Deep Learning Assignment 3

Yke Rusticus
Student nr.: 11306386
University of Amsterdam
yke.rusticus@student.uva.nl
06/12/2019

1 Variational Auto Encoders

Question 1.1.1
Whereas a standard autoencoder (SAE) projects inputs onto latent space using single values for features, the variational autoencoder (VAE) represents feature values as probability distributions [1]. Instead of producing one encoding vector to describe the input, the VAE produces a vector of means and a vector of standard deviations. From these probability distributions, the decoder can sample a "new" feature vector to decode. So in terms of their main function, the SAE and the VAE are similar. They both consist of an encoder that projects input onto latent space, and an decoder that essentially does the opposite. The difference lies in the distribution of samples in latent space both models produce. The SAE produces a highly discontinuous distribution, whereas the VAE is designed to produce a continuous distribution. For this reason, the latter can be used for different and/or more purposes than the former.

Question 1.1.2
The way the VAE is designed makes sure that the full latent space is used according to a probability distribution around the origin. Any sample from this would therefore generate (in general) reasonable outputs, as the decoder is trained to handle any of the samples from the latent space. For the SAE, this is not the case. Since the SAE uses single feature values to project inputs onto latent space, the latent space is not fully used. Only certain regions in latent space are used, and the decoder will only know how to decode samples from these regions properly; any sample from other regions are never seen by the decoder, and outputs for these samples would not be as good. An SAE is trained to reconstruct its input, but its application does not reach far beyond that. So as a generative model, a VAE is preferred over an SAE.

Question 1.2
Since each x_n depends on corresponding latent variable z_n, we use ancestral sampling to sample from the model. With ancestral sampling, we first sample the independent "parent" variables, in this case the latent variable z_n. Then, we sample the variables that are dependent on the parent. We do this by conditioning the "child", in our case x_n, on the parent z_n. So in summary, to sample the n'th image, do:

(1) Sample z_n from $\mathcal{N}(0, I_D)$

(2) Apply our neural network $f_\theta : \mathbb{R}^D \rightarrow [0, 1]^M$ to z_n, so we obtain $f_\theta(z_n)$

(3) Then, for each pixel m, we sample according to Bern$(x_n^{(m)}|f_\theta(z_n)_m)$ to get our image x_n

Question 1.3
As is explained in Question 1.2, a neural network f_θ is applied to the latent variable before we use it to sample other variables. This neural network can learn a mapping from the latent space to the desired output variable space. Thus, it does not really matter how we initialise the distribution over z, as the neural network can adapt to different distributions as input. A Gaussian distribution is simply a convenient choice.

Question 1.4a
Monte Carlo Integration can be used to approximate an integral[1]. In our case of evaluating $\log p(x_n) = \log \int p(x_n|z_n)p(z_n)dz_n$, we can approximate the (intractable) RHS of this equation by sampling N samples z_i from $p(Z)$ and averaging the probabilies of x_n given these samples:

$$\log p(x_n) \approx \log \Big[\frac{1}{N} \sum_{i=1}^{N} p(x_n|z_i) \Big] := F \tag{1}$$

$p(z_i)$ is incorporated in the sampling of z_i, therefore it is not included in the expression of the approximation. We have that $\lim_{N\to\infty} F = \log p(x_n)$.

Question 1.4b
Ideally we would sample each element in latent space to build up an as good approximation of $\log p(x_n)$ as possible. However, the number of elements we would sample scales exponentially with dimensionality of z. Furthermore, many of the probabilities $\log p(x_n|z_i)$ contribute only a small amount to the final approximation, but we don't know beforehand for which z_i this is the case. This makes it a very costly procedure, and because it is so inefficient it is not used for training VAE type of models.

Question 1.5a
Using the source[2] given in the next question (1.5b), the formula for $D_{\text{KL}}(q\|p)$ can be written as:

$$D_{\text{KL}}(q\|p) = \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2}, \tag{2}$$

where σ_p = 1 and $\mu_p = 0$, so:

$$D_{\text{KL}}(q\|p) = \log \frac{1}{\sigma_q} + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2} \tag{3}$$

$$= -\log \sigma_q + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2} \tag{4}$$

For q we can plug in a distribution identical to p, i.e. $\sigma_q = 1$, $\mu_q = 0$. Since the KL-divergence measures information loss for an approximation of a distribution, this will result in a KL-divergence of zero, as no information is lost when we approximate a distribution with the distribution itself. For any other (larger) values of σ_q, μ_q, the KL-divergence increases. So a large KL-divergence would be obtained if for example, $\sigma_q = 1000$ and $\mu_q = 1000$.

