Assignment 3. VAE's and Adversarial Attacks

University of Amsterdam – Deep Learning Course

December 2, 2024

The deadline for this assignment is December 15$^{\text{th}}$ at 23:59.

This assignment consists of two parts. The first part will be about Deep Generative Models. Modelling distributions in high dimensional spaces is difficult. Simple distributions such as multivariate Gaussians or mixture models are not powerful enough to model complicated high-dimensional distributions. The question is: How can we design complicated distributions over high-dimensional data, such as images or audio? In concise notation, how can we model a distribution $p(\boldsymbol{x}) = p(x_1, x_2, \ldots, x_M)$, where M is the number of dimensions of the input data $\boldsymbol{x}$? The solution: Deep Generative Models.

Deep generative models come in many flavors, but all share a common goal: to model the probability distribution of the data. Examples of well-known generative models are Variational Autoencoders (VAEs) [Kingma and Welling, 2014], Generative Adversarial Networks (GANs) [Goodfellow et al., 2014], Adversarial Autoencoder Networks (AAEs) [Makhzani et al., 2015], and Normalizing Flows (NF) [Rezende and Mohamed, 2015]. In this assignment, we will focus on VAEs. The assignment guides you through the theory of VAEs with questions along the way, and finally you will implement one yourself in PyTorch.

In the second part of this assignment, we will dive into adversarial attacks and what they can teach us about neural networks. You will both implement attacks on a ResNet, and defend against attacks.

This assignment contains 50 points: 32 on VAEs, and 18 on Adversarial Attacks.

Note: for this assignment you are not allowed to use the `torch.distributions` package. You are, however, allowed to use standard, stochastic PyTorch functions like `torch.randn` and `torch.multinomial`, and all other PyTorch functionalities (especially from `torch.nn`). Moreover, try to stay as close as you can to the template files provided as part of the assignment.

1 Variational Autoencoders **(Total: 35 points)**

1.1 Prerequisites

VAEs leverage the flexibility of neural networks (NN) to learn and specify a latent variable model. Before we get into the details of VAEs, we first provide a brief recap of latent variable models and the KL divergence.

1.1.1 Latent Variable Models

A latent variable model is a statistical model that contains both observed and unobserved (i.e. latent) variables. Mathematically, they are connected to a distribution $p(\boldsymbol{x})$ over $\boldsymbol{x}$ in the following way: $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$. This integral is typically too expensive to evaluate. However, in this assignment, you will learn an approximate solution via VAEs.

Assume a dataset $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$, where $\boldsymbol{x}_n \in \{0, 1, \ldots, k-1\}^M$. For example, $\boldsymbol{x}_n$ could be the pixel values for an image, in which each pixel can take values 0 through $k-1$ (for example, $k = 256$). A simple latent variable model for this data is shown in Figure 1, which we can also summarize with the following generative story:

$$\boldsymbol{z_n} \sim \mathcal{N}(0, \boldsymbol{I}_D), \tag{1}$$

$$\boldsymbol{x_n} \sim p_X(f_\theta(\boldsymbol{z}_n)), \tag{2}$$

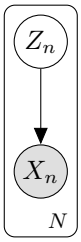

Figure 1. Graphical model of VAE. N denotes the dataset size. The gray variable is observed.

where f_θ is some function – parameterized by θ – that maps $\boldsymbol{z}_n$ to the parameters of a distribution over $\boldsymbol{x}_n$. For example, if p_X would be a Gaussian distribution we will use $f_\theta : \mathbb{R}^D \to (\mathbb{R}^M, \mathbb{R}_+^M)$ for the mean and diagonal elements of the covariance matrix, or if p_X is a product of Bernoulli distributions, we have $f_\theta : \mathbb{R}^D \to [0, 1]^M$. Here, D denotes the dimensionality of the latent space. Likewise, if pixels can take on k discrete values, p_X could be a product of Categorical distributions, so that $f_\theta : \mathbb{R}^D \to (p_1, ..., p_k)^M$, where $p_1, ..., p_k$ are event probabilities of the pixel belonging to value k, where $p_i \geq 0$ and $\sum_{i=1}^k p_i = 1$. Note that our dataset $\mathcal{D}$ does not contain $\boldsymbol{z}_n$, hence $\boldsymbol{z}_n$ is a latent (or unobserved) variable in our statistical model. In the case of a VAE, a (deep) NN is used for $f_\theta(\cdot)$.

