Assignment 3: Generative Models

Ipek Ganiyusufoglu | 12225428
ipek.ganiyusufoglu@student.uva.nl

1 VAE

1.1 Question 1.1

1. Autoencoders learn a compressed representation of the input images while VAEs learn the underlying probability distribution of the data. This means that in contrast to auto-encoders, VAEs can capture attributes of the data encoded in the latent space.
2. Autoencoders are not generative in the sense that they don't generate new samples, they rather decompress/reconstruct compressed input. This is not necessarily generation, as there's no sampling, no learned data distribution, no new instance creation. It is simply learning a mapping.
3. As said above we can think of the standard auto-encoder as a mapping from input to a compressed representation which is then decompressed to reconstruct the same input, rather then generate a new one. So can VAEs do the same thing? Well a default VAE is aimed at generating new input, so it is meant for a larger task and somewhat capturing the semantics. But we can perhaps adapt it and remove the sampling mechanism, such that the mean/variance layer instead acts as a mapping for the input-output decompression.
4. As mentioned, if we remove the sampling mechanism, which enables the generalization and generative ability of VAEs we can perhaps turn it into an auto-encoder. So this is exactly what is missing in the standard auto-encoders that prevents them from generalizing or being generative. The data probability distribution, the attributes aren't learned as the latent space, hence not sampled as well, meaning we can not generalize to a larger variety of input (fonts etc.) or generate novel instances (different combinations of attributes).

1.2 Question 1.2

Ancestral sampling is when we first sample an independent variable, then sample the following variables conditioned on the previously sampled variables. In the decoder network we sample the output x from $p(x|z)$ conditioned on z. But z is also a randomly sampled variable. During training it is sampled from a learned Gaussian, and at inference, after we throw away the encoder network, it can be sampled from any Gaussian. Hence the sampling of x is conditioned on the previously sampled variable z, hence the ancestral sampling.

1.3 Question 1.3

Instead of trying to model and restrain the latent structure z with a certain description, VAEs sample it from a standard Gaussian. This seems like a *too simple* solution to modelling the latent space, however the if we map the normal distribution through a complex enough function, then we can model any distribution. Carl Doersch expresses this like this, [7]:

> The key is to notice that any distribution in d dimensions can be generated by taking a set of d variables that are normally distributed and mapping them through a sufficiently complicated function.

Submitted to the Spring 2019 Amsterdam Deep Learning course

In VAEs this *complex function* is a neural network, namely the encoder, which can simply learn how to map simple Gaussians to a more complicated latent distribution. Hence there's no restriction on the latent space rooting from using a standard Gaussian.

1.4 Question 1.4

(a) We can approximate the intractable integral by sampling, as it is a probability distribution, I.e. we can instead sample as below using Monte Carlo Integration, which approximates by evaluating the integral at sampled points. The stochasticity introduced by this sampling can be beneficial in terms of generalization (diminishing over-fitting/regularization).

$$\log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)\right] = \frac{1}{J}\sum_{j=1}^{J} p(x|z_j) \quad \text{(where } J \text{ is latent dimensionality)} \tag{1}$$

(b) We can optimize and approximate $P(x)$ by $\approx \frac{1}{n}\sum_i P\left(X|z_i\right)$, where we would sample n z values. As Doersch explains [7], the sample number n becomes extremely large as the dimensionality increases (if we want to have a good estimate). Doersch explains this by showing us three different samples of the digit 2, he points out that a good sample of 2 can be more similar to a bad sample (cropped, not exactly 2), than the same good sample but shifted, translated, rotated etc. To avoid such similarity errors we would have to restrain the model even more with a small variance, to the extend that it would have to generate almost the same good sample 2, and we would have to sample a huge amount before we find such a sample. Because of this poor similarity comparison of samples and the necessary sampling amount, this is inefficient, especially in higher dimensionality.

1.5 Question 1.5

(a) KL divergence indicates how far away or *not* similar two distributions are. So for a small D_{KL}, an example would be two Gaussians with very close means and variances, then for a large D_{KL}, simply, far away means or different variances.

