# Assignment 2: Recurrent Neural Networks and Graph Neural Networks

**Steven van de Graaf**
12234036
steven.vandegraaf@student.uva.nl

All code and output for this assignment are available here.

## 1 Variational Auto Encoders

### 1.1 Latent Variable Models

**Question 1.1**

In short, a VAE learns the parameters of a probability distribution (two vectors of size $n$, one for the means $\mu$, and one for the standard deviations $\sigma$) that represents the data in the latent space, rather than a compressed representation of the data (an encoding vector of size $n$). As such, a VAE is able to sample from its probability distribution, thereby generating new data.

1. *Are they different in terms of their main function? How so?*
   I would say yes. While both allow for data compression in the latent space, the VAE adds the ability to generate new data by sampling from its probability distribution.
2. *A VAE is generative. Can the same be said of a standard autoencoder? Why or why not?*
   No, a standard autoencoder is not generative. While, if trained right, it will be able to reconstruct the original data (with some error) from its encoded representation, it is not able to generate new, unseen data, while a VAE is able to do so.
3. *Can a VAE be used in place of a standard autoencoder?*
   Yes. A VAE will be able to generalize better than a standard autoencoder, but for some tasks (like compression) it might not be the best fit, per se.
4. *Generative models need to be able to generalize. What aspect of the VAE missing in the standard autoencoder enables it to be generative.*
   It's precisely the probability distribution (of which the parameters are learned during training) from which samples are drawn in order to generate the data, which allows the VAE to generalize. Instead of learning representations for the data points already seen, the VAE learns the underlying (latent) distribution instead, therefore extending itself well to new data points not in the training set.

### 1.2 Decoder: The Generative Part of the VAE

**Question 1.2**

*Ancestral sampling* entails sampling variables in topological order: First one samples the variables with no parents, then sampling the next generation by conditioning on the parent values, andsoforth.

In our case, in order to sample a datapoint $\mathbf{x}_n$ from its distribution:

$$p(\mathbf{x}_n \,|\, \mathbf{z}_n) = \prod_{m=1}^{M} \text{Bern}\left(\mathbf{x}_n^{(m)} \,|\, f_\theta(\mathbf{z}_n)_m\right), \tag{1}$$

Submitted to the Spring 2019 Amsterdam Deep Learning course

we first have to sample its parent, the latent variable $\mathbf{z}_n$, from its respective distribution:

$$p(\mathbf{z}_n) = \mathcal{N}\big(0, \mathbf{I}_D\big). \tag{2}$$

**Question 1.3**

The assumption that the latent variable $\mathbf{z}_n \sim \mathcal{N}\big(0, \mathbf{I}_D\big)$ follows a standard-normal distribution is not restrictive in practice, because we can map it to the probability distribution of the data using a complex, parameterized function $f_\theta(\cdot)$ (an NN in our case), which can be learned through (stochastic) gradient descent.

**Question 1.4**

(a) Monte-Carlo integration is an estimator which uses random sampling to approximate the integral. Under i.i.d. assumptions, we can draw a large number $M$ of samples from the distribution of our latent variable $\mathbf{z}^{(m)}$, after which we can sample from the distribution of our data $\mathbf{x}^{(m)}$:

$$\begin{aligned}\sum_{n=1}^{N}\left(\log\left(\int p(\mathbf{x}_n \,|\, \mathbf{z}_n)p(\mathbf{z}_n)d\,\mathbf{z}_n\right)\right) &\approx \sum_{n=1}^{N}\left(\log\left(\frac{1}{M}\sum_{m=1}^{M} f_\theta(\mathbf{z}^{(m)})\right)\right) \\ &= \sum_{n=1}^{N}\left(\log\left(\frac{1}{M}\sum_{m=1}^{M} \mathbf{x}^{(m)}\right)\right)\end{aligned} \tag{3}$$

(b) It is inefficient, because it might take many samples (large $M$) $\mathbf{z}^{(m)}$ to estimate the integral. Furthermore, by increasing the dimensionality of $\mathbf{z}$, the sparsity of $p(\mathbf{x} \,|\, \mathbf{z})$ is increased also, as illustrated by Fig. 1.

