Course: **Generative AI Models (2AMU20)** *Responsible Lecturer:* Jakub Tomczak

Report Assignment 2

Name: Luca Mainardi, *Student ID:* 2014602

Please provide (concise) answers to the questions below. If you do not know an answer, please leave it blank. If necessary, please provide a (relevant) code snippet. If relevant, please remember to support your claims with data/figures.

Question 1 (1pt)

Please explain the reparameterization trick and provide a mathematical formula for it.

Answer The reparameterization trick is a technique used in variational autoencoders (VAEs) to allow the gradient of a stochastic variable to be propagated through a stochastic sampling process during backpropagation. This is crucial for optimizing the parameters of the model using gradient descent because, without this trick, it would be difficult to update the model weights using backpropagation, since, the sampling process is non-differentiable. In VAEs, we aim to approximate a posterior distribution $q(z|x)$, which is often modelled as a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ Kingma and Welling (2013). However, direct sampling from this distribution does not allow gradient propagation through the network. The Reparameterization Trick reformulates the sampling process to make it differentiable.

Mathematical Formulation: The Reparameterization Trick rewrites this sampling as:

$$z = \mu + \sigma \cdot \epsilon$$

where $\epsilon \sim \mathcal{N}(0, 1)$ is standard Gaussian noise Rezende et al. (2014).

In this formulation, the random variable z follows a normal distribution $\mathcal{N}(\mu, \sigma^2)$ parameterized by mean μ and standard deviation σ which are deterministic and differentiable, while ϵ is an additional independent random variable. This allows us to rewrite the sampling as a differentiable function of the parameters μ and σ, enabling the application of gradient-based optimisation methods.

Details

- **Latent Space Representation**: In a VAE, the encoder network maps input data x to the parameters of the latent distribution, typically the mean μ and the standard deviation σ. Mathematically, this can be represented as:
$$\mu, \log(\sigma^2) = f_{\text{encoder}}(x)$$
where f_{encoder} is the encoder neural network.
- **Sampling**: Instead of sampling z directly from $\mathcal{N}(\mu, \sigma^2)$, we sample ϵ from a standard normal distribution $\mathcal{N}(0, 1)$. Then, we reparameterize z as:
$$z = \mu + \sigma \cdot \epsilon$$
- **Differentiability**: This reparameterization allows the gradient to propagate through μ and σ, as the sampling process is now a differentiable operation.

By using this trick, the gradients can now flow through μ and σ during backpropagation, allowing us to optimize the encoder network parameters Kingma and Welling (2019).

Question 2 (2pts)

Please write down mathematically the log-probability of the encoder (variational posterior) for a Gaussian distribution with a diagonal covariance matrix.

Answer In a Variational Autoencoder (VAE), we approximate the true posterior $p(z|x)$ with a variational posterior $q(z|x)$, which we assume to be a Gaussian distribution with a diagonal covariance matrix. Mathematically, this can be expressed as:

$$q(z|x) = \mathcal{N}(z; \mu(x), \Sigma(x))$$

where $\mu(x)$ is the mean vector and $\Sigma(x)$ is the diagonal covariance matrix Bishop (2006).

The log-probability of the encoder (variational posterior) is given by the logarithm of the probability density function of the multivariate Gaussian distribution with a diagonal covariance matrix Tomczak (2022):

$$\mathcal{N}(z; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(z-\mu)^T\Sigma^{-1}(z-\mu)\right)$$

Since Σ is diagonal, let $\Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2)$, where d is the dimensionality of the latent space. The determinant $|\Sigma|$ is the product of its diagonal elements, and the inverse Σ^{-1} is the diagonal matrix with elements $1/\sigma_i^2$. Thus, the log-probability is:

$$\log q(z|x) = \log\left(\frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(z-\mu)^T\Sigma^{-1}(z-\mu)\right)\right)$$

Breaking this down, we get:

$$\log q(z|x) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}(z-\mu)^T\Sigma^{-1}(z-\mu)$$

Since Σ is diagonal:

$$\log|\Sigma| = \log\left(\prod_{i=1}^{d}\sigma_i^2\right) = \sum_{i=1}^{d}\log(\sigma_i^2) \quad \text{and} \quad (z-\mu)^T\Sigma^{-1}(z-\mu) = \sum_{i=1}^{d}\frac{(z_i-\mu_i)^2}{\sigma_i^2}$$

Putting these together, we have:

$$\log q(z|x) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\sum_{i=1}^{d}\log(\sigma_i^2) - \frac{1}{2}\sum_{i=1}^{d}\frac{(z_i-\mu_i)^2}{\sigma_i^2}$$

Question 3 (4pts)

Please do the following:

1. *(1pt) Please explain your choice of the distribution (the conditional likelihood) for image data used in this assignment. Please remember to motivate it properly!*
2. *(1pt) Please write the conditional likelihood down mathematically (i.e., present it as the log-probability).*
3. *(2pts) Please explain how one can sample from the distribution chosen by you. Please provide a mathematical formula and a code snippet.*

Answer

1. In the context of generative models, specifically Variational Autoencoders (VAEs), conditional likelihood is a crucial component. It represents the probability of the observed data given the latent variables. Mathematically, the conditional likelihood of a data point $\mathbf{x}$ given a latent variable $\mathbf{z}$ is denoted as $p(\mathbf{x}|\mathbf{z})$. We often use the log-probability form to work with this in optimization frameworks. In this assignment, the chosen distribution for the conditional likelihood $p(x|z)$ when dealing with image data is a Gaussian distribution. This choice is based on several key motivations outlined below:

   (a) **Natural Fit for Continuous Data**: Image pixel values are continuous, ranging from 0 to 255 in standard 8-bit grayscale images or 0.0 to 1.0 when normalized. The Gaussian distribution is naturally suited to model continuous data due to its ability to represent data spread (variance) and central tendency (mean).

   (b) **Flexibility**: The Gaussian distribution is highly flexible, capable of modelling a wide range of data distributions by adjusting its mean (μ) and variance (σ^2). This flexibility allows the model to learn the underlying data distribution more effectively.

(c) **Reparameterization Trick**: The Gaussian distribution facilitates the use of the reparameterization trick, essential for backpropagation in VAEs.

(d) **Mathematical Tractability**: The Gaussian distribution is mathematically tractable, allowing for analytical solutions to the Kullback-Leibler (KL) divergence term in the VAE loss function. This tractability simplifies the implementation and optimization of the VAE model.

2. The conditional likelihood in the context of Variational Autoencoders (VAEs) for continuous image data, modelled as a Gaussian distribution, can be written down mathematically as follows:

   Given:

   - x: the observed data (image data).
   - z: the latent variable.
   - $\mu(z)$: the mean of the Gaussian distribution, parameterized by the decoder network.
   - σ^2: the variance of the Gaussian distribution, often assumed to be a constant or a diagonal covariance matrix for simplicity.

   The conditional likelihood $p(x|z)$ is:

   $$p(x|z) = \mathcal{N}(x; \mu(z), \sigma^2 I)$$

   The log-likelihood for a single data point x given the latent variable z is:

   $$\log p(x|z) = -\frac{1}{2}\left(\log(2\pi\sigma^2) + \frac{(x-\mu(z))^2}{\sigma^2}\right)$$

   When dealing with a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$ consisting of N samples, we sum the log-probabilities over all data points to obtain the total log-likelihood is:

   $$\log p(X|Z) = \sum_{i=1}^{N} \log p(x_i|z_i)$$

   where X and Z represent the sets of all data points and their corresponding latent variables, respectively.

   This summation is a key term in the Evidence Lower Bound (ELBO) objective, which VAEs maximize during training. Maximizing the ELBO ensures that the generative model produces data that closely matches the true data distribution while maintaining a well-structured latent space.