(b) Looking at the original VAE paper [2], we can write the KL divergence as below, where J is the dimensionality of latent space z. I actually use this in implementation as well (then mean over batch).

$$D_{KL}(p||q) = \frac{1}{2}\sum_{j=1}^{J}\left(1+\log\left(\left(\sigma_j\right)^2\right)-\left(\mu_j\right)^2-\left(\sigma_j\right)^2\right) \tag{2}$$

Question 1.6 & Question 1.7

In the original log probability function, we have a KL term that is **intractable**, i.e. $D_{KL}(q(z|x)||p(z|x))$. To be more specific, this term contains $p(x)$ which is an intractable integral. This can be seen by expanding $p(z|x)$ with Bayes rule. If we rephrase the entire given equation (11) as simply:

$$\log p\left(x_n\right) - D_{\mathrm{KL}}\left(q\left(Z|x_n\right)\|p\left(Z|x_n\right)\right) = \mathbb{E}_{q(z|x_n)}\left[\log p\left(x_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|x_n\right)\|p(Z)\right) \tag{3}$$

$$\log p = term1 - term2 + term3 \tag{4}$$

the third term is what we are talking about (intractable). So, we can compute $term1 - term2$ but not $term3$. But we know that KL divergence distance between any two distributions is always bigger than or equal to zero, i.e. $term3 \geq 0$. If we maximize the first two terms, we now that the rest; $term3$ is at least zero, so this means by maximizing right hand side in (5), we are pushing a lower bound ($logp + term3$) up, such that optimized function has at least that maximized lower bound. Hence instead of maximizing the log probability, which we literally can't do, the lower bound of the function is maximized. This is called a variational (or evidence) lower bound or ELBO. We can express it as:

$$\log p\left(x_n\right) \geq argmax\left(\mathbb{E}_{q(z|x_n)}\left[\log p\left(x_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|x_n\right)\|p(Z)\right)\right) \tag{5}$$

1.8 Question 1.8

If we manage to maximize $term1$,$-term2$, this means we have successfully maximized the lower-bound. As a consequence of this since we have two different terms on the left hand side, two different things can happen. Either the log likelihood (the data likelihood distribution) will increase, or the KL divergence between the encoder distribution q and the posterior distribution $p(Z|x_n)$ get closer, (resemble each other more), which is exactly what we wanted since we modeled the distribution q because we did not possess p, so q resembling p is what we want.

1.9 Question 1.9

Reconstruction loss In the loss term L^{rec} we see decoders log probability of reconstructing an x given the z sampled from the encoder distribution. We want to maximize this probability or minimize the negative expectation (loss) like given. Intuitively this loss penalizes the decoder network depending on how much of a different/wrong input it generated. This loss can be defines as L2, cross entropy etc. w.r.t the prior distribution we have assumed.

Regularization Minimizing L^{reg}, the KL divergence means that we are pushing the encoder probability distribution q to resemble target distribution (e.g. Gaussian) more and more by optimizing μ and Σ. But how is this term a regularizer? As this post [1] explains, e.g. a Gaussian pushes the encoder to, well, encode the input evenly around the center of the latent space. If it tries otherwise, i.e. put them further away or cluster them concentrated into an area, then it is penalized. Because it is *diverging* away from the prior that we have defined for the latent variable. In this way, this KL divergence term acts as regularization loss.

The post I cited above has pretty cool visualizations of what kind of a latent feature space you would end up with if you use just either one or both of the losses, [1].

1.10 Question 1.10

We can formulate the losses as below, which can be implemented in the same way, also shown below. Then as in the given formula $\mathcal{L}$, they are added and averaged.

$$\begin{aligned}
\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta\left(\boldsymbol{x}_n|Z\right)\right] && (6)\\
&= \text{Element-wise Binary Cross Entropy between input and output.} && (7)\\
\mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi\left(Z|\boldsymbol{x}_n\right)\|p_\theta(Z)\right) && (8)\\
&= \frac{1}{2}\sum_{j=1}^{J}\left(1+\log\left(\left(\sigma_j\right)^2\right)-\left(\mu_j\right)^2-\left(\sigma_j\right)^2\right) && (9)
\end{aligned}$$

