Practical 3. Deep Generative Models and Transformer-based Models

University of Amsterdam – Deep Learning Course

December 1, 2023

The deadline for this assignment is December 16$^{\text{th}}$ at 23:59.

This assignment consists of three parts. The first and second 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 and AAEs. The assignment guides you through the theory of VAEs and AAEs with questions along the way, and finally you will implement them yourself in PyTorch.

The last part is an investigation into Transformer self-attention building blocks, and the Transformer-based models. It covers the mathematical properties of Transformers and a small-scale implementation of a transformer-based model.

Throughout this assignment, you will see a new type of boxes between questions, namely **Food for thought** boxes. Those contain questions that are helpful for understanding the material, but are not essential and **not required to submit on ANS** (no points are assigned to those questions). Still, try to think of a solution for those boxes to gain a deeper understanding of the models.

This assignment contains 66 points: 34 on VAEs, 22 on AAEs, and 10 on language transformers.

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: 34 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:

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

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

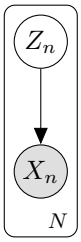

Figure 1. Graphical model of VAE. N denotes the dataset size. The gray variable is observed.

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.

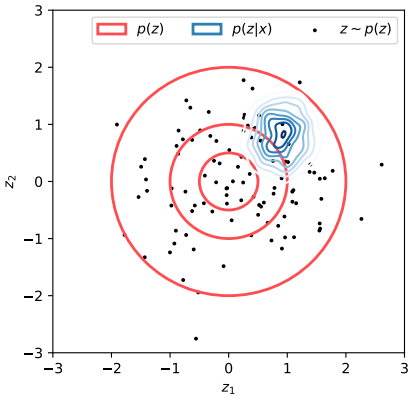

Figure 2. Plot of 2-dimensional latent space and contours of prior and posterior distributions. The red contour shows the prior $p(z)$ which is a Gaussian distribution with zero mean and standard deviation of one. The black points represent samples from the prior $p(z)$. The blue contour shows the posterior 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 (3 points)

Now, looking at the two terms on the left-hand side of 10: Two things can happen when the lower bound is pushed up. Can you describe what these two things are?

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:

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

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

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 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 1.6 (3 points)

Passing the derivative through samples can be done using the *reparameterization trick*. In a few sentences, explain why the act of sampling usually prevents us from computing $\nabla_\phi \mathcal{L}$, and how the reparameterization trick solves this problem.

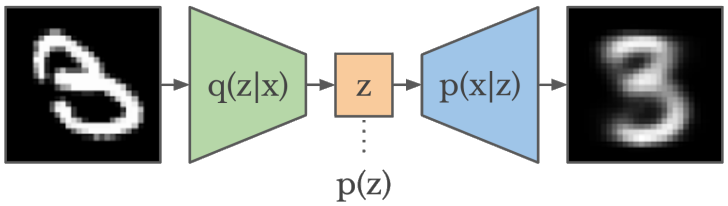

Figure 3. A VAE architecture on MNIST. The encoder distribution $q(z|x)$ maps the input image into latent space. This latent space should follow a unit Gaussian prior $p(z)$. A sample from $q(z|x)$ is used as input to the decoder $p(x|z)$ to reconstruct the image.

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 1.7 (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 1.8 (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 1.9 (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 Autoencoder Networks (Total: 22 points)

Generative Adversarial Networks (GANs) are another type of deep generative models. Similar to VAEs, GANs can generate images that mimic images from the dataset by sampling an encoding from a noise distribution. In contrast to VAEs, in vanilla GANs there is no inference mechanism to determine an encoding or latent vector that corresponds to a given data point (or image). Figure 4 shows a schematic overview of a GAN. A GAN consists of two separate networks (i.e., there is no parameter sharing or the like) called the generator and the discriminator. Training a GAN leverages an adversarial training scheme. In short, that means that instead of defining a loss function by hand (e.g., cross-entropy or mean squared error), we train a network that acts as a loss function. In the case of a GAN this network is trained to discriminate between real images and fake (or generated) images, hence the name discriminator. The discriminator (together with the training data) then serves as a loss function for our generator network that will learn to generate images to are similar to those in the training set. Both the generator and discriminator are trained jointly.

