Autoencoder and Normaling Flow

Sagar Shrestha
Oregon State University
Oregon, USA
shressag@oregonstate.edu

Sriranga Chaitanya Nandam
Oregon State University
Oregon, USA
nandams@oregonstate.edu

June 12, 2021

Abstract

In this project, we propose ”flow autoencoder” which is a deep generative model that combines an autoencoder with normalizing flow. The normalizing flow maps an arbitrary prior to the aggregated posterior of the latent vector of the autoencoder. The flow and the decoder of the autoencoder, together, define a generative model from which an independent sample can be generated with a single forward pass. We show in our experiments that the resulting generative model performs better than existing alternatives in many downstream applications where pixel-wise loss is important.

1 Introduction

One of the important tasks in unsupervised learning is to learn to represent a data distribution such that one can generate samples and evaluate the likelihood following that distribution. Deep generative models aim at accomplishing this task using a machine learning model. Past few years have seen a plethora of deep generative modeling techniques that try to generate realistic data samples using different neural architectures and optimization criteria. For example, normalizing flow [4, 5, 8] learns a bijective mapping between a prior and data distribution and are trained to maximize the exact likelihood. Variational autoencoder (VAE) [9] are trained to maximize a lower bound of data likelihood. Aside from likelihood based models, other distribution matching criteria may be employed. For example, generative adversarial networks (GANs) [7] employ a discriminator network to distinguish between model and data distribution and optimize over the discriminator output. It does not allow for a direct evaluation of data likelihood but generates very realistic looking data samples. Similarly, Generative moment matching network (GMMN) [10] employ maximum mean discrepancy objective to match all moments of the model distribution with that of the data distribution.

Qualitative evaluation of samples have played as important a role as quantitave evaluation in the development of these generative models. For natural images, realistic looking images are generally preferred and optimization criterion of GANs are naturally suited for such tasks. But for many engineering applications, realistic looking data samples may not be the best choice. Take for example, a wireless communication task, radio map reconstruction. Fig. 1 shows the a wireless signal strength over a geographical region represented as an image. This is not a natural image but representing it as an image makes some deep learning research to be directly applicable. For example, completing the radio map from limited observation can be thought of as an image inpainting task. However, in this task we do not care about how realistic the reconstruction is compared to the ground truth, but how close the approximation at each pixel (location) is to the true signal strength. Therefore, although incorporating adversarial loss (4th image Fig.1) produces realistic looking image, it actually incurs higher reconstruction loss

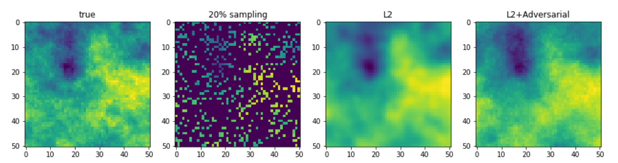

Figure 1: Example application in radio map reconstruction. From left to right, ground truth, sampled overvation, L-2 reconstruction, L-2 + adversarial reconstruction.

compared to the blurry reconstruction produced by minimizing L-2 loss alone (3rd image Fig. 1).

While autoencoder architectures have proved to be useful in tasks such as denoising, inpainting, etc, the need to learn a separate model for separate task or separate sensing model but involving the same data is prohibitively expensive. A common deep generative model, on the other hand, can be used as a deep prior for all the tasks involving the same data. This motivates us to combine the favorable traits of autoencoder and normalizing flow to build a generative model which is particularly useful in the aforementioned applications.

In this project, we propose to combine an autoencoder and a normalizing flow model learnt on the code space of the trained autoencoder in order to map the aggregated posterior of the latent representation to a simple prior. The decoder of the autoencoder and normalizing flow, together, define a deep generative model, which we refer to as *flow autoencoder* that maps the given prior to the data distribution. The resulting model ensures that sampling from the prior results in meaningful samples from the data distribution. Flow autoencoder admits several advantages in downstream tasks over related models which are demonstrated in the experimental section.

2 Proposed method

Flow autoencoder consists of an autoencoder and a normalizing flow model. Training of flow autoencoder is carried out via SGD in two stages. First an autoencoder is trained with the usual reconstruction loss. Second, the parameters of autoencoder are frozen and a normalizing flow is trained on the code space of the trained autoencoder. During sampling, the flow model takes a random sample from the prior to the code space of the autoencoder and the decoder maps the code to a meaningful data sample.

