HY673 - Spring 2024 Problem Set #3
Sifakis Emmanouil, csdp1288 (26/05/2024)

Problem 1. *Implementation of RealNVP on MNIST dataset*
In this exercise, you will extend the NICE model which was presented in Tutorial 7.

(a) *Implement RealNVP model. Essentially, you have to implement*

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} = \begin{bmatrix} z_A \\ z_B \odot e^{\alpha_\theta(z_A)} + m_\theta(z_A) \end{bmatrix}$$

where $\{x_A, x_B\}$, $\{z_A, z_B\}$ are partitions of the data and noise vectors, respectively, $\odot$ denotes element-wise product and both $m_\theta(\cdot)$ and $\alpha_\theta(\cdot)$ are neural networks with parameters θ. Note that $dim(x_A) = dim(z_A)$ and $dim(x_B) = dim(z_B)$.

(b) *Train RealNVP on MNIST dataset. Keep in mind that the above RealNVP equation applies to all transformation steps. Experiment with 5 and 10 coupling layers and compare the results.*

(c) *Perform linear and sinusoidal interpolations between two MNIST digits in the latent space (i.e., the base space of z). In particular, let $z^{(1)}, z^{(2)}$ be two MNIST digits in the latent space, you will generate and plot:*
$z_\lambda = (1-\lambda)z^{(1)} + \lambda z^{(2)}$ (linear) and
$\bar{z}_\lambda = (1 - \sin(\lambda\pi/2))z^{(1)} + \sin(\lambda\pi/2)z^{(2)}$ (sinusoidal)
interpolations for $\lambda = 0 : 0.1 : 1$.

Solution

(a,b) The implementation I've used is based on NICE model that is described on the Tutorial 7: github-link, of the course. For NICE there is the need to split the input $\mathbf{x}$ into two equal parts $\mathbf{x}_1$ and $\mathbf{x}_2$. This allows for a part of the input to remain unchanged during the transformation and is essential for maintaining invertibility. I will achieve this halving by utilizing masks like the ones in the original code.

NICE Transformation:
In additive coupling, the transformation is parametrized by a function $m_\theta(\cdot)$, which in the original code is a neural network with parameters θ.

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x}_1, \\ \mathbf{y}_2 &= \mathbf{x}_2 + m_\theta(\mathbf{x}_1), \end{aligned}$$

In practice, input splitting is achieved with masks, which simplifies the process. Instead of concatenating, I've just added the results. To ensure m_θ does not alter the masked values, just reapply the mask afterwards.

Invertibility: Given the output $\mathbf{y}$, the inverse is directly:

$$\mathbf{x}_1 = \mathbf{y}_1,$$
$$\mathbf{x}_2 = \mathbf{y}_2 - m_\theta(\mathbf{y}_1).$$

RealNVP Transformation:
To implement the RealNVP model, I had to define the forward and inverse transformations. In the RealNVP framework, the data vector is partitioned into two parts and transformed using neural networks for scaling and translation. The transformation is given by:

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} = \begin{bmatrix} z_A \\ z_B \odot e^{\alpha_\theta(z_A)} + m_\theta(z_A) \end{bmatrix}$$

and the inverse is given by:

$$\begin{bmatrix} z_A \\ z_B \end{bmatrix} = \begin{bmatrix} x_A \\ (x_B - m_\theta(x_A) \odot \exp(-\alpha_\theta(x_A)) \end{bmatrix}$$

In this equation, $\{x_A, x_B\}$ and $\{z_A, z_B\}$ are partitions of the data and noise vectors, respectively and both $m_\theta(\cdot)$ and $\alpha_\theta(\cdot)$ are neural networks parameterized by θ. The original paper does not specify which activation function was used, so I've used just use LeakyReLU.