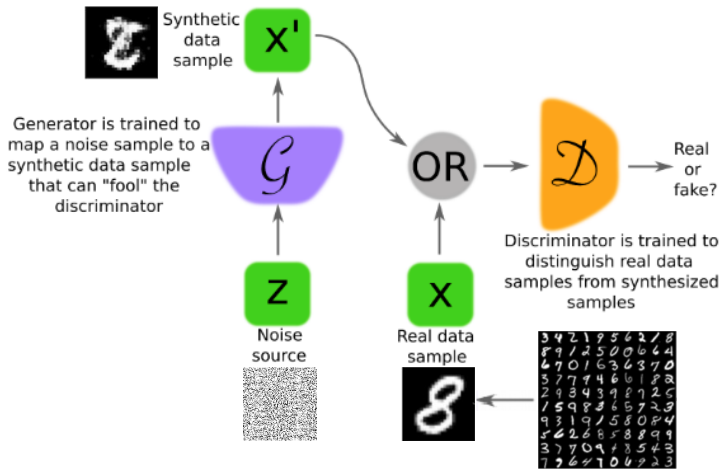

Figure 4. Schematic overview of a Generative Adversarial Network (GAN) [Creswell et al., 2018] on MNIST data. There two networks, generator $\mathcal{G}$ and discriminator $\mathcal{D}$, are learned during optimization for a generative adversarial network. While the generator samples new images, the discriminator has to distinguish between generated images and the real dataset.

Traditional or vanilla GANs have no control over the generated samples and it is hard to predict which noise signal will produce our desired sample. Adversarial autoencoders solve this problem by mixing the benefits from both GANs and autoencoders. Adversarial autoencoders are similar to variational autoencoders, in that they also try to impose a prior on the hidden code vector of the autoencoder, however they use an adversarial training procedure to match the hidden vector distribution with the prior distribution [Makhzani et al., 2015].

The adversarial autoencoder, similar to a standard autoencoder, has an encoding and decoding component. The input $\mathbf{x}$ has a data distribution as $p(\mathbf{x})$. Then, the encoder tries to find a latent code $\mathbf{z}$ from $\mathbf{x}$ which minimizes the decoder reconstruction error. Let us define the distribution of the latent code $\mathbf{z}$ conditioned on input $\mathbf{x}$ with $q(\mathbf{z}|\mathbf{x})$. Meanwhile, the encoder, which is also the generator of adversarial autoencoders, tries to match the distribution of latent code $q(\mathbf{z}|\mathbf{x})$ to a desired prior distribution $p(\mathbf{z})$. The discriminator for adversarial autoencoder should detect whether the samples are fake, *i.e.* the latent samples extracted from the input by the encoder, or they are real samples drawn from the prior distribution $p(\mathbf{z})$. Meanwhile, the decoder objective is to reconstruct the original input from its latent or encoded code.