3. To sample from the learned latent distribution in a Variational Autoencoder (VAE), we typically assume a Gaussian distribution for the latent variables $\mathbf{z}$. The encoder network maps an input $\mathbf{x}$ to the parameters of this Gaussian distribution: the mean μ and the standard deviation σ.

   Mathematical Formulation: In a Variational Autoencoder (VAE), the variational posterior $q(z|x)$ is modeled as a Gaussian distribution with a mean vector $\mu(x)$ and a diagonal covariance matrix represented by the standard deviation vector $\sigma(x)$. Mathematically, this is expressed as:

   $$q(z|x) = \mathcal{N}(z; \mu(x), \text{diag}(\sigma^2(x))).$$

   To sample from this distribution, we use the reparameterization trick, which allows us to backpropagate through the sampling operation. The reparameterization trick expresses the sampling operation as:

   $$z = \mu(x) + \sigma(x) \odot \epsilon,$$

   where $\epsilon \sim \mathcal{N}(0, I)$ (a standard normal distribution), and $\odot$ denotes element-wise multiplication.

Encorder Class

```
    class Encoder(nn.Module):
    def __init__(self, encoder_net):
        super(Encoder, self).__init__()

        # The init of the encoder network.
        self.encoder = encoder_net

    # The reparameterization trick for Gaussians.
    @staticmethod
    def reparameterization(mu, log_var):
        # The formulat is the following:
        # z = mu + std * epsilon
        # epsilon ~ Normal(0,1)

        # First, we need to get std from log-variance.
        std = torch.exp(0.5*log_var)

        # Second, we sample epsilon from Normal(0,1).
        eps = torch.randn_like(std)

        # The final output
        return mu + std * eps

    # This function implements the output of the encoder network (i.e., parameters
    of a Gaussian).
    def encode(self, x):
        # First, we calculate the output of the encoder netowork of size 2M.
        h_e = self.encoder(x)
        # Second, we must divide the output to the mean and the log-variance.
        mu_e, log_var_e = torch.chunk(h_e, 2, dim=1)

        return mu_e, log_var_e

    # Sampling procedure.
    def sample(self, x=None, mu_e=None, log_var_e=None):
        #If we don't provide a mean and a log-variance, we must first calcuate it:
        if (mu_e is None) and (log_var_e is None):
            mu_e, log_var_e = self.encode(x)
        # Or the final sample
        else:
        # Otherwise, we can simply apply the reparameterization trick!
            if (mu_e is None) or (log_var_e is None):
                raise ValueError('mu and log-var can't be None!')
        z = self.reparameterization(mu_e, log_var_e)
        return z

    # This function calculates the log-probability that is later used for
    calculating the ELBO.
    def log_prob(self, x=None, mu_e=None, log_var_e=None, z=None):
        # If we provide x alone, then we can calculate a corresponsing sample:
        if x is not None:
            mu_e, log_var_e = self.encode(x)
            z = self.sample(mu_e=mu_e, log_var_e=log_var_e)
        else:
        # Otherwise, we should provide mu, log-var and z!
            if (mu_e is None) or (log_var_e is None) or (z is None):
                raise ValueError('mu, log-var and z can't be None!')

        return log_normal_diag(z, mu_e, log_var_e)

    # PyTorch forward pass: it is either log-probability (by default) or sampling.
    def forward(self, x, type='log_prob'):
        assert type in ['encode', 'log_prob'], 'Type could be either encode or
    log_prob'
        if type == 'log_prob':
            return self.log_prob(x)
        else:
            return self.sample(x)
```

Explanation

(a) **Compute the Standard Deviation**: Since the encoder outputs the log-variance ($\log \sigma^2$), we compute the standard deviation by taking the exponential of half the log-variance.

(b) **Sample from Standard Normal**: ϵ is sampled from a standard normal distribution using `torch.randn_like(std)`, which generates a tensor of random values with the same shape as `std`.

(c) **Reparameterization**: The sample $\mathbf{z}$ is obtained using the reparameterization trick: $\mathbf{z} = \mu + \sigma \cdot \epsilon$.

This approach ensures that the sampling process is differentiable, allowing gradients to propagate through the sampling operation during backpropagation. This is crucial for training the VAE using gradient-based optimization techniques Tomczak (2022).

Question 4 (3pts)

Please do the following:

1. *(1pt) Please explain your prior and write it down mathematically.*
2. *(1pt) Please write down its sampling procedure (incl. a code snippet).*
3. *(1pt) Please write down its log-probability (a mathematical formula).*

Answer

1. In this assignment, the prior distribution $p(z)$ for the latent variable z is modelled as a Mixture of Gaussians (MoG). This prior is more flexible than a standard Gaussian, allowing the model to capture more complex latent structures, indeed, it uses K different components to model a wider range of distributions, including multi-modal distributions. In addition, the MoG prior can act as a regularizer that encourages the latent representations to be well-separated according to the modes of the mixture, this helps prevent overfitting, improve generalization and consequently the generative capabilities of the VAE Tomczak (2022).

   Mathematical Formulation:

   The MoG prior can be mathematically expressed as a weighted sum of K Gaussian components. Each component k has its own mean μ_k and variance σ_k^2, and the mixture is weighted by the mixing coefficients ω_k. Formally, the probability density function of the MoG prior is:

$$p_\lambda(z) = \sum_{k=1}^{K} \omega_k \mathcal{N}(z|\mu_k, \sigma_k^2)$$

   where:

   - K: Number of components
   - μ_k: Means of the Gaussian components
   - σ_k^2: Variances of the Gaussian components (computed as exp(logvars))
   - ω_k: Mixing weights (computed as softmax of `self.w`)
   - $\lambda = \{\{\omega_k\}, \{\mu_k\}, \{\sigma_k^2\}\}$ are trainable parameters.

2. The function `sample` samples a batch of latent variables z_i from the MoG prior. It first samples component indices based on the mixture weights ω_k, and then samples from the corresponding Gaussian distributions.

   (a) **Initialization**: The `means` and `logvars` for each Gaussian component are initialized. While the `w` parameter is used to compute the mixture weights ω_k.

   (b) **Compute Mixture Probabilities**: The mixture weights are computed by applying the softmax function to `self.w` to get the probabilities π_k.

   (c) **Sample Component Indices**: Based on the mixture weights π_k, sample indices to determine from which Gaussian component each sample should be drawn.

   (d) **Sample from the Selected Gaussian Components**: For each selected component, it samples from the corresponding Gaussian distribution using its mean and variance.

Mathematically:

$$z_i \sim \mathcal{N}(\mu_{c_i}, \sigma^2_{c_i})$$

where c_i is the component index sampled from $\{1, \ldots, K\}$ with probability ω_k.

Sampling function

```
def sample(self, batch_size):
    # initialization
    means, logvars = self.get_params()

    # compute mixing probabilities
    w = F.softmax(self.w, dim=0)
    w = w.squeeze()

    # sample component indices
    indexes = torch.multinomial(w, batch_size, replacement=True)

    # Initialize the noise epsilon for sampling
    eps = torch.randn(batch_size, self.L)
    for i in range(batch_size):
        indx = indexes[i]
        m = means[[indx]].to('cuda')
        e = eps[[i]].to('cuda')
        l = torch.exp(logvars[[indx]]).to('cuda')
        #sample from the selected gaussian components
        if i == 0:
            z = m + e * l
        else:
            z = torch.cat((z, m + e * l), 0)
    return z
```

Where `z` represents the sample from the Gaussian component indexed by `indx`.

3. The `log_prob` function calculates the log-probability of a given latent variable z under the MoG prior. This involves calculating the log-probability under each Gaussian component and then combining them using the log-sum-exp trick to ensure numerical stability.