In the RealNVP model each coupling layer applies the transformation using a mask to split the input vector, followed by neural networks to compute the scaling and translation. The scaling layer applies a global scaling transformation to ensure the overall scaling effect is accounted for in the log-likelihood computation. To train the RealNVP model on the MNIST dataset, the data is first dequantized and scaled using a pre-processing step. The model is trained using the Adam optimizer with a learning rate scheduler, and the negative log-likelihood (NLL) is minimized. The training process includes saving the best model based on the NLL.

The training experiments were conducted with two configurations: one with 5 coupling layers and another with 10 coupling layers. The primary goal was to compare the performance of these configurations in terms of their ability to learn the data distribution and generate realistic samples. For both configurations, the training involved initializing the model, setting up the optimizer and learning rate scheduler, and iterating over the training epochs. At each epoch, the model's parameters were updated to minimize the NLL. The best model was saved based on the lowest NLL observed during training.

Using 5 coupling layers, it is expected that the generated digits will not be clearly understandable. In comparison, with 10 coupling layers, the digits are expected to be slightly less noisy. However, in both cases, it is not accurate to say that the generated digits will be readable, although the model does seem to be learning. If the model was conditioned on the MNIST labels it will be helped to learn more information about the digits (following exercise). The learning progress is evidenced by the plot of the loss over the epochs, which reaches a lower value with 10 coupling layers (shown below). Althought those will be the expected results in practice for some reason that was not clarified 5 coupling layers resulted in more 'clear' generated digits (probably a mistake in the code).

This highlights the importance of sufficient transformation steps in capturing the complexity of the data distribution. The generated results for the training for 5 and 10 coupling layers are shown below.

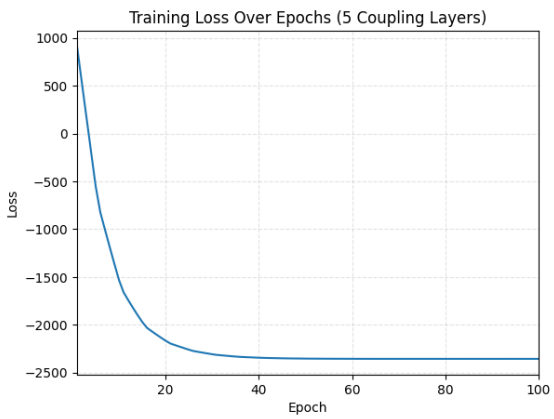

Figure 1: Loss over 100 epochs for 5 coupling layers.
(Best Loss: -2356.8834, epoch 100/100)

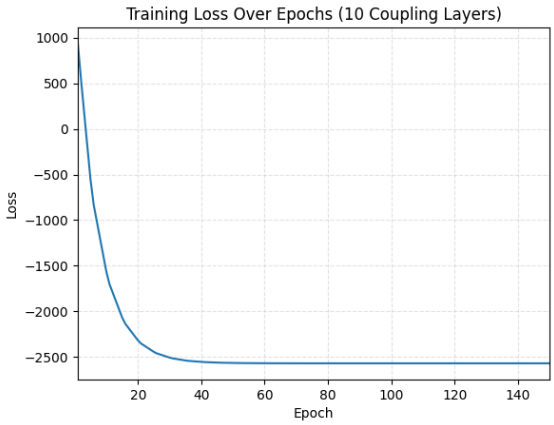

Figure 2: Loss over 150 epochs for 10 coupling layers.
(Best Loss: -2567.0491, epoch 109/150)