$\mathcal{L}_n^{\text{recon}}$ `= (torch.sum(self.BCELoss(output, input),dim=-1))`
$\mathcal{L}_n^{\text{reg}}$ `= (- 0.5* torch.sum(1 + torch.log(std**2)-mu**2-std**2, dim=-1))`

1.11 Question 1.11

(a) The parameter ϕ represents the distribution $q_\phi(z|x)$ which we learn in place of the actual posterior $p(z|x)$. Here we use a Gaussian meaning ϕ is μ_ϕ and σ_ϕ. We need the gradient $\nabla_\phi\mathcal{L}$ to tie the gradients of the decoder and encoder network together, as we are performing the sampling in between networks. So we somehow need to define this gradient to properly back-propagate loss.

(b) The act of sampling is **non-differentiable**. Because the parameters ϕ (μ_ϕ, σ_ϕ) are changing every step, the sampled z is not a function, meaning its derivative is 0. We can not back propagate this.

(c) The trick basically rephrases z as a linear function. Instead of sampling z directly from a Gaussian: $z \sim \mathcal{N}(\mu_\phi, \sigma_\phi)$, we define ϵ which we always sample from a standard Gaussian: $\epsilon \sim \mathcal{N}(0,1)$ and use it to compute the *function* $z = \epsilon \cdot \sigma_\phi + \mu_\phi$, which is equivalent to sampling from the Gaussian directly, but it is also differentiable.

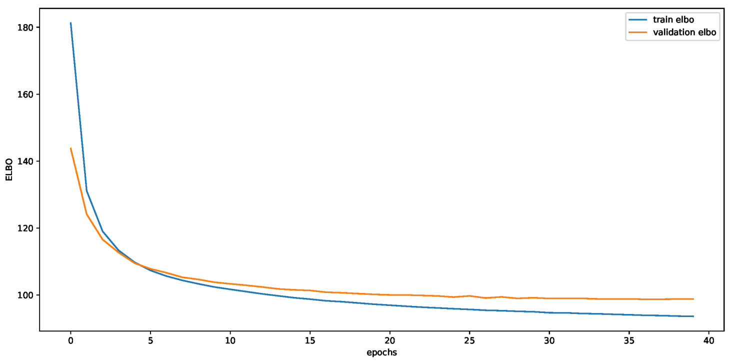
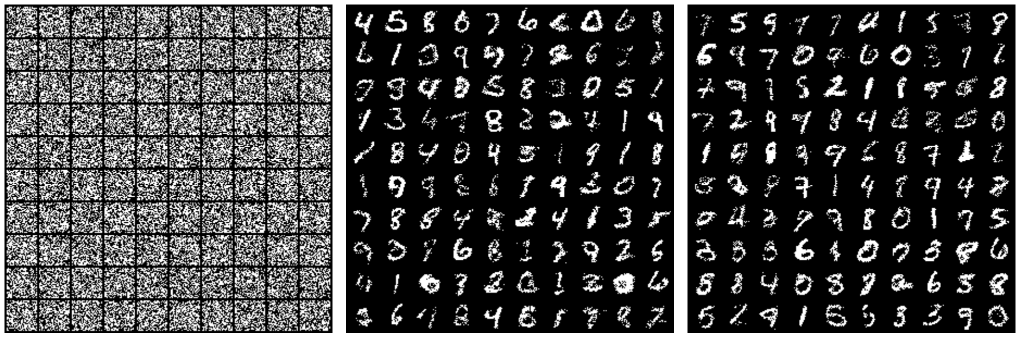


Figure 1: VAE ELBO loss over epochs.



(a) Before training (b) 20 Epochs (c) 40 Epochs

Figure 2: 100 sampled images at different stages in training VAEs. Visualized images are values sampled from the output of the network which are Bernoulli means.

1.12 Question 1.12

For the VAE implementation, I've used simple 2-layer networks, where the encoder is just 2 linear layers (both 500 hidden) followed by ReLUs, then finally the 2 linear layers which take the same final ReLU as input to produce mean and log-variance. The decoder similarly has 2 hidden linear layers and a final output linear layer followed by a sigmoid.