   Mathematically:

$$\log p_\lambda(z) = \log \left(\sum_{k=1}^{K} \omega_k \exp \left(\log \mathcal{N}(z|\mu_k, \sigma_k^2) \right) \right)$$

   where:

$$\log \mathcal{N}(z|\mu_k, \sigma_k^2) = -\frac{1}{2} \left(\log(2\pi) + \log(\sigma_k^2) + \frac{(z-\mu_k)^2}{\sigma_k^2} \right)$$

Question 5 (1pt)

Please derive the Negative ELBO including intermediate steps (be as specific as possible).

Answer In a Variational Autoencoder (VAE), we approximate the true posterior $p(z|x)$ with a variational posterior $q(z|x)$, which we assume to be a Gaussian distribution with a diagonal covariance matrix. The goal is to maximize the log-likelihood of the observed data x. This log-likelihood can be decomposed using the Evidence Lower Bound (ELBO) Doersch (2016); Rezende et al. (2014).

1. **Marginal Log-Likelihood**

$$\log p(x) = \log \int p(x, z)\, dz, \tag{0.1}$$

   where z is the latent variable.

2. **Introducing the Variational Distribution** $q(z|x)$

$$\log p(x) = \log \int \frac{p(x, z)}{q(z|x)} q(z|x)\, dz. \tag{0.2}$$

3. **Applying Jensen's Inequality**

$$\log p(x) \geq \mathbb{E}_{q(z|x)} \left[\log \frac{p(x,z)}{q(z|x)} \right]. \tag{0.3}$$

4. **Rewriting the Joint Distribution**

$$\log p(x) \geq \mathbb{E}_{q(z|x)}[\log p(x|z) + \log p(z) - \log q(z|x)]. \tag{0.4}$$

5. **Separating the Expectation Terms**

$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x|z)] + \mathbb{E}_{q(z|x)}[\log p(z) - \log q(z|x)]. \tag{0.5}$$

6. **Recognizing the KL Divergence**

$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \text{KL}(q(z|x)\|p(z)), \tag{0.6}$$

where the KL divergence is defined as:

$$\text{KL}(q(z|x)\|p(z)) = \mathbb{E}_{q(z|x)} \left[\log \frac{q(z|x)}{p(z)} \right]. \tag{0.7}$$

7. **Final Form of the Negative ELBO**:

$$-\text{ELBO} = -\mathbb{E}_{q(z|x)}[\log p(x|z)] + \text{KL}(q(z|x)\|p(z)). \tag{0.8}$$

The first term represents the reconstruction error which measures how well the VAE can reconstruct the data x from the latent representation z, and the second term is a regularization term that ensures the learned distribution $q(z|x)$ is close to the prior $p(z)$.

Question 6 (1pt)

Please explain your choice of the optimizer, and comment on the choice of the hyperparameters (e.g., the learning rate value).

Answer For training our Variational Autoencoder (VAE), we selected the `Adamax` optimizer from the `torch.optim` library. This choice is motivated by the following reasons:

- **Adaptive Learning Rate:** Like Adam, `Adamax` incorporates adaptive learning rate adjustments for each parameter. This feature is particularly advantageous when dealing with high-dimensional data or models where different parameters exhibit gradients of varying magnitudes. The adaptive nature helps in stabilizing the training process and speeds up convergence Kingma and Ba (2014).
- **Handling Sparse Gradients:** `Adamax` performs well with sparse gradients, a common occurrence in large-scale neural networks. This robustness arises from its use of the infinity norm ($\|\cdot\|_\infty$) in the denominator of the update rule, which can effectively manage the sparsity and ensure more consistent updates across all parameters Reddi et al. (2018).
- **Empirical Performance:** Empirical studies and past research have demonstrated that `Adamax` can outperform other optimizers in terms of both convergence speed and final model performance for certain types of neural networks, including VAEs. Its stability and reliability make it a suitable choice for complex models.

Optimizer

```
lr = 1e-3 # learning rate
optimizer = torch.optim.Adamax([p for p in model.parameters() if p.requires_grad == True
    ], lr=lr)
```

The only hyperparameter we focused on was the learning rate. For `Adamax`, we set the learning rate (`lr`) to 1×10^{-3}. This value was chosen based on theoretical considerations and empirical tuning.

- **Theoretical Considerations:** The default learning rate for `Adamax` in the literature and many practical implementations is 0.002. However, we opted for a slightly lower value to ensure more stable updates during the initial training epochs, which can be crucial for the convergence of VAEs. Indeed, the theoretical guidelines suggesting that the learning rate should be small enough to ensure convergence but large enough to escape local minima efficiently
- **Empirical Tuning:** Through cross-validation and experimentation, we observed that a learning rate of 1×10^{-3} provided a good balance between convergence speed and model stability. Higher learning rates tended to cause divergence or unstable training dynamics, while lower learning rates significantly increased the time to convergence without noticeable improvements in model performance.

Question 7 (1pt)

Please show 16 real images and the final 16 generated images from a fully-trained model. By looking at the generations, was the model properly trained? Please motivate your answer well.

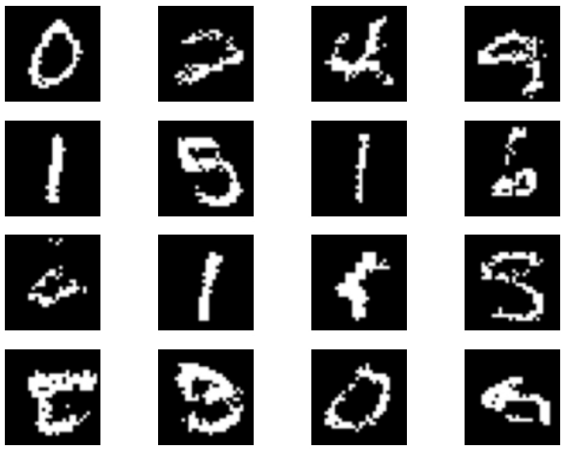

Figure 1: Generated Images

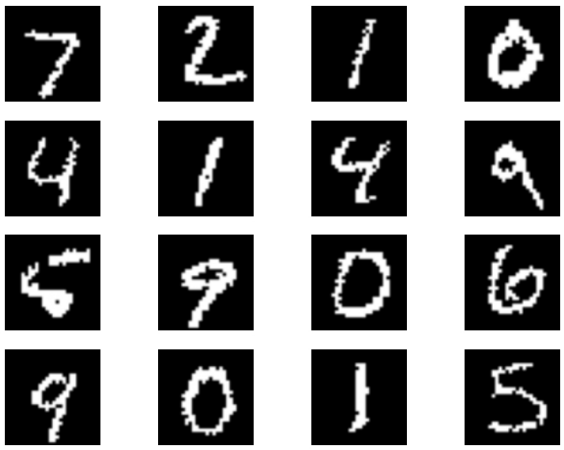

Figure 2: Real Images

```
FINAL LOSS: nll=92.0731965576172
```

Answer By looking at the generations, we can easily deduce that the model wasn't properly trained. Indeed, after comparing the generated images with the real ones, we can make some considerations to support our claim. First of all, it emerges that some of the digits generated by the VAE model are recognizable and visually similar to hand-written digits, however, this does not apply to all images, in fact only about 8 out of 16 show an acceptable quality. This indicates that the model has a partial ability to learn and generate realistic figures, but has failed to generalize completely. Moreover, the generated images do not show a wide variety of styles and shapes, suggesting a limited and less diversified latent representation. Further confirmation of our assessment is the final negative log-likelihood (NLL) of 92.0731965576172, which is relatively high, indicating significant inaccuracies in the reconstruction of images and a discrepancy between the distribution of the original data and that learned from the model. These findings suggest that the model, while capable of generating some realistic images, fails to maintain sufficient consistency with the original input and does not adequately capture the variety of digits present in the MNIST dataset. This may be due to problems in model structure, insufficient training, or latent representation that is not adequately constrained.

