Assignment 3. Deep Generative Models

Maurice Frank
11650656
maurice.frank@posteo.de
Code: github

1 Variational Auto Encoders

We have:

$\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$ the dataset

$\boldsymbol{x}_n \in \{0,1\}^M$ one datapoint, from Bernoulli distribution

1.1 Variational autoencoders vs standard autoencoders

Let us first note down the two objectives of the VAE and a standard sparse autoencoder [1], respectively:

$$\mathbb{E}_{q(z|\boldsymbol{x}_n)}[\log p(\boldsymbol{x}_n|Z)] - D_{KL}(q(Z|\boldsymbol{x}_n)||p(Z)) \tag{1}$$

$$||D_{AE}(E_{AE}(\boldsymbol{X})) - \boldsymbol{X}||^2 + \lambda||E_{AE}(\boldsymbol{X})||_0 \tag{2}$$

For the sparse autoencoder we use D_{AE}, E_{AE} for the Decoder model and Encoder model and λ is our regularization hyperparameter. The main similarity between both is that they have the encoder and decoder pair. The encoder encodes the data into a smaller latent space and the decode generates samples from this space back into our datasets space X.

a) How do they differ in their main function?

We see that both methods vary considerably in their main optimization objective. The VAE objective includes modeling the intractable encoding distribution$p(Z|X)$ through the surrogate $q(Z|X)$. The standard autoencoder on the other side does not actually model the encoding distribution but only has the reconstruction capabilities of those encodings as its objective.

b) Is the standard autoencoder a generative model?

The standard autoencoder is not a generative model as one cannot sample from the implicit latent space. We do have the latent representation and thus can vary the input into the decoder but this does not relate to sampling from the latent manifold of all possible encodings. We can sample from the space the encodings live in but as we do not model the actual distribution this is not a generative model.

c) Can the VAE be a drop-in replacement for a normal autoencoder?

The VAE can be a drop-in replacement for the standard autoencoder in respective applications. Autoencoders are used for compression as they can reduce the input data into the

smaller informative latent space. The VAE does do they same and can be used for the same. Further autoencoders can denoise data by mapping the noisy data close to the noise-less data in the latent space. Again the VAE can do the same. Generally the addition of the VAE is enforcing the continuity of the latent space by introducing $q(\cdot)$. While this opens up new (see generative) applications it does not restrictive the former.

d) Why does VAE generalize better?

The standard autoencoder is missing the is not modelling the encoding distribution. In the VAE our $q(Z|X)$ forces the latent space to be continuos. As such interpolating in the latent space does correspond to interpolating in the data space. During training we therefore build a latent space that is better capable to handling unseen data and can therefore also generalize beyond the trainings data.

1.2 How to sample from $p(Z)$?

We sample using ancestral sampling. First we sample $\boldsymbol{z}_i \sim p(\boldsymbol{z}_n) = \mathcal{N}(0, \mathbb{1}_D)$. We put this sample into the Decoder $f_\theta(\cdot)$ to get our parameters for the bernoulli distribution. Than we sample each pixel from each bernoulli $\text{pixel}_m \sim \text{Bern}(\boldsymbol{x}_n^{(m)}|f_\theta(\boldsymbol{z}_n)_m)$ to get the sample from the joint distribution.

1.3 Why is the assumption of $p(Z)$ not restrictive?

Our assumption that $p(Z)$ is a standard-normal distribution does not hinder us from learning a complex manifold for X as we transform it using $f_\theta(\cdot)$. Thus the complexity lies in the mapping from latent z-space and x-space.

1.4 Evaluating $\log p(\boldsymbol{x}_n)$

a) Approximating using Monte-Carlo

We can sample from our distribution of $\boldsymbol{z}_n$ to compute the integral for one $\boldsymbol{x}_n$:

$$\log p(\boldsymbol{x}_n) = \log \int p(\boldsymbol{x}_n|\boldsymbol{z}_n)p(\boldsymbol{z}_n)\, d\boldsymbol{z}_n \tag{3}$$

$$\approx \log \frac{1}{n}\sum_{i=1}^{n} p(\boldsymbol{x}_n|\boldsymbol{z}_i), \quad \boldsymbol{z}_i \sim p(z) \tag{4}$$

b) Why is Monte-Carlo integration here inefficient?