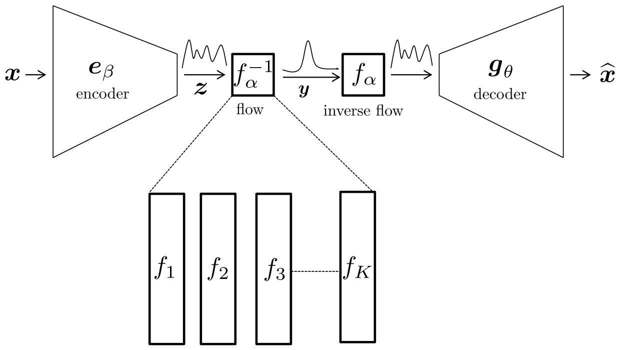

Figure 2: Architecture of flow autoencoder. The latent code $\boldsymbol{z}$ produced by the encoder of autoencoder is mapped to the given prior by the flow model. The inverse of the map can exactly recover the latent code. The composition of the flow and decoder defines a generative model that can generate data samples given a sample from the prior $p(\boldsymbol{y})$.

Fig. 2 illustrates the proposed model. The following two subsections gradually develops the flow autoencoder, and Sec. 2.3 describes the training and sampling procedure.

2.1 Autoencoder

Let $\boldsymbol{x} \in \mathbb{R}^N$ denote the input data and $\boldsymbol{z} \in \mathbb{R}^D$ the corresponding latent representation. We can think of an autoencoder as a feedforward network for which the target is the same as its input $\boldsymbol{x}$. Given a latent code vector $\boldsymbol{z}$, the decoding function $g_{\boldsymbol{\theta}}(\cdot) : \mathbb{R}^D \to \mathbb{R}^N$ can be interpreted as providing the conditional distribution $p(\boldsymbol{x}|\boldsymbol{z})$. When we use the output units of the autoencoder to parameterize the mean of a Gaussian distribution, minimizing the negative log-likelihood corresponds to mean squared error criterion normally used to train the autoencoder. Similarly, we can generalize the notion of the encoding function $e_{\boldsymbol{\beta}}(\cdot)$ as the encoding conditional distribution $q(\boldsymbol{z}|\boldsymbol{x})$ [6].

Under this model, if we let $p_d(\boldsymbol{x})$ denote the underlying data distribution, then the encoding distribution defines an aggregated posterior distribution $q(\boldsymbol{z})$ over the latent variable $\boldsymbol{z}$ as follows:

$$q(\boldsymbol{z}) = \int_{\boldsymbol{x}} q(\boldsymbol{z}|\boldsymbol{x}) p_d(\boldsymbol{x}) d\boldsymbol{x} = \mathbb{E}_{p_d(\boldsymbol{x})}[q(\boldsymbol{z}|\boldsymbol{x})]$$

In the absence of any regularization, $q(\boldsymbol{z})$ is learnt implicitly and can follow any arbitrary distribution. If one were to use the decoding distribution $p(\boldsymbol{x}|\boldsymbol{z})$ as a generative model, one can use $x \sim p(\boldsymbol{x}|\boldsymbol{z})$, where $\boldsymbol{z} \sim q(\boldsymbol{z})$. However, without access to $q(\boldsymbol{z})$, it is not possible to generate meaningful samples from $p(\boldsymbol{x}|\boldsymbol{z})$ that are not in the training examples $\{\boldsymbol{x}_i\}_{i=1}^m$.

In many latent variable models [9], a simple prior $p(\boldsymbol{z})$ is assumed over the latent variable (e.g. standard normal) and the training criterion consists of an additional regularization term that tries to minimize the cross entropy between $q(\boldsymbol{z})$ and $p(\boldsymbol{z})$, or equivalently KL divergence between $q(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{z})$. This allows for the aggregated posterior $q(\boldsymbol{z})$ to be close to $p(\boldsymbol{z})$. As such, samples from $p(\boldsymbol{z})$ can approximate the samples from $q(\boldsymbol{z})$.