In the main VAE model, in a forward pass, we first encode the input and get μ & σ, which we use to sample ϵ from a standard normal distribution, and compute $z = \epsilon * \sigma + \mu$. We forward z through the decoder and get probability results, which we can refer to as Bernoulli means. Then we compute the ELBO using the losses we've defined before. That's basically it, nothing special or different in the rest.

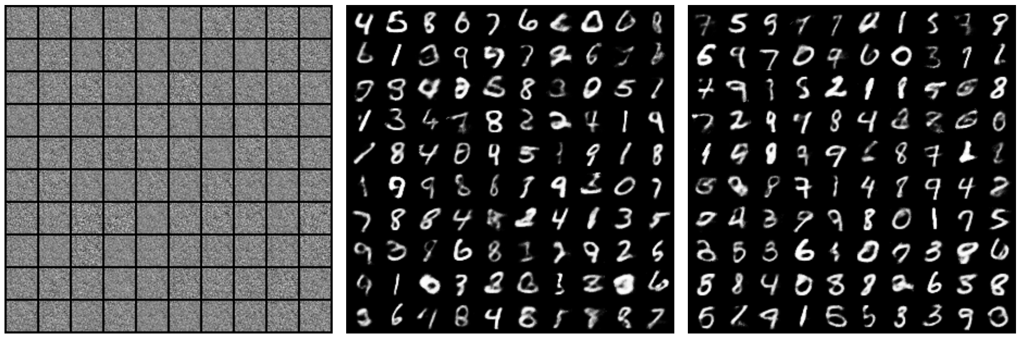
Here are some sources that helped me in my implementation, e.g. for the loss or manifold visualization etc, [4], [5].

1.13 Question 1.13

Estimated lower bound ELBO across epochs can be seen in figure 1. The values seem reasonable for negative ELBO considering it is recorded after learning for one epoch. At the beginning, before anything is learned, we expect random performance from the decoder $p(x|z)$, so a value around $784 \cdot log 0.5$. The value we see in the plot is a bit lower as learning has already progressed for 1 epoch.

1.14 Question 1.14

Sampled images during training can be found in figures 2 & 3. We can see that it gets much more precise from epoch 20 to epoch 40, though sampled images, 2, appear much more grainy/salty



(a) Before training (b) 20 Epochs (c) 40 Epochs

Figure 3: Output Bernoulli means used for sampling in figure 2.

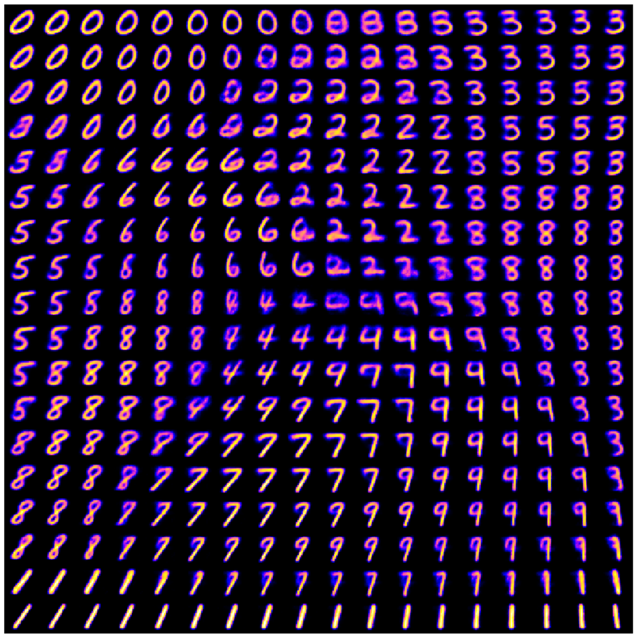


Figure 4: Data manifold of VAE with 2-dimensional latent space.

compared to e.g. GANs, which we'll see shortly. Direct output of the decoder looks better in this sense, 3.