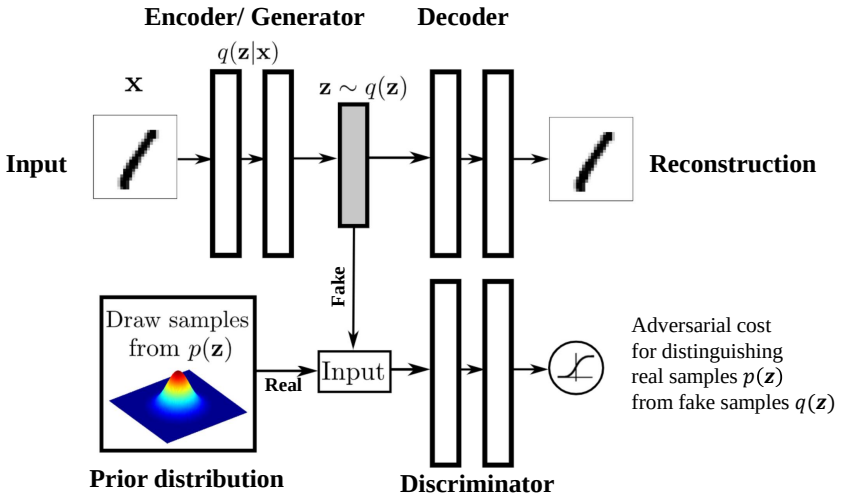

Figure 5. Architecture of an adversarial autoencoder. The top row is a standard autoencoder that reconstructs an image $\mathbf{x}$ from a latent code $\mathbf{z}$. The bottom row diagrams a second network trained to discriminatively predict whether a sample arises from the hidden code of the autoencoder or from a sampled distribution specified by the user [Makhzani et al., 2015].

Question 2.1 (3 points)

Explain how to compute $q(\mathbf{z})$ based on each of the following encoders:

- A deterministic function, i.e. $q(z|x)$ being a dirac delta distribution
- Gaussian posterior, i.e. the encoder predicts a mean and std per latent
- A universal approximator posterior, i.e. $q(z|x)$ being arbitrary complex distributions

Question 2.2 (2 points)

Explain how adversarial auto-encoders can reduce the mode collapse problem compared to the vanilla GAN.

2.1 Training objective: A Minimax Game

The training objective for adversarial autoencoder is a mix of the *vanilla* GAN and the Autoencoder training objectives: a two-player minimax game. The adversarial autoencoder has three main parts, as shown in Figure 5. The encoder first encodes the input to a hidden code vector, from which the decoder tries to reconstruct the original input. Meanwhile, the encoder also tries to match the distribution of the latent codes to a particular prior distribution. In the adversarial objective, the real vector codes come from the prior distribution. The discriminator is then trained to distinguish between hidden code vectors that are created from input data, i.e. $q(\mathbf{z}|\mathbf{x})$, and the ones that are drawn from the prior distribution, $p(\mathbf{z})$. Note that here samples from the prior distribution are "true" samples, while hidden code vectors from $q(\mathbf{z}|\mathbf{x})$ / $q(\mathbf{z})$ are the "fake" examples for the Discriminator.

For a more mathematical rigorous explanation, let us define the Discriminator, Encoder and Decoder respectively with functions D, E and R (R for reconstruction function). Further, suppose we define the adversarial loss with $V(D, E)$ and the reconstruction loss with $Rec(R, E)$. With the hyperparameter $0 \leq \lambda \leq 1$, we can define the adversarial

autoencoder loss as:

$$\begin{aligned} &\min_{E,R} \{(1-\lambda) \cdot \max_{D} V(D,E) + \lambda \cdot Rec(R,E)\} \\ &= \min_{E,R}\{(1-\lambda) \cdot \ (\max_{D} \underline{\mathbb{E}_{\boldsymbol{p(x)}}[\mathbf{log(1-D(E(x)))}]} + \underline{\mathbb{E}_{\boldsymbol{p(z)}}[\mathbf{log\,(D\,(z))}]}) \\ &\qquad\qquad + \lambda \cdot \ \underline{\mathbb{E}_{\boldsymbol{p(x)}}[||\boldsymbol{x} - \boldsymbol{R(E(x))}||^{\mathbf{2}}]}\} \end{aligned} \tag{15}$$

In this equation, $p(\mathbf{x})$ is the data distribution that our input samples are coming from, while $p(\mathbf{z})$ is the targeted prior distribution that we want our hidden vector code matches. Finally, λ shows how much we want the adversarial autoencoder emphasis on reconstruction part as opposed to $1-\lambda$ for the adversarial part. If we consider the optimization as a game, a traditional GAN is a two player game between the Generator and the Discriminator. Each one's goal is achieved if the other player loses. For a traditional or variational autoencoder, the decoder and encoder's goal is to collaborate with each other to maximize the reconstruction.

