
Assignment 3: Deep Generative Models

Florian Wolf
University of Amsterdam
Amsterdam, 1012 WX Amsterdam
florian.wolf@student.uva.nl

Contents

1	Variational Auto Encoders	1
1.1	Latent Variable Models	1
1.2	Decoder: The Generative Part of the VAE	2
1.3	The Encoder	2
1.4	Specifying the Encoder	3
1.5	The Reparametrization Trick	3
1.6	Putting things together: Building a VAE	4
2	Generative Adversarial Networks	6
2.1	Training objective: A Minimax Game	6
2.2	Building a GAN	7
3	Generative Normalizing Flows	8
3.1	Change of variables for Neural Networks	8
3.2	Building a flow based model	9
4	Conclusion	10

Abstract

This is the report for the third assignment of the the course 'Deep Learning' for the Master program 'Artificial Intelligence' at University of Amsterdam. Enjoy reading.

1 Variational Auto Encoders

1.1 Latent Variable Models

- **Question 1.1:**

1. A standard AE uses a function to to encode the input to smaller dimension. It then decodes this lower dimension through a unrelated function back to original dimensionality. The VAE maps the input to a latent space which represent parameters for a

distribution. Sampling from this distribution will generate parameters for the outcome distribution from which we can sample new images.

2. VAE can generate different outputs with the same input. Since they sample from a distribution they will produce a different output every time. The standard AE uses a deterministic function to produce an output similar to the input. The same input will generate the every time the same image. This does not qualify as generating new data.
3. This depends highly on the task. For compressing and restoring of data it is not suitable. For denoising it is not suitable neither, since it will create a whole new image out of random noise. We can study the latent space of the VAE and use this knowledge to generate certain subclasses of the learned dataset.
4. The sampling aspect of the VAE makes it able to generate new images and therefore a generalizing generative model.

1.2 Decoder: The Generative Part of the VAE

- **Question 1.2:**

The model samples from the Bernoulli distribution. The weights for this distribution are generated by the a latent space input. This input has again been sampled from the normal distribution. Sample on top of a previous sample is called ancestral sampling.

- **Question 1.3:**

We assume that the latent space is normal distributed to make the model simpler and computation easier. This does not restrict the complexity of our model since the function which maps the latent space back to our input space is a deep neural network f_θ . This network is non-linear and can reproduce complex behaviour such as mapping a normal distribution to a Bernoulli distribution.

- **Question 1.4:**

- (a) Monte-Carlo Integral: Take random samples of a distribution over it and average over them. Requires numerical computation. The given equation hints towards Monte-Carlo integration. Instead of solving the integral we can take several samples from the distribution and average over them. This is the Monte-Carlo integration for one pixel:

$$\begin{aligned}\log p(x) &= \sum_{n=1}^N \log \mathbb{E}_{p(z_n)} [p(x|z_n)] \\ &= \log \left[\frac{1}{N} \sum_{i=1}^N f(x|z_n) \right]\end{aligned}\tag{1}$$

- (b) Monte-Carlo integration comes with the downside of high variance. Further we need to sample sufficient points to cover the whole latent space. For higher dimensions it requires too many points and becomes computationally expensive Bottou [2003].

1.3 The Encoder

- **Question 1.5:**

- (a) The KL divergence measures how different two distributions are. Therefore the smallest achievable value is $D_{\text{KL}}(q||p) = 0$ for $\mu_q = 0$ and $\sigma_q^2 = 1$ similar to distribution p . For the KL divergence to result in a larger value especially the mean of both distributions must differ. The standard deviation plays a smaller role. For example $\mu_q = 100$ and $\sigma_q^2 = 10$ would result in a high KL divergence.
- (b) The equation for the closed form of the KL divergence is:

$$KL(p||q) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}\tag{2}$$

This way we can calculate the divergence knowing the mean and standard deviation of each distribution Hershey and Olsen [2007].

- **Question 1.6:**

For the following equation we will call the right-hand side the 'lower bound'.

$$\log p(x_n) - D_{\text{KL}}(q(Z|x_n) \| p(Z|x_n)) = \mathbb{E}_{q(z|x_n)} [\log p(x_n|Z)] - D_{\text{KL}}(q(Z|x_n) \| p(Z)) \quad (3)$$

Since the KL divergence is always greater equal to zero we get $\log p(x_n)$ greater equal the lower bound. Lower bound means that this is lowest value possible for $\log p(x_n)$.