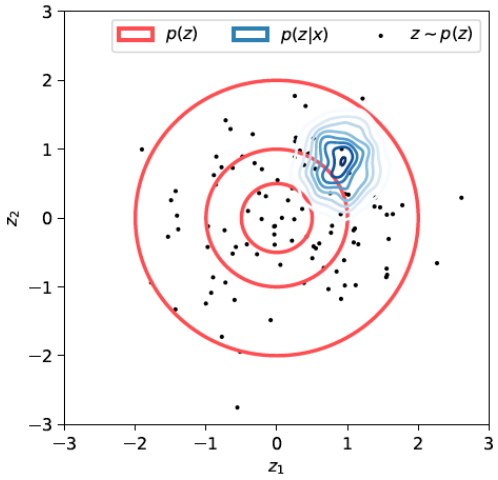



Figure 1: Plot of the latent space, including the prior $p(\mathbf{z})$ and posterior $p(\mathbf{x} \,|\, \mathbf{z})$ distributions

### 1.3 The Encoder: $q_\phi(\mathbf{z}_n \,|\, \mathbf{x}_n)$

**Question 1.5**

(a) i (0, 0.05)
ii (5, 1)

(b)

$$\begin{aligned}D_{KL}(q||p) &= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \\ &= \frac{1}{2}\left(\mathrm{tr}(\Sigma_p^{-1}\Sigma_q) + (\mu_p - \mu_q)^T\Sigma_p^{-1}(\mu_p - \mu_q) - D + \log\left(\frac{|\Sigma_p|}{|\Sigma_q|}\right)\right)\end{aligned} \tag{4}$$

For $p = \mathcal{N}\big(0, \mathbf{I}_D\big)$, we have:

$$D_{KL}(q||p = \mathcal{N}\big(0, \mathbf{I}_D\big)) = \frac{1}{2}\left(\text{tr}(\Sigma_q) + \mu_q^T \mu_q - D + \log\left(|\Sigma_q|\right)\right) \tag{5}$$

**Question 1.6**

As $D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z|\,\mathbf{x}_n)\right) \geq 0$, we have that

$$\begin{aligned}\log p(\mathbf{x}_n) &= D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z|\,\mathbf{x}_n)\right) + \mathbb{E}_{q(z|\,\mathbf{x}_n)}\left[\log p(\mathbf{x}_n\,|Z)\right] - D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z)\right) \\ &\geq \mathbb{E}_{q(z|\,\mathbf{x}_n)}\left[\log p(\mathbf{x}_n\,|Z)\right] - D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z)\right)\end{aligned} \tag{6}$$

**Question 1.7**

As $D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z|\,\mathbf{x}_n)\right)$ is intractable, directly optimizing the log-probability directly is simply impossible.

**Question 1.8**

The two things that can happen when the lower bound is pushed up are:

1. The log-probability of the data $\log p(\mathbf{x}_n)$ is maximized
2. The KL-divergence $D_{KL}\left(q(Z|\,\mathbf{x}_n)||p(Z|\,\mathbf{x}_n)\right)$ is minimized

Ideally both of these things happen when the lower pound is pushed up, as the approximation of the posterior $q(Z|\,\mathbf{x}_n)$ would resemble the true posterior $p(Z|\,\mathbf{x}_n)$ , which is highly informative.

### 1.4 Specifying the encoder $q_\phi(\mathbf{z}_n\,|\,\mathbf{x}_n)$

**Question 1.9**

The name *reconstruction* is appropriate, because

$$\mathcal{L}_n^{\text{recon}} = -\,\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta(\mathbf{x}_n\,|Z)\right] \tag{7}$$

represents the error between the reconstructed data as output by the decoder, and the true data which it is trying to reconstruct.

