Homework 5 — GENERATIVE MODELS

1 Scope

In this assignment, you will be learning and implementing a few variations of Autoencoders namely: **Autoencoders (AEs)** and **Variational Autoencoders (VAEs)**.

2 Instructions

- Deadline for this assignment is **March 29, 2024** via Gradescope
- The submission portal will remain open till end of the semester. Do keep in mind the late submission policy (Check slides from first lecture)
- Your submission has two parts: (i) a PDF report that details all the requested deliverables and (ii) the code for the assignment. You will upload the PDF report in gradescope, marking which parts of the report corresponds to which deliverable. The code will be uploaded separately (instructions TBA), and we will run "software similarity" software on it to detect violations of the integrity policy.
- You can discuss this HW with others, but you are **not** allowed to share, borrow, copy or look at each other's codes.
- All code that you submit must be yours (except for the starter code that we provide). Specifically, you are **not** allowed to use any software packages or code from the internet or other resources, except for `numpy`, `torch`, `torchvision`, `scipy`, `PIL`, `matplotlib`, `torchsummary` and other built-in packages such as `math`.
- We recommend that you look through all of the problems before attempting the first problem. However, we do recommend you complete the problems in order.

3 Introduction

3.1 Autoencoders (AE)

Autoencoders are neural networks that can learn a compressed/low-dimensional representation of input data in a latent space. The compressed representation is then used to generate new data that is similar to the original data. In the case of image data, the autoencoder can learn to generate new images that are similar to the original images.

Since an autoencoder just reconstructs the input, it is not regarded as a generative model. By feeding different vectors, the decoder might be used as a generative model. But, the vanilla Autoencoder, however, mostly learns a latent space with different clusters. The decoder will output very abstract results, primarily junk, because it has never learned to reconstruct gaps that are between the clusters.

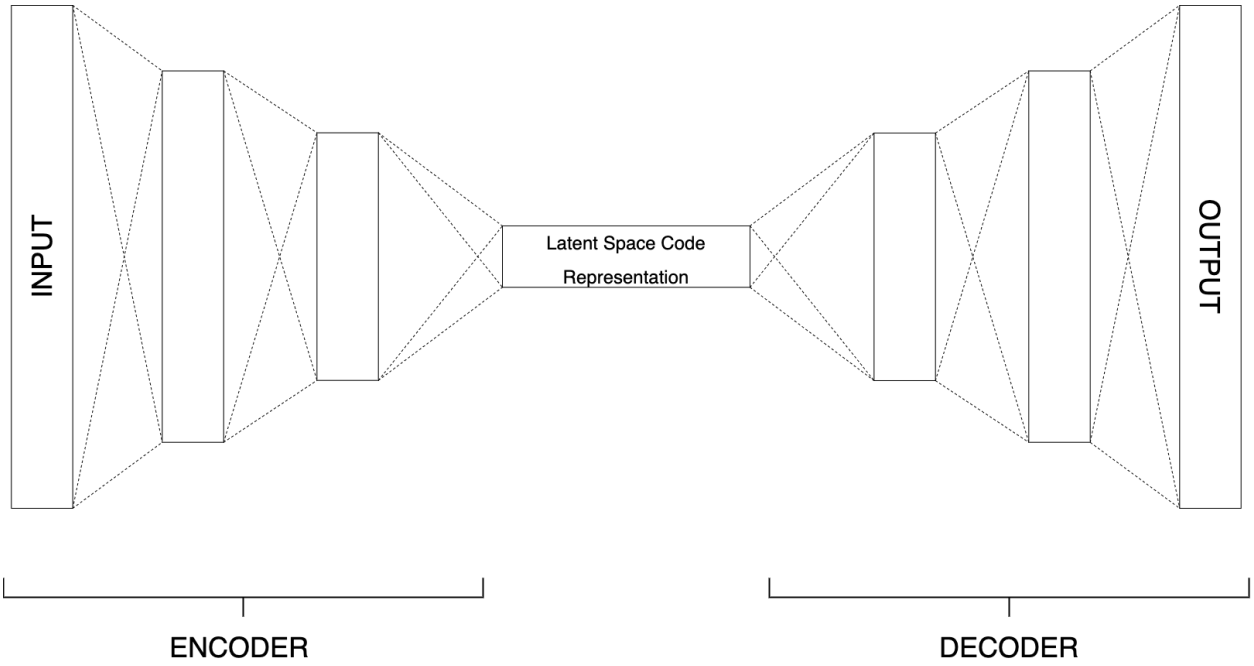


Figure 1: Example AE architecture.