- **Question 1.7:**

To optimize the log-probability of the above equation either the KL divergence needs to be minimized or the lower bound optimized. Since the lower bound is fully tractable we can maximize it. To make the KL divergence explicitly computable we have to choose assume that the latent space is of the same normal distribution as the distribution we sample from.

- **Question 1.8:**

By pushing the lowerbound up, either the log-probability rises or the KL divergence get close to zero. The second is use-full for encoders since it confirms that the distribution q approximates p well. once we minimized the KL divergence we can approximate $\log p(x_n)$ by the lower bound Alemi et al. [2017].

1.4 Specifying the Encoder

- **Question 1.9:**

$$\begin{aligned} \mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|x_n)} [\log p_\theta(x_n|Z)] \\ \mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}(q_\phi(Z|x_n) \| p_\theta(Z)) \end{aligned} \quad (4)$$

The first term is called reconstruction term because it gives the expectation of generating the input given our latent space approximation by a distribution. By minimizing this term we maximize the probability of generating the original pixel.

The use of the second term is to regularize the reconstruction by the KL divergence between the original and the approximation distribution of the latent space. Minimizing this term makes sure we can later sample from the model using the intended Normal distribution.

- **Question 1.10:**

With this knowledge we can define the final loss function which will then be minimized. The reconstruction term can be formulated as Cross-Entropy-Loss. In this case, considering that our input image is binary and our output a Bernoulli distribution it becomes the Binary-Cross-Entropy-Loss (BCL). For the second term we plug in the values for the approximates and the normal distribution into the KL divergence.

$$\mathcal{L} = \sum_{n=1}^N \mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}} \quad (5)$$

$$\mathcal{L}_n^{\text{recon}} = \text{BCL}_{x_n}(f_\theta(z_n)) \quad (6)$$

$$(7)$$

$$\begin{aligned} \mathcal{L}_n^{\text{reg}} &= \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \\ &= \frac{\text{tr}(\Sigma(x_n)) + (\mu_{(x_n)})^2}{2} - \frac{1}{2} - \log \det(\Sigma(x_n)) \end{aligned} \quad (8)$$

1.5 The Reparametrization Trick

- **Question 1.11:**

- (a) The gradient $\nabla_\phi \mathcal{L}$ is needed for the learning process of the model. After each batch step, the weights and bias of both Encoder and Decoder are updated by subtracting their gradient times a specified learning rate.

- (b) Inside the model is a sampling unit to represent the latent space. It is not possible to backpropagate through this unit. The reason for this is that backpropagation requires deterministic mathematical operations and sampling from a distribution is stochastic.
- (c) The 'parametrization trick' offers a solution to the problem of impossible backpropagation through a stochastic unit. This trick takes the sampling part outside the model as stochastic input. Our latent space is defined as standard normal, therefore the input will be a sample from the standard normal distribution defined as ϵ . As described earlier the encoder gives out the mean and standard deviation as distribution parameters. Instead of sampling from a distribution with these parameters we take these parameters and the external randomness as input for the Decoder. The Decoder input z is also normal distributed Shu et al. [2019].

$$z = \mu + \epsilon \cdot \sigma \quad (9)$$

$$z \sim \mathcal{N}(\mu, \sigma) \quad (10)$$

1.6 Putting things together: Building a VAE

- **Question 1.12:**

A model of the presented VAE was implemented using PyTorch. The VAE consists of two classes, the Encoder and the Decoder. The Encoder takes the flattened image as input and gives out the mean and standard deviation for every dimension in latent space (20). The Neural Network inside the Encoder consists of the following architecture:

Layer	Value
Linear	input - 500
ReLU	
Dropout	0.2
Linear	500 - latent*2

Table 1: Encoder Neural Network architecture.

The weights are initialized using Xavier initialization. The bias are set to an initial value of 0.01.

