Practical 3. Deep Generative Models

University of Amsterdam – Deep Learning Course

Huishi Qiu
12955582
huishiqiu@gmail.com

December 23, 2020

1 Variational Auto Encoders

1.1 Latent Variable Models

Firstly we sample latent variable z_n from to the normal distribution $\mathcal{N}(0, I_D)$. Secondly we compute $f_\theta.(z_n)$ according to the previous latent variable z_n and neural network f_θ, i.e. generator in the circumstance of VAE. Finally, we sample every pixel m in x_n from Bernoulli distribution over $f_\theta(z_n)$, to determine whether it is 1 or 0.

1.2 Decoder: The Generative Part of the VAE

Monte Carlo estimation need to sample from the entire dataset. However, not all of the latent variable z are useful. In figure 2 of assignment of a 2-dimensional space, the blue contour of posterior $p(z|x)$ is much smaller than the red contour space of p(z), which indicates only a small number of latent variable z are related to input variable x. Besides, the higher is the dimensional space, the less overlapped space are two contour, which lead to even less efficiency.

1.3 KL Divergence

Example 1:$\{\mu_p = 0, \mu_q = 0.1, \sigma_p^2 = 1, \sigma_q^2 = 1\}$
In this situation, two distribution has the same variance and very close mean. which lead to very small KL-Divergence of $D_{KL}(q||p) = 5.0$

Example 2:$\{\mu_q = 0, \mu_p = 0, \sigma_p^2 = 1, \sigma_q^2 = 10\}$
This time the mean value of two distribution are both 0 while the variance is different, which lead to a very big KL-Divergence of $D_{KL}(q||p) = 8266.078$

1.4 The Encoder

$$logp(x_n) \geq \mathbb{E}_{q(z_n|x_n)}[logp(x_n|z_n)) - KL(q(Z|x_n)||p(Z))] \tag{1}$$

According to the definition of KL-divergence, the second KL term in equation 1 on the right hand side is always bigger than 0, thus $logp(x_n) \geq ELBO(q)$, i.e. ELBO(q) determine the minimum value of log$p(x_n)$. This is why it named as Excepted Lower Bound. When $q(z_n) = p(z_n|x_n)$, the equation is set up by equal.

Commonly directly optimize P_{xn} need to cost exponentially time to calculate. Same as the reason why gave up Monte Carlo method, we need better efficiency. Thus we choose to optimize over ELBO instead of $logp(x)$.

1.5

When the right hand side ELBO value is pushed up, firstly it indicates KL-divergence $KL(q(Z|x_n)||p(Z|x_n))$ becomes smaller, i.e. the posterior distribution q is better fitted with our prior distribution p. Secondly, $logp(x)$ increase, which is the term we want to maximize during the training.

1.6 Reconstruction and Regularization

$$\mathcal{L}_n^{recon} = -\mathbb{E}_{q\phi}(z|x_n)[logp\theta(x_n|z)] \tag{2}$$

Looking at equation2 and translate the formula in terms of words: given x, we encode the variable by hidden state z and reconstruct the expectation E of x. If the Expectation E is bigger enough, indicates hidden states z is a good representation for x, where enough data could be derived to represent. This is why it called reconstruct term.

$$\mathcal{L}_n^{reg} = D_{KL}(q\phi(Z|x_n)||p_\theta(Z))] \tag{3}$$

Equation 3 of KL-Divergence is trying to minimize the distance between posterior distribution q and the prior distribution of z. In other words it works a regulator to minimize the difference between two distributions, thus called regularization term.