3.1.1 Encoder

The encoder is a neural network that learns to transform the input data into a smaller-dimensional vector or matrix for storage. One or more neural network layers, such as convolutional layers or fully connected layers, may make up the encoder. Each layer decreases the dimensionality of the incoming data and learns a new degree of abstraction. The encoder's objective is to extract the most crucial aspects of the input data and omit unnecessary details.

3.1.2 Latent Space Code

The input data that the encoder learned is represented in a compressed form by the latent space code. It is often a lower-dimensional vector or matrix and contains the key components of the input data. For tasks like classification or clustering, the latent space code is frequently used as a feature representation. The quality of the latent space code is a crucial factor in the performance of an autoencoder.

3.1.3 Decoder

The decoder is another neural network that learns to reconstruct the original input from the latent space code. It takes the encoded representation produced by the encoder as an input and applies a series of transformations to reconstruct the original input. The decoder's goal is to produce an output that is as close as possible to the input.

3.2 Variational Autoencoders (VAE)

Variational Autoencoder (VAE) is a generative model that is based on the autoencoder architecture. The key difference is that a VAE learns a probability distribution over the latent space code, allowing it to generate new samples from the learned distribution.

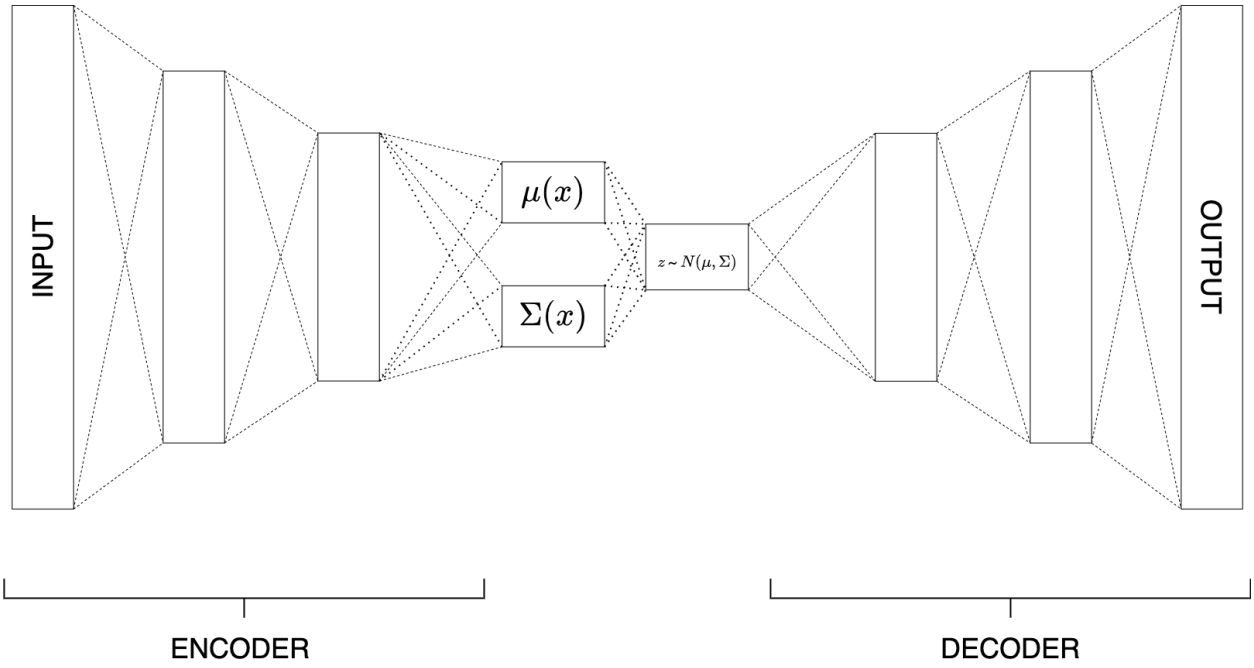


Figure 2: Example VAE architecture.

In a VAE, the encoder learns to encode the input data into a mean and a variance vector in the latent space code. The mean vector represents the most likely values for the latent variables, while the variance vector represents the uncertainty or the spread of the latent variables. The encoder produces a sample from the learned distribution by sampling from the latent variables' mean and variance vectors.