2.2 Normalizing Flows

One way to sample from $q(\boldsymbol{z})$ is to learn a mapping that takes us from a different variable $\boldsymbol{y}$ with a simple (easy to evaluate density and sample from) distribution $p(\boldsymbol{y})$ to $q(\boldsymbol{z})$. To be specific, we wish to learn a function $f_\alpha(\cdot) : \mathbb{R}^D \to \mathbb{R}^D$ such that if $\boldsymbol{y}$ follows $p(\boldsymbol{y})$ then $\boldsymbol{z} = f_{\boldsymbol{\alpha}}(\boldsymbol{y})$ follows $q(\boldsymbol{z})$. If the mapping is smooth and bijective (i.e. the inverse mapping exists and is represented by $g_{\boldsymbol{\alpha}}(\cdot) = f_{\boldsymbol{\alpha}}^{-1}(\cdot)$), then one can use the change of variables formula to represent $q(\boldsymbol{z})$ as follows:

$$\begin{aligned} q(\boldsymbol{z}) &= p(\boldsymbol{y}) \left| \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}} \right| \\ &= p(g_{\boldsymbol{\alpha}}(\boldsymbol{z})) \left| \frac{\partial g_\alpha(\boldsymbol{z})}{\partial \boldsymbol{z}} \right| \end{aligned} \tag{1}$$

(1) is the basis for a number of recent works on normalizing flow [4,5,8] (in the literature on normalizing flows, $f_{\boldsymbol{\alpha}}$ is defined from the input data $\boldsymbol{x}$ to a prior of the same dimension). We hope to learn the parameters $\boldsymbol{\alpha}$ of the model in such a way that the log likelihood of the data is maximized, i.e.

$$\begin{aligned} &\max_{\boldsymbol{\alpha}} \quad \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}[\log(q(\boldsymbol{z}))] \\ &= \max_{\boldsymbol{\alpha}} \quad \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}[\log(g_{\boldsymbol{\alpha}}(\boldsymbol{z}))] + \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\left[\log\left|\frac{\partial g_\alpha(\boldsymbol{z})}{\partial \boldsymbol{z}}\right|\right] \end{aligned} \tag{2}$$

It is important to see that (2) represents the exact likelihood as opposed to approximations of the likelihood using lower bounds (as in [9]). The sample estimate of (2) can be obtained by using samples of $\{\boldsymbol{z}_i\}_{i=1}^m$ as follows:

$$\begin{aligned} &\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m} \log(q(\boldsymbol{z}_i)) \\ &= \max_{\boldsymbol{\alpha}} \sum_{i=1}^{m} \left[\log p(g_{\boldsymbol{\alpha}}(\boldsymbol{z}_i)) + \log\left|\frac{\partial g_\alpha(\boldsymbol{z}_i)}{\partial \boldsymbol{z}}\right|\right] \end{aligned} \tag{3}$$

(3) is easy to evaluate and run gradient descent on. Note that the training samples $\{\boldsymbol{z}_i\}_{i=1}^m$ can be obtained from $\{\boldsymbol{x}_i\}_{i=1}^m$ and the encoding function $e_{\boldsymbol{\beta}}(\cdot)$.

The requirements on $f_{\boldsymbol{\alpha}}(\cdot)$ is, however, quite stringent and widely used neural network architectures can not be used. Specifically, the neural network needs to be smooth, invertible and the second term on the right hand side of (3) (i.e. the log determinant of the jacobian) should be easy to evaluate. Popular fully connected or convolutional linear layers do not satisfy these constraints. As such special architecture designs are required to ensure feasibility of the network. Popular architectures in the literature include the composition of affine coupling layers [5], 1×1 convolutional layers [8], act-norm layers, etc. In this work, we employ a composition of multiple affine coupling layers which are described in Sec. 4

2.3 Training, Sampling and Density Evaluation

The training procedure of flow autoencoder proceeds in the following way:

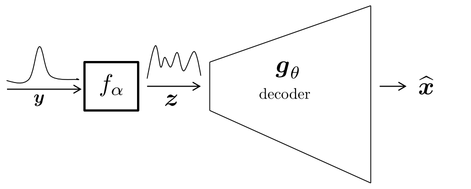

Figure 3: Sampling from flow autoencoder

1. Train the autoencoder $g_{\boldsymbol{\theta}}(e_{\boldsymbol{\beta}}(\cdot))$ as follows:

$$\max_{\boldsymbol{\beta},\boldsymbol{\theta}} \sum_{i=1}^{m} \|\boldsymbol{x} - g_{\boldsymbol{\theta}}(e_{\boldsymbol{\beta}}(\boldsymbol{x}_i))\|_2^2 \tag{4}$$

2. Fix $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ generate $\{\boldsymbol{z}_i\}_{i=1}^m$ using $\boldsymbol{z}_i = e_{\boldsymbol{\beta}}(\boldsymbol{x}_i)$, and obtain $\boldsymbol{\alpha}$ using (3).