We sample from $p(Z)$. As we nee the integral for each datapoint this is inefficient with any non trivial dimensionality in Z. The precision of our density estimation of $p(\boldsymbol{x}_n|\boldsymbol{z}_n)$ decreases exponentially as the dimension of $\boldsymbol{x}_n$ increases. As such we would need to increase the number of samples exponentially with higher dimension of $\boldsymbol{z}_n$ to keep the estimation precise. In the real world the hidden embeddings often has hundreds of dimensions. A density dimension that we would have to do for every single input in every inference step is too inefficient to be usable.

1.5 Kullback-Leibler divergence of Gaussians

We assume: $q = \mathcal{N}(\mu_q, \sigma_q^2), \quad p = \mathcal{N}(0, 1)$

a) Two examples of KL-divergence

Checking with the closed form solution in Section b) we see as expected that the divergence is zero for $\sigma_q = 1, \mu_q = 0$ and increases for any deviation from either of those values:

$$Big: \quad (1 + 10^5, 10^5) \implies D_{\text{KL}}(q||p) = 1.0 \times 10^{10} \tag{5}$$

$$Small: \quad (1 + 10^{-5}, 10^{-5}) \implies D_{\text{KL}}(q||p) = 1.49 \times 10^{-10} \tag{6}$$

b) Closed-form solution of the KL-divergence

$$q(x) = \frac{1}{\sqrt{2\pi}\sigma_q} \exp\left[-\frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2\right] \tag{7}$$

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{1}{2}x^2\right] \tag{8}$$

$$D_{\mathrm{KL}}(q||p) = -\mathbb{E}_{q(x)}\left[\log \frac{p(x)}{q(x)}\right] \tag{9}$$

$$= -\mathbb{E}_{q(x)}\left[\log\left(\frac{1}{\sqrt{2\pi}}\right) - \frac{1}{2}x^2 - \log\left(\frac{1}{\sqrt{2\pi}}\right) + \log\sigma_q + \frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2\right] \tag{10}$$

$$= -\mathbb{E}_{q(x)}\left[\frac{1}{2}\left(\left(\frac{x-\mu_q}{\sigma_q}\right)^2 - x^2\right)\right] - \mathbb{E}_{q(x)}\left[\log\sigma_q\right] \tag{11}$$

$$= -\frac{1}{2}\mathbb{E}_{q(x)}\left[\left(\frac{x-\mu_q}{\sigma_q}\right)^2 - x^2\right] - \log\sigma_q \tag{12}$$

$$= -\frac{1}{2}\left(\frac{1}{\sigma_q^2}\mathbb{E}_{q(x)}[(x-\mu_q)^2] - \mathbb{E}_{q(x)}[x^2]\right) - \log\sigma_q \tag{13}$$

$$= -\frac{1}{2}\left(\frac{\sigma_q^2}{\sigma_q^2} - \sigma_q^2 - \mu_q^2\right) - \log\sigma_q \tag{14}$$

$$= \frac{\sigma_q^2 + \mu_q^2 - 1}{2} - \log\sigma_q \tag{15}$$

1.6 Why is ELBO a lower bound?

The evidence lower bound (ELBO) for our VAE is given by:

$$\mathbb{E}_{q(z|\boldsymbol{x}_n)}[\log p(\boldsymbol{x}_n|Z)] - D_{\mathrm{KL}}(q(Z|\boldsymbol{x}_n)||p(Z)) \tag{16}$$

We want to maximize our data model $p(\boldsymbol{x}_n)$, or written including the ELBO:

$$p(\boldsymbol{x}_n) = \mathbb{E}_{q(z|\boldsymbol{x}_n)}[\log p(\boldsymbol{x}_n|Z) - D_{\mathrm{KL}}(q(Z|\boldsymbol{x}_n)||p(Z))] + D_{\mathrm{KL}}(q(Z|\boldsymbol{x}_n)||p(Z|\boldsymbol{x}_n)) \tag{17}$$

As KL divergence is strictly > 0 we directly see that the ELBO is a lower bound for our actual objective. Now the distance from the lower bound to the actual evidence is the divergence of our surrogate approximate posterior $q(Z|\boldsymbol{x}_n)$ from the actual posterior $p(Z|\boldsymbol{x}_n)$. The lower bound always stays a lower bound so maximizing the lower bound does also maximize the actual objective.

1.7 Why do we optimize the lower bound instead of the log-probability?

We must optimize the lower bound as we cannot compute $D_{\mathrm{KL}}(q(Z|\boldsymbol{x}_n)||p(Z|\boldsymbol{x}_n))$. Computing this divergence would mean we need to compute $p(z|\boldsymbol{x}_N)$. If we could do this we would not have introduced $q(z|\boldsymbol{x}_n)$.

