Deep Probabilistic Generative Models - Final project

Jiangnan Huang, You Zuo

December 20, 2020

Instructor: Caio Corro

Abstract

This report aims to make a discussion about three different generative models: Variational Auto-Encoders, Generative Adversarial Nets, Adversarial Auto-Encoders and demonstrate how to implement them in practice. In sections 1,2 and 3 we introduce the main ideas behind these three generative models and give some points to note when implementing these models in practice. Finally, in Section 4 we demonstrate and analyze the results obtained by the three different models implemented from our experiments.

Keywords: VAE; GAN; AAE

1 Variational Auto-Encoders

As the details about the Variational Method have already been discussed in our previous work of Lab1, in this section, we will only give a very brief review about VAE. We will then discuss the objective function in the specific case where both the latent and observed distributions are assumed to be Gaussian.

1.1 Main ideas of VAEs

VAEs is a classical generative model, which can be used to generate new data. While generative model is a model parameterized by θ which can sample new points from the observed variable distribution $P(x^{(d)})$. To train this model, we need to learn θ to maximize the log-likelihood of training data. The objective function can be written as:

$$\hat{\theta} = \arg\max_{\theta} \sum_{d} \log p(x^{(d)}; \theta) \tag{1}$$

As the $P(x^{(d)})$ in real world could be complicated, and thus hard to estimated. We would like to introduce a latent variable z to better interpret the data, and assume it follows a distribution $P(z)$:

$$\hat{\theta} = \arg\max_{\theta} \sum_{d} \log \int p(x^{(d)}|z;\theta)p(z)dz \tag{2}$$

In VAE, we usually assume $P(z)$ to be standard Gaussian for its nice form and properties. But since z is a continuous variable, the computing process of equation (2) can be intractable. To solve this problem, we introduce **Monte Carlo Sampling Method**(MC):

$$\int p(x^{(d)}|z;\theta)p(z)dz = \frac{1}{m}\sum_{i=1}^{m} p(x^{(d)}|z_i) \tag{3}$$

To make the MC step more efficient, we need our model be capable of reconstructing the data x_i with fewer times of sampling m. Intuitively, if we sample from $p(z|x;\theta)$ instead of $P(z)$, the samples may have higher probability to well reconstruct the data. As the $p(z|x;\theta)$ can not be computed directly, we propose another distribution $q(z|x;\Phi)$ which is actually the encoder network to approximate it by minimizing the KL-divergence between these two distributions (basically this is where the 'V' in VAEs comes from). We add this to our objective function:

$$\hat{\theta} = \arg\max_{\theta,\phi}[\log p(x;\theta) - KL[q(z|x;\Phi)||p(z|x;\theta)]] \tag{4}$$

And this is exactly the definition of evidence lower bound of log-likelihood (ELBO). It is a lower bound because the KL term is always greater than or equal to zero. The previous equation can be reorganized as:

$$\hat{\theta} = \arg\max_{\theta,\phi}[\mathbb{E}_{z\sim q(z|x;\Phi)}[\log p(x|z;\theta)] - KL[q(z|x;\Phi)||p(z)]] \tag{5}$$

The first term of ELBO is called the **reconstruction term**. It is actually our decoder network which gives $p(x|z;\theta)$. The second term is a **KL-divergence term**, it simply tells that we should make the approximate posterior distribution $q(z|x;\Phi)$ close to the prior $p(z)$.

1.2 Distribution Family of Variables

1.2.1 Latent Variables

In this work, we assumed the distributions of latent variables $P(z)$ are Gaussian. Then from the output of the encoder network we obtain the mean vector $\mu_{z|x}$ and the variance $\sigma^2_{z|x}$ for each proposed distribution $q(\{z_i\}_i|\{x_i\}_i;\Phi)$. To remove the stochastic node from the path of gradient, we sample from the proposed distribution by applying the **reparameterization trick**: Sampling from

another random variable $\epsilon \sim \mathcal{N}(0, I)$ each time, then rescale ϵ with the output of encoder to represent a sample of z as:

$$z = \epsilon \cdot \sigma_{z|x} + \mu_{z|x} \tag{6}$$

Hence the gradients $\frac{\partial z}{\partial \sigma}$ and $\frac{\partial z}{\partial \mu}$ can be computed via the reparameterization expression above.