Question 8 (3pts)

Please provide answers to the following questions:

1. *(1pt) What are the potential problems with evaluating generative models by looking at generated data?*
2. *(2pts) How can we evaluate the perceptual quality of generative models (NOTE: ELBO or NLL do not count as answers)? Please provide two specific quantitative metrics (mathematical formulas and explanations). At most two, 1pt per metric.*

Answer

1. Evaluating generative models solely based on the inspection of generated data can present several significant challenges:

   - **Subjectivity:** Visual inspection relies heavily on human judgment, which is inherently subjective. Different observers may have varying opinions on what constitutes a "good" sample.
   - **Bias:** Evaluators may possess inherent biases based on their expectations or previous experiences with similar data, leading to inconsistent evaluations.
   - **Time-Consuming:** Evaluating large datasets by visual inspection is impractical and time-consuming. It is not feasible to assess thousands or millions of samples manually.
   - **Limited Sample Size:** Human evaluators can only assess a limited number of samples, which may not be representative of the overall performance of the model.
   - **Lack of Standard Metrics:** Visual inspection does not provide a quantitative metric that can be used to objectively compare different models. This makes it difficult to benchmark performance across different models or improvements over time.
   - **Reproducibility:** Quantitative metrics are essential for reproducibility in scientific research. Without them, it is challenging to replicate results and verify claims.
   - **Mode Collapse:** Generative models can suffer from mode collapse, where the model generates a limited variety of samples, missing out on other modes in the data distribution. Visual inspection might not detect this issue if the evaluator does not see the full range of diversity in the data.
   - **Misleading Samples:** A small set of visually inspected samples might look good, but the overall distribution might still be flawed.
   - **Overfitting to Human Preferences:** If models are tuned based on visual inspection, there is a risk of overfitting to human preferences rather than improving the underlying generative process. This can lead to models that produce visually pleasing but statistically invalid samples.

2. To evaluate the perceptual quality of generative models, two commonly used quantitative metrics are the Inception Score (IS) and the Fréchet Inception Distance (FID). These metrics focus on the visual quality and diversity of the generated samples.

   Inception Score (IS) The Inception Score uses a pre-trained Inception v3 model to assess the quality of generated images. The score considers two main aspects: the clarity of individual samples and the diversity across multiple samples Barratt and Sharma (2018). It is defined as follows:

   $$IS(G) = \exp\left(\mathbb{E}_{\mathbf{x}\sim p_g}\left[\mathrm{KL}(p(y|\mathbf{x})\|p(y))\right]\right)$$

   where:

   - G is the generative model.
   - $\mathbf{x}$ represents the generated samples.
   - p_g is the distribution of generated samples.
   - $p(y|\mathbf{x})$ is the conditional label distribution given by the Inception v3 network for a generated image $\mathbf{x}$.
   - $p(y) = \int p(y|\mathbf{x})p_g(\mathbf{x})d\mathbf{x}$ is the marginal distribution over all generated images.
   - $\mathrm{KL}(p\|q)$ represents the Kullback-Leibler divergence between distributions p and q.

   Explanation:

   - The Inception Score rewards generative models that produce images which are easily classifiable into a single category (high $p(y|\mathbf{x})$) and that also generate a diverse set of images covering all possible categories (high entropy of $p(y)$).
   - A high IS indicates both high quality and high diversity in the generated images.

Fréchet Inception Distance (FID) The Fréchet Inception Distance measures the distance between the distributions of real and generated images in the feature space of a pre-trained Inception v3 model Heusel et al. (2017). It is defined as follows:

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$$

where:

- μ_r and μ_g are the means of the feature vectors of the real and generated samples, respectively.
- Σ_r and Σ_g are the covariance matrices of the feature vectors of the real and generated samples, respectively.
- $\|\cdot\|_2$ represents the Euclidean norm.
- $\text{Tr}(\cdot)$ denotes the trace of a matrix.

Explanation:

- The FID score measures the distance between the multivariate Gaussian distributions fitted to the real and generated data in the feature space of the Inception network.
- A lower FID indicates that the distributions of real and generated images are more similar, implying that the generated images are of higher perceptual quality and more closely resemble the real data distribution.

Both Inception Score and Fréchet Inception Distance provide quantitative ways to evaluate the perceptual quality of generative models, focusing on the clarity, diversity, and similarity to real data. The IS captures the quality and diversity of individual images, while the FID provides a holistic measure of the overall similarity between real and generated data distributions.

Question 9 (4pt)

After training, a validation loss curve is plotted. Please comment on the following:

1. *(1pt) Please add the plot. Based on that plot, can you say that the training of your VAE is stable or unstable? Why?*
2. *(3pts) Please run your model with three different sets of hyperparameters. Are the hyperparameter values of the optimizer important and how do they influence the training? Motivate well your answer (e.g., run the script with three values of the learning rate, present three plots here and provide a discussion).*

Answer

1. Observing the trend of the negative log-likelihood (NLL) during the training of the model we can observe a clear tendency to decrease rapidly in the first 50 epochs where the NLL decreases rapidly from about 135 to about 100, This indicates that the model is learning effectively and is rapidly improving its ability to reconstruct input images. Then, it is followed by a period of slow and constant reduction, until stabilizing around a value of about 92 after about 200 epochs but without significant fluctuations or sudden increases, the NLL remains relatively stable throughout the training. This is a positive indicator of stability since it means that the model is not developing numerical instability or difficulty, therefore, we can conclude that the VAE training is stable.

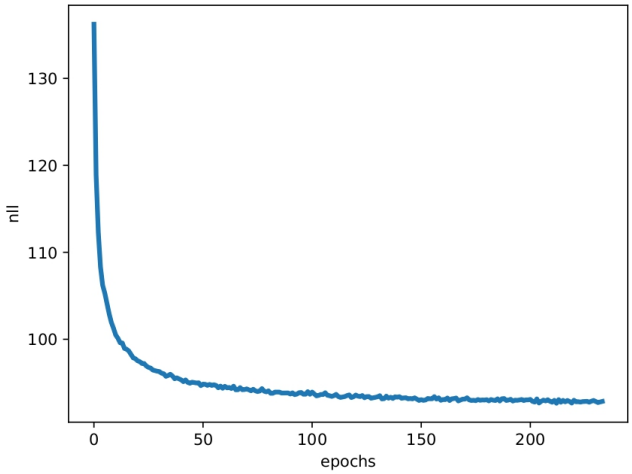

Figure 3: `learning_rate = 1e-3`

2. The hyperparameters values of the optimizer, especially the learning rate, play a crucial role in training a machine learning model. The learning rate determines the size of the steps the optimizer takes during gradient descent, directly influencing the quality, speed and stability of the model's convergence. Therefore, the selection of a balanced learning rate is essential to ensure that the model can generate good results but above all learn effectively and stably.

   Here, we evaluate the influence of the learning rate values on training a Variational Autoencoder (VAE), presenting and comparing the negative log-likelihood (NLL) plots obtained after the training of two new models using respectively learning rates equal to 1×10^{-5}, and 1×10^{-2} with the initial model in terms of *Speed, Stability and Convergence*.