The decoder takes this sample from the latent space code and reconstructs the original input. The reconstruction loss, which measures the difference between the input and the output, is used to train the model.

The VAE architecture also includes a regularization term that encourages the learned distribution to be close to a standard normal distribution, also known as the KL divergence term. This regularization term ensures that the VAE learns a meaningful and smooth latent space that can be easily sampled from and interpolated between.

VAEs are capable of generating new data samples from the learned distribution by sampling from the latent space. This allows them to generate new and diverse data samples, making them useful for tasks such as image synthesis, text generation, and anomaly detection.

3.3 Vector Quantized VAE (VQ-VAE)

VQ-VAE is a variant of VAE that introduces a discrete latent space representation. Unlike traditional VAEs where the latent space is continuous, VQ-VAE employs a discrete latent space where encoder outputs are quantized to a set of codebook vectors. This quantization step, often using vector quantization, enables VQ-VAE to learn a more structured and interpretable latent space. The encoder encodes input data into continuous latent vectors, which are then mapped to the nearest vectors in the discrete codebook. The decoder then reconstructs the input data from the quantized latent vectors. VQ-VAE has shown promising results in tasks such as image generation,

representation learning, and speech synthesis, offering improved disentanglement and robustness to variations in input data.

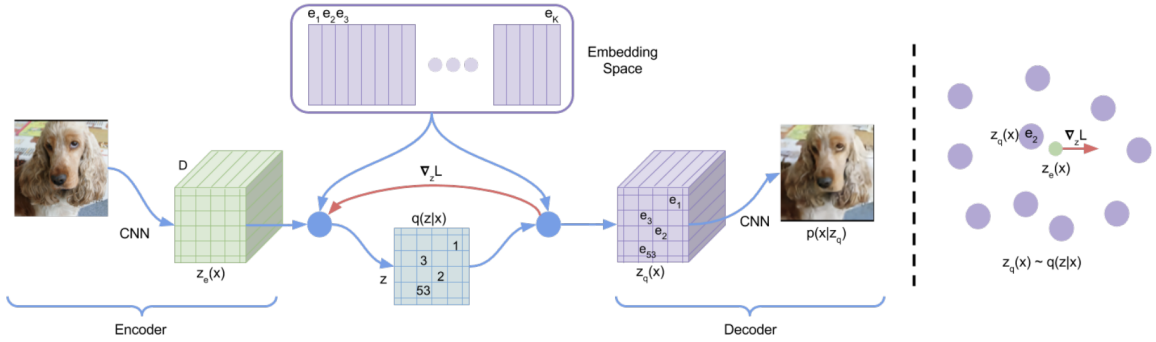


Figure 3: VQ-VAE architecture.

4 Problem Set

4.1 Deliverable 1: Vanilla Autoencoders (AEs)

In this deliverable you will learn how to implement your own Autoencoder that can generate new images of handwritten digits that are similar to the images in the MNIST dataset.

Your implementation will consist solely of fully connected layers. In this section, you'll define the Encoder and Decoder models in the `Autoencoder` class of `Autoencoder_Starter_Notebook.ipynb` and implement the forward pass and loss function to train your first AE.

Encoder
The encoder will take the vectorized images as input and pass them through Linear + Activation layers. Specifically:

- This network will take a batch of images as input `(N, 1, H, W)` and produces a batch of latent vectors as output `(N, Z)`. You can use come up with multiple hidden layers that map the input to hidden features and then to the latent space. For example, we use three layers of the same `hidden_dim (H_d)`.
- Try experimenting with the Z and understanding how the results change with the size of the bottleneck. For submission, use latent_size as 2.
- Use `nn.LeakyReLu` for the activation function, and do not use anactivation function in the final layer.
- Use `nn.Sequential` to define the encoder layers.

Decoder
The decoder takes the encoded representation produced by the encoder as input, pass them through Linear + Activation Layers.

- The decoder will take the latent vectors as input `(N,Z)` and output an image that has the same dimensions of the original image. `(N,1,H,W)`

- Use `nn.LeakyReLu` for the activation function in hidden layers, and use `nn.Sigmoid` for the activation function in the final layer to ensure that the output values are between 0 and 1.
- Use `nn.Sequential` to define the decoder layers.