The name *regularization* error is appropriate, because

$$\text{and } \mathcal{L}_n^{\text{reg}} = D_{\text{KL}}\left(q_\phi(Z|\,\mathbf{x}_n)||p_\theta(Z)\right) \tag{8}$$

represents the distance between the true posterior $p_\theta(Z)$ and the variational approximation $q_\phi(Z|\,\mathbf{x}_n)$, which is to be minimized to reduce overfitting, which is exactly what regularization does.

**Question 1.10**

For this question, we consider the partial losses:

$$\begin{aligned}\mathcal{L}_n^{\text{recon}} &= -\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta(\mathbf{x}_n\,|Z)\right], \\ \text{and } \mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi(Z|\,\mathbf{x}_n)||p_\theta(Z)\right).\end{aligned} \tag{9}$$

First, the *reconstruction* error:

$$\begin{aligned}\mathcal{L}_n^{\text{recon}} &= -\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta(\mathbf{x}_n\,|Z)\right] \\ &= -\log\left(\prod_{m=1}^{M}\text{Bern}\left(\mathbf{x}_n^{(m)}\,|f_\theta(\mathbf{z}_n^{(m)})_m\right)\right)\end{aligned}$$

$$
\begin{aligned}
&= -\sum_{m=1}^{M} \log\left(\text{Bern}\left(\mathbf{x}_n^{(m)} \,|f_\theta(\mathbf{z}_n^{(m)})_m\right)\right) \\
&= -\sum_{m=1}^{M} \log\left((f_\theta(\mathbf{z}_n^{(m)})_m)^{\mathbf{x}_n^{(m)}} \cdot (1-(f_\theta(\mathbf{z}_n^{(m)})_m)^{1-\mathbf{x}_n^{(m)}}\right) \\
&= -\sum_{m=1}^{M} \left(\log\left((f_\theta(\mathbf{z}_n^{(m)})_m)^{\mathbf{x}_n^{(m)}}\right) + \log\left((1-(f_\theta(\mathbf{z}_n^{(m)})_m)^{1-\mathbf{x}_n^{(m)}}\right)\right) \\
&= -\sum_{m=1}^{M} \left(\mathbf{x}_n^{(m)} \cdot \log\left(f_\theta(\mathbf{z}_n^{(m)})_m\right) + (1-\mathbf{x}_n^{(m)}) \cdot \log\left(1-(f_\theta(\mathbf{z}_n^{(m)})_m\right)\right). \qquad (10)
\end{aligned}
$$

Second, the *regularization* error:

$$
\begin{aligned}
\mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi(Z|\,\mathbf{x}_n)||p_\theta(Z)\right) \\
&= D_{\text{KL}}\left(\mathcal{N}\big(\mathbf{z}_n, \mu_\phi(\mathbf{x}_n)\big), \text{diag}\left(\Sigma_\phi(\mathbf{x}_n)\right)||\mathcal{N}\big(0, \mathbf{I}_d\big)\right) \\
&= \frac{1}{2}\left(\text{tr}(\Sigma_{q_\phi}) + \mu_{q_\phi}{}^T \mu_{q_\phi} - D + \log\left(|\Sigma_{q_\phi}|\right)\right) \qquad (11)
\end{aligned}
$$

### 1.5 The Reparametrization Trick

**Question 1.11**

(a) We need $\nabla_\phi \mathcal{L}$, because we want to train our VAE end-to-end. The gradient of the loss, with respect to the variational parameters $\phi$, are used to update the weights through backpropagation of our VAE, thereby training it.

(b) In order to propagate the loss through the entire VAE, we need to propagate it through a sampling layer, which is inherently stochastic, and therefore is a non-continuous operation and has no gradient. Thus it is non-differentiable.