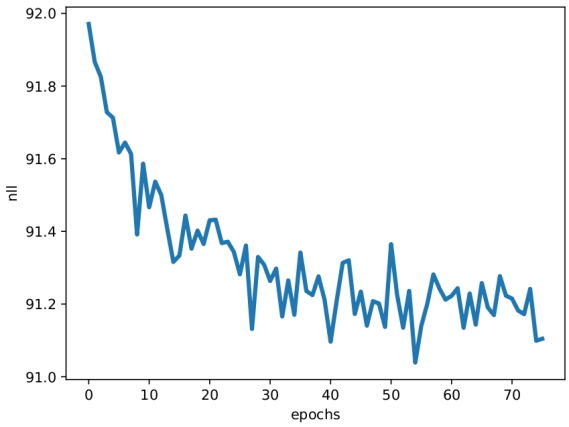

Figure 4: `learning_rate = 1e-5`

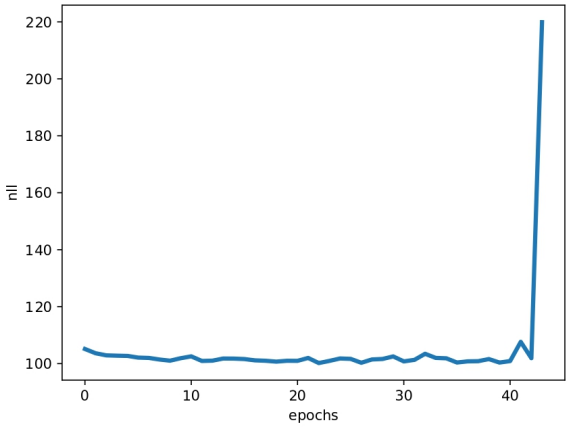

Figure 5: `learning_rate = 1e-2`

Influence of Learning Rate on training

- **Learning Rate** $= 1 \times 10^{-3}$:
  - **Speed**: A learning rate of 1×10^{-3} allows the model to learn quickly initially, as indicated by the rapid drop in NLL during the first 50 epochs.
  - **Stability**: After the initial phase, the NLL decreases slowly and stabilizes. This suggests the model can make fine adjustments without significant fluctuations, indicating good training stability.
  - **Convergence**: The stabilization of NLL indicates that the model is converging towards an optimal solution.
- **Learning Rate** $= 1 \times 10^{-5}$:

  - **Speed**: A learning rate of 1×10^{-5} is too low, resulting in extremely slow learning. This can be observed from the fact that the NLL does not significantly decrease even after many epochs.
  - **Stability**: Although the training appears stable, the slow learning rate is counterproductive, as it requires an excessive number of epochs to reach a good solution.
  - **Convergence**: The risk with such a low learning rate is that the model may get stuck in local minima or simply not reach an optimal solution in a reasonable time.
- **Learning Rate** $= 1 \times 10^{-2}$:
  - **Speed**: A learning rate of 1×10^{-2} allows very large updates to the model weights, which may initially seem advantageous for a rapid drop in NLL.
  - **Stability**: However, the plot clearly shows significant oscillations and a sudden spike in NLL, indicating instability. The model fails to stabilize, leading to potential continuous oscillations and lack of convergence.
  - **Convergence**: The presence of spikes and oscillations in NLL suggests that the model may not be able to converge to an optimal solution, risking overshooting the minima of the cost function.

References

Barratt, S. and Sharma, R. (2018). A note on the inception score. *arXiv preprint arXiv:1801.01973*.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Doersch, C. (2016). Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Kingma, D. P. and Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392.

Reddi, S. J., Kale, S., and Kumar, S. (2018). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286. PMLR.

Tomczak, J. M. (2022). *Deep Generative Modeling*. Springer Nature.

Assignment 2 - Variational Auto-Encoders

Generative AI Models 2024

Instructions on how to use this notebook:

This notebook is hosted on `Google Colab`. To be able to work on it, you have to create your own copy. Go to *File* and select *Save a copy in Drive*.

You can also avoid using `Colab` entirely, and download the notebook to run it on your own machine. If you choose this, go to *File* and select *Download .ipynb*.

The advantage of using **Colab** is that you can use a GPU. You can complete this assignment with a CPU, but it will take a bit longer. Furthermore, we encourage you to train using the GPU not only for faster training, but also to get experience with this setting. This includes moving models and tensors to the GPU and back. This experience is very valuable because for various models and large datasets (like large CNNs for ImageNet, or Transformer models trained on Wikipedia), training on GPU is the only feasible way.

The default `Colab` runtime does not have a GPU. To change this, go to *Runtime - Change runtime type*, and select *GPU* as the hardware accelerator. The GPU that you get changes according to what resources are available at the time, and its memory can go from a 5GB, to around 18GB if you are lucky. If you are curious, you can run the following in a code cell to check:

```
!nvidia-smi
```

Note that despite the name, `Google Colab` does not support collaborative work without issues. When two or more people edit the notebook concurrently, only one version will be saved. You can choose to do group programming with one person sharing the screen with the others, or make multiple copies of the notebook to work concurrently.

Submission: Please bring your (partial) solution to instruction sessions. Then you can discuss it with intructors and your colleagues.

```
!nvidia-smi
```

```
Sun Jun  9 20:52:45 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   40C    P8              9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Introduction

In this assignment, we are going to implement a Variational Auto-Encoder (VAE). A VAE is a likelihood-based deep generative model that consists of a stochastic encoder (a variational posterior over latent variables), a stochastic decoder, and a marginal distribution over latent variables (a.k.a. a prior). The model was originally proposed in two concurrent papers:

- Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114.
- Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra. "Stochastic backpropagation and approximate inference in deep generative models." International conference on machine learning. PMLR, 2014.

You can read more about VAEs in Chapter 4 of the following book:

- Tomczak, J.M., "Deep Generative Modeling", Springer, 2022

In particular, the goals of this assignment are the following:

- Understand how VAEs are formulated.
- Implement components of VAEs using PyTorch.
- Train and evaluate your VAE model on image data.

This notebook is essential for preparing a report. Moreover, please remember to submit the final notebook together with the report (PDF).

Theory behind VAEs

VAEs are latent variable models trained with variational inference. In general, the latent variable models define the following generative process:

$$1.\ \mathbf{z} \sim p_\lambda(\mathbf{z})$$
$$2.\ \mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$$

In plain words, we assume that for observable data $\mathbf{x}$, there are some latent (hidden) factors $\mathbf{z}$. Then, the training objective is log-likelihood function of the following form:

$$\log p_\vartheta(\mathbf{x}) = \log \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\lambda(\mathbf{z}) \mathrm{d}\mathbf{z}.$$

The problem here is the intractability of the integral if the dependencies between random variables $\mathbf{x}$ and $\mathbf{z}$ are non-linear and/or the distributions are non-Gaussian.

By introducing variational posteriors $q_\phi(\mathbf{z}|\mathbf{x})$, we get the following lower bound (the Evidence Lower Bound, ELBO):

$$\log p_\vartheta(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \left[\log p_\theta(\mathbf{x} \mid \mathbf{z})\right] - \mathrm{KL}\left(q_\phi(\mathbf{z} \mid \mathbf{x}) \| p_\lambda(\mathbf{z})\right).$$

IMPORTS

```
# DO NOT REMOVE!
import os

import numpy as np
import matplotlib.pyplot as plt

import torch

from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F

import torchvision
from torchvision.datasets import MNIST
```

```
# DO NOT REMOVE!
# Check if GPU is available and determine the device
if torch.cuda.is_available():
  device = 'cuda'
else:
  device = 'cpu'

print(f'The available device is {device}')
```

```
The available device is cuda
```

```
# DO NOT REMOVE! (unless you work locally)
# mount drive: WE NEED IT FOR SAVING IMAGES! NECESSARY FOR GOOGLE COLAB!
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
# DO NOT REMOVE! (unless you work locally)
# PLEASE CHANGE IT TO YOUR OWN GOOGLE DRIVE!
images_dir = '/content/gdrive/My Drive/Results/'
```

Auxiliary functions

Let us define some useful log-distributions:

```
# DO NOT REMOVE
PI = torch.from_numpy(np.asarray(np.pi))
EPS = 1.e-5

def log_categorical(x, p, num_classes=256, reduction=None):
    x_one_hot = F.one_hot(x.long(), num_classes=num_classes)
    log_p = x_one_hot * torch.log(torch.clamp(p, EPS, 1. - EPS))
    if reduction == 'mean':
        return torch.mean(log_p, list(range(1, len(x.shape))))
    elif reduction == 'sum':
        return torch.sum(log_p, list(range(1, len(x.shape))))
    else:
        return log_p

def log_bernoulli(x, p, reduction=None):
    pp = torch.clamp(p, EPS, 1. - EPS)
    log_p = x * torch.log(pp) + (1. - x) * torch.log(1. - pp)
    if reduction == 'mean':
        return torch.mean(log_p, list(range(1, len(x.shape))))
    elif reduction == 'sum':
        return torch.sum(log_p, list(range(1, len(x.shape))))
    else:
        return log_p

def log_normal_diag(x, mu, log_var, reduction=None):
    D = x.shape[1]
    log_p = -0.5 * torch.log(2. * PI) - 0.5 * log_var - 0.5 * torch.exp(-log_var) * (x - mu)**2.
    if reduction == 'mean':
        return torch.mean(torch.sum(log_p, list(range(1, len(x.shape)))))
    elif reduction == 'sum':
        return torch.sum(torch.sum(log_p, list(range(1, len(x.shape)))))
    else:
        return log_p


def log_standard_normal(x, reduction=None):
    D = x.shape[1]
    log_p = -0.5 * torch.log(2. * PI) - 0.5 * x**2.
    if reduction == 'mean':
        return torch.mean(log_p, list(range(1, len(x.shape))))
    elif reduction == 'sum':
        return torch.sum(log_p, list(range(1, len(x.shape))))
    else:
        return log_p
```

Implementing VAEs

The goal of this assignment is to implement four classes:

- `Encoder` : this class implements the encoder (variational posterior), $q_\phi(\mathbf{z}|\mathbf{x})$.
- `Decoder` : this class implements the decoded (the conditional likelihood), $p_\theta(\mathbf{x}|\mathbf{z})$.
- `Prior` : this class implements the marginal over latents (the prior), $p_\lambda(\mathbf{z})$.
- `VAE` : this class combines all components.

Encoder

We start with `Encoder` . Please remember that we assume the Gaussian variational posterior with a diagonal covariance matrix.

```
# YOUR CODE GOES IN THIS CELL
# NOTE: The class must containt the following functions:
# (i) reparameterization
# (ii) sample
# Moreover, forward must return the log-probability of variational posterior for given x, i.e., log q(z|x)

class Encoder(nn.Module):
    def __init__(self, encoder_net):
        super(Encoder, self).__init__()

        # The init of the encoder network.
        self.encoder = encoder_net

    # The reparameterization trick for Gaussians.
    @staticmethod
    def reparameterization(mu, log_var):
        # The formulat is the following:
        # z = mu + std * epsilon
        # epsilon ~ Normal(0,1)

        # First, we need to get std from log-variance.
        std = torch.exp(0.5*log_var)

        # Second, we sample epsilon from Normal(0,1).
        eps = torch.randn_like(std)

        # The final output
        return mu + std * eps

    # This function implements the output of the encoder network (i.e., parameters of a Gaussian).
    def encode(self, x):
        # First, we calculate the output of the encoder netowork of size 2M.
        h_e = self.encoder(x)
        # Second, we must divide the output to the mean and the log-variance.
        mu_e, log_var_e = torch.chunk(h_e, 2, dim=1)

        return mu_e, log_var_e

    # Sampling procedure.
    def sample(self, x=None, mu_e=None, log_var_e=None):
        #If we don't provide a mean and a log-variance, we must first calcuate it:
        if (mu_e is None) and (log_var_e is None):
            mu_e, log_var_e = self.encode(x)
        # Or the final sample
        else:
        # Otherwise, we can simply apply the reparameterization trick!
            if (mu_e is None) or (log_var_e is None):
                raise ValueError('mu and log-var can`t be None!')
        z = self.reparameterization(mu_e, log_var_e)
        return z

    # This function calculates the log-probability that is later used for calculating the ELBO.
    def log_prob(self, x=None, mu_e=None, log_var_e=None, z=None):
        # If we provide x alone, then we can calculate a corresponsing sample:
        if x is not None:
            mu_e, log_var_e = self.encode(x)
            z = self.sample(mu_e=mu_e, log_var_e=log_var_e)
        else:
        # Otherwise, we should provide mu, log-var and z!
            if (mu_e is None) or (log_var_e is None) or (z is None):
                raise ValueError('mu, log-var and z can`t be None!')

        return log_normal_diag(z, mu_e, log_var_e)

    # PyTorch forward pass: it is either log-probability (by default) or sampling.
    def forward(self, x, type='log_prob'):
        assert type in ['encode', 'log_prob'], 'Type could be either encode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x)
        else:
            return self.sample(x)
```

Decoder

The decoder is the conditional likelihood, i.e., $p(x|z)$. Please remember that we must decide on the form of the distribution (e.g., Bernoulli, Gaussian, Categorical).

```
# YOUR CODE GOES IN THIS CELL
# NOTE: The class must containt the following function:
# (i) sample
# Moreover, forward must return the log-probability of the conditional likelihood function for given z, i.e., log p(x|z)
class Decoder(nn.Module):
    def __init__(self, decoder_net, distribution='categorical', num_vals=None):
        super(Decoder, self).__init__()

        # The decoder network.
        self.decoder = decoder_net
        # The distribution used for the decoder (it is categorical by default, as discussed above).
        self.distribution = distribution
        # The number of possible values. This is important for the categorical distribution.
        self.num_vals=num_vals

    # This function calculates parameters of the likelihood function p(x|z)
    def decode(self, z):
        # First, we apply the decoder network.
        h_d = self.decoder(z)

        # In this example, we use only the categorical distribution...
        if self.distribution == 'categorical':
            # We save the shapes: batch size
            b = h_d.shape[0]
            # and the dimensionality of x.
            d = h_d.shape[1]//self.num_vals
            # Then we reshape to (Batch size, Dimensionality, Number of Values).
            h_d = h_d.view(b, d, self.num_vals)
            # To get probabilities, we apply softmax.
            mu_d = torch.softmax(h_d, 2)
            return [mu_d]
        # ... however, we also present the Bernoulli distribution. We are nice, aren't we?
        elif self.distribution == 'bernoulli':
            # In the Bernoulli case, we have x_d \in {0,1}. Therefore, it is enough to output a single probability,
            # because p(x_d=1|z) = \theta and p(x_d=0|z) = 1 - \theta
            mu_d = torch.sigmoid(h_d)
            return [mu_d]

        else:
            raise ValueError('Either `categorical` or `bernoulli`')

    # This function implements sampling from the decoder.
    def sample(self, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            # We take the output of the decoder
            mu_d = outs[0]
            # and save shapes (we will need that for reshaping).
            b = mu_d.shape[0]
            m = mu_d.shape[1]
            # Here we use reshaping
            mu_d = mu_d.view(mu_d.shape[0], -1, self.num_vals)
            p = mu_d.view(-1, self.num_vals)
            # Eventually, we sample from the categorical (the built-in PyTorch function).
            x_new = torch.multinomial(p, num_samples=1).view(b, m)

        elif self.distribution == 'bernoulli':
            # In the case of Bernoulli, we don't need any reshaping
            mu_d = outs[0]
            # and we can use the built-in PyTorch sampler!
            x_new = torch.bernoulli(mu_d)

        else:
            raise ValueError('Either `categorical` or `bernoulli`')

        return x_new

    # This function calculates the conditional log-likelihood function.
    def log_prob(self, x, z):
        outs = self.decode(z)

        if self.distribution == 'categorical':
            mu_d = outs[0]
            log_p = log_categorical(x, mu_d, num_classes=self.num_vals, reduction='sum').sum(-1)

        elif self.distribution == 'bernoulli':
            mu_d = outs[0]
            log_p = log_bernoulli(x, mu_d, reduction='sum')

        else:
            raise ValueError('Either `categorical` or `bernoulli`')
```

```
        return log_p

    # The forward pass is either a log-prob or a sample.
    def forward(self, z, x=None, type='log_prob'):
        assert type in ['decoder', 'log_prob'], 'Type could be either decode or log_prob'
        if type == 'log_prob':
            return self.log_prob(x, z)
        else:
            return self.sample(x)
```

Prior

The prior is the marginal distribution over latent variables, i.e., $p(z)$. It plays a crucial role in the generative process and also in synthesizing images of a better quality.

In this assignment, you are asked to implement a prior that is learnable (e.g., parameterized by a neural network). If you decide to implement the standard Gaussian prior only, then please be aware that you will not get any points.

For the learnable prior you can choose the **Mixture of Gaussians**.