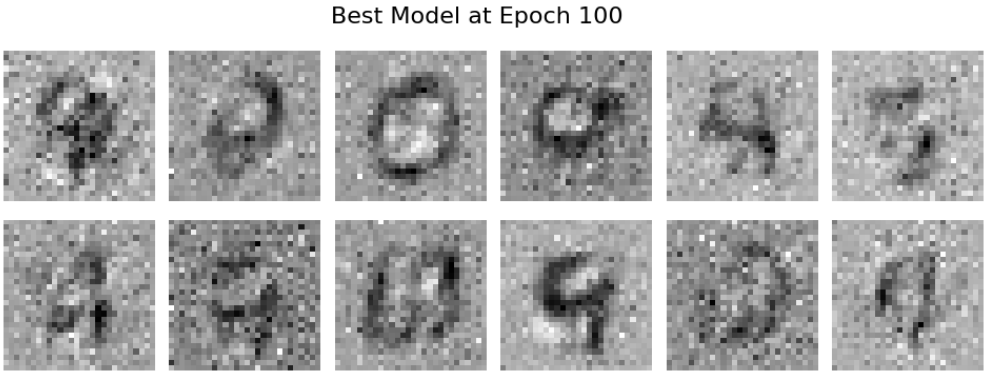

Figure 3: Generation for the best model using 5 coupling layers.

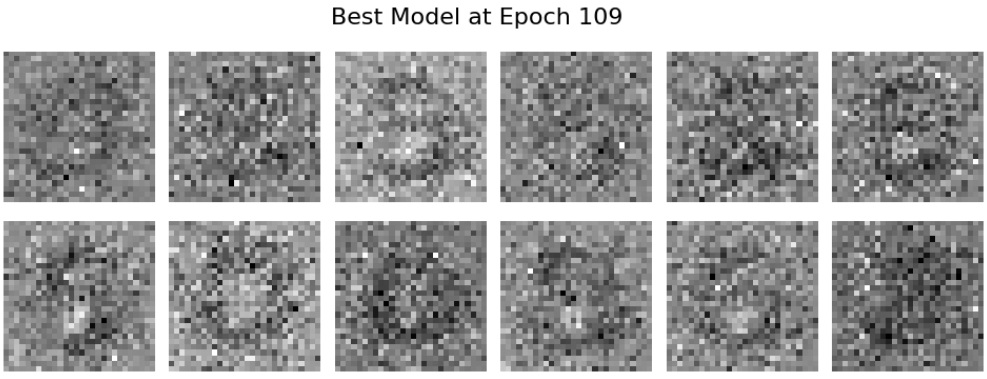

Figure 4: Generation for the best model using 10 coupling layers.

(c) Perform linear and sinusoidal interpolations between two MNIST digits in the latent space.

To perform linear and sinusoidal interpolations between two MNIST digits in the latent space

using the RealNVP model, there is the need to firstly define the interpolation functions.

For linear interpolation:

$$z_\lambda = (1 - \lambda)z^{(1)} + \lambda z^{(2)}$$

For sinusoidal interpolation:

$$\bar{z}_\lambda = (1 - \sin(\lambda\pi/2))z^{(1)} + \sin(\lambda\pi/2)z^{(2)}$$

where λ ranges from 0 to 1 in steps of 0.1.

In my code the 'linear_interpolation' and 'sinusoidal_interpolation' functions compute the interpolated latent vectors for given λ values. The 'interpolate_and_plot' function handles the interpolation process, encoding two images into the latent space, performing the interpolation, and then decoding the interpolated latent vectors back to the image space. This function plots the resulting images, showing the gradual transformation between the two original images.

With this code one is able to visualize the smooth transition between two MNIST digits in the latent space using both linear and sinusoidal interpolations. The results are shown below. I used a threshold of 0.3 to limit low-value pixels in the plot, making the results clearer.

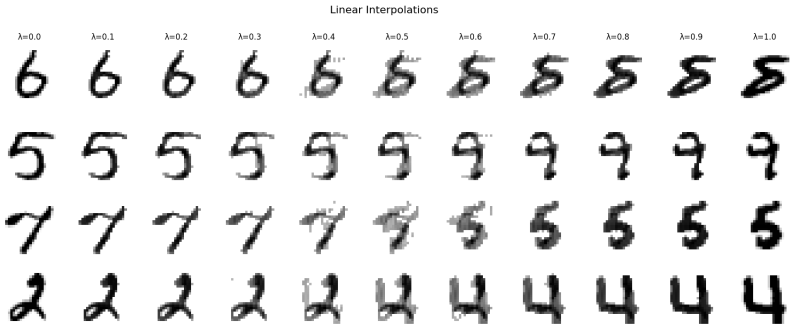

Figure 5: Linear nterpolation between two digits for 4 examples.