The Decoder has a similar architecture only adding a Sigmoid layer in the end. The input are samples from the latent distribution. The output is normalized to the range $[0, 1]$ by the Sigmoid layer and has the same dimension as the flattened image fed into the Encoder. These two parts are joint together in the VAE. The loss function is the negative lower bound. Since the Decoder output are the means for binomial-sampling we take the binomial cross-entropy and penalize further with the KL divergence between the standard Normal distribution and the latent distribution from the Encoder. To weight both parts equally they are averaged over their own dimension and then summed up. To generate an output image we reshape the output of the Decoder and sample from the Bernoulli-distributions.

- **Question 1.13:**

The estimated lower bound serves as loss function for the VAE. The following plot shows the loss convergence during training. The loss function sums over the latent space dimension and over the output dimension. This is the reason for the high bias.

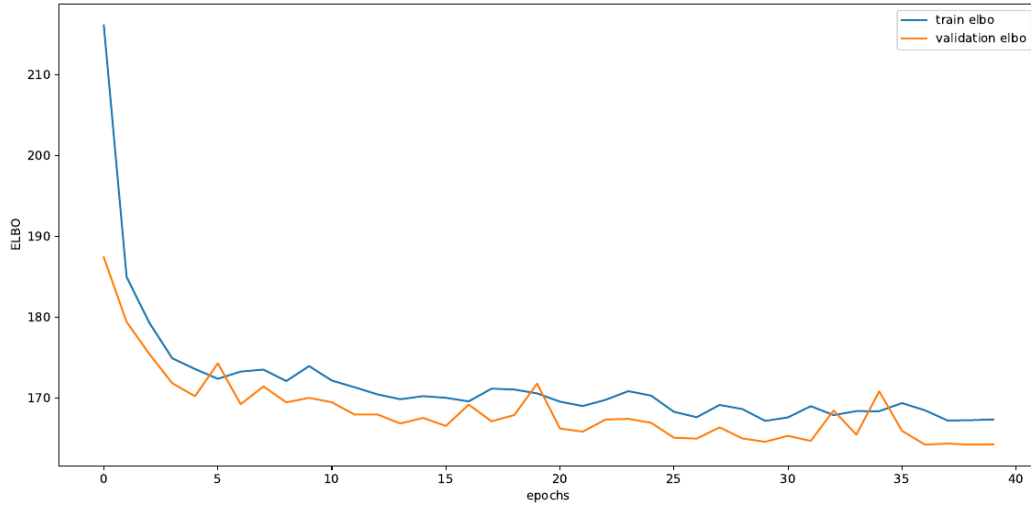


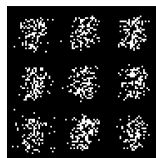
Figure 1: Convergence of the Estimated Lower bound during training and validation.

• **Question 1.14:**

The input for the VAE are batches of handwritten black-and-white numbers. Over time the Decoder learns to produce very similar outputs. The first results show high variance and appear rather randomly distributed. At the point of convergence in Epoch 6 the results show a clearer shape. The final result shows hand-written number close to the input but more pixelated. This lets us conclude that the VAE successfully learned to generate handwritten numbers by it self.



(a) Sample from Dataset



(b) Sample Epoch 1



(c) Sample Epoch 6



(d) Sample Epoch 40

Figure 2: Plot of the input image on the left and three generated image

• **Question 1.15:**

To generate new outputs we input random normal noise as vector in latent-space dimension into the Decoder. Applying the inverse cumulative distribution we find the significant range for the latent space and can use this to linearly explore it. The inverse cdf starts at probability 0.1 and ends at 0.9 we used a linearly spaced range from -2 to 2 with 10 steps. The following figure shows the plots of the Bernoulli probabilities of these latent values (note: before we sampled from the Bernoulli distributions now we plot the Bernoulli probabilities). This visualization of the latent space gives us control over our Decoder. Now it is possible to generate specific numbers. If we would want to generate a 2 for example we would feed the Decoder the vector $[-0.4, -0.4]$ Kingma and Welling [2013].



Figure 3: Manifold representation of the latent space.

2 Generative Adversarial Networks

GANs are generative networks consisting of two parts, a Generator (**G**) and a Discriminator (**D**). Both are individual networks and share the same loss function. In this section the architecture of GANs will be explained as well as how to train and implement them.

2.1 Training objective: A Minimax Game

The training of GANs relies on minimizing the loss function. This is a 'minmax game' between **D** and **G**.