1.15 Question 1.15

See figure 4. We can see a nice and smooth transition between digits, and a distribution. [5]

2 GANs

2.1 Question 2.1

The input for the generator is random noise. The output is an image which will initially also resemble random noise, but in time generator learns to turn the input noise z into a sensible fake image.

The discriminator takes images (both fake or real depending on which aspect we are training), and outputs a single value (sigmoid) indicating the probability of the input being fake/real. Depending on your network architecture all the images can be flattened (our case), or if you have a CNN they can be 2D & gray-scale or RGB etc.

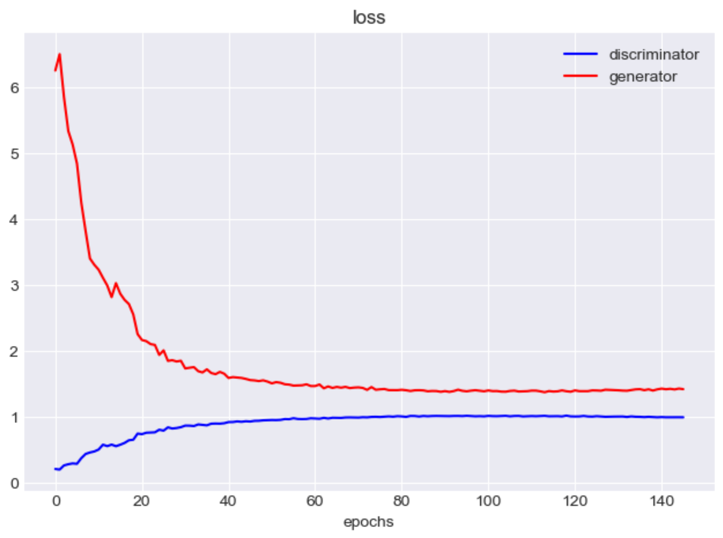


Figure 5: Generator vs. Discriminator loss in GAN training.

2.2 Question 2.2

- $\mathbb{E}_{p_{\text{data}}(x)}[\log D(X)]$: The term $D(X)$ shows certainty of the discriminator that X is real, i.e. we input X into D and get this probability. We can see in the expectation that the input X is the real data. Ideally, to have a decent discriminator we would want it to always classify truly real images as real $D(X) = 1$.

- $\mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))]$: The term $D(G(z))$ is the probability that D thinks a fake image generated by G is real. Naturally the complementary probability $1 - D(G(Z)))$ corresponds to D catching a fake, hence it wants to maximize this.

So to put it simply, D maximizes the probabilities of it thinking real images are real and fake images are fake. And the G minimizes the second term (which is the only term containing it), thus maximizing the chance of it fooling D.

2.3 Question 2.3

Assuming both of our networks are decent at their jobs, ideally at convergence, we would want the generator to fool the discriminator all the time. This means that the discriminator will be as good as random meaning both probabilities $D(G(z))$ and $D(x)$ will be 0.5. Thus we have: $V(D, G) = 2\log(0.5)$

2.4 Question 2.4

The term $\log(1 - D(G(Z)))$ expresses the probability of discriminator seeing that the input is fake. The generator minimizes this loss. However at the start of training the output of the generator is basically random, i.e. random noise input multiplied with un-learnt random weights. Hence it is very different than the real output and the discriminators job is really easy. If the discriminator is doing a great job $D(G(Z)) \approx 1$, meaning $\log(1 - D(G(Z))) \approx 0$. If the loss is close to zero0, there's a very little loss or nothing for the Generator to minimize/back-propagate, thus it doesn't learn.

To prevent this prematurely-great-Discriminator problem, we basically have to slow the discriminator down to allow generator to improve it self from random to sensible images. This can be done by having a more shallow network for D, or randomly changing some labels (real/fake) to fool D into thinking its doing a bad job, etc. Techniques used to improve training GANs can be tried, [3].