Question 1.5b
As was shown in the previous question, the formula for $D_{\text{KL}}(q\|p)$ can be written as:

$$D_{\text{KL}}(q\|p) = -\log \sigma_q + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2}, \tag{5}$$

for $q = \mathcal{N}(\mu_q, \sigma_q^2)$ and $p = \mathcal{N}(0, 1)$.

Question 1.6
Given:

$$\log p(x_n) - D_{\text{KL}}(q(Z|x_n)\|p(Z|x_n)) = \mathbb{E}_{q(Z|x_n)}[\log p(x_n|Z)] - D_{\text{KL}}(q(Z|x_n)\|p(Z)). \tag{6}$$

Now call the RHS of the equation "RHS" and the KL-divergence on the LHS "KL". Then

$$\log p(x_n) - \text{KL} = \text{RHS} \tag{7}$$

$$\log p(x_n) = \text{RHS} + \text{KL} \tag{8}$$

[1] https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration

[2] https://stats.stackexchange.com/questions/7440/kl-divergence-between-two-univariate-gaussians

Since the KL-divergence KL is non-negative, the minimum value for $\log p(x_n)$ is RHS for KL $= 0$. For this reason, RHS is called the lower bound on the log-probability.

Question 1.7
We have seen in Question 1.4a that evaluating $\log p(x_n)$ involves an intractable integral. Furthermore, approximating this integral is computationally inefficient and impractical. We therefore optimize the lower-bound, instead of optimizing the log-probability directly.

Question 1.8
Recall Eq. 6 and 7, respectively:

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

Increasing RHS will result in an increase of $(\log p(x_n) - \text{KL})$. For this to happen, either (1) $\log p(x_n)$ must increase, or (2) KL must decrease. In situation (1), this means that we are optimizing the log-probability. In situation (2), this means that our approximation $q(Z|x_n)$ gets closer to $p(Z|x_n)$.

Question 1.9
We want the model to be able to reconstruct inputs from their representation in latent space. Say, an image A has an embedding in latent space z_A. Then we want that our model is able to perfectly reconstruct A, given z_A. In general, the larger $\mathbb{E}_{q_\phi(z|x_n)}[\log p_\theta(x_n|Z)]$, the better the mapping from latent variable Z to the desired data point x_n, or the better the "reconstruction". Reconstruction is therefore an appropriate name for he loss term $\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|x_n)}[\log p_\theta(x_n|Z)]$. However, we do not want our model to only work for a few values of Z, but cover the whole latent space well within our distribution over Z. For this reason, we want to train our encoder to produce encodings over a distribution that is similar to $p_\theta(Z)$. The regularization term $\mathcal{L}_n^{\text{reg}}$ ensures this, as it brings the objective to minimize the KL-divergence between $q_\phi(Z|x_n)$ and $p_\theta(Z)$. The name "regularization" is therefore appropriate for this loss term.

Question 1.10

$$\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(Z|x_n)}[\log p_\theta(x_n|Z)] \quad \text{approximate with 1 sample} \tag{9}$$
$$\approx -\log p_\theta(x_n|z_n) \tag{10}$$
$$= -\log \prod_{m=1}^{M} \text{Bern}(x_n^{(m)})|f_\theta(z_n)_m) \tag{11}$$
$$= -\sum_{m=1}^{M} \log \text{Bern}(x_n^{(m)}|f_\theta(z_n)_m) \tag{12}$$

For $\mathcal{L}_n^{\text{reg}}$ we can factorize $q_\phi(z_n|x_n)$ and $p_\theta(z_n)$ because they have diagonal covariance matrices.

$$q_\phi(z_n|x_n) = \mathcal{N}(z_n|\mu_\phi(x_n), \text{diag}(\Sigma_\phi(x_n))) \tag{13}$$
$$= \prod_{d=1}^{D} \mathcal{N}((z_n)_d|(\mu_\phi(x_n))_d, (\Sigma_\phi(x_n))_d) \tag{14}$$
$$\tag{15}$$