(c) The *reparametrization trick* involves moving the sampling to an input layer: First we sample $\epsilon \sim \mathcal{N}\big(0, \mathbf{I}\big)$, after which we sample $\mathbf{z}_n$ by $\mathbf{z}_n = \mu(\mathbf{x}_n) + \Sigma^{\frac{1}{2}}(\mathbf{x}_n) + \epsilon$, where $\mu(\mathbf{x}_n)$ and $\Sigma^{\frac{1}{2}}(\mathbf{x}_n)$ are the mean and covariance of $q(\mathbf{z}_n \,|\, \mathbf{x}_n)$. As the stochastic sampling is now performed in an input layer, for which we do not need to compute the gradient. As such, the loss can now be backpropagated through the encoder, allowing us to train the VAE.

### 1.6 Putting things together: Building a VAE

**Question 1.12**

Please see my `a3_vae_template.py` for my implementation.

The encoder has one hidden layer with a dimension of 500 and uses the ReLU for the activation function. The hidden layer is mapped to the **mean** through a linear layer to the mean. The same goes for the **standard deviation**, but after it's passed through a ReLU activation function (as to avoid negative values).

The reparametrization trick is used to "sample" a latent z: a sample $n \sim \mathcal{N}\big(0, 1\big)$ is combined with the mean and standard deviation from the encoder as follows:

$$
z = \mu + \sigma * n \qquad (12)
$$

The decoder also has one hidden layer with a dimension of 500 mapped from the latent z, and uses a tanh activation function. This hidden layer is mapped to Bernouilli means by another linear layer, this time with a sigmoid activation function.

In all experiments, the Adam optimizer was used. Latent z dimensions of 2 and 20 were tried and tested.

**Question 1.13**

The plot of the estimated lower-bounds for the VAE (using `z_dim = 20`) during training is presented in Fig. 2.

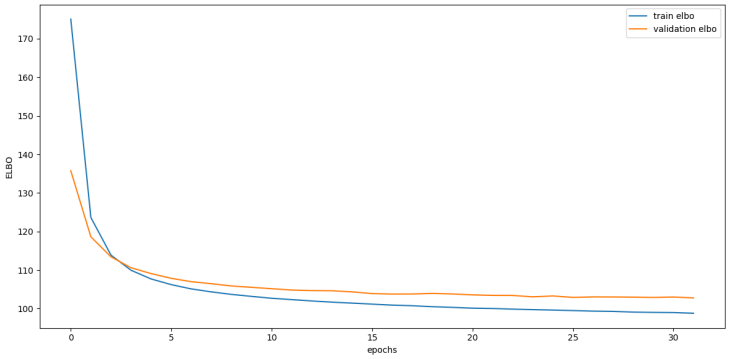
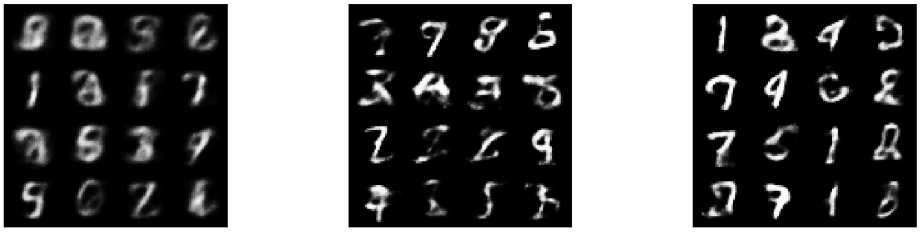



Figure 2: Plot of the estimated lower-bounds during training for `z_dim = 20`

Unfortunately, I only was able to run my VAE for 35 epochs (instead of 40), after which I get only `NaN`s for the estimated lower-bounds for both the training and validation data.

**Question 1.14**



(a) Epoch 1 (b) Epoch 18 (c) Epoch 35

Figure 3: Plots of samples from my VAE model at various stages of training

As can be seen in Fig. 3, the quality of the samples do improve over (training) time!

**Question 1.15**

## 2 Generative Adverserial Networks

**Question 2.1**

The **generator** takes a latent vector **z** (random noise) as input and gives an image as output.

The **discriminator** takes an image as input and gives a binary label of the image,classifying it either as real or fake. The image can come from the generator, but also from the real "training" data.