Food for thought

How does the VAE relate to a standard autoencoder (see e.g. Tutorial 9)?

1. Are they different in terms of their main purpose? How so?
2. A VAE is generative. Can the same be said of a standard autoencoder?
3. Can a VAE be used in place of a standard autoencoder for its purpose you mentioned above?

1.1.2 KL Divergence

Before covering VAEs, we will need to learn about one more concept that will help us later: the Kullback-Leibler divergence (KL divergence). It measures how different one probability distribution is from another:

$$D_{\mathrm{KL}}(p||q) = \mathbb{E}_{p(x)}\left[\log \frac{p(X)}{q(X)}\right] = \sum_x p(x)\left[\log \frac{p(x)}{q(x)}\right], \tag{3}$$

where q and p are probability distributions in the space of some random variable X. [1] In Appendix A, we provide more intuition on how the KL divergence achieves this effect of measuring the difference between two distributions. Note that, while such an intuition can be useful in general, it is not strictly necessary to make this assignment.

[1] The KL divergence of a continuous RV is obtained by replacing the summation with an integral.

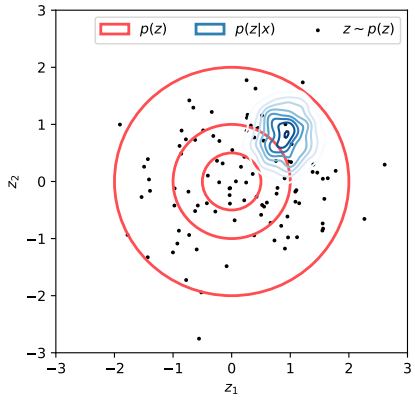

Figure 2. Plot of 2-dimensional latent space and contours of prior and posterior distributions. The red contour shows the prior $p(z)$ which is a Gaussian distribution with zero mean and standard deviation of one. The black points represent samples from the prior $p(z)$. The blue contour shows the posterior distribution $p(z|x)$ for an arbitrary x, which is a complex distribution and here, for example, peaked around $(1, 1)$.

1.2 Decoder: The Generative Part of the VAE

In Section 1.1.1, we described a general graphical model which also applies to VAEs. In this section, we will define a more specific generative model that we will use throughout this assignment. This will later be refered to as the decoding part (or decoder) of a VAE. For this assignment we will assume the pixels of our images $\boldsymbol{x}_n$ in the dataset $\mathcal{D}$ are Categorical(p) distributed.

$$p(\boldsymbol{z}_n) = \mathcal{N}(0, \boldsymbol{I}_D) \tag{4}$$

$$p(\boldsymbol{x}_n|\boldsymbol{z}_n) = \prod_{m=1}^{M} \text{Cat}(\boldsymbol{x}_n^{(m)}|f_\theta(\boldsymbol{z}_n)_m) \tag{5}$$

where $\boldsymbol{x}_n^{(m)}$ is the m-th pixel of the n-th image in $\mathcal{D}$, $f_\theta : \mathbb{R}^D \to (p_1, ..., p_k)^M$ is a neural network parameterized by θ that outputs the probabilities of the Categorical distributions for each pixel in $\boldsymbol{x}_n$. In other words, $\boldsymbol{p}_m = (p_{m1}, ..., p_{mk})$ are event probabilities of the m-th pixel having intensities $1, \ldots, k$ respectively. Hence, for all m, we have $p_{mi} \geq 0$ and $\sum_{i=1}^{k} p_{mi} = 1$.

Question 1.1 (2 points)

Given a decoder f_θ, describe the steps needed to sample an image.

Now that we have defined the model, we can write out an expression for the log probability of the data $\mathcal{D}$ under this model:

$$\begin{aligned} \log p(\mathcal{D}) &= \sum_{n=1}^{N} \log p(\boldsymbol{x}_n) \\ &= \sum_{n=1}^{N} \log \int p(\boldsymbol{x}_n|\boldsymbol{z}_n) p(\boldsymbol{z}_n) d\boldsymbol{z}_n \\ &= \sum_{n=1}^{N} \log \mathbb{E}_{p(\boldsymbol{z}_n)} \left[p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] \end{aligned} \tag{6}$$