Write $(z_n)_d$ as $z_n^{(d)}$, $(\mu_\phi(x_n))_d$ as $\mu_n^{(d)}$, and $(\Sigma_\phi(x_n))_d$ as $(\sigma_n^{(d)})^2$, then:

$$q_\phi(z_n|x_n) = \prod_{d=1}^{D} \mathcal{N}(z_n^{(d)}|\mu_n^{(d)}, (\sigma_n^{(d)})^2) \tag{16}$$

$$p_\theta(z_n) = \mathcal{N}(z_n|0, I_D) \tag{17}$$
$$= \prod_{d=1}^{D} \mathcal{N}(z_n^{(d)}|0, 1) \tag{18}$$

The regularization term then becomes:

$$\mathcal{L}_n^{\text{reg}} = D_{\text{KL}}(q_\phi(Z|x_n)\|p_\theta(Z)) \tag{19}$$

$$= \mathbb{E}_{z\sim q_\phi}\left[\frac{\log q_\phi(Z|x_n)}{\log p_\theta(Z)}\right] \tag{20}$$

$$= \sum_{d=1}^{D} D_{\text{KL}}(\mathcal{N}(\mu_n^{(d)}, (\sigma_n^{(d)})^2) \parallel \mathcal{N}(0,1)) \qquad \text{use Eq. 16 and 18} \tag{21}$$

$$= \sum_{d=1}^{D}\Big[-\log\sigma_n^{(d)} + \frac{(\sigma_n^{(d)})^2 + (\mu_n^{(d)})^2}{2} - \frac{1}{2}\Big] \qquad \text{use Eq. 5} \tag{22}$$

In the last line we used the notations introduced in Eq. 16. In vector form this is equal to [2]:

$$\mathcal{L}_n^{\text{reg}} = \frac{1}{2}\Big(\text{tr}\big(\Sigma_\phi(x_n)\big) + \big(\mu_\phi(x_n)\big)^\top\big(\mu_\phi(x_n)\big) - D - \log\det\big(\Sigma_\phi(x_n)\big)\Big) \tag{23}$$

We have now derived the two terms that together make up the objective $\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}$.

Question 1.11

(a) We need $\nabla\phi\mathcal{L}$ because the gradients allow us to perform gradient descent.

(b) The act of sampling prevents us from computing $\nabla\phi\mathcal{L}$, because the sampling a non-continuous operation and is therefore not differentiable [2].

(c) The *reparametrization trick* moves the sampling to a separate input layer, over which we do not perform backpropagation. For this reason it solves our problem.

Question 1.12
The VAE was implemented using the formulas derived in previous questions. For the encoder, a linear layer was implemented that projects the input onto the hidden layer. For this layer, tanh was used as activation function. Two separate linear layers then produce the mean and standard deviation (std) vectors with the same dimensionality as the latent space. The mean vector is passed through a tanh function to force values between 1 and -1 and the std through a sigmoid to force values to be larger than 0 (and smaller than 1). Later on I had my doubts about using a sigmoid, but it seemed to work. The decoder takes as input a sample from this latent space, which is during training given by using the means and stds that were produced by the encoder. The models were trained using the given default parameters, i.e. a latent dimension of 20 and a learning rate of 0.001.

Question 1.13
Figure 1a shows the losses that the model returns. These are defined to be the estimated negative ELBOs (evidence lower bounds), so taking their nagative values gives the estimated ELBOs, which are shown in Figure 1b. Note that the ELBO is defined to be the lower bound on the logarithm of the evidence in the assignment, and therefore the numbers do not represent a probability. If we take the inverse logarithm of the ELBOs however, we do get values between 0 and 1. During training this lower bound is pushed up as far as possible, which is also shown in Figure 1b. The reason why the model seems to do better on the validation set at first, is that the losses are avaraged over one epoch during training. In one epoch the model could improve a lot, resulting in a loss that is not representative of the model at the end of the epoch. However, the validation losses are calculated at the end of each epoch, so the training losses lag behind.

Question 1.14
Figures 2 to 4 show samples of the model during different stages in training. Before training, we generated image that resemble the noise they were created from. After 20 epochs, halfway through training, the model generates already reasonable images. The numbers 3 and 5 become recognizable, however it is still a bit messy. After training for 40 epochs, the model still does not generate perfect digits, but it seems to have improved compared to the model from 20 epochs. The numbers 8, 2, 5, and 9 are recognizable in this last case.

Question 1.15
Not implemented.