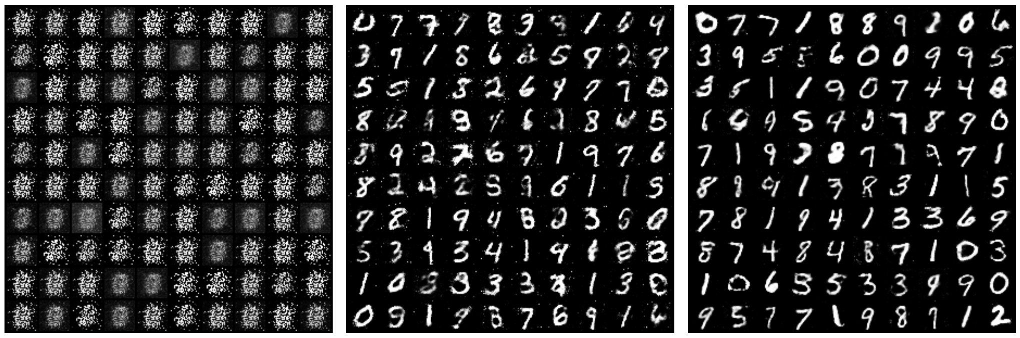

(a) 1K steps (b) 45K steps (c) 140K steps

Figure 6: 100 sampled images at different stages in training. Middle image is sampled a bit earlier than halfway for the sake of a better comparison. The input noise to generator is same for the shown samples.

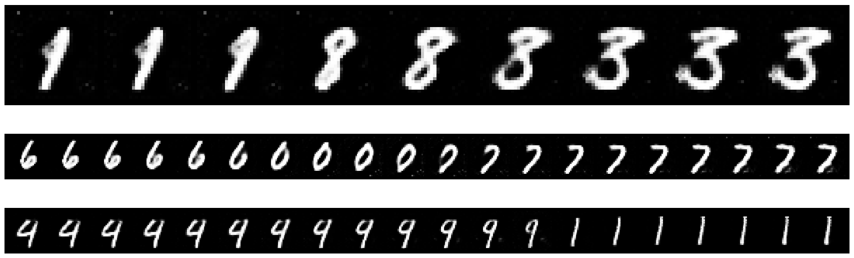


Figure 7: Interpolations between digits.

2.5 Question 2.5

I followed the baseline network architectures given, but then added some stuff following some tips [3]. I changed the output non-linearity of the Generator to Tanh which gave a significant improvement over previous results. Following this I also normalize the *real* images to between $[-1, 1]$ so that they're on the same scale as the Generator output. Finally, I added Dropout after every LeakyReLU in the Generator. There nothing added to the given Discriminator.

The discriminator minimizes the loss: `- log(real) + log(1-fake)`, while the generator minimizes: `- log(fake)` averaged over batches, where `real` and `fake` are the discriminators output probability for real and fake images indicating how confidently it thinks they're real.

2.6 Question 2.6

On figure 5, we see 100 images sampled at different stages in training. We can see that initially the GAN, even though it still produces random, learns to concentrate values in the center. Then we start to see some digit like formations, which are already very apparent in (b), and gets cleaner and more precise as generator gets better. We also see that generator captures different hand writing styles of the same digit in a very clean manner.

2.7 Question 2.7

Multiple interpolation results can be seen in figure 7. Implementation-wise we simply sample (normal) two distinct random noise vectors and get equally spaced vectors in between by using `np.linspace` on every dimension. We forward all the vectors as a batch through the generator. As a results we can see a smooth transition from one digit class to the other in all cases. Interestingly, we also see that some digits can act as transition digits as they posses properties similar to both ends of the transition.

3 Generative Normalizing Flows

For questions below, I mostly followed the theory from lectures and this blog-post.

3.1 Question 3.1

We see the multi-variate version below, where the determinant is of the Jacobian f w.r.t. z.

$$\mathbf{z} \sim \pi(\mathbf{z}),\ \mathbf{x} = f(\mathbf{z}),\ \mathbf{z} = f^{-1}(\mathbf{x}) \tag{10}$$

$$p(\mathbf{x}) = \pi(\mathbf{z}) \left| \det \frac{d\mathbf{z}}{d\mathbf{x}} \right| = \pi\left(f^{-1}(\mathbf{x})\right) \left| \det \frac{df^{-1}}{d\mathbf{x}} \right| \tag{11}$$

$$\log p(x) = \log \pi\left(f^{-1}(\mathbf{x})\right) + \sum_{l=1}^{L} \log \left| \det \frac{df^{-1}}{d\mathbf{x}} \right| \tag{12}$$

3.2 Question 3.2

X and Z need to have the same dimensionality otherwise, the Jacobian, we've shown above, won't be a square matrix, hence won't have a determinant. Intuitively, we cant think of the abs (determinant) matrix as controlling how much we expand or shrink space. For instance in our MNIST implementation both will have dimensionality 784.