1.8 What can happen if we push ELBO up?

Basically the two things that can happen is either our data log probability $\log p(\boldsymbol{x}_N)$ rises or the divergence between $q(Z|\boldsymbol{x}_n)$ and $p(Z|\boldsymbol{x}_n)$ gets lower. Both ways are good for us. In the first case the likelihood of the trainings data rises meaning we get a better ML model. The reconstruction gets better. In the second case our surrogate gets more similar to the actual encoding which means we improve our embeddings.

1.9 How can we name the parts of the loss of an VAE?

We can write the loss of the VAE as the per-sample sum of the *reconstruction* and the *regularizer* loss:

$$\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)] \tag{18}$$

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

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}(\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}) \tag{20}$$

As written before $\mathcal{L}_n^{\text{recon}}$ gives us the negative data likelihood of the input sample $\boldsymbol{x}_n$ using its embedding from the encoder $q_\phi(\cdot)$. In other words this gives us the reconstruction probability of $\boldsymbol{x}_n$ using the decoder $p_\theta(\cdot)$ under an already set embedding for $\boldsymbol{x}_n$, but negative (to get a loss).

Our second objective wants to minimize the divergence of $q_\phi(Z|\boldsymbol{x}_n)$ with respect to $p_\theta(Z)$. Now that means that every posterior of the encoders input wants to be similar to the distribution of our latent embeddings Z. Of course this is impossible as in the case of no divergence all posteriors would be the same. If the model wants to have reconstruction capabilities while minimizing the divergence of the embedding posteriors it is best to map similar inputs to similar posteriors as that allows them to be larger, hence less diverged from the $p(Z)$. Further this will lead to a distribution of posteriors as such that:

$$D_{\text{KL}}\left(\sum_n q_\phi(Z|\boldsymbol{x}_n)||p_\theta(Z)\right) \approx 0 \tag{21}$$

This is exactly what we want as we thus make a partition of the embedding manifold based on latent feature similarity (in the case of MNIST, same number are close).

1.10 Writing down the complete loss

We assume for our encoder:

$$q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n) = \mathcal{N}(\boldsymbol{z}_n|\mu_\phi(\boldsymbol{x}_n), \text{diag}(\Sigma_\phi(\boldsymbol{x}_n))) \tag{22}$$

$$\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)] \tag{23}$$

$$\approx -\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n) \quad \text{with } \boldsymbol{z}_n \sim q_\phi(Z|\boldsymbol{x}_n) \tag{24}$$

$$\mathcal{L}_n^{\text{reg}} = D_{\text{KL}}(q_\phi(Z|\boldsymbol{x}_n)||p_\theta(Z)) \tag{25}$$

$$= \frac{\Sigma_\phi(\boldsymbol{x}_n)^2 + \mu_\phi(\boldsymbol{x}_n)^2 - 1}{2} - \log \Sigma_\phi(\boldsymbol{x}_n) \tag{26}$$

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}(\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}) \tag{27}$$

$$= \tag{28}$$

TODO: FiNISH this and make loss multivariate

1.11 The reparametrization trick

a) Why do we need $\nabla_\phi \mathcal{L}$?

We need the gradient of our loss with respect to the parameters ϕ of our encoder $q_\phi(\cdot)$ for backpropagation. To update the weights ϕ we need to know their influence on the loss.

b) Why can we not compute $\nabla_\phi \mathcal{L}$?

We cannot compute this gradient because we sample from $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$. As we sample from we cannot attribute the effect of this inference step to and of the weights ϕ in $\mu_\phi(\cdot)$ or $\Sigma_\phi(\cdot)$.

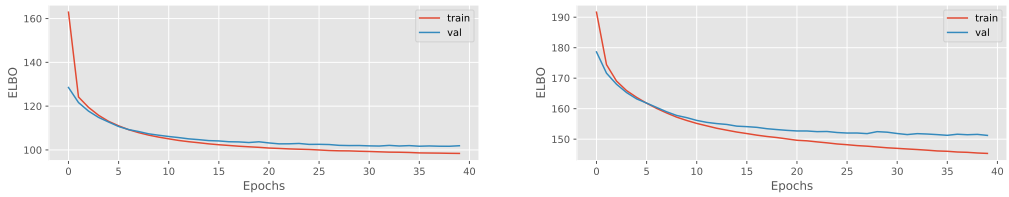

Figure 1: The ELBO curves of the trainings and validation set for encoding sizes z of *left* 20 and *tight* 2.

