Advanced Deep Learning **VU Anh Thu** : 21322736
2024-2025 **LE Thi Minh Nguyet** : 21401438

Lab report - Normalizing Flows

Abstract

In this report, we implement a flow-based generative model called Glow, as proposed by D. P. Kingma and P. Dhariwal in [1]. We train the model on the CIFAR10 and MNIST datasets and show that, while the model is able to learn key patterns in the data, the generated images exhibit several issues. These include blurriness and a lack of fine details for CIFAR10, as well as unrealistic numbers in some generated images for MNIST. We then perform a linear interpolation in the latent space to prove that the model has indeed constructed an organized latent space, as indicated by smooth transitions between consecutive interpolated images. Finally, a return to the original paper suggests that the issues observed in the generative images may stem from insufficient depth in the model's architecture.

1 Background: Flow-based generative models

Glow is a so-called flow-based generative model, developed based on the concept of normalizing flows. A normalizing flow $\mathbf{g}$ is a sequence of invertible transformations: $\mathbf{g} = \mathbf{g}_M \circ \cdots \circ \mathbf{g}_2 \circ \mathbf{g}_1$. The idea of flow-based generative models is to apply a sequence of invertible functions to latent variables, gradually transforming their probability distribution into a more complex distribution over the input data. Therefore, in these models, the generative process is designed as

$$\begin{aligned}\mathbf{z} &\sim p_{\boldsymbol{\theta}}(\mathbf{z}) \\ \mathbf{x} &= \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{z}) = \mathbf{g}_{\boldsymbol{\theta},M} \circ \cdots \circ \mathbf{g}_{\boldsymbol{\theta},2} \circ \mathbf{g}_{\boldsymbol{\theta},1}(\mathbf{z})\end{aligned}$$

where $p_{\boldsymbol{\theta}}(\mathbf{z})$ is a (typically simple) tractable probability density over the latent variable $\mathbf{z}$ and $\{g_{\boldsymbol{\theta},i}\}_1^M$ is a sequence of M invertible functions. The relationship between $\mathbf{x}$ and $\mathbf{z}$ can be written as

$$\mathbf{z} = \mathbf{h}_0 \overset{\mathbf{g}_{\boldsymbol{\theta},1}}{\longleftrightarrow} \mathbf{h}_1 \overset{\mathbf{g}_{\boldsymbol{\theta},2}}{\longleftrightarrow} \mathbf{h}_2 \cdots \overset{\mathbf{g}_{\boldsymbol{\theta},M}}{\longleftrightarrow} \mathbf{h}_M = \mathbf{x}$$

By change of variables, we can express the probability density of $\mathbf{x}$ as a function of $\mathbf{z}$ as follows

$$\begin{aligned}\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \log \left|\det \left(d\mathbf{z}/d\mathbf{x}\right)\right| \\ &= \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \sum_{i=1}^{M} \log \left|\det \left(d\mathbf{h}_{i-1}/d\mathbf{h}_i\right)\right| \end{aligned} \tag{1}$$

Given a training set $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$, the training objective of flow-based models is then to minimize the average negative log-likelihood (NLL) of the training data. As we can see,

one advantage of these flow-based models is that they allow the exact calculation of the log-likelihood of the data, $\log p_{\boldsymbol{\theta}}(\mathbf{x})$, which mean there is no loss of information. However, the price to pay is a lower expressiveness because the functions $\mathbf{g}_{\boldsymbol{\theta},i}$ must be invertible and have a Jacobian whose determinant can be calculated.