Note that any out-of-the-box gradient based optimizer can be used for running (4) and (3).

Sampling from flow autoencoder is also straightforward. Using the trained parameters one can sample from the prior $\boldsymbol{y} \sim p(\boldsymbol{y})$ and evaluate $\boldsymbol{x}_{\text{sample}} = g_{\boldsymbol{\theta}}(f_{\boldsymbol{\alpha}}(\boldsymbol{y}))$. This is illustrated in Fig. 3

Flow autoencoder does not directly allow for density evaluation. While density evaluation in flow models alone is straightforward, only the density of the latent code can be evaluated accurately using the flow model of flow autoencoder, i.e. flow model allows for evaluation of $q(\boldsymbol{z})$ but not of $p(\boldsymbol{x})$.

3 Related Works

It is useful to compare the proposed method to related work in the literature. For comparison we consider the following generative models which have similarities with the flow autoencoder.

3.1 Generative Moment Matching Network (GMMN)

Perhaps the closest architecture in terms of training and sampling is GMMN [10] which employ maximum mean discrepancy (MMD) loss to measure the discrepancy between two distributions. Like flow autoencoder, GMMN learns a mapping from a simple prior to the distribution represented by the code space of a trained autoencoder. However, GMMN is not invertible and therefore does not provide a recognition model.

3.2 Variational Autoencoder (VAE)

VAE is a latent variable model that minimizes an upper bound on the data likelihood:

$$\begin{aligned}\min_{q(\boldsymbol{z}|\boldsymbol{x}),\boldsymbol{\theta}} & \mathbb{E}_{\boldsymbol{x}}[\mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}[-log(p(\boldsymbol{x}|\boldsymbol{z}))]] \\ & + \mathbb{E}_{\boldsymbol{x}}[\mathrm{KL}(q(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})].\end{aligned}$$

The first term corresponds to the reconstruction loss and the second term can be decomposed into:

$$\begin{aligned}&\mathbb{E}_{\boldsymbol{x}}[KL(q(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})] \\ &= -\mathbb{E}_{\boldsymbol{x}}[\mathrm{H}(q(\boldsymbol{z}|\boldsymbol{x}))] + \mathbb{E}_{q(\boldsymbol{z})}[-\log p(\boldsymbol{z})] \\ &= -\mathrm{Entropy} + \mathrm{CrossEntropy}(q(\boldsymbol{z}), p(\boldsymbol{z})).\end{aligned}$$

The KL term encourages the aggregated posterior to be close to the prior. Essentially, the loss function is a tradeoff between reconstruction accuracy and regularization of the posterior distribution. In flow autoencoder, instead of regularizing the posterior distribution, it learns an independent mapping function from the posterior to the given prior.

3.3 Adversarial Autoencoder (AAE)

Adversarial Autoencoder [11] is quite similar to VAE. Like VAE, it regularizes the aggregated posterior to be close to the given prior. However, unlike VAE, AAE uses adversarial loss instead of the KL loss. The training of AAE alternates between training the autoencoder with usual reconstruction loss and training the encoder and discriminator with adversarial loss. To clarify, the encoder acts as a generator of GAN which produces latent codes (fake samples), and sample from the prior $p(\boldsymbol{z})$ acts as the real prior. A discriminator is trained to distinguish between latent code generated by the encoder of the autoencoder and the sample from the prior. Encoder parameters are updated against the discriminator output. This results in the aggregated posterior being close to the prior.

4 Experiments

In order to demonstrate proof of concept of the proposed method, we run experiments on MNIST digits dataset and compare our result with related methods. Specifically, we train a variational autoencoder(VAE) and an adversarial autoencoder (AAE) as these methods provide recognition models so that the latent representation can be obtained for test data samples and visualized (see Fig. 6).

4.1 Model Description and training

4.1.1 Autoencoder

In order to make a fair comparison we use the same encoder and decoder for all the models being compared, i.e. flow autoencoder, VAE, and AAE. The autoencoder employed is a fully convolutional neural network with architecture detailed in table 1. We chose the latent dimension to be 2 so that the empirical distribution of the code space can be easily visualized. For the discriminator of AAE, two hidden layers were employed with dimension of 512 and 256 respectively. The output layer was a sigmoid unit.

All models were trained with Adam optimizer with initial learning rate of 0.001 for 100 epochs on a batch size of 32.