```
# YOUR CODE GOES IN THIS CELL
# NOTES:
# (i) Implementing the standard Gaussian prior does not give you any points!
# (ii) The function "sample" must be implemented.
# (iii) The function "forward" must return the log-probability, i.e., log p(z)

class MoGPrior(nn.Module):
    def __init__(self, L, num_components):
        super(MoGPrior, self).__init__()

        self.L = L
        self.num_components = num_components

        # params
        self.means = nn.Parameter(torch.randn(num_components, self.L))
        self.logvars = nn.Parameter(torch.randn(num_components, self.L))

        # mixing weights
        self.w = nn.Parameter(torch.zeros(num_components, 1, 1))

    def get_params(self):
        return self.means, self.logvars

    def sample(self, batch_size):
        # mu, lof_var
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)
        w = w.squeeze()

        # pick components
        indexes = torch.multinomial(w, batch_size, replacement=True)

        # means and logvars
        eps = torch.randn(batch_size, self.L)
        for i in range(batch_size):
            indx = indexes[i]
            m = means[[indx]].to('cuda')
            e = eps[[i]].to('cuda')
            l = torch.exp(logvars[[indx]]).to('cuda')
            if i == 0:
                z = m + e * l
            else:
                z = torch.cat((z, m + e * l), 0)
        return z

    def log_prob(self, z):
        # mu, lof_var
        means, logvars = self.get_params()

        # mixing probabilities
        w = F.softmax(self.w, dim=0)

        # log-mixture-of-Gaussians
        z = z.unsqueeze(0) # 1 x B x L
        means = means.unsqueeze(1) # K x 1 x L
        logvars = logvars.unsqueeze(1) # K x 1 x L

        log_p = log_normal_diag(z, means, logvars) + torch.log(w) # K x B x L
        log_prob = torch.logsumexp(log_p, dim=0, keepdim=False) # B x L

        return log_prob
```

Complete VAE

The last class is `VAE` tha combines all components. Please remember that this class must implement the **Negative ELBO** in `forward`, as well as `sample` (*hint*: it is a composition of `sample` functions from the prior and the decoder).

```
# YOUR CODE GOES HERE
# This class combines Encoder, Decoder and Prior.
# NOTES:
# (i) The function "sample" must be implemented.
# (ii) The function "forward" must return the negative ELBO. Please remember to add an argument "reduction" that is either "
class VAE(nn.Module):
    def __init__(self, encoder_net, decoder_net, prior, num_vals=256, likelihood_type='categorical'):
        super(VAE, self).__init__()

        self.encoder = Encoder(encoder_net=encoder_net)
        self.decoder = Decoder(distribution=likelihood_type, decoder_net=decoder_net, num_vals=num_vals)
        self.prior = prior

        self.num_vals = num_vals

        self.likelihood_type = likelihood_type

    def forward(self, x, reduction='mean'):
        # encoder
        mu_e, log_var_e = self.encoder.encode(x)
        z = self.encoder.sample(mu_e=mu_e, log_var_e=log_var_e)

        # ELBO
        RE = self.decoder.log_prob(x, z)
        KL = (self.prior.log_prob(z) - self.encoder.log_prob(mu_e=mu_e, log_var_e=log_var_e, z=z)).sum(-1)

        if reduction == 'sum':
            return -(RE + KL).sum()
        else:
            return -(RE + KL).mean()

    def sample(self, batch_size=64):
        z = self.prior.sample(batch_size=batch_size)
        return self.decoder.sample(z)
```

Evaluation and training functions

Please DO NOT remove or modify them.

```
# DO NOT REMOVE
def evaluation(test_loader, name=None, model_best=None, epoch=None):
    # EVALUATION
    if model_best is None:
        # load best performing model
        model_best = torch.load(name + '.model')

    model_best.eval()
    loss = 0.
    N = 0.
    for indx_batch, (test_batch, _) in enumerate(test_loader):
        test_batch = test_batch.to(device)
        loss_t = model_best.forward(test_batch, reduction='sum')
        loss = loss + loss_t.item()
        N = N + test_batch.shape[0]
    loss = loss / N

    if epoch is None:
        print(f'FINAL LOSS: nll={loss}')
    else:
        print(f'Epoch: {epoch}, val nll={loss}')

    return loss


def samples_real(name, test_loader, shape=(28,28)):
    # real images-------
    num_x = 4
    num_y = 4
    x, _ = next(iter(test_loader))
    x = x.to('cpu').detach().numpy()

    fig, ax = plt.subplots(num_x, num_y)
    for i, ax in enumerate(ax.flatten()):
        plottable_image = np.reshape(x[i], shape)
        ax.imshow(plottable_image, cmap='gray')
        ax.axis('off')

    plt.savefig(name+'_real_images.pdf', bbox_inches='tight')
    plt.close()


def samples_generated(name, data_loader, shape=(28,28), extra_name=''):
    x, _ = next(iter(data_loader))
    x = x.to('cpu').detach().numpy()

    # generations-------
    model_best = torch.load(name + '.model')
    model_best.eval()

    num_x = 4
    num_y = 4
    x = model_best.sample(num_x * num_y)
    x = x.to('cpu').detach().numpy()

    fig, ax = plt.subplots(num_x, num_y)
    for i, ax in enumerate(ax.flatten()):
        plottable_image = np.reshape(x[i], shape)
        ax.imshow(plottable_image, cmap='gray')
        ax.axis('off')

    plt.savefig(name + '_generated_images' + extra_name + '.pdf', bbox_inches='tight')
    plt.close()


def plot_curve(name, nll_val):
    plt.plot(np.arange(len(nll_val)), nll_val, linewidth='3')
    plt.xlabel('epochs')
    plt.ylabel('nll')
    plt.savefig(name + '_nll_val_curve.pdf', bbox_inches='tight')
    plt.close()
```

```
# DO NOT REMOVE
def training(name, max_patience, num_epochs, model, optimizer, training_loader, val_loader, shape=(28,28)):
    nll_val = []
    best_nll = 1000.
    patience = 0

    # Main loop
    for e in range(num_epochs):
        # TRAINING
        model.train()
        for indx_batch, (batch, _) in enumerate(training_loader):
            batch = batch.to(device)
            loss = model.forward(batch, reduction='mean')

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Validation
        loss_val = evaluation(val_loader, model_best=model, epoch=e)
        nll_val.append(loss_val)  # save for plotting

        if e == 0:
            print('saved!')
            torch.save(model, name + '.model')
            best_nll = loss_val
        else:
            if loss_val < best_nll:
                print('saved!')
                torch.save(model, name + '.model')
                best_nll = loss_val
                patience = 0

                samples_generated(name, val_loader, shape=shape, extra_name="_epoch_" + str(e))
            else:
                patience = patience + 1

        if patience > max_patience:
            break

    nll_val = np.asarray(nll_val)

    return nll_val
```

Setup

NOTE: ***Please comment your code! Especially if you introduce any new variables (e.g., hyperparameters).***

In the following cells, we define `transforms` for the dataset. Next, we initialize the data, a directory for results and some fixed hyperparameters.