### 2.1 Training objective: A Minimax Game

**Question 2.2**

The loss function of the GAN is given by:

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{\text{data}(x)}} \left[\log D(X)\right] + \mathbb{E}_{p_z(z)} \left[\log(1 - D(G(Z)))\right], \tag{13}$$

where

$$\mathbb{E}_{p_{\text{data}(x)}} [\log D(X)] \tag{14}$$

is the expected log-probability that a real data sample is classified as real. The discriminator seeks to maximize this log-probability.

$$\mathbb{E}_{p_z(z)} [\log(1 - D(G(Z)))] \tag{15}$$

is the expected log-probability that a generated data sample is classified as fake. Again, the discriminator seeks to maximize this log-probability, while the generator seeks to minimize it (= min-max game).

**Question 2.3**

As Goodfellow et al. [2014] show, the value of $V(D, G)$ is $-\log 4$ after training has converged. This also makes intuitive sense! Consider that the generator $G$ seeks to fool the discriminator $D$ into thinking that the images it generates (which are fake) are real. As such, when the discriminator $D$ becomes better at discriminating the real from the fake images, the generator $G$ has to become better at generating more realistic (fake) images, and vice versa. In the extreme, the (fake) images generated by the generator $G$ will have become so realistic that the discriminator $D$ would not be able to tell the difference between real or fake images anymore:

$$\begin{aligned} V(D^*, G^*) &= \mathbb{E}_{p_{\text{data}(x)}} [\log D(X)] + \mathbb{E}_{p_z(z)} [\log(1 - D(G(Z)))] \\ &= \log(\frac{1}{2}) + \log(\frac{1}{2}) = 2\log(\frac{1}{2}) = -2\log(2) = -\log(4), \end{aligned} \tag{16}$$

where $D^*$ and $G^*$ are the optimal discriminator and generator after training has converged, respectively.

**Question 2.4**

Early on during training, the generator outputs images which are closer to white noise more than anything (as will be shown later in 2.2), instead of resembling the digits present in the `MNIST` dataset. As such, the discriminator will easily be able to discriminate these images as fake.

This means that the $\log(1 - D(G(Z)))$ loss-term (which the generator seeks to minimize) will go to $0$, which means that the generator has little to train upon. As such, it is better, instead, for the generator to maximize $\log(D(G(Z)))$, as this will give strong gradients when $D(G(Z))$ has values close to $0$.

### 2.2 Building a GAN

**Question 2.5**

Please see my `a3_gan_template.py` for my implementation.

The generator $G$ is implemented according to the example architecture outlined in the template, with the exception that a dropout layer was added after the intermediate hidden layers, as introduced by Isola et al. [2017]. The loss for the generator $G$ is given by:

$$\mathcal{L}_G = -\frac{1}{N} \sum_{n=1}^{N} \log(D(G(Z_n))), \tag{17}$$

which is the batch mean of the negative log-likelihood.

The discriminator $D$ is equally implemented according to the example architecture outlined in the template, again with the exception that a dropout layer was added after every immediate hidden layer.

The loss for the discriminator $D$ is given by:

$$\mathcal{L}_D = -\frac{1}{N}\sum_{n=1}^{N}\left(\log\left(D(X_n)\right) + \log\left(1 - D(G(Z_n))\right)\right), \tag{18}$$

which, again, is the batch mean of the negative log-likelihood.

Both the generator and discriminator were trained during every training step. No further alterations were made to the training process (such as clamping losses, halting the discriminator, etc.).

The loss curves for both the generator $G$ and the discriminator $D$ can be found in Fig. 4.