Another constraint on f is of course that it must be invertible.

3.3 Question 3.3

We want f to be *easily* invertible and its Jacobian determinant to be easy to compute. Both of these depend on the smart selection of the transformation function f, otherwise can end up being expensive to compute. For example, if we have a triangular Jacobian, the determinant is simply the product of the diagonal elements.

3.4 Question 3.4

As pointed out by Ho et al. [9], modelling discrete data with a continuous density, causes the probability mass to get concentrated around discrete data-points, thus a distribution-like structure isn't captured, which is what we want to learn. To avoid this problem we can apply Dequantization, i.e. convert the discrete into a continious distribution. Ho et al. listed two different ways to do this, uniform and variational. Lets handle one of them in more detail.

Uniform dequantization simply adds uniform noise $u \sim [0,1)^D$ over the width of bins in between discrete data points, $x' = x + u$. Ho et al. note that then the model now is maximizing a lower bound on the log-likelihood of modelling discrete data.

3.5 Question 3.5

Input of the network during training is simply the image, the output of the network is the learned log probability of the input image ($logp(x)$). At inference time, if the purpose of the inference is to get probability, it can be retrieved the same way as training. If it is meant by inference that we are **sampling**, then as the input we first sample a random vector (from a standard Gaussian) and propagate it through our network in reverse, i.e. inverse flow, and get a new generated image.

3.6 Question 3.6

Training. We can think of the input as mini-batches of vectorized real images.

1. Forward input images through the flow. To get into more detail, let's follow the model Real-NVP, [8].
   - The flow consists of multiple alternating Coupling layers.

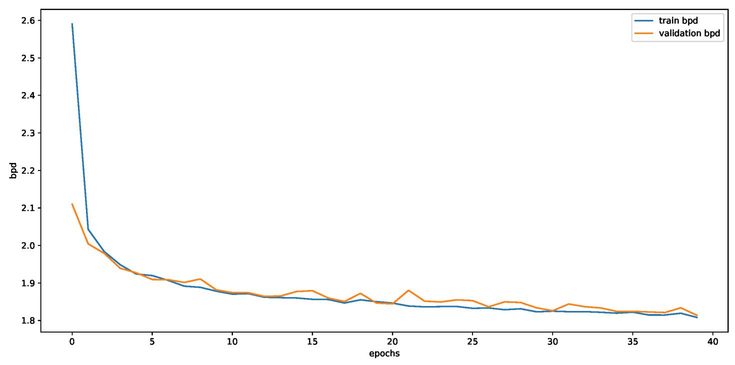


Figure 8: Normalizing Flows mean bits per dimension across epochs

- A coupling layer is a network that learns *scale* and *translation* parameters from half of its input z, and applies the learned parameters to the other half, and does this by using a mask.
- As a result, a coupling layer outputs a semi-altered version of its input z.
- z is passed through many such alternating layers.

2. During the forward through flow, we keep track of Jacobian determinant as well. Thus flow outputs are that and z.
3. Finally as output of our model, we compute $\log p(x)$, similar to how we have defined in equation 12.
4. Loss of a forward pass is negative log likelihood, namely $-\log p(x)$ mean over mini-batch.
5. We train by back-propagating this loss.

Inference (Sampling).

1. Sample vector z (e.g. standard Gaussian).
2. Forward through the flow in reverse.
   - This means we propagate through coupling nets in reverse.
   - It also means that in the coupling nets we are computing the inverse transformation (of scale & translation).
3. The output of the inverse flow is the new generated image.