2 Glow architecture

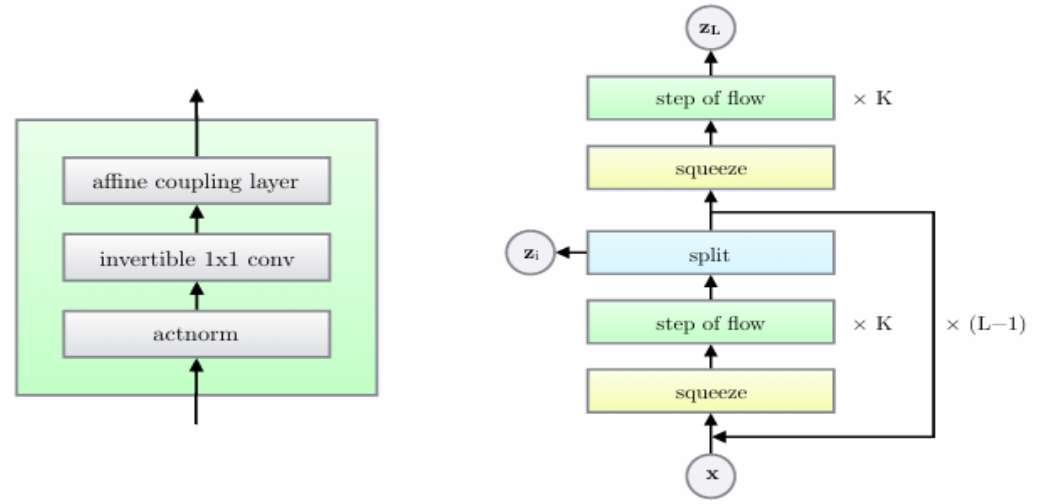


Figure 1: Glow architecture with the components of one flow step on the left and the combined multi-scale structure on the right. Image from [1].

The architecture of Glow is illustrated in Figure 1. As shown, Glow follows a multi-scale structure consisting of a series of steps of flow, with each step separated by *squeeze/split* operations. Each flow step (Section 2.1) comprises an *actnorm* layer, followed by an *invertible* 1×1 *convolution*, and then an *affine coupling* layer. The Glow implementation is detailed in the following subsections.

2.1 One step of flow

In this subsection, let us denote by $\mathbf{x}$ and $\mathbf{y}$ the input and output of each layer, respectively. Both $\mathbf{x}$ and $\mathbf{y}$ are tensors of shape $h \times w \times c$, with spatial dimensions (h, w) and a channel dimension c. We additionally denote (i, j) the spatial indices of the tensors $\mathbf{x}$ and $\mathbf{y}$.

- *actnorm*: Normalization layer similar to batch norm, except that the mean and standard deviation statistics are trainable parameters rather than estimated from the data. These parameters are initialized such that the post-actnorm activations per channel have zero mean and unit variance given an initial minibatch of data. This is particularly useful

here because the model sometimes has to be trained with very small batch sizes due to memory requirements.

Forward: $\forall i, j$: $\mathbf{y}_{i,j} = (\mathbf{x}_{i,j} - \mu)/\sigma$

Log-det: $\log\det = -h \times w \times \sum \log(|\sigma|)$

- *Invertible* 1×1 *convolution*: This layer is indeed a convolution of kernel 1×1, which corresponds to a linear transformation $\mathbf{W}$ of shape $c \times c$. The same transformation will be applied to each $\mathbf{x}_{i,j}$.

  Forward: $\forall i, j$: $\mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$

  Log-det: $\log\det = h \times w \times \sum \log(|\det(\mathbf{W})|)$

  As noted in the original paper, we use LU decomposition of $\mathbf{W}$ to reduce the cost of computing $det(\mathbf{W})$. This matrix is initialized as a random orthogonal (rotation) matrix. This rotation matrix $\mathbf{W}$ helps the *invertible* 1×1 *convolution* act as a generalization of channel permutations before the *affine coupling* layer, ensuring that after sufficient steps of flow, all channels are affected.

- *Affine coupling*: This layer splits the input data into two halves along the channel dimension, then use the second half to estimate the parameters of a transformation before applying it to the first half.

  Forward:
  $\mathbf{x}_a, \mathbf{x}_b = \text{split}(\mathbf{x})$
  $(\log\sigma, \mu) = \texttt{NN}(\mathbf{x}_b)$
  $\mathbf{y}_a = \sigma \odot \mathbf{x}_a + \mu$
  $\mathbf{y} = \text{concat}(\mathbf{y}_a, \mathbf{x}_b)$

  Log-det: $\log\det = \sum \log(|\sigma|)$

  We would like to have two remarks here.

  - In our implementation, `NN()` is a shallow convolutional network consisting of three convolutions (3×3, 1×1, then 3×3), with a ReLU activation and 512 channels for hidden layers. Following the original paper, we initialize the last convolution of each `NN()` with zeros, such that each *affine coupling* layer initially performs an identity function. This is proven to help training deep networks.
  - Following the original implementation, we apply the `sigmoid` function instead of `exp` to $\log\sigma$ to compute σ in order to avoid numerical instability.

2.2 Squeeze

This layer simply splits the feature maps in $2 \times 2 \times c$ blocks and flatten each block to shape $1 \times 1 \times 4c$. This can be considered as a way to do the Pooling.