This quantity is useful for obtaining the parameters of the decoder θ by Maximum Likelihood Estimation. However, evaluating $\log p(\boldsymbol{x}_n) = \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right]$ involves a very expensive integral. Equation 6 hints at a method for approximating it, namely Monte-Carlo Integration. The log-likelihood can be approximated by drawing samples $\boldsymbol{z}_n^{(l)}$ from $p(\boldsymbol{z}_n)$:

$$\log p(\boldsymbol{x}_n) = \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \tag{7}$$

$$\approx \log \frac{1}{L}\sum_{l=1}^{L} p(\boldsymbol{x}_n|\boldsymbol{z}_n^{(l)}), \quad \boldsymbol{z}_n^{(l)} \sim p(\boldsymbol{z}_n), \tag{8}$$

where '$\sim$' means 'sampled from'. As the number of samples L tends to infinity, the gap between the approximation and the actual expectation becomes tight. This estimator has the nice property of being unbiased when approximating expectations, though not necessarily log-expectations. Nonetheless, it can be used to approximate $\log p(\boldsymbol{x}_n)$ with a sufficiently large number of samples.

Question 1.2 (3 points)

Although Monte-Carlo Integration with samples from $p(\boldsymbol{z}_n)$ can be used to approximate $\log p(\boldsymbol{x}_n)$, it is not used for training VAE type of models, because it is inefficient. In a few sentences, describe why it is inefficient and how this efficiency scales with the dimensionality of $\boldsymbol{z}$. (Hint: you may use Figure 2 in your explanation.)

1.3 The Encoder: $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ - Efficiently evaluating the integral

In Section 1.2, we have developed the intuition why we need the posterior $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$. Unfortunately, the true posterior $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$ is as difficult to compute as $p(\boldsymbol{x}_n)$ itself. To solve this problem, instead of modeling the true posterior $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$, we can learn an approximate posterior distribution, which we refer to as the variational distribution. This variational distribution $q(\boldsymbol{z}_n|\boldsymbol{x}_n)$ is used to approximate the (very expensive) posterior $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$.

Now we have all the tools to derive an efficient bound on the log-likelihood $\log p(\mathcal{D})$. We start from Equation 6 where the log-likelihood objective is written, but for simplicity in notation we write the log-likelihood $\log p(\boldsymbol{x}_n)$ only for a single datapoint.

$$\begin{aligned}
\log p(\boldsymbol{x}_n) &= \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \\
&= \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[\frac{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \quad \text{(multiply by } q(\boldsymbol{z}_n|\boldsymbol{x}_n)/q(\boldsymbol{z}_n|\boldsymbol{x}_n)\text{)} \\
&= \log \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}\left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \quad \text{(switch expectation distribution)} \\
&\geq \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \log\left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \quad \text{(Jensen's inequality)} \\
&= \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}\left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] + \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \log\left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}\right] \quad \text{(re-arranging)} \\
&= \underbrace{\mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}\left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] - KL\left(q(\boldsymbol{z}_n|\boldsymbol{x}_n)||p(\boldsymbol{z}_n)\right)}_{\text{Evidence Lower Bound (ELBO)}} \quad \text{(writing 2nd term as KL)}
\end{aligned} \tag{9}$$

This is awesome! We have derived a bound on $\log p(\boldsymbol{x}_n)$, exactly the thing we want to optimize, where all terms on the right hand side are computable. Let's put together what

we have derived again in a single line:

$$\log p(\boldsymbol{x}_n) \geq \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] - KL\left(q(\boldsymbol{z}_n|\boldsymbol{x}_n)||p(\boldsymbol{z}_n)\right).$$

The right side of the equation is referred to as the *evidence lowerbound* (ELBO) on the log-probability of the data.

This leaves us with the question: How close is the ELBO to $\log p(\boldsymbol{x}_n)$? With an alternate derivation[2], we can find the answer. It turns out the gap between $\log p(\boldsymbol{x}_n)$ and the ELBO is exactly $KL(q(\boldsymbol{z}_n|\boldsymbol{x}_n)||p(\boldsymbol{z}_n|\boldsymbol{x}_n))$ such that:

$$\log p(\boldsymbol{x}_n) - KL(q(\boldsymbol{z}_n|\boldsymbol{x}_n)||p(\boldsymbol{z}_n|\boldsymbol{x}_n)) = \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] - KL\left(q(\boldsymbol{z}_n|\boldsymbol{x}_n)||p(\boldsymbol{z}_n)\right) \tag{10}$$

Now, let's optimize the ELBO. For this, we define our loss as the mean negative lower bound over samples:

$$\mathcal{L}(\theta, \phi) = -\frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_{q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n)] - D_{\mathrm{KL}}(q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)||p_\theta(\boldsymbol{z}_n)) \tag{11}$$

Note, that we make an explicit distinction between the generative parameters θ of the decoder, and the variational parameters ϕ of the encoder.