1.3 Observed Variables

In this work, we assume that the observed variables $\{x_i\}_i$ are Gaussian, i.e. $p(x_i|z_i;\theta)$ given by the decoder is a Gaussian distribution. The standard deviation σ_i is fixed to a constant value 0.1, therefore the output of the decoder is only the mean parameters μ_i. In this case, the reconstruction term in our objective function can be simplified to L2 loss. Here we give the derivation:

As in each time we only sample one random value from $p(z|x;\theta)$(actually from the proposed distribution $q(z|x;\phi)$) by applying MC estimation, then we have:

$$\begin{aligned}
\arg\max_{\theta,\phi} \mathbb{E}_{z\sim q(z|x;\Phi)}[\log p(x|z;\theta)] &= -\arg\max_{\theta} \log p(x|z;\theta) \\
&= \arg\max_{\theta} log(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}) \\
&= \arg\max_{\theta} (log\frac{1}{\sigma\sqrt{2\pi}} - \frac{(x-\mu)^2}{2\sigma^2}) \\
&= \arg\max_{\theta} -\frac{(x-\mu)^2}{2\sigma^2} \\
&= \arg\max_{\theta} -(x-\mu)^2 \\
&= \arg\min_{\theta} (x-\mu)^2
\end{aligned} \tag{7}$$

Where x is the real data points, and μ is the corresponding output of the decoder network. The final loss function of VAEs can then be written as follows:

$$loss = \arg\min_{\theta,\phi}[(x-\mu)^2 + KL[q(z|x;\Phi)||p(z)]] \tag{8}$$

1.4 Construction of VAEs

The construction of VAEs includes two networks: encoder and prior decoder. Our neural networks are constructed in the following way.

Encoder:

From input x to the hidden layer h_1, we have a full linear connection and a non-linear relu connection, same structure for hidden layer h_1 to h_2:

$$\begin{aligned}
h_1 &= \max(W_1 \cdot x + b_1, 0) \\
h_2 &= \max(W_2 \cdot h_1 + b_2, 0)
\end{aligned} \tag{9}$$

From hidden layer h_2 to the output of encoder network we have:

$$y_1 = \begin{pmatrix} \mu_x \\ \sigma_x \end{pmatrix} = \begin{pmatrix} W_{31} \cdot h_2 + b_{31} \\ W_{32} \cdot h_2 + b_{32} \end{pmatrix} \tag{10}$$

PriorDecoder:

We have two hidden layer with a full linear connection and a relu, then a full linear connection and a sigmoid to the output:

$$\begin{aligned} h_3 &= \max(W_4 \cdot y_1 + b_4, 0) \\ h_4 &= \max(W_5 \cdot h_3 + b_5, 0) \\ \hat{x} &= \sigma(W_6 \cdot h_4 + b_6) \end{aligned} \tag{11}$$

The sigmoid function was added for the decoder since the images in the training set are normalized in $[0, 1]$, the output which are actually the means of Gaussian should also be in $[0, 1]$.

2 Generative Adversarial Net

2.1 Main ideas of GAN

Generative Adversarial Network(GAN), is a combination of two neural networks, a generative network $G(x)$ which is called generator, and a discriminative network $D(x)$ called discriminator. The generator maps samples z from the prior $p(z)$ to the data space. While the discriminator, which is actually a 2-classes classifier, computes the probability of a point x in data space.

To train a GAN, we need 2 SGD steps in each training epoch. Firstly the discriminator is trained to maximally distinguish the real point from the generated point given by the generator. Then the generator is trained to maximally confuse the discriminator into believing that samples it generates come from the real data distribution. The main ideas of GAN is the adversarial game between these two neural networks, the objective function can then be expressed as following:

$$\min_G \max_D \mathbb{E}_{z \sim p_{data}}[logD(x)] + \mathbb{E}_{z \sim p_z}[log(1 - D(G(z)))] \tag{12}$$

In this work, the prior distribution $p(z)$ was assume to be a Gaussian with 100 dimensions, where $p(z_i) \sim N_i(0, 1)$.

2.2 Construction of GAN

The construction of GAN also includes two networks: generator and discriminator.They are constructed in the following way.