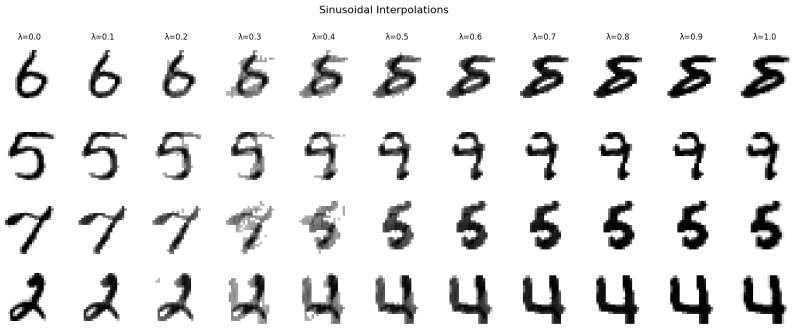

Figure 6: Sinusoidal nterpolation between two digits for 4 examples.

Problem 2. *Conditional RealNVP on MNIST*

(a) *Add digit information to the RealNVP model and learn the conditional distributions. In particular, use one-hot encoding for the digit labels and concatenate them in the input of the neural nets. The equation of the conditional RealNVP model is given by*

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} = \begin{bmatrix} z_A \\ z_B \odot e^{\alpha_\theta(z_A, y)} + m_\theta(z_A, y) \end{bmatrix}$$

where $y \in \mathbb{R}^{10}$ corresponds to the one-hot encoding vector of the MNIST digit labels.

(b) *Train conditional RealNVP on MNIST dataset. Compare the obtained results with the generated digits from 1(b). Can we say something about the number of required transformations when conditional generation is utilized?*

Solution

I've implemented a Conditional RealNVP model to generate MNIST digits by incorporating digit information through one-hot encoding of the labels. Everything in my code remains the same in relation to the previous question (EX1_CS673.ipynb), with the primary difference being that the network is now conditioned on the digit labels of the MNIST dataset. This conditioning allows the model to incorporate additional information about the digit classes, potentially improving its ability to generate and interpolate between specific digit images. The 'RealNVPCouplingLayer' class conditions the transformations on the MNIST labels by concatenating the one-hot encoded labels with the input data. During both the 'forward' and 'inverse' passes, the input is split using binary masks, concatenated with the labels, and passed through scale and translation subnetworks to apply the transformations. The 'ScalingLayer' class globally scales the data, computing the log-determinant of the Jacobian for the scaling transformation (same as before).

The incorporation of conditional information in the RealNVP model results in more readable and clearer digit generation compared to the non-conditional version due to several key factors. By conditioning on one-hot encoded labels, the model receives explicit information about the digit it needs to generate. This guidance helps the coupling layers learn transformations specific to each digit class, reducing ambiguity and improving image quality. Conditional labels provide additional features, enabling the model to focus on relevant characteristics, leading to more accurate and distinct digits. Conditional generation mitigates mode collapse by providing clear directives for each digit class, ensuring diverse image outputs. It also stabilizes training by reducing distribution complexity, allowing the model to learn simpler conditional distributions, leading to better convergence and more reliable training. More layers increase the model's capacity for complex transformations, resulting in clearer and more refined images. Each additional layer enhances the model's ability to approximate the target distribution, capturing finer details and reducing noise. Below the results of conditional generation are presented for 5 and 10 coupling layers along with their training loss plots over epochs.