Question 1.3 (2 points)

Explain how you can see from Equation 10 that the right hand side has to be a *lower bound* on the log-probability $\log p(\boldsymbol{x}_n)$?

Question 1.4 (2 points)

Describe what happens to the ELBO as the variational distribution $q(z_n|\mathbf{x}_n)$ approaches the true posterior $p(z_n|\mathbf{x}_n)$?

Food for thought

Consider the third line of Equation 9, before we apply Jensen's inequality. We change the expectation distribution after multiplying and dividing by the variational distribution.

Intuitively, this process corresponds to sampling from a different distribution ($q(\boldsymbol{z}_n|\boldsymbol{x}_n)$ instead of $p(\boldsymbol{z}_n)$) with the correction weight $\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}$ offsetting the change in distribution we made. Assuming that $q(\boldsymbol{z}_n|\boldsymbol{x}_n)$ is a good approximation to the true $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$, why would Monte-Carlo integration described in Section 1.2 be more sample efficient after changing the expectation distribution?

(Hint: Consider again Figure 2 and how the samples would be distributed under the variational distribution.)

1.4 Defining the optimization objective

The loss in Equation 11:

$$\mathcal{L}(\theta, \phi) = -\frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_{q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n)] - D_{\mathrm{KL}}(q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)||p_\theta(\boldsymbol{z}_n))$$

[2]This derivation is not done here, but can be found in for instance Bishop sec 9.4.

can be rewritten in terms of per-sample losses:

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

where

$$\begin{aligned}\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n)] \\ \mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}(q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)||p_\theta(\boldsymbol{z}_n))\end{aligned}$$

can be seen as a reconstruction loss term and a regularization term, respectively.

Question 1.5 (2 points)

Explain shortly why the names reconstruction and regularization are appropriate for these two losses.

(Hint: Suppose we use just one sample to approximate the expectation $\mathbb{E}_{q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)}[p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n)]$ – as is common practice in VAEs.)

First, we write down the **reconstruction term**:

$$\begin{aligned}\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)] \\ &= -\frac{1}{L}\sum_{l=1}^{L}\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n^{(l)}), \quad \boldsymbol{z}_n^{(l)} \sim q(\boldsymbol{z}_n|\boldsymbol{x}_n)\end{aligned}$$

here we used Monte-Carlo integration to approximate the expectation

$$= -\frac{1}{L}\sum_{l=1}^{L}\sum_{m=1}^{M}\log \text{Cat}(\boldsymbol{x}_n^{(m)}|f_\theta(\boldsymbol{z}_n^{(l)}))$$

Remember that $f_\theta(\cdot)$ denotes our decoder. Now let $\boldsymbol{p}_{nl}^{(m)} = f_\theta(\boldsymbol{z}_n^{(l)})_m$, then

$$= -\frac{1}{L}\sum_{l=1}^{L}\sum_{m=1}^{M}\sum_{k=1}^{K}\boldsymbol{x}_{nk}^{(m)}\log p_{nlk}^{(m)}.$$

where $\boldsymbol{x}_{nk}^{(m)} = 1$ if the m-th pixel has the value k, and zero otherwise. In other words, the equation above represents the common cross-entropy loss term. When setting $L = 1$ (i.e. only one sample for $\boldsymbol{z}_n$), we obtain:

$$= -\sum_{m=1}^{M}\sum_{k=1}^{K}\boldsymbol{x}_{nk}^{(m)}\log p_{nk}^{(m)}$$

where $\boldsymbol{p}_n^{(m)} = f_\theta(\boldsymbol{z}_n)_m$ and $\boldsymbol{z}_n \sim q(\boldsymbol{z}_n|\boldsymbol{x}_n)$. Thus, we can use the cross-entropy loss with respect to the original input $\boldsymbol{x}_n$ to optimize $\mathcal{L}_n^{\text{recon}}$