1.7 Object Function

1. Passing x to encoder q_ϕ, generate μ_q and Σ_q.
2. Sample z_n from $q_\phi(z_n|x_n)$, which is $\mathcal{N}(\mu_\phi(x_n), diag(\Sigma_\phi(x_n)))$
3. Compute regression loss $\mathcal{L}_n^{reg} = D_{KL}(q\phi(Z|x_n)||p_\theta(Z))] = \frac{\sigma_q^2+\mu_q^2-1-log\sigma_q^2}{2}$
4. Pass z_n though decoder p_θ generate $x_n^{'} = p_\theta(z_n)$

5. Compute $logp_\theta(x_n|z)$

$$\begin{aligned} logp_\theta(x_n|z) &= log(p_{mn}^{(x_{mn})}(1-p_{mn})^{(1-x_{mn})}) \\ &= x_{mn}(logp_{mn})(1-x_{mn})log(1-p_{mn}) \end{aligned} \tag{4}$$

6. Compute reconstruction loss L_n^{recon}

$$L_n^{recon} = -\mathbb{E}_{q\phi}(z|x_n)[logp_\theta(x_n|z)] = -log(p_{mn}^{(x_{mn})}(1-p_{mn})^{(1-x_{mn})}) \tag{5}$$

7. Sum up $L_n = \mathcal{L}_n^{recon} + \mathcal{L}_n^{reg}$

1.8 Reparameterization Trick

To calculate the $\bigtriangledown_\phi \mathcal{L}$, we are not able to compute the expectation output of latent layer, considering it is a sampling procedure which can not be formulated. Also in the structure of neural network, back propagation is not workable with sampling. Reparameterization trick reshaped the output z by introducing a new variable ϵ, which follow the Gaussian Distribution. It represents the stochastic part of the sampling process. Afterwards, the expected output can be rewritten as $z = \mu + \sigma\dot{\epsilon}$, where we can only focus on the update of μ and σ.

1.9 Training Process

For MLP implemented model, the encoder firstly transformed the input dimension to hidden dimension by Linear module, following by a ReLU activation function. Afterwards, the hidden layer transformed to two separate modules, representing mean and log_std, respectively, which is built by a Linear module with output latent dimension, z=20 in our circumstances.

In contrast to encoder, the decoder firstly convert the latent dimension to hidden dimension , following by ReLU as activation function. Then decoder reshape the input to output dimension which fit the image shape. It's noticeable that we didn't applied a Sigmoid by the end of the decoder for the sake of stability, thus need to apply a sigmoid function to the final output when computing loss and sampling.

Hyper parameters hadn't been changed, which are: training epochs=80, learning rate=0.001 and Adam as optimizer. From figure 1 we observed that bpd value over both training set and validation steadily converged below the threshold of 0.16, while validation set appeared some turbulence in the latter training. The final bpd score on test set is 0.1377.

1.10 Sampled Image

Figure 2 shown the sampled digits in a 8x8 grid. At epoch0 sampled digits are completely noise by generator. With training going on, at epoch=10 we could recognize by the outline of digits. By the end of training of 80 epochs, the quality of sampling barely improved from epoch10's digits. The main problem

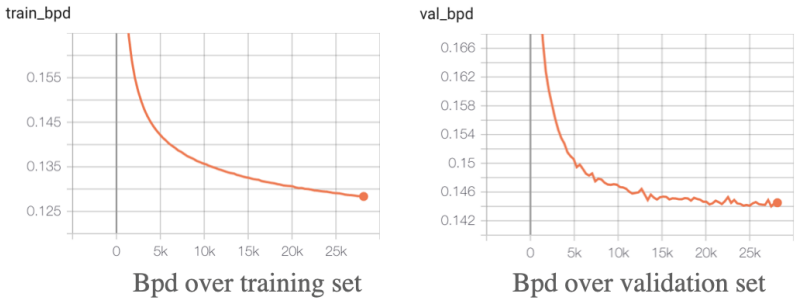

Figure 1: MLP as encoder and decoder training process

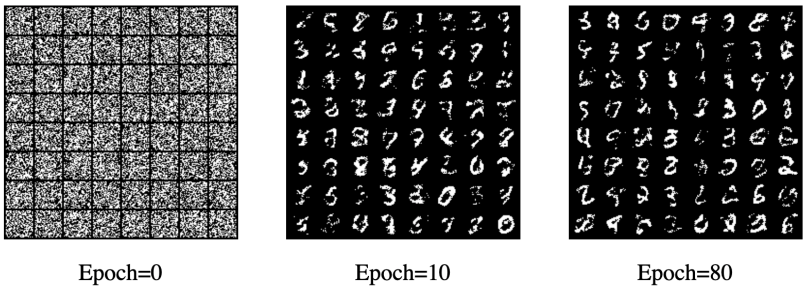

Figure 2: Sampled digits from the model in different training epoch

of the image is it's non-coherency shape and blurred outline, which made some generated digits unrecognizable.