- **Question 2.1:**

G takes as input samples from a distribution. The choice of distribution does not matter. For simplicity reasons it is recommended to choose a Gaussian. This input is called noise. The output of **G** is the generated item, in our case an image.

D is trained on the generated images plus a training set. It returns a classification of the input image as 'real' or 'fake'.

- **Question 2.2:**

Term 1:

$$\min_G \max_D \mathbb{E}_{p_{\text{data}}(x)} [\log D(X)] \quad (11)$$

The expected log of the output distribution of **D** identifying a real image. This Term is to be maximized.

Term 2:

$$\mathbb{E}_{p_z(z)} [\log(1 - D(G(Z)))] \quad (12)$$

This Term is the expected log of \mathbf{D} identifying by \mathbf{G} generated data. This shall be minimized. The part inside the log of both terms is the opposite and the equation can be rewritten in dependence of x . Both terms can be rewritten as KL divergences.

- **Question 2.3:**

The Term $V(D, G)$ depends on the KL divergence of \mathbf{D} and \mathbf{G} performing with optimal probability and therefore result in $KL = 1$. There remains a constant $\log 4$ so it would converge to ~ 1.5

- **Question 2.4:**

During the beginning phase of the training the generated images will be rather easy to identify. This will lead to a high probability of \mathbf{D} detecting the fake image $D(G(Z)) \sim 1$. This again will lead to an exponential negative increase of the term $\log(1 - D(G(Z)))$ or even a nan error.

2.2 Building a GAN

- **Question 2.5:**

The architecture of the GAN was build after the given template. The following table shows the Generator network:

Module	Arguments
Linear	100 -> 128
LeakyReLU(0.2)	
Linear	128 -> 256
Bnorm	
LeakyReLU(0.2)	0.2
Linear	256 -> 512
Bnorm	
LeakyReLU(0.2)	0.2
Linear	512 -> 1024
Bnorm	
LeakyReLU(0.2)	0.2
Linear	1024 -> 784
Sigmoid	

Table 2: Generating Network architecture of the GAN.

We will highlight the network choices for \mathbf{G} . This network has the objective to generate images and is therefore more complex where as \mathbf{D} is a simple image classifier. The presented network uses LeakyReLU as activation function. This type of ReLU allows negative values and allows the model to be more complex. Further the network uses batch normalization to minimize problems induced through too high variance within each batch. The final output image is normalized using a Sigmoid function. The image is represented in a flat vector. The same applies for the training images which get flattened before they are input into

D. This is a six layer network counting three linear layers. The output is a normalized probability for each input image.

The loss function for **D** is the Binary cross entropy between the output probability for the training set input and ones-label or the generated images and zero-label

• **Question 2.6:**

The following figure shows the input plots compared to generated plots during the training. At first the generated image is nothing but random noise. This shows that the generator does not have any prior knowledge on the dataset. At batch 80 the numbers look already better than the dataset. **G** produces smoother images with less noise than the original images.

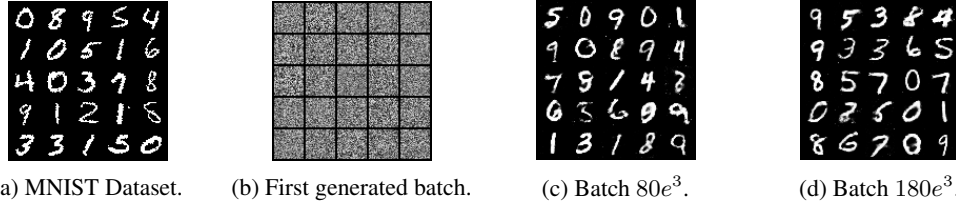


Figure 4: Plot of samples from the input images on the left and three generated images throughout training on the right.

• **Question 2.7:**

To get an insight into the GAN latent state we sample two inputs and interpolate linearly between them. The following plot shows the two numbers 1 and 9 interpolated with 7 steps. The numbers in between show blurriness and are partly not easy to classify. The reason for this is that we linearly interpolate through the Gaussian latent space. Spherical interpolation would grant better results.



Figure 5: Plot of the interpolation between two random samples in latent space.