To **compute the regularization term**, we must specify the posterior distribution of the latent variable $p(\boldsymbol{z})$ as well the encoder distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$. Similarly to having the flexibility to model the output (i.e. image) distribution $p(\boldsymbol{x}|\boldsymbol{z})$, we also have some freedom

here. We usually set the prior to be a normal distribution with a zero mean and unit variance: $p = \mathcal{N}(\mathbf{0}, \boldsymbol{I}_D)$ and the encoder to be:

$$q_\phi(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\mu_\phi(\boldsymbol{x}), \text{diag}(\sigma_\phi(\boldsymbol{x}))), \tag{12}$$

where $\mu_\phi : \mathbb{R}^M \to \mathbb{R}^D$ maps an input image to the mean of the multivariate normal over $\boldsymbol{z}_n$ and $\sigma_\phi : \mathbb{R}^M \to \mathbb{R}^D_+$ maps the input image to the diagonal of the covariance matrix of that same distribution. For this case, we can actually find a closed-form solution of the KL divergence:

$$\begin{aligned}\mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}(q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)||p_\theta(\boldsymbol{z}_n)) \\ &= D_{\text{KL}}(\mathcal{N}(\boldsymbol{z}|\mu_\phi(\boldsymbol{x}), \text{diag}(\sigma_\phi(\boldsymbol{x})))||\mathcal{N}(\mathbf{0}, \boldsymbol{I}_D))\end{aligned}$$

Using the fact that both probability distributions factorize and that the KL-divergence of two factorizable distributions is a sum of KL terms, we can rewrite this to

$$\begin{aligned}&= \sum_{d=1}^{D} D_{\text{KL}}(\mathcal{N}(z_{nd}|\mu_\phi(\boldsymbol{x}_n)_d, \sigma_\phi(\boldsymbol{x}_n)_d \,||\, \mathcal{N}(z_{nd}|0,1)) \\ &= \sum_{d=1}^{D} \frac{\sigma_\phi(\boldsymbol{x}_n)_d^2 + \mu_\phi(\boldsymbol{x}_n)_d^2 - 1 - \log \sigma_\phi(\boldsymbol{x}_n)_d^2}{2} \\ &= \frac{1}{2}\sum_{d=1}^{D} \sigma_{nd}^2 + \mu_{nd}^2 - 1 - \log \sigma_{nd}^2 \\ &= \frac{1}{2}\sum_{d=1}^{D} \exp(2\log\sigma_{nd}) + \mu_{nd}^2 - 1 - 2\log\sigma_{nd}.\end{aligned}$$

For simplicity, we skipped most of the steps in the derivation, but you can find more details here if you are interested (it is not essential for understanding the VAE).

Plugging everything together, our final loss is:

$$\mathcal{L}_n(\theta, \phi) = \underbrace{-\sum_{m=1}^{M}\sum_{k=1}^{K} \boldsymbol{x}_{nk}^{(m)} \log p_{nk}^{(m)}}_{\mathcal{L}_n^{\text{recon}}} + \underbrace{\sum_{d=1}^{D} \frac{1}{2}(\exp(2\log\sigma_{nd}) + \mu_{nd}^2 - 1 - 2\log\sigma_{nd})}_{\mathcal{L}_n^{\text{reg}}} \tag{13}$$

$$\mathcal{L}(\theta, \phi) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}_n(\theta, \phi). \tag{14}$$

Question 1.6 (3 points)

The above derivation of closed form expression for the regularization term requires Gaussian prior and variational distributions. Assume that we want to model the prior $p(\boldsymbol{z})$ with a more complex distribution — it is likely the closed form expression would not exist.

Keeping in mind that $D_{\text{KL}}(q(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) = \mathbb{E}_{z\sim q(\boldsymbol{z}|\boldsymbol{x})}[\log \frac{q(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{x})}]$, propose an alternative way of estimating the regularization term for a given sample $\boldsymbol{x}_n$.

1.5 The Reparametrization Trick

Although we have written down (the terms of) an objective above, we still cannot simply minimize this by taking gradients with regard to θ and ϕ. This is due to the fact that we

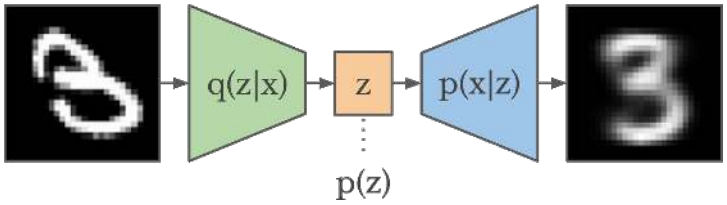

Figure 3. A VAE architecture on MNIST. The encoder distribution $q(z|x)$ maps the input image into latent space. This latent space should follow a unit Gaussian prior $p(z)$. A sample from $q(z|x)$ is used as input to the decoder $p(x|z)$ to reconstruct the image.

sample from $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ to approximate the $\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)]$ term. Yet, we need to pass the derivative through these samples if we want to compute the gradient of the encoder parameters, i.e., $\nabla_\phi \mathcal{L}(\theta, \phi)$. Our posterior approximation $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ is parameterized by ϕ. If we want to train $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ to maximize the lower bound, and therefore approximate the posterior, we need to have the gradient of the lower-bound with respect to ϕ.