1.11 Manifold

The manifold of digits is shown in the figure 3. This is generated in a 2-dimensional latent space, sampling from the percentile by 20 steps in range of a $[0.5/(grid_size + 1), (grid_size + 0.5)/(grid_size + 1)]$ interval. From the manifold we can see how digits are related in the 2D space. For instance, digit 1 on the top left corner can be transformed in to 2 when go vertically, and 8 when go horizontally.

1.12 CNNEncoder

The training result of CNN-encoder-decoder can be found in following figure 4. Each of corresponding bpd score of CNN-model is smaller than MLP-based model, indicating higher image quality. This can be also verified by the sample image, which is shown in figure 5. The digits are better in terms of coherency,

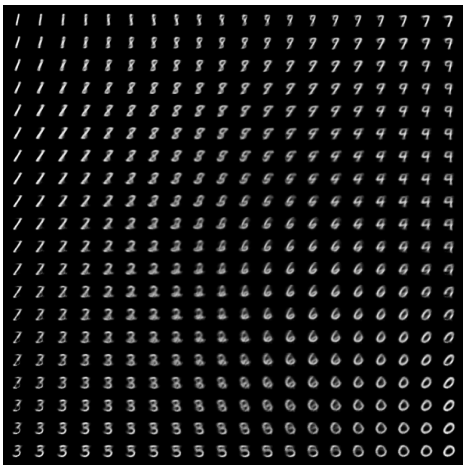

Figure 3: Generated manifold after 80 epochs training

less noise and sharper edge compare to it's MLP correspondent. However, due to the complicated structure of CNN, it also cost much more time to train compared with MLP model, which is a trade off between efficiency and quality.

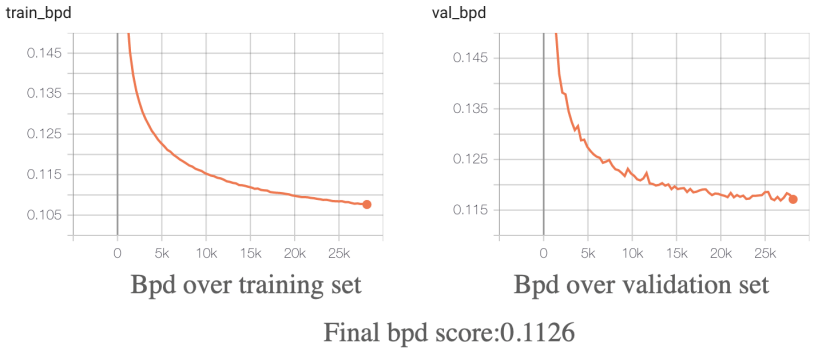

Figure 4: Generated manifold after 80 epochs training

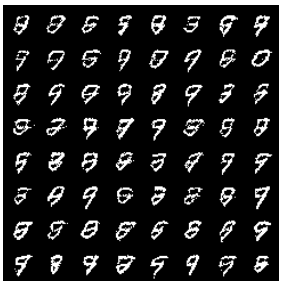

Figure 5: Generated manifold after 80 epochs training

2 Generative Adversarial Networks

2.1 Minimax

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{data}(x)}[logD(X)] + \mathbb{E}_z(z)[log(1-D(G(Z)))] \quad (6)$$

1. Firstly looking at the inner part of the $\max_D$, which is the loss function for the discriminator. The goal for discriminator is to identify real image as 1 and generated image as 0, i.e. $D(X) = 1$, and $D(G(Z)) = 0$ thus $1-D(G(Z)) = 1$. Therefore the loss value as big as possible for discriminator $\max_D$.
   Looking at the outside $\min_G$, the goal of generator is to deceive discriminator, i.e. $D(G(Z)) = 1$ in the last of equation 9, which makes the entire loss function as small as possible.