Generator:

From the Gaussian prior $P(z)$ to the hidden layer h_1, we have a full linear connection and a non-linear tanh connection, same structure for hidden layer h_1 to h_2, also for hidden layer h_2 to output of the network:

$$\begin{aligned} h_1 &= \tanh(W_1 \cdot z + b_1) \\ h_2 &= \tanh(W_2 \cdot h_1 + b_2) \\ \hat{x} &= \tanh(W_3 \cdot h_2 + b_3) \end{aligned} \tag{13}$$

The output $\hat{x}$ has the same dimension as (a batch of) real image.

Discriminator:

We have two hidden layers with a full linear connection and a leakyrelu activation function, then a full linear connection and a sigmoid to the output:

$$\begin{aligned} h_3 &= \max(W_4 \cdot x + b_4, 0) + \alpha \cdot \min(W_4 \cdot x + b_4, 0) \\ h_4 &= \max(W_5 \cdot h_3 + b_5, 0) + \alpha \cdot \min(W_5 \cdot h_3 + b_5, 0) \\ \hat{y} &= \sigma(W_6 \cdot h_4 + b_6) \end{aligned} \tag{14}$$

The leakyrelu was used for the discriminator instead of relu because we don't want to loss to much information when backprop, the α was set to be 0.01. If the input x of the discriminator is from the real training set, we wish the output $\hat{y}$ to be close to 1. Otherwise, if the input is $\hat{x}$ generated by the generator, then we wish the output $\hat{y}$ to be close to 0.

3 Adversarial Auto-Encoders

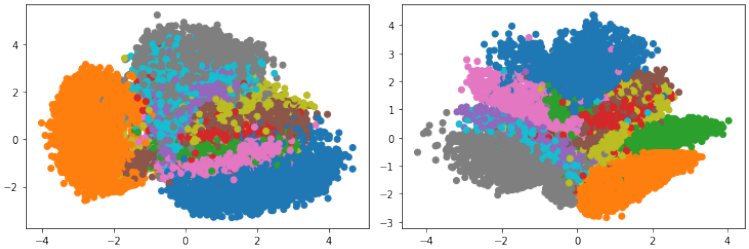
3.1 Main ideas of AAEs

According to the original paper[1], the Adversarial Auto-Encoder is also an auto-encoder, which can be trained to compress the input data through the encoder and then reconstruct it by the decoder.

But instead of proposing a certain type of distribution for the posterior $q(z|x)$ (the output of encoder) and minimizing the KL divergence $KL(q(z|x)||p(z))$, Adversarial Auto-Encoder applies a generative adversarial networks(GANs) to match the aggregated posterior of the hidden space $q(z) = \int_x q(z|x)p_d(x)dx$ directly to an arbitrary imposed prior $p(z)$ like a swiss roll distribution or mixture Gaussian.

They proposed this method because, to compute the KL divergence, VAEs requires a rather strict assumption for the distribution of latent variable z. It should have a nice form to compute the KL divergence (usually $\mathcal{N}(0, 1)$ for the

prior and a Gaussian distribution for the conditional posterior). Moreover, it is actually hard to have a good approximation from Monte-Carlo sampling for computing KL divergence. All these limitations make it tricky for VAEs to have a good latent representation compatible with p(z). For example, from our previous experiments based on MINIST in Figure 2, we can see that data with different labels mixture together in latent space, which is hard for interpreting different latent variables and controlling the decoder's output as expected.



latent space of VAEs with binary observed space

latent space of VAEs with Gaussian observed space

Figure 1: two Gaussian latent spaces of VAEs

However, in AAEs, the discriminator helps to push the encoder to approximate the prior directly, without knowing the exact functional form of distribution of z. Thus, we can impose a latent distribution without considering too much, and it can even have a complicated form like a swiss roll or a mixture Gaussian. The decoder will learn the mapping from this imposed latent distribution $p(z)$ to our data distribution $p_d(x)$.

So, when we want to generate data via a more complicated latent manifold, we can choose AAEs instead of VAEs.

3.2 Construction of AAEs

The construction of AAEs includes three networks: encoder, decoder and discriminator.