Figure 2: Samples drawn from the VAE with encoding size 20 at epoch $0, 10, 20, 30, 40$ (from *top left* to *bottom right*). At each epoch we sample 16 samples from $p(Z)$.

c) How does the reparametrization trick solve that problem?

The reparametrization trick solves this problem by taking the stochastic non-deterministic sampling operation out of the encoder. This uses the special property of normal distribution, that:

$$\mathcal{N}(z_n|\mu_\phi(x_n), \text{diag}(\Sigma_\phi(x_n))) = \text{diag}(\Sigma_\phi(x_n)) \cdot \mathcal{N}(0, \mathbb{1}_D) + \mu_\phi(x_n) \tag{29}$$

The only remaining stochastic process $\mathcal{N}(0, \mathbb{1}_D)$ is therefore than outside the parameterized model and we can compute the gradient.

1.12 Building a VAE

See the code in `code/vae.py`

1.13 ELBO during training

See Figure 1 for the plots of the ELBO curves.

1.14 Samples during training

See Figure 2 for samples of the VAE during training.

1.15 Manifold during training

See Figure 3 for three manifolds during training.

2 Generative Adversarial Networks

We have:

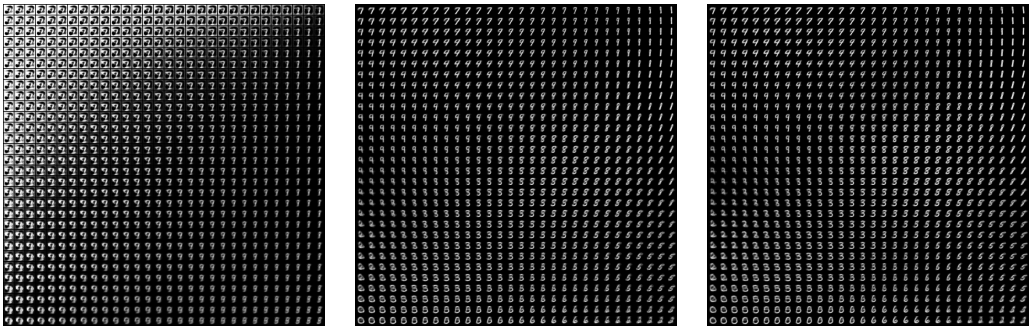

Figure 3: The manifold of the VAE with encoding size 2 at epoch $0, 20, 40$.

$D(\cdot)$ our discriminator function/network

$G(\cdot)$ our generator function/network

$p(\boldsymbol{z})$ the noise distribution, here $\sim \mathcal{N}(0, \mathbb{1}_d)$

$\mathcal{D} = \ldots, i_i, \ldots$ dataset of images, each $i_i \in \mathbb{R}^{h \times w}$

and our GAN objective:

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[\log D(X)] + \mathbb{E}_{p_z(\boldsymbol{z})}[\log(1 - D(G(Z)))] \tag{30}$$

2.1 Inputs and outputs of $D(\cdot)$ and $G(\cdot)$

For the Generator we have a randomly sampled input of size d and an output in image size.

$$input_G \sim p(\boldsymbol{z}) = \mathcal{N}(0, \mathbb{1}_d) \in \mathbb{R}^D \tag{31}$$

$$output_G \in \mathbb{R}^{h \cdot w} \tag{32}$$

For the Discriminator we have an image sized input and a single output.

$$input_D \in \mathbb{R}^{h \cdot w} \tag{33}$$

$$output_D \in \mathbb{R} \tag{34}$$

2.2 What are the parts of the GANs objective?

The left part $\mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[\log D(X)]$ is the data likelihood of the trainings data under the discriminator. Basically it maximizes the discriminators output for samples in the ground truth data set.

The right part $\mathbb{E}_{p_z(\boldsymbol{z})}[\log(1 - D(G(Z)))]$ gives the inverse score of the discriminator for sampled $\boldsymbol{z}$ that were put through the generator. The generator wants to minimize this as than outputs are similar for p_{data} and p_z. The discriminator wants to maximize to have reciprocal outputs for the two distributions.