4.1.2 Normalizing Flow

We employ a composition of 4 affine transform layers to construct our flow model. Affine transform layers can be described by the following set of equations:

$$\begin{aligned} \boldsymbol{y}_{1:d} &= \boldsymbol{z}_{1:d} \\ \boldsymbol{y}_{d+1:D} &= \boldsymbol{z}_{d+1:D} \odot \exp(\boldsymbol{z}_{1:d}s) + \boldsymbol{z}_{1:d}t. \end{aligned} \tag{5}$$

In (5), input $\boldsymbol{z}$ is partitioned into $\boldsymbol{z}_{1:d}$, which represents the first d elements of $\boldsymbol{z}$, and $\boldsymbol{z}_{d+1:D}$. The first partition is copied to the first d elements of $\boldsymbol{y}_{1:d}$ without any transformation. The remaining partition undergoes elementwise scaling and translation.

Layer	#filters	kernel	stride	padding
Encoder				
conv.	32	4	2	1
conv.	64	4	2	1
conv.	128	3	2	1
conv.	256	2	2	0
conv.	2	2	2	0
Decoder				
deconv.	256	2	1	0
deconv.	128	4	2	1
deconv.	64	4	1	0
deconv.	32	4	2	1
deconv.	1	4	2	1
conv.	1	3	1	1

Table 1: Encoder and Decoder architectures. Each layer, except the output layers of encoder and decoder, was followed by batch normalization and leaky ReLU activation.

The scaling and translation parameters are learnt using any arbitrarily complex neural architecture. In our implementation we employ a multilayer perceptron with two hidden layers and 256 units each with ReLU activations except for the last layer which is kept without activation. Fig. 5 [left] illustrates (5). The inverse of affine transform layer is straightforward to obtain and is depicted in Fig. 4 [right]. It can be represented by the following set of equations:

$$\begin{aligned} \boldsymbol{z}_{1:d} &= \boldsymbol{y}_{1:d} \\ \boldsymbol{z}_{d+1:D} &= (\boldsymbol{y}_{d+1:D} - \boldsymbol{y}_{1:d}) \odot 1/\exp(\boldsymbol{y}_{1:d}s) + \boldsymbol{z}_{1:d}t. \end{aligned} \tag{6}$$

The jacobian of this transformation is:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^{\top}} = \begin{bmatrix} d & \mathbf{0} \\ \frac{\partial \boldsymbol{y}_{d+1:D}}{\partial \boldsymbol{x}_{1:d}} & \operatorname{diag}(\exp(\boldsymbol{x}_{1:d}s)) \end{bmatrix} \tag{7}$$

Since the jacobian is triangular, the determinant is just the product of the diagonal entries, $\exp(\boldsymbol{x}_{1:d}s)$.

The flow model is trained with a batch size of 128 using Adam optimizer with an initial learning rate of 0.01 for 100 epochs.

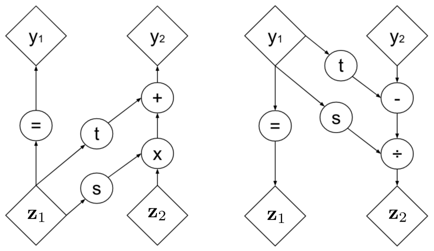

Figure 4: RealNVP affine transform layers. [Left] forward. [Right] reverse flow. $\boldsymbol{y}_1$ represents $\boldsymbol{y}_{1:d}$ and $\boldsymbol{y}_2$, $\boldsymbol{y}_{d+1:D}$. Similar notation is used for $\boldsymbol{z}$

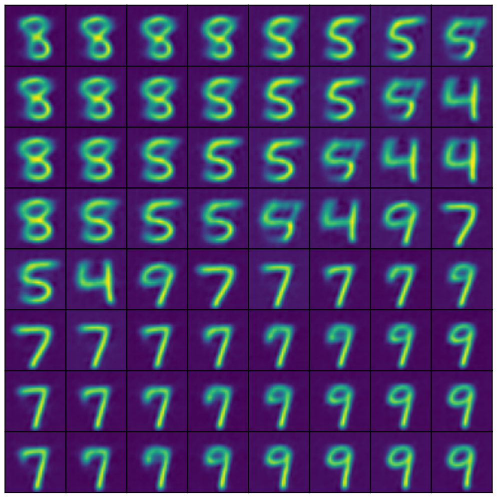

Figure 5: Data manifold of flow autoencoder.