3.7 Question 3.7

Implemented as instructed following real-NVP, [8]. I didn't add anything special besides the necessary parts.

3.8 Question 3.8

See figure 8 for bits per dimension across epochs and figure 9 for the samples acquired in different epochs. A notable observation is that in contrast to other models we have implemented, this one seems to be learning/focusing on certain digits much more. This is best observable through a GIF of all samples.

4 Conclusion

4.1 Question 4.1

Recap of generative results. Lets start of by qualitatively comparing generated final images. In VAE's, 3, we see that it captures the numbers quite well in general, but there are still a bunc of images

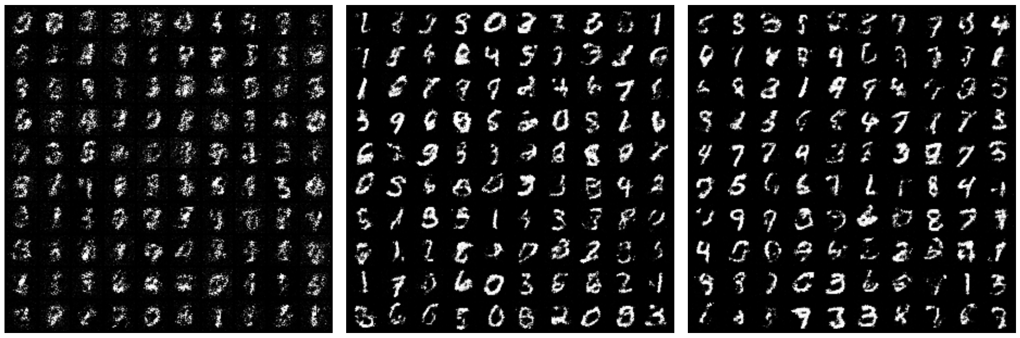

(a) 1 epoch (b) 20 epochs (c) 40 epochs

Figure 9: 100 sampled images at different stages in training NF model.

that seem really faded or unsure. This weakness if especially present when we sample from these mean images to get figure 2 where these unsure digits appear the most grain-y. In normalizing flows we also see a similar pattern 9, where some digits seem very clear and precise while others are almost not visible. In contrast to these, GANs 6, seem to generate very clean numbers, where we can spot different hand-writing styles as well. We can also see that there's one or two troubled digits as well but in that case they still resemble just very poorly written numbers.

Complexity of understanding Conceptually GANs are easy to understand and implement, while explicit density models; VAEs and NFs are relatively harder to grasp and has more math to work out.

Computation & Training GANs are infamously harder to train, and may need many tricks to accomplish depending on the task. Here we have also trained GANs for 150 epochs while the others just 40.

Final thoughts GANs are better if the sole purpose of the task is to generate clean and realistic images. Explicit density models VAEs and NFs are better for tasks where we may need a density function. With a density, we can for instance have an idea how likely a generate sample is, it is also more open to handling problematic data (e.g. missing). So they are more open to statistical analysis. If we where to make a preference between VAEs and NFs, NFs are said to adapt better to the data distribution while also not scaling well to higher dimensionality. Today, VAEs seem to be not preferable in practice.

References

[1] Medium post, Intuitively Understanding Variational Autoencoders, link

[2] Auto-Encoding Variational Bayes, Diederik P Kingma, Max Welling, arXiv:1312.6114

[3] How to Train a GAN? Tips and tricks to make GANs work, ganhacks

[4] ML-cheat-sheet implementation of VAEs, ml-cheat-sheet-vae

[5] Visualizing MNIST with a Deep Variational Autoencoder kaggle-vae

[6] Generative Adversarial Nets in TensorFlow, gan-tensorflow

[7] Carl Doersch, Tutorial on Variational Auto Encoders, link

[8] Laurent Dinh, Jascha Sohl-Dickstein, Samy Bengio, Density estimation using Real NVP, ICLR 2017, arXiv:1605.08803

[9] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. CoRR, abs/1902.00275, 2019. link

[10] The advantages of normalizing flow over VAEs

[11] The-difference-between-a-Variational-Autoencoder-VAE-and-an-Autoencoder