2. When V(D,G) converged, the generator can always deceive the generator while the discriminator can not justify whether the image is fake, which lead to a Nash equilibrium. The decision on whether the image is fake for discriminator would be a chance of 50-50, which lead to V(D,G)=2log(1/2)=-2log2

2.2

If we looking at the plot of log(1-x) (Lecture 10.2, Page 15), it can be seen that when x is close to 0, i.e. the discriminator identified the graph as fake, which is common at the first of training GAN, the curve of log(1-x) is rather flat. Flat curve indicates small gradient, which lead to slow learning process and low training speed. To solve the problem, commonly during the training we use a heuristic that convert

$$\min_G \mathbb{E}_z(z)[log(1 - D(G(Z)))] \longrightarrow \max_G \mathbb{E}_z(z)[log(D(G(Z)))] \quad (7)$$

which has high gradient at the early training stage and small gradient in the end(Lecture 10.2, Page 15), fit the training process more properly.

2.3 Implementation

The generator is constructed by a typical MLP structure, with a input layer, 4 hidden layers and a output layer. LeakyReLU as activation function after each Linear module, except Tanh for final activation. Moreover, according to the GAN suggestion by paper[1], after activation, a dropout layer with p=0.2 have been added to prevent over fitting. As for Discriminator, it took a flattened image tensor as input, after transforming by 2 hidden layers with activation, the final output layer has to decide whether the image is generated or real, by a 1 output dimension output Layer.

During the training step, the model optimize two separate training steps: generator step and discriminator step respectively. The generator compute the Binary Cross Entropy(BCE) loss between the discriminator's output of generated image and True labels. The discriminator compute both BCE loss of real images and fake images, and average the result. Since we didn't add a Sigmoid by the end of the Neural Network of discriminator, thus we need to use BCEWithLogitsLoss as loss function.

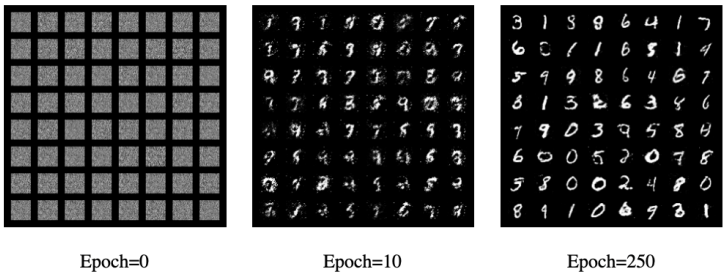

Figure 6: Generated digits by GAN in different epochs.

Figure 6 shown the sampled images by GAN. Like VAE, at the beginning, the generated digits from normal distribution sampled z is completely noise. After 10 epochs, basically the outline of digits are recognizable, though still remaining some noise and is are blurred. By the end of 250 epochs, the GAN generated very promising result, indicating the model learned digits correctly.

2.4 Interpolation

Figure 7 displayed the interpolating process between two digits over latent space. Looking vertically, transforming from one digits to another often require some intermediate digit. For instance the example on the third column, digit 1 changed to 4 is completed with the help of changing to digit 7 as intermediate step.

2.5 GAN Issue: Model Collapse

Due to none of the mentioned issue was experienced, I would like to discuss the issue of model collapse of GAN. Model collapse is the result of non-convergence.

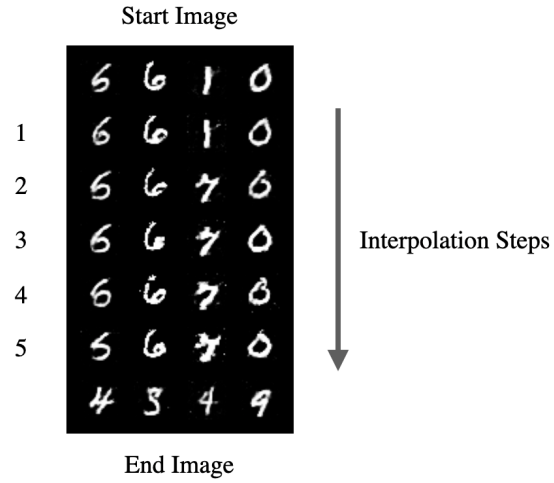

Figure 7: Interpolation between 4 pairs of digit

The training objective is to make model obtain global optimization, however the model is likely to arrive at local optimum instead of global optimum in real situation. This lead to the problem that generator always produced the most likely point while other sampling points have been ignored, i.e., poor sample diversity.