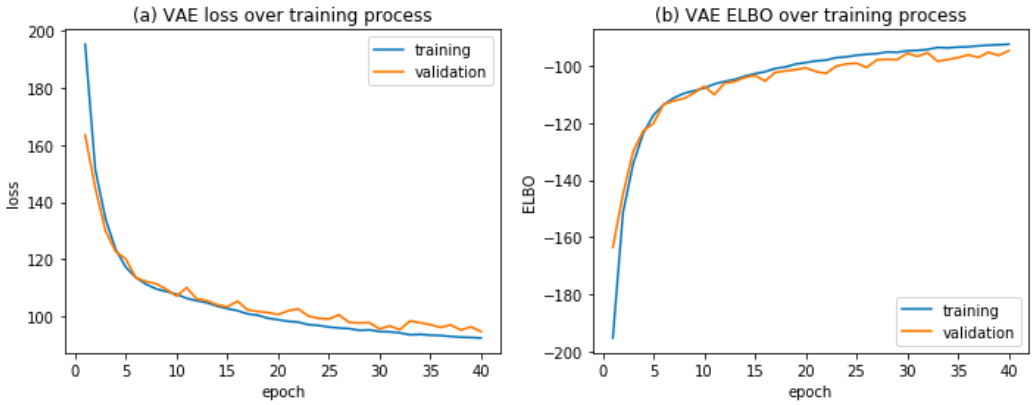

Figure 1: Loss and ELBO for the VAE during the training process.

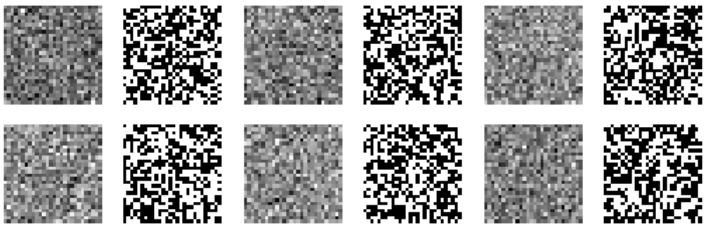

Figure 2: Six samples of the model before training. The grayscale images show the means of the samples, and the generated images are shown next to that in binary.

Figure 3: Six samples of the model mid-training (20 epochs). The grayscale images show the means of the samples, and the generated images are shown next to that in binary.

Figure 4: Six samples of the model after training (40 epochs). The grayscale images show the means of the samples, and the generated images are shown next to that in binary.

2 Generative Adversarial Networks

Question 2.1
The generator takes as input a noise sample [3]. This could be a vector of any size. It takes this noise and generates from it, in our case, an image as output. The discriminator takes an image as input, which could be a generated image from the generator or a real image that is included our data set. The discriminator then outputs a probability of the given input being real, rather than being generated. This can be represented as a single number between 0 and 1.

Question 2.2
If we think of the generator G and the discriminator D as two players that want to win a game. By doing so, the goal of G is to minimize

$$\mathcal{L} := \mathbb{E}_{p_{\text{data}}(x)}[\log D(X)] + \mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))] \tag{24}$$

and the goal of D is to maximize the same expression. D wants to be as good as possible at recognizing when an image is real or not, but G wants to "trick" D. In other words, for a real image X, D wants $D(X)$ to be in general high, so it wants to maximize the first term in Eq. 24. It also wants to output a low probability of a generated image $G(Z)$ being real, so it can distinguish in what is real or not. Here Z is noise and $G(Z)$ is the output image of G given Z. So D wants $D(G(Z))$ to be in general low, which is equivalent to $1 - D(G(Z))$ being high. For this reason it wants to maximize the second term in Eq. 24. Since it is G's task to trick D, it tries to achieve the exact opposite as D. That is, it wants $D(G(Z))$ to be in general high. It wants D to make mistakes in its predictions. However, since G can't really have any effect in the predictions of D over real images X, G is only present in the second term. G will try to minimize $\mathcal{L}$, and D will try to maximize the $\mathcal{L}$, and by doing so, they will both learn and get better at what they do (in theory).