Encoder

Since we do not need to compute the approximate posterior anymore, the output of encoder network are directly the latent variables. The structure of the encoder is defined as below:

From input x to the hidden layer h_1, we have a full linear connection and a non-linear relu connection:

$$h_1 = \max(W_1 \cdot x + b_1, 0) \tag{15}$$

where W_1 a matrix of dimension $(512, 28 \times 28)$

Then we have again a full linear connection and do a batch norm for each mini-batch, and a relu as activation function:

$$\begin{aligned} h'_2 &= W_2 \cdot h1 + b_2 \\ h''_{2B} &= \frac{h'_{2B} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ h_2 &= max(h''_2, 0) \end{aligned} \tag{16}$$

where W_2 a matrix of dimension $(512, 512)$, and μ_B σ_B^2 are mean and variance of each mini-batch.

Finally from hidden layer to the output of encoder network is a fully linear connection:

$$z = W_3 \cdot h_2 + b_3 \tag{17}$$

where W_3 a matrix of dimension $(2, 512)$, so the output space of encoder will be dimension of 2.

Decoder

For decoder, we will have the output of encoder as its input directly, and the reconstructed data as its output. We first implement a fully connected layer and a relu activation function as usual:

$$h_3 = \max(W_4 \cdot z + b_4, 0) \tag{18}$$

where W_4 a matrix of dimension $(512, 2)$.

Then again like what we have done in the encoder, we applied a batch norm after the full linear connection and used a relu as activation function:

$$\begin{aligned} h'_4 &= W_5 \cdot h3 + b_5 \\ h''_{4B} &= \frac{h'_{4B} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ h_4 &= max(h''_4, 0) \end{aligned} \tag{19}$$

where W_2 a matrix of dimension $(512, 512)$, and μ_B σ_B^2 are mean and variance of each mini-batch.

Finally a linear connection followed by a Tanh() as non-linear activation function:

$$\hat{x} = \tanh(W_6 \cdot h4 + b_6) \tag{20}$$

where W_3 a matrix of dimension $(28 \times 28, 512)$, so the output space of encoder will be a vector of length 28×28 .

Discriminator

The task for Discriminator is figuring out whether the latent variables are generated from the Encoder or chosen from the imposed prior. So it is a network as binary classifier.

$$y1 = max(W_8 \cdot \max(W_7 \cdot z + b_7, 0) + b_8, 0) \tag{21}$$

where W_7 is matrix of dimension $(512, 2)$ and W_8 is matrix of dimension $(256, 512)$

For a binary classifier, we want to give outputs as probabilities. So we implement first a linear connection and then a sigmoid to give a probability:

$$d = \sigma(W_9 \cdot z + b_9) \tag{22}$$

where W_9 a matrix of dimension $(1, 256)$.

Choice of hyperparameters

For the hyperparameters, we have set the default values for most of the hyperparameters.

- learning rate = 0.001 for all the optimizers
- Other Adam hyperparameters as default: betas=(0.9, 0.999), eps=1e-08, weight decay=0
- number of epochs = 20
- batch size = 64 instead of 256, since we have found that training with small size of mini-batch gives less noise in latent space
- number of updates of encoder in each iterition = 1 (to avoid overfitting)

4 Results of Experiments

In this work, our model were trained on the **MNIST** dataset. There are 50000 images in our training set and each image is of size 28x28.

4.1 Results of VAE

First we visualize the latent space by displaying the mean values:

From the result we can see that almost all the points are close to 0 and there are several clusters of the same color, which means some of the classes are well

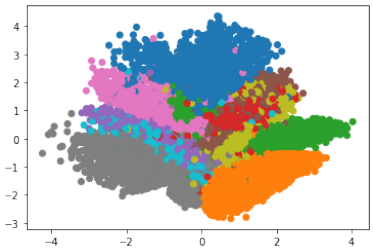


Figure 2: Latent space

delimited, but there also exist some classes with mean values really close to 0 are mixed together.

Then we generate some new distributions of random variable X by applying the trained decoder on some new samples from the latent space. The new images are then generated by sampling from the output Gaussian distribution or directly using the mean value for each random variable. In this work, we display the output image with by using the mean value.