2.3 How does a converged objective look like?

If training is converged than the generator produces perfect samples which mean that the distribution of generated samples is equal to the trainings set:

$$p_{\text{data}}(\boldsymbol{x}) = G(\boldsymbol{z}) \quad \boldsymbol{z} \sim p_z(\boldsymbol{z}) \tag{35}$$

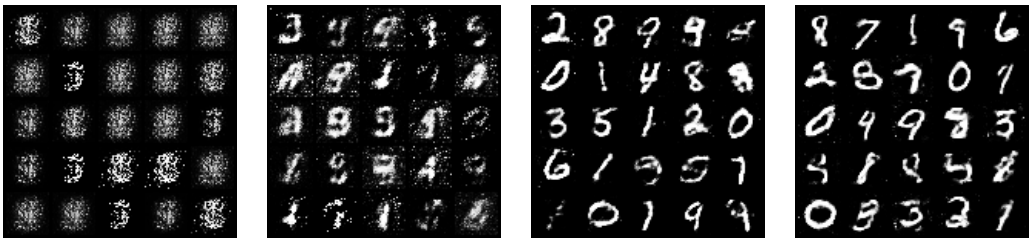

Figure 4: Samples from our GAN implementation during training. From *left* to *right*: 2 epochs, 8 epochs, 100 epochs, 200 epochs.

at that point a perfectly discriminator is no different from randomness as it is impossible to distinguish the two datasets:

$$\mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[D(\boldsymbol{x})] = \mathbb{E}_{p_z(\boldsymbol{z})}[D(G(\boldsymbol{z}))] = \frac{1}{2} \tag{36}$$

$$\implies \tag{37}$$

$$V(D,G) = \log\frac{1}{2} + \log\frac{1}{2} \tag{38}$$

$$= -\log 4 \tag{39}$$

2.4 Why can $\log(1 - D(G(Z)))$ be problematic?

In the beginning $G(\cdot)$ and $D(\cdot)$ are shit. But than in early training it is easier for the discriminator to differ between real and fake than it is for the generator to fool it. Hence in our left side objective we get near 0 early on. This saturating effect makes backpropagation for the generator difficult with the losses near zero. The solution to this problem is fairly simple. We separate the two objectives into:

$$\mathcal{L}_D = -\mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[\log D(X)] - \mathbb{E}_{p_z(\boldsymbol{z})}[\log(1 - D(G(Z)))] \tag{40}$$

$$\mathcal{L}_G = -\mathbb{E}_{p_z(\boldsymbol{z})}[\log(D(G(Z)))] \tag{41}$$

With the changed objective for the generator we averted this problem and as we will update the two models consecutively its is not a problem that we formulated separate objectives.

2.5 Building the GAN

See code in `code/gan.py` for the implementation.

We build the Generator G as follows: a linear layer from latent space into a hidden size $h = 128$ followed by three linear layers with adjacent leaky ReLU (with $\alpha = 0.2$) and batch normalization, respectively. Each linear layer doubles the hidden size. The final linear layer projects down to the inputs image size and we squash the outputs with a tanh.

We build the discriminator D as two linear layers again each followed by a leaky ReLU ($\alpha = 0.2$) and a dropout layer ($p = 0.3$) for better regularization. The first takes the input size down to $h \cdot 4$ and subsequently to the half. Finally we have linear classifier with one output and squash this with a sigmoid.

We use optimizer Adam for both models and a fixed learning rate of $\eta = 5e - 4$. We separate the objectives for better stability as explained before and we sample from G for each objective again.

2.6 Sampling from GAN

Refer to Figure 4 for samples taking during the training of our GAN.

2.7 Interpolation using the GAN

TODO: INTERPOLATION FIGURE

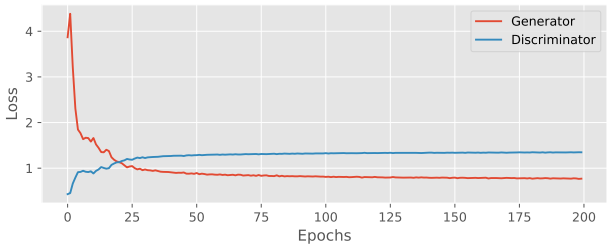

Figure 5: The loss of the two models D and G of the GAN during training.

3 Generative Normalizing Flows

3.1 Update equation for the *multivariate* flow

In our flow model we are using an invertible function $f(\cdot) : \mathbb{R}^m \to \mathbb{R}^m$ to map an input $\boldsymbol{x}$ to its next step $\boldsymbol{z} = f(\boldsymbol{x})$ (therefore $x = f^{-1}(\boldsymbol{z})$). The transformation $f(\cdot)$ changes our density manifold $p(X)$ of the input. Now this change can be seen locally as contractions and stretching of the manifold (like pulling around the manifold, intuitively). The local change of our transformation is exactly its Jacobian $\frac{\partial f}{\partial \boldsymbol{x}^T}$. The change in local volume (the contractions) is the determinant of a transformation and therefore globally the determinant of the Jacobian $\det \frac{\partial f}{\partial \boldsymbol{x}^T}$. Using this we can write the updated density as:

$$p(\boldsymbol{x}) = p(\boldsymbol{z}) \cdot |\det \frac{\partial f}{\partial \boldsymbol{x}}| \tag{42}$$