In adversarial autoencoders, the encoder, which is also the generator of the latent vectors, tries to defeat the discriminator. Simultaneously, it collaborates with the decoder to reconstruct the input. Meanwhile, the discriminator tries to defeat the encoder by detecting codes from the prior distribution $p(\mathbf{z})$ versus those created by the actual input, i.e. $q(\mathbf{z}|\mathbf{x})$ for some $\mathbf{x}$.

As we can see from the Equation 15, there are three different terms in adversarial autoencoder objective function. The first term is calculated for the discriminator D applied to the encoder E for samples from the data distribution:

$$\mathbb{E}_{p(\mathbf{x})}[(1 - \log D(E(x)))] \tag{16}$$

The second term applies D to samples drawn from the prior distribution:

$$\mathbb{E}_{p(\mathbf{z})}[\log\,(D\,(z))] \tag{17}$$

Finally, the third term is calculated using the decoder R and encoder E for samples from the data distribution:

$$\mathbb{E}_{p(\mathbf{x})}[||x - R(E(x))||^2] \tag{18}$$

Question 2.3 (2 points)

a) Explain the three terms in the adversarial autoencoder training objective defined in Equation 15 based on what part of the adversarial or reconstruction loss they represent. Refer to equations 16, 17 and 18, for your answer.
b) How does λ affect the training objective? what happens in case of $\lambda = 1$ and $\lambda = 0$?

2.2 Building an Adversarial Autoencoder

Now that we have gained a sound understanding of how adversarial autoencoder works, let's start implementing a simple one. In this exercise, we develop the different modules of an adversarial autoencoder step by step. An adversarial autoencoder can be trained in two phases: *reconstruction* and *regularization*. In phase one, similar to an ordinary autoencoder, we optimize the network to minimize the reconstruction error of the input. For this part, use the 'README.md' as a guide for implementing the adversarial autoencoder. In this assignment, we only will consider deterministic encoder.

2.2.1 Reconstruction Phase

In this phase, the focus of the model is to train its autoencoder part to improve the reconstruction of its input. To this end, a simple MSE loss between the actual input and the reconstructed one is considered.

Question 2.4 (3 points)

In the file 'models.py', complete the encoder and decoder part of the adversarial autoencoder and then complete the 'train.py' to train the network on the MNIST dataset. Show the reconstruction results of an ordinary autoencoder for images with different labels after 0, 20, 50, 100 [a] epochs. Consider the dimension of latent code $z_{dim} = 8$. (First train a model with $\lambda = 1$, i.e. standard autoencoder, and verify that your model is able to reconstruct the images well.)

[a] Instead of 100^{th} epoch, it is fine to show results of 95^{th} epoch too.

2.2.2 Regularization Phase

In regularization phase, the adversarial network first updates its discriminator to distinguish between true samples and fake ones. Then it will update its generator or encoder to match the latent variable distribution with its targeted prior distribution to confuse the discriminator. In our experiments, the prior noise distribution of $p(\mathbf{z})$ will be a standard normal distribution.

Question 2.5 (6 points)

In the file 'models.py', complete the regularization phase by completing the discriminator code for the Adversarial Autoencoder. Train the whole network from scratch, by considering both reconstruction and regularization losses. Show the results of reconstruction and sampling after 0, 20, 50, 100 [a] epochs. For this assignment, you can consider $z_{dim} = 8$ and $\lambda = 0.995$.

Note: using the default hyperparameters is sufficient to obtain full points. As a reference, the reconstruction loss should go below 0.05 with this configuration.

[a] Instead of 100^{th} epoch, it is fine to show results of 95^{th} epoch too.