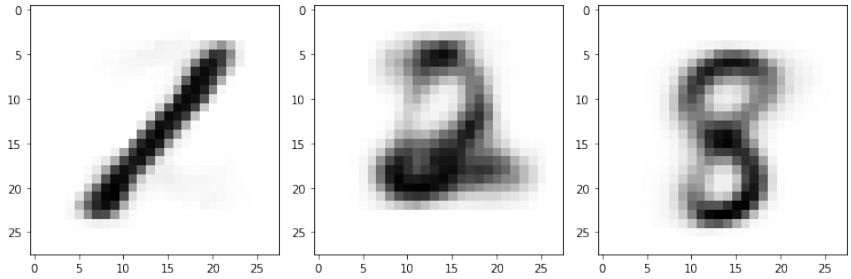


Figure 3: Distributions generated by VAE

From the sampled distributions we can see that the VAE with continuous Gaussian latent space and continus Gaussian observed spaces works pretty well on this task. Even though the generated distributions are a bit blurry, we can still observe the shape of the numbers.

Then we try to interpolate in the latent space. First we randomly take two images from the training set and compute the mean of the posterior by using the encoder, and then decode with different convex combinations of these two means. In one of our experiments, the random images selected represent numbers 2 and 6.

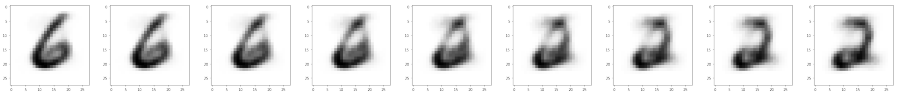


Figure 4: Interpolate in the latent space

From the results we can see that, with the interpolation, the number 6 gradually and smoothly becomes the number 2, the generated images in the middle contains the characteristics of both numbers.

4.2 Results of GAN

We demonstrate the results of GAN by giving the images generated by the generator of GAN with different number of running epochs. In this way we can also observed how did the model "learn" the observed variable distribution and whether it was converged.

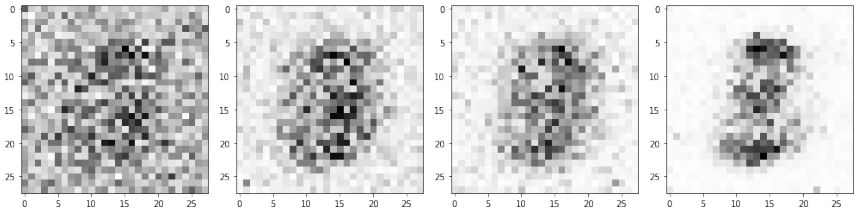


Figure 5: Images generated by GAN with number of epoch = 1,5,10,20

From the first four images we can see that, for the first 20 epochs, the generator began to realize that in the training set, most of the useful pixels are concentrated in the middle of the images. So it tended to generate images with some cloud like pixels in the middle and try to confuse the discriminator. At epoch 20 we can actually observed the shape of several numbers. (which is like '3' for the example in the Figure 5)

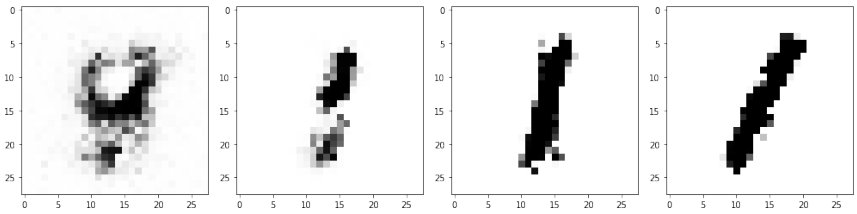


Figure 6: Images generated by GAN with number of epoch = 50,100,200,300

From the last four images, we can observed that from 50 epochs to 300 epochs, the images generated by the generator became cleaner and more clear, which

means the model became more and more stable and tended to converge. But the big problem is that after converging, the generator could only generate images which represent number '1'.

This problem may be caused by a mixture of numbers when training GAN. As most of the numbers had some part of pixels which are included in the pixels distribution of number '1', i.e. number '1' is partially included in every images. So if we train our GAN with all the images with different numbers mixed together, finally the generator will believe that generating the '1' is the most conservative result.

How to solve this problem will be the future work, one idea is that we can first train a classifier using the given label to classify the images with different numbers. Then we can have a group of GANs trained for each specific number. But considering that the GAN should actually be an unsupervised learning model, this may not be a good solution.