4.2 Results

4.2.1 Manifold Visualization

Fig. 5 shows the learned data manifold of the flow autoencoder. Data points were collected by smooth interpolation between four samples at the endpoints of a square in the 2-dimensional code space. Note that the code dimension of 2 results in relatively aggressive compression and the samples drawn may not be of acceptable quality in this setting.

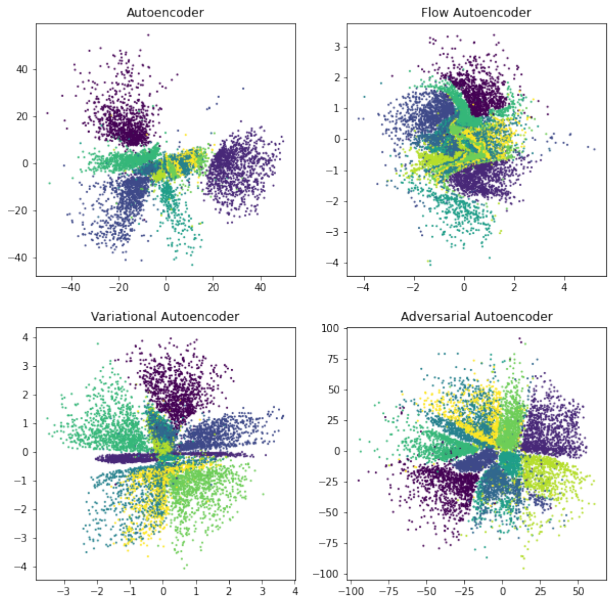

Figure 6: Visualization of code space (dim=2) of MNIST (for test set). Ideally they should be close to standard normal distribution. [Top Left] code space of autoencoder is not under our control, so it is irregular and have fractures/discontinuities. [Top Right] combination of flow and autoencoder produces code space closest to standard normal. [Bottom Left] VAE is close to gaussian but contains some discontinuities. [Bottom Right] Shape is very close to gaussian but has large variance and greater fractures than flow autoencoder.

4.2.2 Code space visualization

Fig. 6 shows the learned code space manifold of the flow autoencoder and other methods. Different colors represent different labels of the MNIST digits. For flow autoencoder these points were generated by passing the 10,000 MNIST test points $\{\boldsymbol{x}\}_{i=1}^{10000}$ to the composition of encoder and the flow, i.e. $e_{\boldsymbol{\beta}} \circ \boldsymbol{f}_{\boldsymbol{\alpha}}$. For AAE and autoencoder, they were generated by passing the test points though the encoder. For VAE, mean vectors generated through the encoder were taken.

The code space of autoencoder is supposed to represent $q(\boldsymbol{z})$ without any regularization. One can see that the code space is quite irregular with gaps/fracture between some clusters. This can result in non-smooth transitions. Any point in code space not belonging to any of the clusters can result in unnatural samples. However flow autoencoder successfully transforms the latent space of autoencoder into the given prior (standard normal in this case). Note that there are no fractures and some sharp transitions between the labels. This is important for the generated samples to look like a digit and not superposition of two digits. VAE and and AAE on the other hand have some gaps and are not as close to standard normal as flow autoencoder. In AAE, one can see that the variance of the data is quite large compared to standard normal.

4.2.3 Log Density and Reconstruction accuracy

Flow autoencoder does not provide a mechanism for likelihood evaluation from its architecture. While flow models alone have exact likelihood evaluation method by just evaluating the cost function, it will only correspond to the likelihood of the latent factor of autoencoder, i.e. $q(\boldsymbol{z})$. In order to approximate the data likelihood, we follow the method used in [1]. The idea is to fit a kernel density estimator (KDE) on the samples generated by the generative model and evaluate the average likelihood of the test point on the KDE. We use gaussian kernels with 10,000 samples from each generative model and 1000 test points from the MNIST test set. The likelihood estimate is tabulated in Fig. 2. Note that low log-likelihood of the AAE is probably due to the overestimation of variance of the code space distribution.

Model	Log-likelihood	Reconstruction Error
Flow Autoencoder	**-256.75**	**0.292**
VAE	-266.14	0.453
AAE	-324.68	0.322
Autoencoder	-322.67	0.292

Table 2: Average log-likelihood score of the samples from the trained models (higher is better). Likelihood was computed by fitting gaussian kernel density estimator on 10000 generated samples and tested using 1000 test set images. The width of the gaussian was selected using a validation set. Reconstruction loss is the $\|\boldsymbol{x} - \widehat{\boldsymbol{x}}\|_2$ on the test set.