Question 2.6 (4 points)

a) Report the reconstruction results of a simple autoencoder with $z_{dim} = 2$ after 0, 20, 50, 100 [a] epochs. This time, also report the generated samples results for the autoencoder even though the model does not train the adversarial part.
b) Now report the reconstruction results and generated samples results for an Adversarial autoencoder with the same architecture. Thus $z_{dim} = 2$ and $\lambda = 0.995$.
c) Explain why the generated samples for simple autoencoder and adversarial autoencoder look similar?

[a] Instead of 100^{th} epoch, it is fine to show results of 95^{th} epoch too.

Question 2.7 (2 points)

If we consider $z_{dim} = 20$ and $\lambda = 0.995$, our generated samples will be as follows:

After 0 epochs | After 20 epochs | After 50 epochs | After 100 epochs

Explain why for bigger than 8 latent dimensions, it is recommended to use a stochastic encoder instead of a deterministic one?

3 Attention, Transformers, and LLMs (Total: 10 points)

In this section, we'll delve into theoretical aspects of the Transformer architecture. It's essential to review the lecture focusing on Transformers. Additionally, it is advisable to explore the University of Amsterdam Deep Learning Tutorial 6, which provides insights on Transformers and Multi-Head Attention. After gaining a theoretical understanding, we'll proceed with a practical exercise involving a small-scale implementation of a transformer-based Large Language Model (LLM).

3.1 Attention for Sequence-to-Sequence models

Conventional sequence-to-sequence models for neural machine translation have a difficulty to handle long-term dependencies of words in sentences. In such models, the neural network is compressing all the necessary information of a source sentence into a fixed-length vector. Attention, as introduced by Bahdanau et al. [2015], emerged as a potential solution to this problem. Intuitively, it allows the model to focus on relevant parts of the input, while decoding the output, instead of compressing everything into one fixed-length context vector. The attention mechanism is shown in Figure 6. The complete model comprises an encoder, a decoder, and an attention layer.

Unlike conventional sequence-to-sequence, here the conditional probability of generating the next word in the sequence is conditioned on a distinct context vector c_i for each target word y_i. In particular, the conditional probability of the decoder is defined as:

$$p(y_i|y_1, \ldots, y_{i-1}, x_1, \ldots, x_T) = p(y_i|y_{i-1}, s_i), \tag{19}$$

where s_i is the RNN decoder hidden state for time i computed as $s_i = f(s_{i-1}, y_{i-1}, c_i)$.

The context vector c_i depends on a sequence of annotations $(h_1, \cdots, h_{T_x})$ to which an encoder maps the input sentence; it is computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \tag{20}$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \tag{21}$$

where $e_{ij} = a(s_{i-1}, h_j)$ is an alignment model which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} (just before emitting y_i, Eq. (19)) and the j-th annotation h_j of the input sentence. The alignment model a is parametrized as a feedforward neural network which is jointly trained with all the other components of the proposed system. Use the Figure 6 to understand the introduced notations and the derivation of the equations above.

3.2 Transformer

Transformer is the first encoder-decoder model based solely on (self-)attention mechanisms, without using recurrence and convolutions. The key concepts for understanding Transformers are: queries, keys and values, scaled dot-product attention, multi-head and self-attention.

Queries, Keys, Values The Transformer paper redefined the attention mechanism by providing a generic definition based on queries, keys, values. The encoded representation of the input is viewed as a set of key-value pairs, of input sequence length. The previously generated output in the decoder is denoted as a query.