Regarding the Conditional RealNVP model applied to the MNIST dataset, conditioning on one-hot encoded labels simplifies the learning process by providing direct information

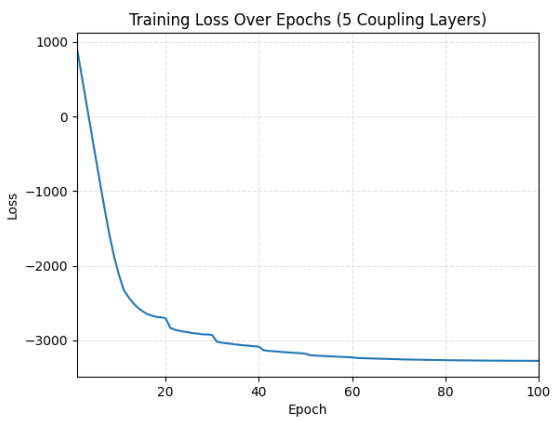

Figure 7: Loss over 100 epochs for 5 coupling layers (conditional).
(Best Loss: -3275.8027, epoch 100/100)

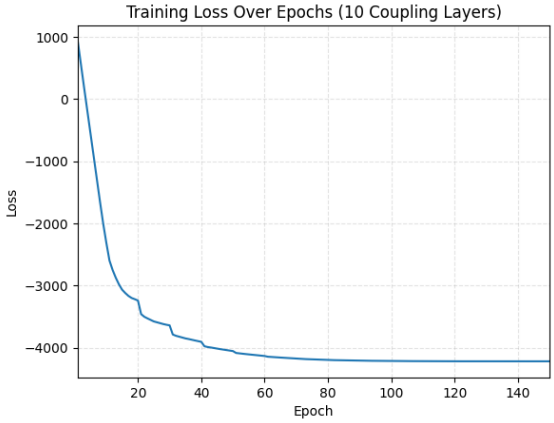

Figure 8: Loss over 150 epochs for 10 coupling layers (conditional).
(Best Loss: -4219.8333, epoch 148/150)

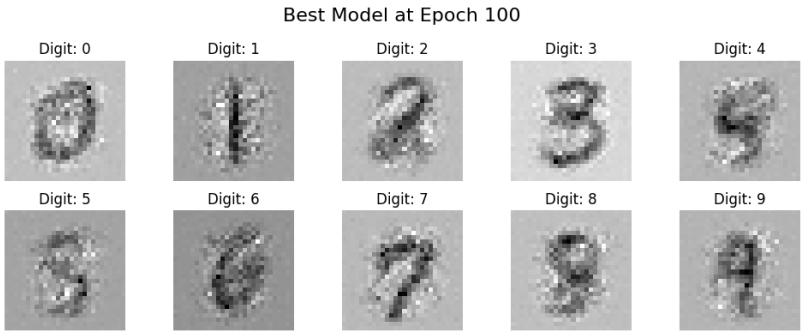

Figure 9: Sinusoidal nterpolation between two digits for 4 examples.

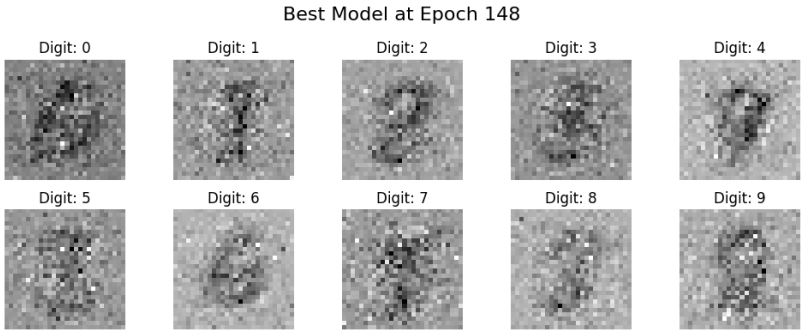

Figure 10: Sinusoidal nterpolation between two digits for 4 examples.

about which digit to generate. This reduces the complexity of transformations needed for each digit class, allowing for clearer and more accurate outputs with potentially fewer layers. Guided by conditional labels, each transformation is more targeted and effective, possibly requiring fewer transformations than an unconditional model. Although this is true, it is shown not by fewer iterations but by the same number of iterations with much more readable

generated digit results. As in the previous question with the original RealNVP, there might be a mistake in my code regarding the coupling layers. Although it is expected to have clearer digit generation with 10 coupling layers compared to 5, my results do not reflect this. In my results, the clearest and most readable output comes from the model with 5 coupling layers.