Question 1.7 (2 points)

Passing the derivative through samples can be done using the *reparameterization trick* — the process of sampling z directly from $\mathcal{N}(\mu(x), \Sigma(x))$ is commonly replaced by calculating $z = \Sigma(x)\epsilon + \mu(x)$, where $\epsilon \sim \mathcal{N}(0, 1)$. In a few sentences, explain why the act of direct way of sampling usually prevents us from computing $\nabla_\phi \mathcal{L}$, and how the reparameterization trick solves this problem.

1.6 Putting things together: Building a VAE

Given everything we have discussed so far, we now have an objective (the evidence lower bound or ELBO) and a way to backpropagate to both θ and ϕ (i.e., the reparametrization trick). Thus, we can now implement a VAE in PyTorch to train on MNIST images. We will model the encoder $q(z|x)$ and decoder $p(x|z)$ by a deep neural network each, and train them to maximize the data likelihood. Figure 3 provides an overview of the components you need to consider in your implementation of the VAE.

In the code directory `part1`, you can find the templates to use for implementing the VAE. For your implementation of the training loop, you will work with PyTorch Lightning - a framework that simplifies your code needed to train, evaluate, and test a model in PyTorch. You do not need to be familiar with PyTorch Lightning to the lowest level, but a high-level understanding as from the introduction in Tutorial 5 is recommended for implementing the template.

You also need to implement additional functions in `utils.py`, and the encoder and decoder in the files `cnn_encoder_decoder.py`. We specified a recommended architecture to start with, but you are allowed to experiment with your own ideas for the models. For the sake of the assignment, it is sufficient to use the recommended architecture to achieve full points. Use the provided unit tests to ensure the correctness of your implementation. Details on the files can be found in the README of part 1.

As a loss objective and test metric, we will use the bits per dimension score (bpd). Bpd is motivated from an information theory perspective and describes how many bits we would need to encode a particular example in our modeled distribution. You can see it as how many bits we would need to store this image on our computer or send it over a network, if we have given our model. The less bits we need, the more likely the example is in our distribution. Hence, we can use bpd as loss metric to minimize. When we test for the bits per dimension on our test dataset, we can judge whether our model generalizes to new samples of the dataset and didn't in fact memorize the training dataset. In order

to calculate the bits per dimension score, we can rely on the negative log-likelihood we got from the ELBO, and change the log base (as bits are binary while NLL is usually exponential):

$$\text{bpd} = \text{nll} \cdot \log_2(e) \cdot \left(\prod_{i=1}^{K} d_i \right)^{-1}$$

where $d_1, ..., d_K$ are the dimensions of the input **excluding any batch dimension**. For images, this would be the height, width and channel number. In other words, for an image of size 28×28 with one channel, we have $d_1 = 28, d_2 = 28, d_3 = 1$ (the order does not matter). We average over those dimensions in order to have a metric that is comparable across different image resolutions. The nll represents the negative log-likelihood loss $\mathcal{L}$ from Equation 11 for a single data point. You should implement this function in `utils.py`.

Question 1.8 (12 points)

Build a Variational Autoencoder in the provided templates, and train it on the MNIST dataset. Both the encoder and decoder should be implemented as a CNN. For the architecture, you can use the same as used in Tutorial 9 about Autoencoders. Note that you have to adjust the output shape of the decoder to output $1 \times 28 \times 28$ for MNIST. You can do this by adjusting the output padding of the first transposed convolution in the decoder. Use a latent space size of `z_dim=20`. Read the provided README to become familiar with the code template.

In your submission, plot the estimated bit per dimension score of the lower bound on the training and validation set as training progresses, and the final test score. You are allowed to take screenshots of a TensorBoard plot if the axes values are clear.

Note: using the default hyperparameters is sufficient to obtain full points. As a reference, the training loss should start at around 4 bpd, reach below 2.0 after 2 epochs, and end around 0.52 after 80 epochs.

Question 1.9 (3 points)

Plot 64 samples (8×8 grid) from your model at three points throughout training (before training, after training 10 epochs, and after training 80 epochs). You should observe an improvement in the quality of samples. Describe shortly the quality and/or issues of the generated images.