Since objectives in VAE and AAE consists of regularization term that introduces a tradeoff between reconstruction error and latent space regularization, they, generally, have higher reconstruction error compared to autoencoder. But since, flow autoencoder does not introduce any changes to the autoencoder loss function and merely learns a mapping from the trained latent space to a given prior, it has the same reconstruction error as the autoencoder. In other words, the invertibility of the flow model ensures that the code space respresentation is recovered exactly.

4.2.4 Downstream task: Compressed Sensing

Generative models have a wide range of applications. The primary motivation of this project is to improve performance in deep prior related tasks where the L2-error is of primary importance. One such task is compressed sensing using generative models [2]. In this task, generative models are used as prior for the popular compressed sensing tasks explored widely in the literature [3].

Consider taking M measurements $\boldsymbol{m} \in \mathbb{R}^M$ of N signals $\boldsymbol{x} \in \mathbb{R}^N$ by using a linear transformation $\boldsymbol{A} \in \mathbb{R}^{M \times N}$ with random gaussian entries. The task is to recover $\boldsymbol{x}$ from $\boldsymbol{m}$ given the sensing model $\boldsymbol{A}$. Using deep generative model $\boldsymbol{G}(\cdot) : \mathbb{R}^D \to \mathbb{R}^N$ learnt on the data distribution $p_d(\boldsymbol{x})$, we aim to minimize the

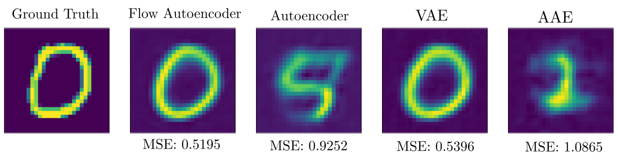

Figure 7: Downstream Task: Compressed Sensing. Number of measurements = 15% with gaussian measurement matrix. Reported MSE = $\|\boldsymbol{x} - \widehat{\boldsymbol{x}}\|_2$. Flow Autoencoder has the best reconstruction result. Autoencoder alone is produces poor reconstruction as the latent space is not smooth and searching over it is error prone.

following optimization criterion:

$$\underset{\boldsymbol{z}}{\text{minimize}} \quad \|\boldsymbol{m} - \boldsymbol{A}\boldsymbol{G}(\boldsymbol{z})\|_2 \tag{8}$$

By fixing the parameters $\boldsymbol{\theta}$, one can run gradient descent on the objective in (8) to find $\boldsymbol{z}^\star$ such that $\boldsymbol{G}(\boldsymbol{z}^\star)$ best explains the observation $\boldsymbol{m}$.

Fig. 7 shows the reconstruction using different models. $\boldsymbol{z}$ was initialized with $\mathbf{0}$. Since the prior for all generative models are standard normal, the center of the gaussian is a good starting point as it corresponds to the region with highest density. We can see that the recovery from flow autoencoder admits the least reconstruction error whereas using autoencoder only is quite error prone and the output image cannot be labelled to a particular digit with high confidence.

5 Conclusion and Future Work

In this project, we proposed a deep generative model by combining useful traits of autoencoder and normalizing flow and provided a proof of concept with experiments. The resulting generative model admitted high reconstruction accuracy in downstream compressed sensing task and provided competitive or better log likelihood scores and latent space regularization. In the future, we aim to explore theoretical foundation and other possible benefits for such model and also apply it to more complicated datasets.

References

[1] Y. Bengio, G. Mesnil, Y. Dauphin, and S. Rifai. Better mixing via deep representations. In *International conference on machine learning*, pages 552–560. PMLR, 2013.

[2] A. Bora, A. Jalal, E. Price, and A. G. Dimakis. Compressed sensing using generative models. In *International Conference on Machine Learning*, pages 537–546. PMLR, 2017.

[3] E. J. Candès, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on information theory*, 52(2):489–509, 2006.

[4] L. Dinh, D. Krueger, and Y. Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[5] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

[6] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

[8] D. P. Kingma and P. Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *arXiv preprint arXiv:1807.03039*, 2018.

[9] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[10] Y. Li, K. Swersky, and R. Zemel. Generative moment matching networks. In *International Conference on Machine Learning*, pages 1718–1727. PMLR, 2015.

[11] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644*, 2015.