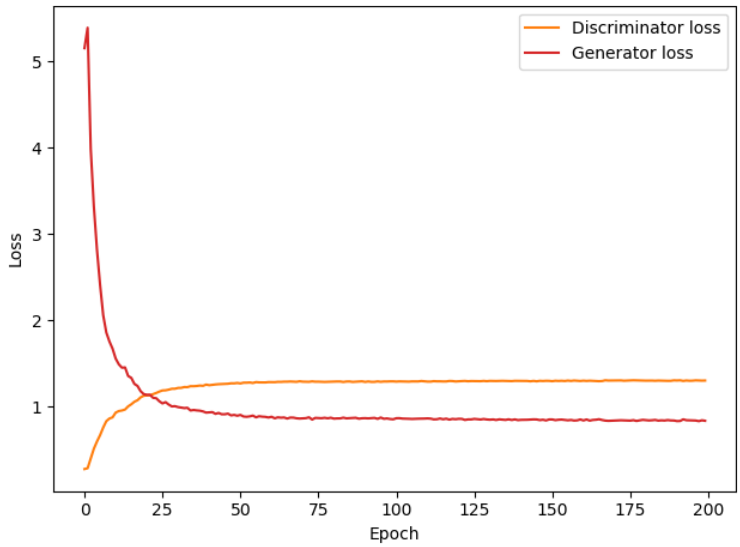



Figure 4: Plot of the losses for both the generator and the discriminator during training

**Question 2.6**

(a) Epoch 1 (b) Epoch 101 (c) Epoch 200

Figure 5: Plots of samples from my VAE model at various stages of training

**Question 2.7**

# 3 Generative Normalizing Flows

## 3.1 Change of variables for Neural Networks

**Question 3.1**

$$z = f(x); x = f^{-1}(z); p(x) = p(z)|\frac{df}{dx}| \tag{19}$$

$$\log p(x) = \log p(z) + \sum_{l=1}^{L} \log |\frac{dh_l}{dh_{l-1}}| \tag{20}$$

Rewriting equations 19 and 20 for the case when $f : \mathbb{R}^m \to \mathbb{R}^m$ is an invertible smooth mapping and $\mathbf{x} \in \mathbb{R}^m$ is a *multivariate* random variable gives:

$$\mathbf{z} = f(\mathbf{x}); \mathbf{x} = f^{-1}(\mathbf{z}); p(\mathbf{x}) = p(\mathbf{z})|\det\left(\frac{\partial f}{\partial \mathbf{x}^T}\right)| \tag{21}$$

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) + \sum_{l=1}^{L} \log |\det\left(\frac{\partial h_l}{\partial h_{l-1}}\right)|, \tag{22}$$

where $\det\left(\frac{\partial f}{\partial \mathbf{x}^T}\right)$ and $\det\left(\frac{\partial h_l}{\partial h_{l-1}}\right)$ are the determinants of the Jacobian matrices, instead of the regular derivatives.

**Question 3.2**

The constraints that have to be set on the function $f(\cdot) : \mathbb{R}^m \to \mathbb{R}^m$ to make this equation computable is that all intermediate functions need to have a square Jacobian with a defined determinant, meaning:

$$f = h_L \circ h_{L-1} \circ \cdots \circ h_2 \circ h_1, \tag{23}$$

with $h_i : \mathbb{R}^m \to \mathbb{R}^m$.

**Question 3.3**

A computational issue that might arise when one optimizes the network using the objective derived in Eqn. 22 is that computing the Jacobian matrix is computationally expensive. Furthermore, computing the determinant of this Jacobian matrix is even more expensive.

As such, one needs to take care such that the Jacobian and its determinant are easily computable.

**Question 3.4**

A possible consequence of the continuous random variables assumption when dealing with images (which have discrete integer values in the range $[0, 255]$), is that the continuous probability distribution might collapse and concentrate its probability mass around these discrete points, which might hurt generalization.

This can be fixed by adding uniform random noise $u \sim U(0, 1)$ to the discrete data $x$, thereby converting the discrete probability distribution into a continuous probability distribution. This process is called "dequantization", as introduced by Ho et al. [2019].

## 3.2 Building a flow-based model

**Question 3.5**

During training time, the input is an image of dimension $m$ ($x \in \mathbb{R}^m$, and the output is the log-likelihood of the image, of the same dimension ($y \in \mathbb{R}^m$).