2.3 Split

This layer cuts the input in half along the channel dimension: the first half will no longer be modified and become the latent variable at the current scale, $\mathbf{z}_i$, while the second half, $\mathbf{x}$, continues to be passed to the next scale. This makes the model slightly more computationally lightweight. In order to estimate $\log p(\mathbf{x})$ in Equation 1, we need to compute the prior $p(\mathbf{z})$ on all the latent variables $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_L)$. Here, we do not use the same prior $\mathcal{N}(0, 1)$ for all $\mathbf{z}_i$; instead, we implement a **learnable prior** $p(\mathbf{z}_i)$ assumed to be Gaussian whose mean μ_i and standard deviation σ_i are learned. More specifically, we add a convolutional layer on top of $\mathbf{x}$ to estimate the μ_i and σ_i parameters of the prior $p(\mathbf{z}_i) = \mathcal{N}(\mu_i, \sigma_i)$. This convolution has the same design as the last convolution of the `NN()` networks in the *affine coupling* layers, meaning that it has a kernel of size 3×3 and is initialized with zeros.

Finally, we also add a convolutional layer to learn the top prior $p(\mathbf{z}_L)$. This layer follows the same design as the convolutions used to learn the prior $p(\mathbf{z}_i)$ in the *split* layers and takes a zero tensor as input.

3 Data preprocessing

In Equation 1, we consider the input data $\mathbf{x}$ as a continuous variable, and we are training on images, which typically have discrete values. Therefore, as suggested in [2], we have to *dequantize* the data by adding some small uniform noise to the pixel values. Moreover, in the original Glow implementation, they also introduce a `num_bits` parameter which allows for further controlling the quantization level of the input images.

4 Training hyperparameters

Our set of hyperparameters can be described as follows.

`L` :	The number of blocks in the model.
`K` :	The number of flow steps in one block.
`num_bits` :	Controls the quantization level of the input images.
`nb_epochs` :	The total number of training epochs.
`lr` :	The learning rate.
`batch_size` :	The size of each training batch.
`nb_warmup_epochs` :	The learning rate is initially zero and increases linearly to reach `lr` after `nb_warmup_epochs`.

The specific values of these hyperparameters used for training each dataset are detailed below.

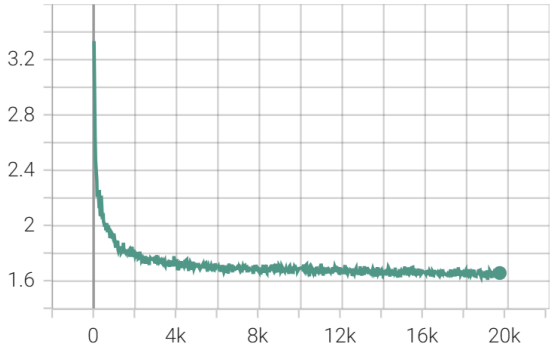
- CIFAR10:

  - Model architecture: We use `L` $= 3$ and `K` $= 16$, a smaller value of `K` compared to the original implementation due to limited GPU resources.
  - Data preprocessing: To compensate for the shallowness of the model, we train the model on 5-bit images (`num_bits` $= 5$), aiming to improve visual quality at the cost of a slight decrease in color fidelity.
  - Training: We train the model for 100 epochs, with 10 warm-up epochs, a learning rate of 10^{-3} after the warm-up phase, and a batch size of 256.

- MNIST: The hyperparameter setting is similar to that of CIFAR10, except that we do not have a warn-up phase for the learning rate, and we fix the learning rate at 10^{-4} for the entire training process.

5 Results and Analysis

(a) Generated images



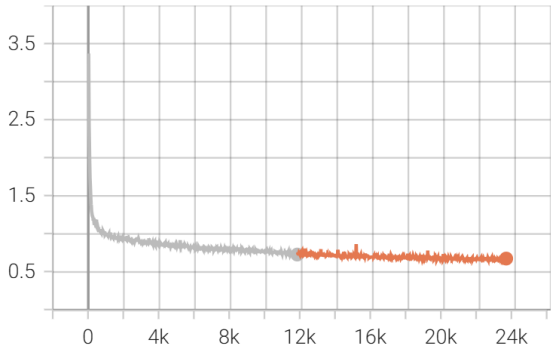
(b) Training loss with the x-axis representing number of iterations and the y-axis representing the negative log-likelihood in bits per dimension

Figure 2: Training results for CIFAR10

Using the hyperparameter settings presented in Section 4, we train the model on CIFAR10 and MNIST, and the results are shown in Figures 2 and 3.