4.3 Results of AAEs

For the part of AAEs, we proposed a swiss roll distribution for the prior in an unsupervised mode. It means for discriminator, it can only tell whether this latent point comes from the real prior or generated by the encoder, and it can not distinguish which number it represents. But we can always give digits labels when demonstrating the latent space:

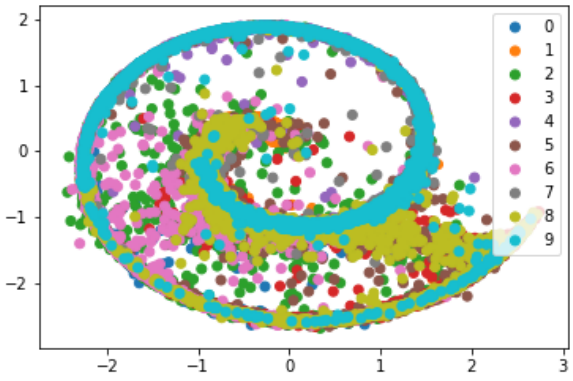


Figure 7: Latent space

As we can see, in this unsupervised mode, it is quite challenging for the encoder to separate different digits in the latent manifold. But we can see that AAEs fits well this complicated imposed prior, which would be almost impossible for

VAEs. Then we want to see what characteristics does the swiss roll represents for different positions. Take 100 points uniformly from the center to the outside, and use the decoder to generate these data, we get:

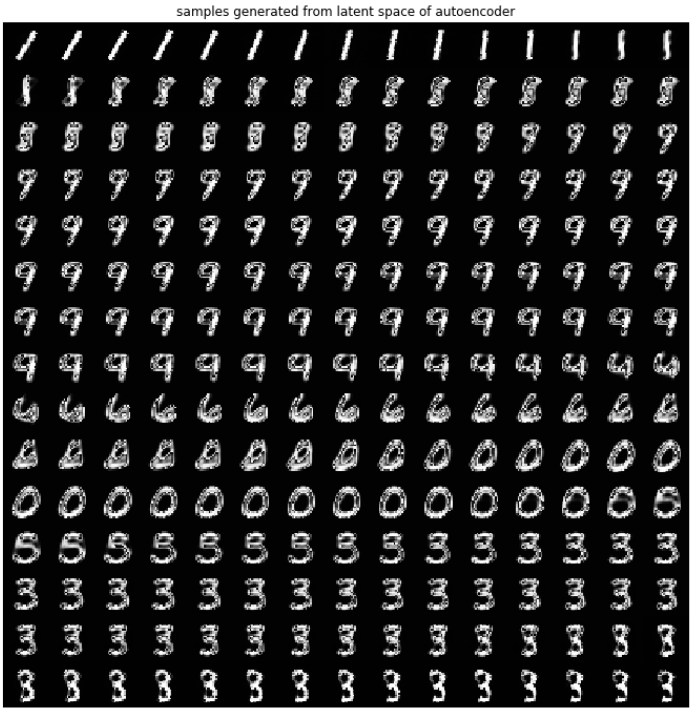


Figure 8: Latent space

We can see from these generated samples: the numbers generated by AAE are pretty real, and the transition between digits is also very natural.

5 Conclusion

By doing this project, we have implemented VAEs again but with observed variables as Gaussian. We can see from the latent space and generated digits that the assumption as Gaussian distribution has better performance than the Bernoulli distribution for MNIST. On top of that, we have implemented GANs, which is a powerful generative model without explicit assumption for the distribution. Finally, we used AAEs that combines an auto-encoder with an adversarial discriminator. This model well solved the problem of VAEs and gave us a very good approximate of any arbitrary imposed prior distribution. After training the AAEs, we get a decoder that learned how to map the imposed prior to our data distribution.

But as we can see, the latent space of AAEs sometimes can not distinguish different digits. They are distributed in a swiss roll mode, but all the ten digits are mixture together. In later exploration, we can change our discriminator to a supervised mode that has digits labels as a one-hot vector. In this way, we cannot only get hidden information about the styles of the manuscript, but we can also have basic information about each digit.

References

Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I., and Frey, B. (2016). Adversarial autoencoders.