During inference time, the input is noise from the prior of dimension $m$ ($z \in \mathbb{R}^m$), and the output is a generated image of dimension $m$ ($x' \in \mathbb{R}^m$).

**Question 3.6**

Following Dinh et al. [2016], the steps during training time are, for every batch of $D$ data points:

1. Dequantize
2. Normalize
3. Forward pass through coupling layers
   (a) Split in two parts of $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$
   (b) Use the first half to compute the *scale* $s$ and *translation* $t$ of the second half
   (c) Compute the output $\mathbf{y}$:

$$\begin{aligned} \mathbf{y}_{1:D} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} \left(\odot \exp\left(s(\mathbf{x}_{1:d})\right) + t(\mathbf{x}_{1:d})\right) \end{aligned} \tag{24}$$

   (d) Concatenate the two parts
4. Compute the log-likelihood
5. Compute the loss
6. Backpropagate the loss

The steps during inference time are:

1. Sample from the prior $\mathbf{z} \sim p(\mathbf{z})$
2. Reverse pass $\mathbf{z}$ through the coupling layers
3. Normalize

**Question 3.7**

Please see my `a3_nf_template.py` for my implementation.

**Question 3.8**

The bits-per-dimension (BpD) curves for the NF model can be found in Fig. 6.

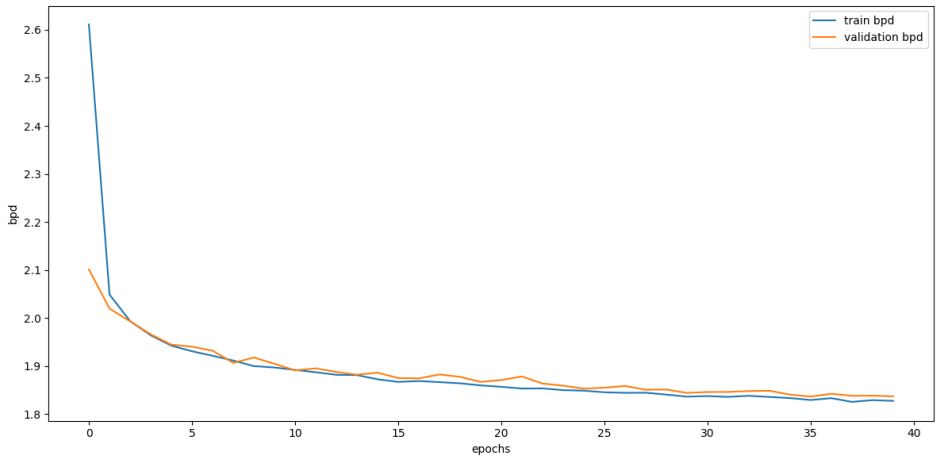



Figure 6: Plot of the bits-per-dimension (BpD) for the NF model during training

(a) Epoch 1 (b) Epoch 21 (c) Epoch 40

Figure 7: Plots of samples from my NF model at various stages of training

## 4 Conclusion

**Question 4.1**

The three models implemented and discussed previously are all able to generate realistic looking images, using different methods, which come with varying model complexity, computational cost and performance.

**VAE**s use an encoder to project the data into a smaller latent space, and a decoder to reconstruct the real data. The model is computationally inexpensive (I was able to run my VAE model on my GPU in 15 minutes or so), being able to generate high-quality images already within 40 epochs of training.

**GAN**s employs game theory in the form of a minimax game between the generator and the discriminator to train both. The generator is responsible for generating the data, using random noise as an input. The model is quite computationally expensive (I was able to run my GAN model on my GPU in 1.5 hours), being able to generate high-quality images within 200 epochs of training.

**NF**s use an invertible smooth mapping from the data to the latent space (which is of the same size as the original data). The model is mildly computationally expensive (I was able to run my NF model on my GPU in 50 minutes or so), being able to generate high-quality images already within 40 epochs of training.

## References


Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. *arXiv preprint arXiv:1902.00275*, 2019.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.