As we can see, both losses decrease smoothly throughout the training process and very rapidly during the initial stages, indicating that the model quickly learns the basic patterns in the data. This observation reflects the model's ability to effectively learn from the training data. The generated images for both datasets further demonstrate the model's learning capability:

(a) Generated images



(b) Training loss with the x-axis representing number of iterations and the y-axis representing the negative log-likelihood in bits per dimension

Figure 3: Training results for MNIST

- CIFAR10: There is good diversity in generated images. In some images, primary shapes and structures resembling objects from the dataset can be observed. This suggests that the model has learned a reasonable representation of the data distribution. However, a noticeable issue is the blurriness and lack of fine detail in most of the generated images, making the objects difficult to define.
- MNIST: Compared to the CIFAR-10 images, the generated digits here are more recognizable. The essential shapes of the digits are generally preserved, and there is a good amount of variation in how the digits are written. However, some noticeable deformations and unrealistic patterns remain in certain cases.

It is worth noting that, despite being trained with `num_bit` $= 5$, the generated images do not match the quality of those produced by the original paper. This highlights the importance of model depth in enabling the model to learn long-range dependencies, as emphasized and illustrated in the original paper (Figure 4).

6 Extension - Interpolation

To test whether our model has successfully constructed a meaningful latent space, we perform a linear interpolation on the test set. The implementation is straightforward: Given two images x_1 and x_2 of the same class, we first pass the images through the network to get their latent representations z_1 and z_2. The latent vectors are then interpolated by calculating $(1 - \alpha)z_1 + \alpha z_2$ for different values of α, and then sent back through the network backwards to get the resulting interpolated images. The results can be found in Figure 5.

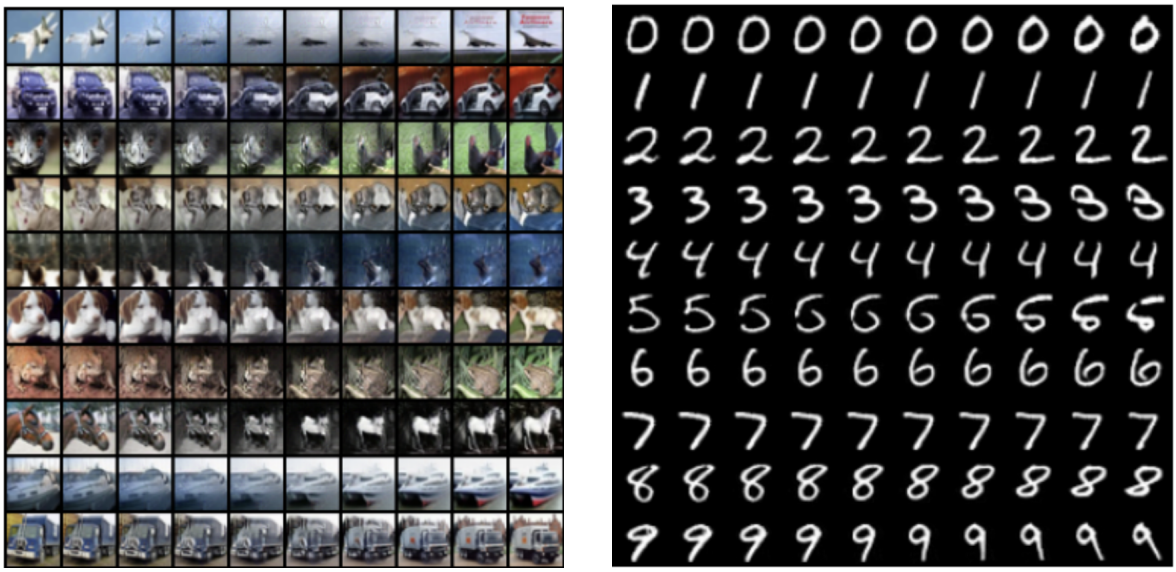
Figure 4: CelebA generated images for shallow model with $\texttt{L} = 4$ levels on the left and for deep model with $\texttt{L} = 6$ levels on the right. Image taken from [1].

As shown, relatively smooth transitions can be observed in most cases for both datasets. However, there are still some artifacts in some interpolated images, such as in the deer and cat classes of CIFAR10, or in class 5 of MNIST. These observations suggest that the model's latent space is well-organized and captures meaningful semantic relationships. While the smooth transitions indicate some level of organization and semantic understanding, the presence of artifacts and mode mixing highlights the model's limitations in fully capturing the complexity of the data.

References

[1] Diederik P. Kingma, Prafulla Dhariwal, *Glow: Generative Flow with Invertible* 1×1 *Convolutions*, arXiv:1807.03039v2, 2018.

[2] L.Theis, A. Oord, M. Bethge, *A note on the evaluation of generative models*, arXiv:1511.01844v3, 2016.



(a) CIFAR10 (b) MNIST

Figure 5: Latent space linear interpolation. For each row, the left and right-most images are real test images, while the images in between are interpolated images.