3 Generative Normalizing Flows

3.1 Change of variables for Neural Networks

• **Question 3.1:**

In the case of a smooth mapping function f and a multivariate variable x we can rewrite the the derivative of f over x as the Jacobian-matrix $\frac{\partial f(x)}{\partial x^T}$.

$$p_X(x) = p_Z(f(x)) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \quad (13)$$

$$\log(p_X(x)) = \log(p_Z(f(x))) + \sum_{l=1}^L \log \left(\left| \det \left(\frac{\partial h_l}{\partial h_{l-1}} \right) \right| \right) \quad (14)$$

• **Question 3.2:**

In order to make this computation possible the function needs to map the input dimensions to the same output dimensions. The multivariate variables x and z have the same dimensionality. The function f has to be traceable, invertible and smooth. This excludes the use of activation functions like ReLU and gives preference to invertible function such as Tangens Hyperbolicus. The shared dimensionality of the input and output create a orthogonal Jacobian matrix for which the determinant is easily computed Berg et al. [2018].

- **Question 3.3:**

A problem arising with the definition of this model is that the computing of the Jacobian-matrix is inefficient and carries numerical uncertainty. We can avoid this bottleneck by having constraining the Jacobian to be orthogonal so that we can calculate the determinant with only the diagonal elements. The determinant will be computed as follows:

$$y_{d+1:D} = x_{d+1:D} \odot \exp(s(x_{1:d})) + t(x_{1:d}) \quad (15)$$

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \text{diag}(\exp[s(x_{1:d})]) \end{bmatrix} \quad (16)$$

This Jacobian-Matrix has a determinant of $\exp\left[\sum_j s(x_{1:d})_j\right]$ which in deed does not require the computation of the Jacobian.

- **Question 3.4:**

Pixel values are mostly integers. This does not allow us to use it as continuous distribution. The probability mass function would spike at discrete values. The solution for this is dequantization. Dequantization takes the discrete value and multiplies it by a random sample from the normal distribution and we get a continuous distribution Rezende and Mohamed [2015].

3.2 Building a flow based model

- **Question 3.5:**

The input during training for the Normalizing Flow model are the training-set images. The output is log-posterior-probability for the image which should be maximized. During inference the input is normal-random distributed noise with the same dimensionality as the training images. Inference is achieved by inverting the Flow and is used to sample or generate new images. Therefore the output is an image with higher or lower similarity to the training set depending on the model Dinh et al. [2016].

- **Question 3.6:**

During training the model parameter get updates in order to generate better images. The steps during training are:

- Data batch input as z
- Dequantization of z
- Sigmoid normalization
- Apply mask
- Flow through the coupling layers
- Inside the Coupling layers generate s and t
- Computation of the negative-log-likelihood
- Backpropagation over the above and update Coupling layer parameters

During inference the model takes the following steps:

- Sampling random noise
- Apply mask
- Flow through the inverse model
- Inside the Coupling layers generate s and t
- Inverse Sigmoid normalization
- Output generated image

- **Question 3.7:**

The model was implemented according to the given template. For the computation of the log-prior we take the logarithm of the Standard-Normal probability-density-function. To generate an image we sample randomly from the Standard-Normal distribution. The figure 6 shows plots of the implemented model through-out the training of 40 epochs.

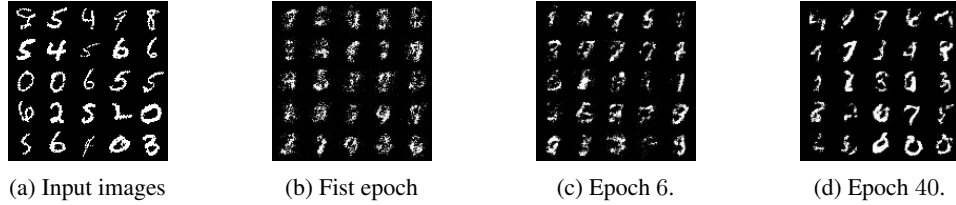


Figure 6: Sample images from the MNIST dataset and generated from our model during training.

• **Question 3.8:**

The model reaches a low in average bits per dimension of 1.83 over 40 epochs and is expected to reach lower values when trained over more epochs. It converges only slowly after 5 epochs where as the generated images keep improving. Adjusting the learning rate resulted in instable training and an oscillating loss.