A intuitive way to solve is to add dropout to leverage the model's generalization ability. This idea is implemented by the Dropout-GAN[2], which produced a more generalized generator. Another popular solution called Multi-Agent Diverse Generative Adversarial Networks(MAD-GAN)[3], trying to incorporating multiple generators instead of one generator, which also produced fruitful result on sample diversity.

3 Generative Normalizing Flows

3.1

Firstly we write down the function f(x) and it's inverse function $f^{-1}(z)$ for following use:

$$\begin{aligned} z &= f(x) = x^3 \\ x &= f^{-1}(z) = z^{\frac{1}{3}} \end{aligned} \tag{8}$$

$$
\begin{aligned}
p_x(x) &= p_z(z)|\frac{df(x)}{dx}| \\
&= p_z(z)|\frac{x^3}{dx}| \\
&= p_z(z)|3x^2| \\
&= \frac{1}{b-a}|3x^2|
\end{aligned} \tag{9}
$$

Since z(x) is uniform distribution, the mapping of $p(x)$ would also be uniform distribution, within the range of $[a^{\frac{1}{3}}, b^{\frac{1}{3}}]$, according to equation 8. In addition, according to the second fundamental theorem of calculus, the integral of a function f can be calculated using f's antiderivatives. i.e. $\int_a^b f(t)dt = F(b) - F(a)$. In our case, this F(x) is x^3 for $f(x) = 3x^2$. Verify by integrating we have:

$$
\begin{aligned}
\int_{-\infty}^{\infty} p_x(x)dx = \int_{a^{\frac{1}{3}}}^{b^{\frac{1}{3}}} p_x(x)dx &= \frac{1}{b-a}|3x^2|dx \\
&= \frac{1}{b-a} * (b^{\frac{1}{3}})^3 - \frac{1}{b-a} * (a^{\frac{1}{3}})^3 \\
&= \frac{b-a}{b-a} \\
&= 1
\end{aligned} \tag{10}
$$

3.2

1. In order to calculate the inverse $x = f^{-1}(z)$, the dimension of h_l and h_{l-1} has to be equal. This is determined the mapping property of function: if z has lower dimension than x, the reverse function f^{-1} would have multiple output value for a single input.

2. In equation 23 of assignment, the probability of $p_x(x)$ is calculated by the multiply. When there are too many layers continuous multiplication may lead to vanishing or exploding gradient during the transformation. Moreover, usually $p(z)$ is normal distribution, which means x=0 has the highest probability p(z), which lead to $p_x(x) = 0$ by multiplication.

   Secondly, the determinant term to convert Jacobian matrix into scalar is resource consuming. With dimension of input of n, the computational complexity would be O(n!). Thus we need to make mapping function f is easily invertible and Jacobian matrix can be computed efficiently.

3.3

3.3.1

Applying normalizing flow on discrete data, instead of continuous probability, the learned distribution will concentrated on a number of discrete data points

with high probability while no volume on other points.

A method called dequantization has been used to solve this issue. This is implemented by adding a small value of noise from normal distribution to the each discrete data point, turning the discrete distribution more equally distributed for further transforming process.

3.3.2

To train a general normalizing flow model, we have following steps.