For the application of normalizing flows we will chain multiple such mapping to get the final $p(\boldsymbol{z})$. We describe the intermediate states with the hidden states $\boldsymbol{h}_i$. One transformation is then $\det \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}^T}$. And thus we can write the complete transformation as:

$$\log p(\boldsymbol{x}) = \log p(\boldsymbol{z}) + \sum_{l=1}^{L} \log |\det \frac{\partial \boldsymbol{h}_l}{\partial \boldsymbol{h}_{l-1}^T}| \tag{43}$$

We use the logarithm for numerical stability as we than have a sum instead of a product.

3.2 How are the formulas for the multivariate case constrained?

As given above the function $f(\cdot)$ has to map to same dimensionality as the inputs. Otherwise the determinant would be zero and we could not find the inverse. The second constraint is that we have to be actually able to compute the determinant of the Jacobian.

3.3 What computational problem might arise with this formulation?

The computational problem with the naïve approach is that we need to compute the determinant of the Jacobian. Computing the determinant of the Jacobian is at least $\mathcal{O}(d^3)$ [2] (where d is the size of the hidden units $\boldsymbol{h}$) and we have to do this L times (for each layer).

3.4 What changes if the inputs are discrete not continuos?

Image pixels have discrete values (mostly from the 8-bit intensity set). Thus our original image distribution $p(X)$ is equal to the sum of Dirac delta functions. If we now apply our density contraction transformation on this we will still end up with a set of dirac deltas. This means our embedding space $p(Z)$ is also discrete just like a standard autoencoder. We do want a continuous regularized embedding space where we can interpolate.

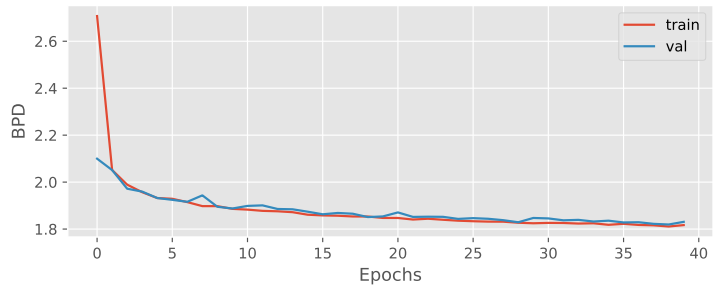

Figure 6: The average bits per dimension during training of the normalizing flow model.

One solution to this problem from [3] is to add some noise to out discrete input labels. In our case we would add some noise $\sim \mathcal{N}(0, 1)$ on our image pixels. The variance acts as a regularization hyperparameter.

3.5 What is the input and output of the flow model?

The input to the flow based model is always of the same size (let different batch sizes). For training we put in the images plus the element wise gaussian noise. For inference we will got through the computational graph in reverse but because we have the bijective mappings our inputs are still of the image's size. Now of course for sampling we will sample from the prior which in our case is just from a normal Gaussian.

3.6 What are the steps for training a flow model?

1. We build a list of bijective mappings. As we have to compute the determinant of the Jacobian we actually estimate two projections instead of one by masking the inputs into triangular matrix (a checkerboard) and its reverse.
2. We take a batch of images $\boldsymbol{x}$ and preprocess them by normalizing and adding noise for regularization.
3. We put the $\boldsymbol{x}$ through the chain of bijective functions. We build up the hidden representation $\boldsymbol{h}_i$ and add up the log of the Jacobian's determinant.
4. The resulting $\log p(\boldsymbol{x})$ from the sum of the final hidden state and the final determinant is our objective we want to maximize.
5. We use backpropagation to estimate the weight gradients and update the functions parameters with a suitable optimizer (in our case Adam).

3.7 Implementation of the flow-based model

See the code in `code/nf.py` for the implementation.

3.8 Bits per dimension during training

4 Conclusion

TODO: Conclusion

References

[1] Carl Doersch. Tutorial on Variational Autoencoders. URL http://arxiv.org/abs/1606.05908.

[2] Danilo Jimenez Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. URL http://arxiv.org/abs/1505.05770.

[3] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design. URL http://arxiv.org/abs/1902.00275.