Problem 3. *Implementation of Conditional VAE*
In this exercise, you will extend the VAE model presented in Tutorial 6. Let x be an MNIST digit image and y be the corresponding one-hot encoded label. We can define $p(x|y)$ to be the conditional data distribution (a.k.a. the conditional marginal likelihood), $p(x, z|y)$ to be the conditional joint generative distribution, $p(z|y)$ to be the conditional prior distribution, $p(x|z, y)$ to be the conditional likelihood of the decoder (a.k.a, the conditional generative distribution) and $q(z|x, y)$ to be the conditional approximate posterior distribution.

(a) *Write down the conditional ELBO (see Lecture 10 for the unconditional formulation) and the ELBO variation with the Kullback-Leibler divergence as one of the two terms of ELBO (see Lecture 11 for the unconditional formulation).*

(b) *Using a (stochastic) decoder with input (z, y) and a (stochastic) encoder with input (x, y), implement the conditional VAE for MNIST digit generation.*

(c) *Using the trained generative model from (b), write a program that takes as input a number and returns an image with the number where each digit has been conditionally generated from the pre-trained model. Using the trained VAE obtained in (b), write a function that generates images based on an array of input labels ranging between 0 and 9 (e.g., if the input is [0, 2] the function should return two generated images conditioned on 0 and 2, respectively). Include examples of images produced by this function, and indicate the corresponding input labels.*

Solution

(a) Conditional ELBO and the ELBO variation with the Kullback-Leibler divergence as one of the two terms of ELBO.

As it is already defined, the following elements:
$\boldsymbol{p(x|y)}$: conditional data distribution, $\boldsymbol{p(x, z|y)}$: conditional joint generative distribution, $\boldsymbol{p(z|y)}$: conditional prior distribution, $\boldsymbol{p(x|z, y)}$: conditional likelihood of the decoder and $\boldsymbol{q(z|x, y)}$: conditional approximate posterior distribution. Other definitions regarding the data are: $\boldsymbol{x}$: Observed data, specifically a MNIST digit image. $\boldsymbol{y}$: Corresponding one-hot encoded label for x, representing the digit class. $\boldsymbol{z}$: Latent variable representing the hidden features of the data x.
The computation of the Evidence Lower Bound (ELBO) for the conditional log-likelihood $\log p(x|y)$ is shown below:

$$\log p(x|y) = \log \left(\int p(x, z|y) \frac{q(z|x, y)}{q(z|x, y)} \, dz \right) = \log \left(\int q(z|x, y) \frac{p(x, z|y)}{q(z|x, y)} \, dz \right)$$

Use of the Jensen's inequality:

$$\log p(x|y) \geq \int q(z|x,y) \log \left(\frac{p(x,z|y)}{q(z|x,y)} \right) dz = \int q(z|x,y) \log \left(\frac{p(x|z,y)p(z|y)}{q(z|x,y)} \right) dz$$

$$= \int q(z|x,y) \left(\log p(x|z,y) + \log p(z|y) - \log q(z|x,y) \right) dz$$

$$= \int q(z|x,y) \log p(x|z,y)\, dz + \int q(z|x,y) \log p(z|y)\, dz - \int q(z|x,y) \log q(z|x,y)\, dz$$

The final conditional ELBO loss function for training the CVAE is:

$$\mathcal{L}_{\text{cond-ELBO}} = \mathbb{E}_{q(z|x,y)} \left[\log p(x|z,y) \right] - \text{KL} \left(q(z|x,y) \| p(z|y) \right)$$

where:
- $\mathbb{E}_{q(z|x,y)} \left[\log p(x|z,y) \right]$ represents the expected log-likelihood of the data given the latent variables and the conditional label.
- $\text{KL} \left(q(z|x,y) \| p(z|y) \right)$ denotes the Kullback-Leibler divergence between the approximate posterior $q(z|x,y)$ and the prior $p(z|y)$.