```
# DO NOT REMOVE
transforms_train = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                   torchvision.transforms.Lambda(lambda x: torch.bernoulli(x)),
                                                   torchvision.transforms.Lambda(lambda x: torch.flatten(x,0)),
                                                   ],
                                                  )

transforms_test = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                                   torchvision.transforms.Lambda(lambda x: torch.bernoulli(x)),
                                                   torchvision.transforms.Lambda(lambda x: torch.flatten(x,0)),
                                                   ],
                                                  )
```

Please do not modify the code in the next cell.

```
# DO NOT REMOVE
#-dataset
dataset = MNIST('/files/', train=True, download=True,
                      transform=transforms_train
                )

train_dataset, val_dataset = torch.utils.data.random_split(dataset, [50000, 10000], generator=torch.Generator().manual_seed(

test_dataset = MNIST('/files/', train=False, download=True,
                      transform=transforms_test
                     )
#-dataloaders
batch_size = 32

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

#-hyperparams (please do not modify them for the final report)
num_epochs = 1000 # max. number of epochs
max_patience = 20 # an early stopping is used, if training doesn't improve for longer than 20 epochs, it is stopped
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to /files/MNIST/raw/train-images-id
100%|██████████| 9912422/9912422 [00:01<00:00, 5671612.66it/s]
Extracting /files/MNIST/raw/train-images-idx3-ubyte.gz to /files/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to /files/MNIST/raw/train-labels-id
100%|██████████| 28881/28881 [00:00<00:00, 1103128.96it/s]
Extracting /files/MNIST/raw/train-labels-idx1-ubyte.gz to /files/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to /files/MNIST/raw/t10k-images-idx3
100%|██████████| 1648877/1648877 [00:00<00:00, 9451063.60it/s]
Extracting /files/MNIST/raw/t10k-images-idx3-ubyte.gz to /files/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to /files/MNIST/raw/t10k-labels-idx1
100%|██████████| 4542/4542 [00:00<00:00, 11140660.10it/s]Extracting /files/MNIST/raw/t10k-labels-idx1-ubyte.gz to /files
```

```
# DO NOT REMOVE
#-creating a dir for saving results
name = 'vae'  # NOTE: if you run multiple experiments, you would overwrite results. Please modify this part if necessary.
result_dir = images_dir + name + '/'
if not(os.path.exists(result_dir)):
  os.mkdir(result_dir)
```

In the next cell, please initialize the model. Please remember about commenting your code!

```
D = 784   # input dimension
L = 20    # number of latents
M = 512   # the number of neurons in scale (s) and translation (t) nets
num_components = 4**2
num_vals = 1


# YOUR CODE COMES HERE:
#
# your code goes here
#
# use all necessary code to initialize your VAE
likelihood_type = 'bernoulli'
encoder_net = nn.Sequential(nn.Linear(D, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, 2 * L))
```

```
decoder_net = nn.Sequential(nn.Linear(L, M), nn.LeakyReLU(),
                        nn.Linear(M, M), nn.LeakyReLU(),
                        nn.Linear(M, num_vals * D))


prior = MoGPrior(L=L, num_components=num_components)

model = VAE(encoder_net=encoder_net, decoder_net=decoder_net, num_vals=num_vals, prior=prior, likelihood_type=likelihood_type
model.to(device) # at the end, your model must be put on the available device
```

```
VAE(
  (encoder): Encoder(
    (encoder): Sequential(
      (0): Linear(in_features=784, out_features=512, bias=True)
      (1): LeakyReLU(negative_slope=0.01)
      (2): Linear(in_features=512, out_features=512, bias=True)
      (3): LeakyReLU(negative_slope=0.01)
      (4): Linear(in_features=512, out_features=40, bias=True)
    )
  )
  (decoder): Decoder(
    (decoder): Sequential(
      (0): Linear(in_features=20, out_features=512, bias=True)
      (1): LeakyReLU(negative_slope=0.01)
      (2): Linear(in_features=512, out_features=512, bias=True)
      (3): LeakyReLU(negative_slope=0.01)
      (4): Linear(in_features=512, out_features=784, bias=True)
    )
  )
  (prior): MoGPrior()
)
```

Please initialize the optimizer

```
# PLEASE DEFINE YOUR OPTIMIZER
#
# your code goes here
#
# please do not forget to define hyperparameters!
# please do it like this: optimizer = ...
lr = 1e-3 # learning rate
optimizer = torch.optim.Adamax([p for p in model.parameters() if p.requires_grad == True], lr=lr)
```

Training and final evaluation

In the following two cells, we run the training and the final evaluation.

```
# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val = training(name=result_dir + name, max_patience=max_patience,
                   num_epochs=num_epochs, model=model, optimizer=optimizer,
                   training_loader=train_loader, val_loader=val_loader,
                   shape=(28,28))
```

```
Epoch: 51, val nll=94.82152692871094
Epoch: 52, val nll=94.69859262695313
Epoch: 53, val nll=94.80602648925782
Epoch: 54, val nll=94.69135776367187
Epoch: 55, val nll=94.76925148925781
Epoch: 56, val nll=94.69232622070312
Epoch: 57, val nll=94.4224361694336
saved!
Epoch: 58, val nll=94.6125564453125
Epoch: 59, val nll=94.36722475585937
saved!
Epoch: 60, val nll=94.59014614257812
Epoch: 61, val nll=94.41292021484375
Epoch: 62, val nll=94.45935026855469
Epoch: 63, val nll=94.33792841796875
saved!
Epoch: 64, val nll=94.60275755615234
Epoch: 65, val nll=94.19123192138672
saved!
Epoch: 66, val nll=94.19753533935547
Epoch: 67, val nll=94.4374841796875
Epoch: 68, val nll=94.19954006347656
Epoch: 69, val nll=94.21720297851563
Epoch: 70, val nll=94.29729348144531
Epoch: 71, val nll=94.18861030273438
saved!
Epoch: 72, val nll=94.07766645507813
saved!
Epoch: 73, val nll=94.2507085571289
Epoch: 74, val nll=94.0691675415039
saved!
```

```
# DO NOT REMOVE OR MODIFY
# Final evaluation
test_loss = evaluation(name=result_dir + name, test_loader=test_loader)
f = open(result_dir + name + '_test_loss.txt', "w")
f.write(str(test_loss))
f.close()

samples_real(result_dir + name, test_loader)
samples_generated(result_dir + name, test_loader, extra_name='_FINAL')

plot_curve(result_dir + name, nll_val)
```

```
FINAL LOSS: nll=92.0731965576172
```

```
name = 'vae_1'  # NOTE: if you run multiple experiments, you would overwrite results. Please modify this part if necessary.
result_dir = images_dir + name + '/'
if not(os.path.exists(result_dir)):
  os.mkdir(result_dir)


lr = 1e-5 # learning rate
optimizer = torch.optim.Adamax([p for p in model.parameters() if p.requires_grad == True], lr=lr)


# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val = training(name=result_dir + name, max_patience=max_patience,
                   num_epochs=num_epochs, model=model, optimizer=optimizer,
                   training_loader=train_loader, val_loader=val_loader,
                   shape=(28,28))
```

```
Epoch: 0, val nll=91.97083553466797
saved!
Epoch: 1, val nll=91.8662556640625
saved!
Epoch: 2, val nll=91.8257950439453
saved!
Epoch: 3, val nll=91.72795203857422
```