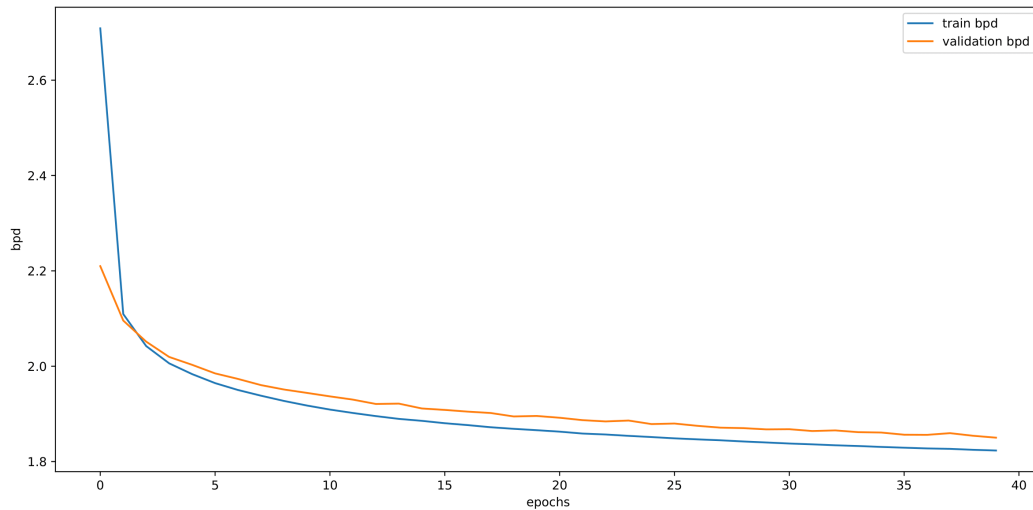


Figure 7: Caption

4 Conclusion

• **Question 4.1:**

Comparing the three presented generative models, it is important to first point out the differences. The most obvious difference is the architecture. The VAE has a rather simple architecture consisting of an encoder, a decoder and an external stochastic unit. The GAN consists of a Generator and a Discriminator, both play against a min-max game against each other. The Generator has a quite complex and deep network whereas the Discriminator is a fairly simple binary classifier. The NF model consists of one big unit, which leads the flow through various coupling layers, which again have deep neural networks to learn their parameters. This model is the most complex one.

In table 3 we notice a difference in image quality between the models. The VAE produces images with much higher variance, when instead of sampling from it the Bernoulli probabilities are plotted directly we get 'smooth' looking images. Images produced by GAN are very clear and realistic, similar to the data set. GAN produces the best results of these three models.

Table 3 also shows the parameter count of each model. VAE has the lowest count what collides with having the shortest training time. GAN required the most runtime and has three times more parameters than VAE. The NF model being the most complex one has 10 times the parameter count of the GAN. The resulting images are not accordingly impressive but far behind the quality of the GAN images. image quality

In table REF the differences in terms of image quality and parameter per model are presented.

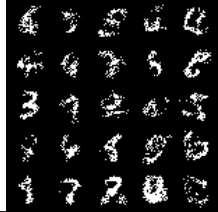


	VAE	GAN	Normalizing flow
Sample quality			
Parameter count	815824	2043537	27685120

Table 3: Comparison between the implemented models VAE, GAN and NF

YES
YOU
CAN

References

- Léon Bottou. Stochastic learning. In *Summer School on Machine Learning*, pages 146–168. Springer, 2003.
- John R Hershey and Peder A Olsen. Approximating the kullback leibler divergence between gaussian mixture models. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, volume 4, pages IV–317. IEEE, 2007.
- Alexander A Alemi, Ben Poole, Ian Fischer, Joshua V Dillon, Rif A Saurous, and Kevin Murphy. Fixing a broken elbo. *arXiv preprint arXiv:1711.00464*, 2017.
- Rui Shu, Hung H Bui, Jay Whang, and Stefano Ermon. Training variational autoencoders with buffered stochastic variational inference. *arXiv preprint arXiv:1902.10294*, 2019.
- Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- Rianne van den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. *arXiv preprint arXiv:1803.05649*, 2018.
- Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.
- Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.