(b) Using a (stochastic) decoder with input (z, y) and a (stochastic) encoder with input (x, y), implement the conditional VAE for MNIST digit generation.

To implement the conditional VAE for MNIST digit generation, I've used a stochastic encoder with inputs (x, y) and a stochastic decoder with inputs (z, y). (This complete implimentation is based on the Tutorial 8: github-link of the course) The encoder network is designed to take an MNIST image x and a one-hot encoded label y, concatenate them, and map the combined input to a latent space. This process produces two outputs: the mean (μ) and the log-variance $(\log \sigma^2)$ of the latent distribution. The decoder network, on the other hand, takes a latent vector z and a one-hot encoded label y, concatenates them, and reconstructs the input image x. To sample from the latent space, the original code used the reparameterization trick, which allows to compute gradients during backpropagation. The latent vector z is computed as $z = \mu + \sigma \cdot \epsilon$, where ϵ is a standard normal variable. The main Conditional VAE class encapsulates the encoder and decoder networks. The forward pass of this cVAE involves reshaping the input image, passing it through the encoder to obtain μ and $\log \sigma^2$, reparameterizing to obtain z, and then passing z and y through the decoder to reconstruct the image. Regarding the loss function used for the cVAE, the original code combines the reconstruction loss and the KL divergence. The reconstruction loss is computed using binary cross-entropy between the original and reconstructed images. The KL divergence term ensures that the learned latent distribution $q(z|x, y)$ remains close to the prior distribution $p(z|y)$, which is typically a standard normal distribution.

The training loop involves iterating over the dataset for a specified number of epochs. For each batch, the inputs are passed through the model, and the loss is computed. The gradients are then backpropagated, and the model parameters are updated using an optimizer (Adam). The training loop also includes periodic evaluation and saving of the model based on the validation performance. Above are the generated digits for every class/label from

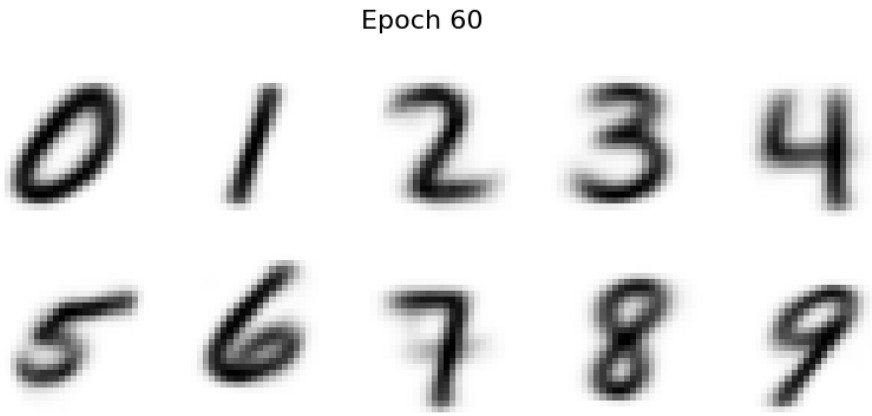

Figure 11: Plots of generated images for all the MNIST Labels.