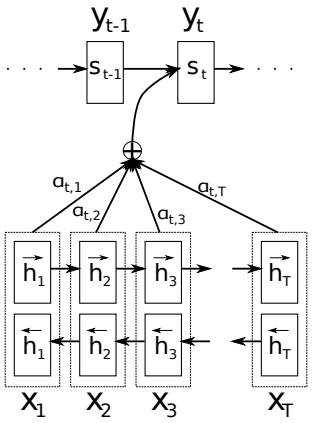

Figure 6. The graphical illustration of the attention mechanism proposed in [Bahdanau et al., 2015]

Scaled dot-product attention In [Vaswani et al., 2017], the authors propose *scaled dot-product attention*. The input consists of queries and keys of dimension d_k, and values of dimension d_v. The attention is computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V. \tag{22}$$

From this equation, it can be noticed that the dot product of the query with all keys represents a matching score between the two representations. This score is scaled by $\frac{1}{\sqrt{d_k}}$, because dot products can grow large in magnitude due to long sequence inputs. After applying a softmax we obtain the weights of the values.

Multi-head attention Instead of performing a single attention function, it is beneficial to linearly project the Q, K and V, h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. The outputs are concatenated and once again projected by an output layer:

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O, \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V).\end{aligned}$$

Self-attention Self-attention (also called intra-attention) is a variant of the attention mechanism relating different positions of a single sequence in order to compute a revised representation of the sequence.

Question 3.1 (3 points)

Discuss the challenge of long input sequence lengths for the Transformer model by:
- Describing the challenge.
- Proposing a way to overcome this challenge.

(Words limit: 200)

3.3 Large Language Models (LLMs)

Transformer-based models, initially developed for natural language processing (NLP), are now pivotal in deep learning. Their key feature, the self-attention mechanism, allows them to process entire sequences of data (like sentences or image pixels) simultaneously.

This contrasts with earlier models that processed data sequentially. In deep learning, this ability facilitates more nuanced understanding and generation of data patterns, whether in text, images, or other complex data types.

LLMs, a specific application of Transformer technology, have become central to deep learning advancements. These models, exemplified by OpenAI's GPT series, are trained on vast amounts of text data. They excel in understanding context, generating human-like text, and learning from new data in a way that mimics human learning. LLMs' deep learning prowess is evident in tasks like translation, content creation, and even in answering complex questions with nuanced understanding.

You are set to implement a small-scale, Transformer-based model for text generation tasks. The great news is that there's no need for you to train the model from scratch. The code is designed to automatically download pre-trained weights, streamlining the process considerably. Your primary task will be to utilize this model to generate content. However, this will involve some hands-on work on your part. Specifically, you'll need to complete the missing lines in certain files and ensure the functions are operational. This step is crucial for the successful application of the model in your text generation project. If everything is filled in correctly, you are also able to train a LLM from scratch yourself!

Question 3.2 (4 points)

In the file 'gpt.py', complete the lines (L90-) of causal self-attention function. Calculate the attention weights and apply the masking, softmax and dropout.

Question 3.3 (3 points)

In the file 'gpt.py', complete the lines (L388-) of generate function. After requiring the probabilities over your vocabulary, implement a greedy algorithm picking the top-1 word and a sampling strategy sampling from a 'torch.multinomial'.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015. URL http://arxiv.org/abs/1409.0473. 15, 16

Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. IEEE Signal Processing Magazine, 35(1):53–65, 2018. 10

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

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, 10, 11

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017. 16

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

$$h(x) = \log\left(\frac{1}{p(x)}\right) = -\log p(x). \tag{23}$$

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:

$$H(p) = \mathbb{E}_{p(x)}\left[-\log p(X)\right] = -\sum_x p(x)\log p(x). \tag{24}$$

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 7, Bernoulli(p) has a high entropy for $p = 0.5$, and a low entropy for $p = 0$ and $p = 1$.

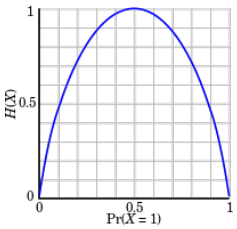

Figure 7. The entropy of a Bernoulli distribution for different values of its parameter p. Source: Wikipedia.

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:

$$H(p,q) = \mathbb{E}_{p(x)}\left[-\log q(X)\right] = -\sum_x p(x)\log q(x). \tag{25}$$

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.