Question 1.10 (4 points)

Train a VAE with a 2-dimensional latent space (`z_dim=2` in the code). Use this VAE to plot the data manifold as is done in Figure 4b of [Kingma and Welling, 2014] and was discussed in Lecture 6. This is achieved by taking a two dimensional grid of points in Z-space, and plotting $f_\theta(Z) = \mu|Z$. Use the percent point function (ppf, or the inverse CDF) to cover the part of Z-space that has significant density. Implement it in the function `visualize_manifold` in `utils.py`, and use a grid size of 20. Are you recognizing any patterns of the positions of the different digits?

2 Adversarial Attacks **(Total: 18 points)**

Deep learning models are used to perform tasks by processing the input. Picture classification. The network outputs the probability that the input belongs to a specific class. With this in mind, a desirable property is that, given two perceptually similar inputs, the output probability also does not change significantly. To put this into more precise terms, assume an input $\boldsymbol{x}$, and a small radius $\epsilon \in \mathbb{R}_{>0}$. We expect that for any vector $\boldsymbol{r}$ with $\boldsymbol{r} < \epsilon$, then the new input $\boldsymbol{x} + \boldsymbol{r}$ does not affect the class predicted by the model. This is often called the smoothness bias of neural networks. This, however, does not hold for deep networks. There exist very small changes in the input that do not effectively change its content but that can drastically affect the prediction of the model. This concept was explored in depth by Szegedy [2013], where the concept of **adversarial attack** was introduced (and several very pedagogical experiments were carried out).

Definition. An adversarial attack is an imperceptible perturbation performed to an input that causes a network to make a mistake.

Such attacks are not just a quirk of neural networks. They are a dangerous effect of having a very complex system that we do not entirely understand trusted to perform critical tasks. This could be modifying the prompt to an LLM-assisted agent to compel it to give you private information, bypass the fraud checks of a bank, or stop a face-recognition system from identifying your face.

The fact that adversarial attacks are possible also highlights that, given inputs that to us look perfectly reasonable, there could be something in them which throws off the performance of a model, even when we have not been attack by a malicious agent. Something in the data could have changed, and the model picks up on it, while we do not.

2.1 Fast Gradient Sign Method (FGSM)

In this assignment we will be attacking an old ResNet with a very straightforward attack. The FGSM uses the gradients with respect to the input to make a barely noticeable perturbation to the input, while significantly changing the output [Goodfellow et al., 2015]. The formula for the perturbation is given by:

$$\boldsymbol{\eta} = \epsilon \cdot \operatorname{sign}\left(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y)\right)$$

Question 2.1 (6 points)

In *adversarial_ attack.py*, implement the *fgsm_ attack* function, and implement the FGSM part in *test_ attack*. For a pretrained ResNet18, compute the accuracy on the test set with and without the FGSM attack. You can use the *train.py* for this, report the accuracies and the configuration you used. In addition, answer the following questions:

- (1pt) Why would adding a random perturbation with size ϵ not have a similar effect as a FGSM perturbation of the same size?
- (1pt) Say we split the training data into two sets A and B, and train models A and B on those datasets, respectively. Then, we have two models trained for the same task but using different subsets of the same dataset. Now for an instance x in the test set, a perturbation built using the gradients from model A, will likely have a similar effect on model B, even though the models don't share weights or the exact training data. What is likely to be the cause of this phenomenon?
- (1pt) What is the effect on using data augmentation compared to using no data augmentation, how would you explain this effect?

Note: you can leave hyperparameters as they are, but feel free to experiment to get a better idea what is happening, try the visualise flag for example. (What do you think would happen if we increase ϵ ?) Or you can try using a pretrained resnet, instead of training from scratch.

It is also possible to defend a model against a certain adversarial attack, this can be done in different ways. In the original papers, the authors proposed an "adversarial objective function". Here the loss takes adversarial examples into account:

$$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{x}, y) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, y) + (1-\alpha) J\big(\boldsymbol{\theta}, \boldsymbol{x} + \epsilon \operatorname{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))\big).$$

Question 2.2 (4 points)

Also train a ResNet18 using this modified loss, this can be done by implementing the *fgsm_ loss* function, and calling the necessary configuration in *train.py*. Report the accuracies with and without attacks, of the models with and without the defense.
Hint: You should see improvements for the defense, but expect it to still perform significantly worse on attacked examples than on normal examples. Hint: Try it using pretraining and augmentation, in which case does the defense work significantly?