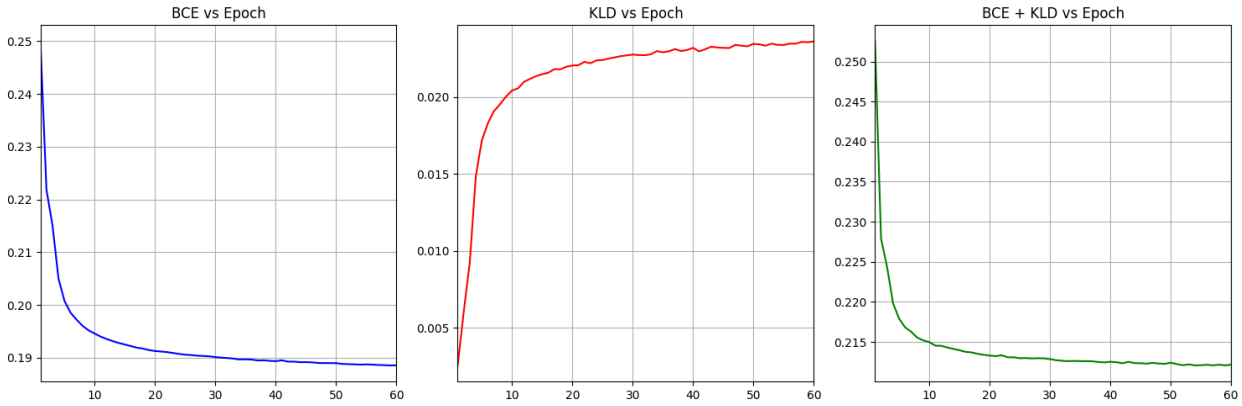

Figure 12: Plots the BCE and KLD loss and their sum over the training epochs.

the MNIST dataset, using the trained cVAE for 60 epochs and batch_size=128 and also the resulting Loss plots over epochs.

After training, the model can be used to generate new images. The sampling function generates random latent vectors and passes them through the decoder along with the specified labels to produce images. These generated images can be visualized to assess the quality of the conditional generation.

(c) Using the trained VAE to generate images conditioned on input labels, returning images for each digit specified in an input array, and to provide examples of the generated images with their corresponding labels.

Using the last trained generative model from part (b), one can input a number and return an image where each digit is conditionally generated by the pre-trained model. This involves two main functions: one for generating images based on input labels and another for plotting these images. The ‘generate_images‘ function takes an array of labels, converts them to one-hot encoded tensors, and uses the trained cVAE model to generate the corresponding images. The labels are transformed into one-hot encoded vectors, converted to floating-point

tensors, and passed to the model's sampling method. The 'plot_generated_images' function takes the generated images and their labels and plots them in a grid, dynamically adjusting the grid size based on the number of labels.

To demonstrate this implementation the previously trained model is loaded (best_vae.pth file that is provided in the MEGA link at the end of this report). Then, I generated images for different sets of labels. For instance, when generating an image for a single label, such as '[8]', the function generates an image conditioned on the digit 8 as shown below:

Figure 13: Generating digit for one label.

For multiple labels, such as '[2, 4, 3]', the function generates three images conditioned on the digits 2, 4, and 3, respectively, also shown below:

Figure 14: Generating digits for a sequence of 3 labels.

Lastly, for a longer sequence of labels, such as '[2, 8, 1, 0, 3, 9, 3, 5, 9, 2]', the function generates ten images, each conditioned on one of these digits. This sequence, which is the Postgraduate studies secretariat Number is shown below:

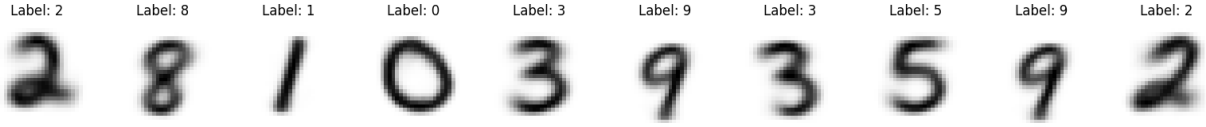

Figure 15: Generating digits for a sequence of 10 labels.

*** **Each question's code is in a separate notebook file (**`EX1_CS673.ipynb`**,** `EX2_CS673.ipynb`**,** `EX3_CS673.ipynb`**) sent in the zip file together with this report. The saved models for the two RealNVP, two conditional RealNVP, and the conditional VAE are stored in the following Mega link:** Saved-Models-Link. Whenever there is a 'c' present in the name of a saved model, it represents a conditional network. ***