(y, axis=-1)
  probs = tf.expand_dims(probs, axis=-1)
  E_qhIx_log_pxIh = -1*tf.reduce_mean(tf.reduce_sum ...
                (keras.losses.binary_crossentropy(y, probs),axis=-1))
  Dkl_qhIx_ph = -0.5 * (1 + 2*tf.math.log(qhLIx.scale) - tf.math.square(qhLIx.loc) ...
            - tf.math.square(qhLIx.scale))
  Dkl_qhIx_ph = tf.reduce_mean(tf.reduce_sum(Dkl_qhIx_ph, axis=-1))
  L = E_qhIx_log_pxIh-Dkl_qhIx_ph
  return L

# MC estimation of log(p(x)) (corresponds to computing L_k(x) for very large k)
def get_NLL(self,x,k=5000):
  return - self.get_L_k(x,k)

def get_NLL_without_inactive_units(self,x,threshold=0.01,n_samples = 5000):
  variances, eigen_values = self.get_levels_of_units_activity(x,n_samples)
  active_units, _, _ = self.get_active_units(variances,eigen_values,threshold)
```

```
    x = keras.layers.Flatten()(x)
    y = x
    assert(y.shape[1] == 28*28)
    qh1Iy = self.encoder.stochastic_layers[0](y)
    h1 = qh1Iy.sample(n_samples)
    modified_h1 = h1*active_units[0]
    log_qh1Iy =  tf.reduce_sum(qh1Iy.log_prob(modified_h1),axis = -1)
    h=[modified_h1]
    distributions = [qh1Iy]
    log_qhip1Ihi = [log_qh1Iy]
    for i in range(1,self.encoder.n_stochastic):
      qhip1Ihi = self.encoder.stochastic_layers[i](h[-1])
      hip1 = qhip1Ihi.sample()
      modified_hip1 = hip1*active_units[i]
      h.append(qhip1Ihi.sample())
      log_qhip1Ihi.append(tf.reduce_sum(qhip1Ihi.log_prob(h[-1]), axis=-1))
      distributions.append(qhip1Ihi)
    log_qhIx = tf.reduce_sum(log_qhip1Ihi,0)

    probs, pxIh = self.decoder(h)
    log_pxIh = self.decoder.get_log_pxIh(x,h)
    log_ph = self.decoder.get_log_ph(h)
    log_pxh = log_ph + log_pxIh
    log_weights=log_pxh-log_qhIx

    return - self.L_k_from_weights(log_weights)

  def get_training_statistics(self, x, k, batch_size=10):
    res = {}
    res2 = {}

    batched_x_test = tf.reshape(x, (-1, batch_size, 28, 28, 1))
    n_batches = batched_x_test.shape[0]

    res["VAE"] = 0
    res["IWAE"] = 0
    res["NLL"] = 0
    res["E_q(h|x)[log(p(x|h))]"] = 0
    res["D_kl(q(h|x),p(h))"] = 0
    res["D_kl(q(h|x),p(h|x))"] = 0
    res["reconstruction_loss"] = 0

    tf.print(f"Evaluating the test statistics")
    for batch in batched_x_test:
      res["VAE"] += keras.backend.get_value(self.get_L(batch, k))/n_batches
      res["IWAE"] += keras.backend.get_value(self.get_L_k(batch, k))/n_batches
      res["NLL"] += keras.backend.get_value(self.get_NLL(batch))/n_batches
      res["E_q(h|x)[log(p(x|h))]"] += keras.backend.get_value(...
              self.get_E_qhIx_log_pxIh(batch, k))/n_batches
      res["D_kl(q(h|x),p(h))"] += keras.backend.get_value(...
              self.get_Dkl_qhIx_ph(batch, k))/n_batches
      res["D_kl(q(h|x),p(h|x))"] += keras.backend.get_value(...
              self.get_Dkl_qhIx_phIx(batch, k))/n_batches
      res["reconstruction_loss"] += keras.backend.get_value(...
              self.get_reconstruction_loss(batch))/n_batches

    variances, eigen_values = self.get_levels_of_units_activity(x, 1000)
    res2["active_units"], res2["number_of_active_units"], res2["number_of_PCA_active_units"] = ...
              self.get_active_units(variances, eigen_values)
    res2["variances"] = variances
```

```
    res["LL_pruned"] = self.get_NLL_without_inactive_units(batched_x_test[0])

    return res, res2


  def tensorboard_log(self, res, epoch_n=-1):
    """
        res: a dictionary of the results generated from self.get_training_statistics
        epoch_n: gives the option to override the actual epoch number
    """

    if epoch_n == -1:
      epoch_n = self.epoch

    # tensorboard logging (special chars not allowed)
    tf.summary.scalar("VAE", res["VAE"], step=epoch_n)
    tf.summary.scalar("IWAE", res["IWAE"], step=epoch_n)
    tf.summary.scalar("NLL", res["NLL"], step=epoch_n)
    tf.summary.scalar("E_q(h|x)[log(p(x|h))]", res["E_q(h|x)[log(p(x|h))]"], step=epoch_n)
    tf.summary.scalar("D_kl(q(h|x),p(h))", res["D_kl(q(h|x),p(h))"], step=epoch_n)
    tf.summary.scalar("D_kl(q(h|x),p(h|x))", res["D_kl(q(h|x),p(h|x))"], step=epoch_n)
    tf.summary.scalar("reconstruction_loss", res["reconstruction_loss"], step=epoch_n)
```

Code 4: The 'Flexible Model' class.