- (1pt) Describe the tradeoff between defending against the attack or not, why does this tradeoff occur?

2.2 Projected Gradient Descent (PGD)

FGSM is a lightweight, simple attack. Using a similar principle, we can achieve even less detectable perturbations. In this part we will be implementing PGD, which iteratively follows the gradient w.r.t. the data, but clips change at each timestep to make sure the perturbed image stays "close" to the original. This is done by iteratively taking a FGSM step with step size α, and then keeping the perturbed image within the ball of size ϵ around the original image.

This time the defense against the attack will have a similar idea, but different implementation. Instead of integrating an adversarial defense into the loss function,

we will add adversarial examples to the batch during training, and train with the original loss.

Question 2.3 (8 points)

Implement the *pgd_ attack* function in *adversarial_ attack.py*, and the PGD section in *test_ attack*. Moreover, for the defense implement adding adversarial examples to the batch in *train* in *utils.py*.

- (1pt) In this implementation, how do "using an adversarial loss" and "adding adversarial examples to the batch" compare? Are they equivalent or different? Provide reasoning for your answer, including under what conditions they might align or diverge.
- (1pt) Describe a tradeoff between using FGSM and PGD. For each method, identify one advantage it has over the other.

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. 1

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015. URL https://arxiv.org/abs/1412.6572. 10

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. International Conference on Learning Representations (ICLR), 2014. 1, 9

Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. Adversarial autoencoders. arXiv preprint arXiv:1511.05644, 2015. 1

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, pages 1530–1538. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045281. 1

C Szegedy. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199, 2013. 10

A Information Theoretic View on the KL Divergence

In order to get a better intuition on how the KL divergence measures how different two probability distributions are, it is useful to treat it from an information theoretic perspective. When we observe an event (e.g., a random variable takes on a specific value), we can think of it as receiving a certain amount of information. Formally, the information content of an event x is

$$h(x) = \log\left(\frac{1}{p(x)}\right) = -\log p(x). \tag{15}$$

Notice that the lower the probability of that event, the more information we will have received after observing that event.

The **entropy** of a discrete random variable X with distribution p is the *average* amount of information we effectively receive if we draw a single sample from the distribution p:

$$H(p) = \mathbb{E}_{p(x)}\left[-\log p(X)\right] = -\sum_x p(x)\log p(x). \tag{16}$$

The larger the uncertainty ('spread') of a random variable, the higher its entropy. For instance, a uniform distribution has a high entropy, whereas any delta-function has a low entropy. Similarly, as shown in Figure 4, Bernoulli(p) has a high entropy for $p = 0.5$, and a low entropy for $p = 0$ and $p = 1$.

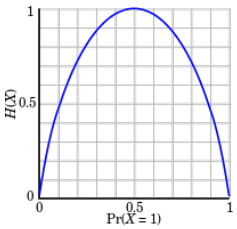

Figure 4. The entropy of a Bernoulli distribution for different values of its parameter p. Source: Wikipedia.

Now, suppose we model the true distribution p with another distribution q, and we use q to construct a 'coding scheme' [3] to transmit information about samples drawn from p. Then, the **cross-entropy** is the average amount of information required (e.g. in bits) to specify the value of a sample drawn from p if we use the coding scheme based on q:

$$H(p,q) = \mathbb{E}_{p(x)}\left[-\log q(X)\right] = -\sum_x p(x)\log q(x). \tag{17}$$

Notice that $H(p,p) = H(p)$ is a lower bound on $H(p,q)$, i.e., the coding scheme based on p is optimal if we want to specify a value of a sample drawn from p. Using these definitions, we can derive an expression of the KL divergence $D_{\mathrm{KL}}(p||q)$ in terms of entropy $H(p)$ and cross-entropy $H(p,q)$:

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \sum_x p(x)\left[\log\frac{p(x)}{q(x)}\right] \\ &= \sum_x p(x)\log p(x) - p(x)\log q(x) \\ &= \sum_x p(x)\log p(x) - \sum_x p(x)\log q(x) \\ &= H(p,q) - H(p). \end{aligned}$$

Food for thought

We have derived that $D_{\mathrm{KL}}(p||q) = H(p,q) - H(p)$. What does this expression tell you about the meaning of the KL divergence from an information theoretic point of view? How does this relate to the idea that the KL divergence measures how different one probability distribution is from the other?

[3] Please have a look at this video for more details.