Question 2.3
When the model has converged, both G and D cannot improve anymore. Here we are not taking into account the situation where D is perfect, and because of this G cannot improve (as there technically still room for improvement). Convergence will be reached when G produces (nearly) perfect images, indistinguishable from real images, i.e. G is able to perfectly mimic the distribution of the data set. In that case D has no way of telling whether a given image is real or generated, i.e. $D(X) = D(G(Z)) = \frac{1}{2}$. In this case:

$$V(D, G) = \log \frac{1}{2} + \log(1 - \frac{1}{2}) \tag{25}$$

$$= 2 \log \frac{1}{2} \tag{26}$$

$$= -2 \log 2 \tag{27}$$

Question 2.4
If the discriminator starts to recognize fake images, then $D(G(Z))$ goes to zero. As a result $\log(1 - D(G(Z)))$ goes to $\log(1) = 0$. So in the case that D starts to learn recognize fake images before G learn to generate proper images, the gradients for G quickly vanish, making G unable to learn. This can be solved by introducing a separate objective for G, that is to maximize $\mathbb{E}_{p_z(z)} \log(D(G(Z)))$. Then, the closer $D(G(Z))$ gets to 0, the larger the gradients for G will be.

Question 2.5
Programming part not implemented.

Question 2.6
Programming part not implemented.

Question 2.7
Programming part not implemented.

3 Generative Normalizing Flows

Question 3.1
Given:

$$z = f(x) \tag{28}$$
$$x = f^{-1}(z) \tag{29}$$
$$p(x) = p(z)\left|\frac{\mathrm{d}f}{\mathrm{d}x}\right| \tag{30}$$

For the multivariate case, $x \in \mathbb{R}^m$ and $f : \mathbb{R}^m \to \mathbb{R}^m$, we get:

$$z = f(x) \tag{31}$$
$$x = f^{-1}(z) \tag{32}$$
$$p(x) = p(z)\left|\det J_{x\to z}\right|, \tag{33}$$

where the Jacobian $J_{x\to z}$ is defined as:

$$J_{x\to z} = \frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_m}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_m}. \end{bmatrix} \tag{34}$$

The objective becomes:

$$\log p(x) = \log p(z) + \sum_{l=1}^{L} \log \left| \det \frac{\partial h_l}{\partial h_{l-1}} \right| \tag{35}$$

Question 3.2
h_l and h_{l-1} must have an equal number of components. In that case only, the derivative $\frac{\partial h_l}{\partial h_{l-1}}$ will result in a square matrix. We need to calculate its determinant, and a determinant can only be calculated for a square matrix. Moreover, the mapping needs to be invertible, this can only be the case for a non-zero determinant. This means that for each h_l, at least one element must be non-zero (however this still does not fully exclude the possibility of a determinant of zero).

Question 3.3
A computational issue that might arise when optimizing the network is that inversions will drastically drain your computation power, as they require a lot of steps. For example, the cost of inverting an $m \times m$ matrix is $\mathcal{O}(m^3)$.[3]. A problem that may arise when using your network after training for sampling datapoints, is that it might turn out that the model worked well for reconstructing data points, but not for generating new data points.

Question 3.4
The consequence of training a continuous density model on discrete integers, which should represent a continuous signal, could be that the model fails to distribute the probability mass over continuous data points [4]. A solution that is proposed, is to perform “dequantization" as pre-processing of the data. In this approach, the discrete integers are transformed to a continuous signal by adding uniform noise to the data.

Question 3.5
Zero-points question. Not submitted.

Question 3.6
The input of the model could be a vector representing data, such as an image. The model maps this input to latent space using invertible functions and by making use of change of variables. After that, the log likelihood is calculated and the gradients are calculated using backpropagation in order to

[3] `https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations`

train the model. After training, we can sample from the latent space and apply the inverse-functions to generate data, such as an image.

Question 3.7
Programming part not implemented.

Question 3.8
Programming part not implemented.

4 Conclusion

Three different kinds of deep generative models were explored in this work: variational autoencoders (VAEs), generative adversarial networks (GANs) and flow-based models. Due to limited time, only a VAE was implemented. Nevertheless, we have had a view on the capabilities of generative models by training the VAE and sampling from it. Using a fairly straightforward architecture, the model learned to generate decent looking images of digits, after training on the MNIST data set. Ideally, all three models would have be implemented and tested for us to be able to compare their behaviour. Since they all have a different approach to essentially the same task, it might be interesting to see if one outperforms the other two. If this is not the case, then we might want to choose the model that is simplest to implement and train. For now however, we will leave this for future work.

References

[1] D. Kingma and M. Welling, "Auto-encoding variational bayes. arxiv 2013," *arXiv preprint arXiv:1312.6114*, 2019.

[2] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[4] J. Ho, X. Chen, A. Srinivas, Y. Duan, and P. Abbeel, "Flow++: Improving flow-based generative models with variational dequantization and architecture design," *arXiv preprint arXiv:1902.00275*, 2019.