1. Training and evaluation

   - Data pre-processing. For data like images, use dequantization to convert discrete integer into continuous number space.
   - Coupling layer. In coupling layer, first we transform the input of previous module, e.g. z, by separating into two parts. The first part $z_{1:j}$ we directly make a copy while we transformed second half $z_{j+1:d}$ by applying two functions μ_θ and σ_θ in following manner:

     $$z'_{j+1:d} = \mu_\theta(z_{1:j}) + \sigma_\theta(z_{1:j}) \odot z_{j+1:d} \tag{11}$$

     The output of $\frac{\partial z'}{\partial z}$ would be a Jacobian matrix . Concatenate multiple layers by shifting each layer's first input part and second input part.
   - Compute loss: Compute the log determinant of previous generated Jacobian matrix. Taking the negative of log probability as loss
   - Back propagation: Backpropagate the loss value of the negative log probability though the model.

2. Sampling

   - Sample latent variable z from a prior distribution p(z), i.e. normal distribution.
   - Pass variable z though the Inverse function f^{-1}, having the generated output x'.

4 Conclusion

In this assignment we studied three generative models, namely VAE, GAN and Normalizing flow. As shown in the figure 8, they all adopt latent representation z to help learn the input x, but in a different manner.

Firstly for GAN, it doesn't learn any distribution but convert the generating problem into a unsupervised learning problem. The generator and discriminator evolve together during the training process. By the end, we only use generator to produce image. It has the best image quality compare to the other 2 models,

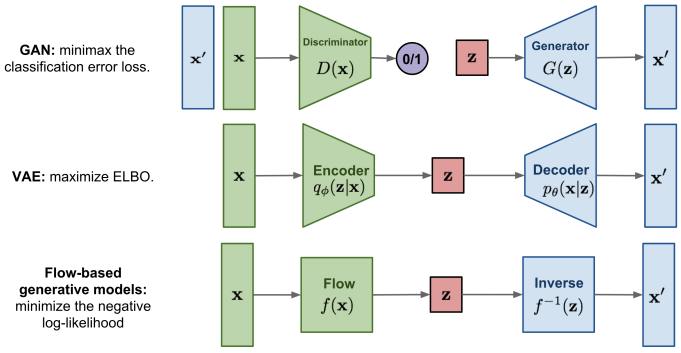

Figure 8: Input and output structure of 3 generative models [4]
1

however also have issues like model collapse and instability, which is hard to train.

Both VAE and Flow-based model trying to learn a distribution over the latent variable z. The difference is VAE use a approximate ELBO to approximate the p(z) while Flow-based model directly construct the distribution by multiple invertible transformations. VAE, which build on Auto-encoder, instead of converting x to z directly, trying to learn μ and *sigma* to construct a distribution over latent representation z. Compare to GAN, the model is more trainable and tractable, while compare to flow-based model, it has more flexibility discard the same dimensional constraint for x and z, and could also learn compressed information over dataset since the dimension of latent space z usually much smaller than the dimension of x. On the other hand, flow-based model use a continuous convention to convert input x to latent z. It has better generated image quality, however, consume more resource to train due to the computation determinant of Jacobian matrix, compared to VAE. The comparison over different aspects is summarized in the table 1.

	Likelihood	Stabality	Training cost	Image quality
GAN	No	Unstable	Medium	Best
VAE	Approximate	Stable	Low	Normal
Flow-based	Tractable	Stable	High	Good

Table 1: Comparison over GAN VAE and Flow-based model

References

1. Isola P, Zhu J, Zhou T, Efros A. Image-to-Image Translation with Conditional Adversarial Networks. arXiv.org. https://arxiv.org/abs/1611.07004v1. Published 2016. Accessed December 23, 2020.
2. Mordido G, Yang H, Meinel C. Dropout-GAN: Learning from a Dynamic Ensemble of Discriminators. arXiv.org. https://arxiv.org/abs/1807.11346. Published 2018. Accessed December 23, 2020.
3. Ghosh A, Kulharia V, Namboodiri V, Torr P, Dokania P. Multi-Agent Diverse Generative Adversarial Networks. arXiv.org. https://arxiv.org/abs/1704.02906. Published 2017. Accessed December 23, 2020.
4. Weng L. Flow-based Deep Generative Models. Lil'Log. https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html. Published 2018. Accessed December 23, 2020.