Training
Finish the remaining TODOs in the ipynb file, including:

- Define the `hidden_dim (H_d)`, `latent_dim (Z)`, `batch_size (N)`, `n_epochs` in the initialization.
- Apply the proper `transform` on the MNIST dataset to convert data into torch.FloatTensor.
- Use the Mean Squared Error (MSE) loss function as `criterion` to measure the difference between the original and reconstructed images.
- Use Adam as the `optimizer`.

Deliverables for the PDF report: Your architecture should not exceed $1.5M$ trainable parameters and achieve a loss of less than 0.04 within 15 epochs. You will need to submit:

- The training loss.
- Snip of your model summary provided by the `torchsummary.summary`
- Latent space representation generated using the `plot_latent_space` function as in Fig. 4 (a).
- Images reconstructed from sampled vectors in the latent space using `plot_latent_space_images` function. As in Fig. 4 (b), you should be able to see some correspondence to the latent space represenation.

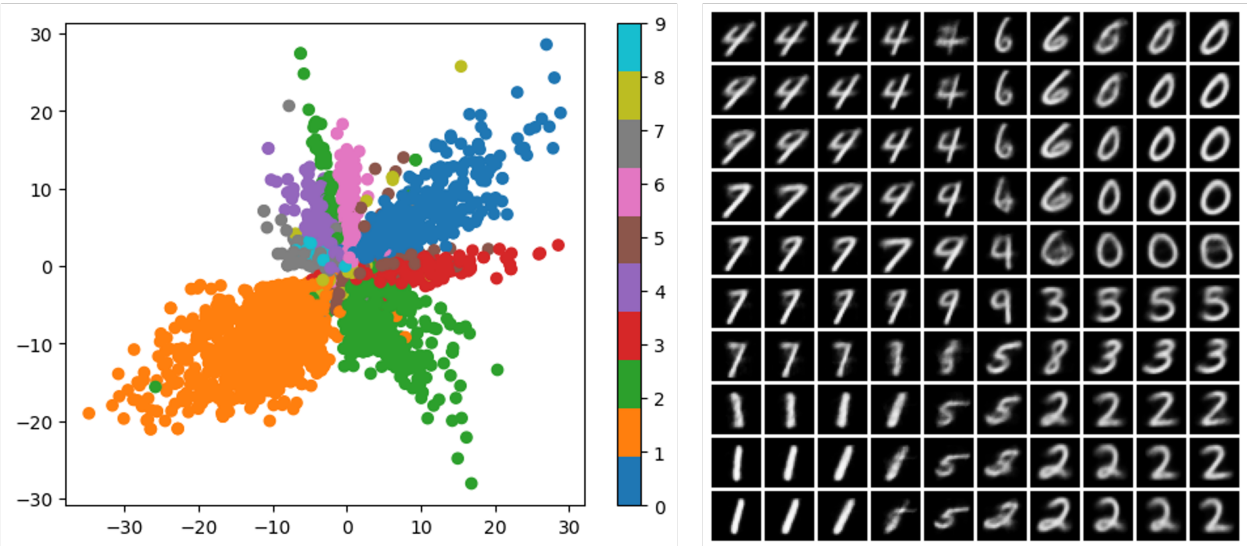


Figure 4: Example latent space and corresponding reconstructed image. Design inspired by this post

4.2 Deliverable 2: Denoising Convolutional Autoencoder

The objective of the assignment is to familiarize you with the concepts of autoencoders, convolutional layers, and how to combine them to create a denoising autoencoder that is capable of denoising any image.

Unlike deliverable 1, your implementation for this part will consist solely of Convolutional layers. In this section, you'll define the Encoder and Decoder models in the `DenoiseAE` class of `Denoising_Autoencoder_Starter_Notebook.ipynb` and implement the forward pass, and loss function to train your AE.

Encoder
The encoder should consist of three convolutional layers with batch normalization and `LeakyReLu` activation functions, followed by max pooling layers to reduce the spatial resolution of the input. You can decide your own convolutional kernel size.

Decoder
The decoder should consist of three upsampling / transposed convolutional layers (Hint: use `nn.ConvTranspose2d`) with batch normalization and `LeakyReLu` activation functions, followed by a final sigmoid activation function to ensure that the output values are between 0 and 1. (*You might not be able to exactly reverse the opertation. Instead play around with kernel size to get the image back to original shape*)

Training
Train the denoising autoencoder on your dataset of images. The autoencoder should learn to reconstruct the original image from the noisy input.

- Define the `batch_size (N), n_epochs` in the initialization.
- Use the Mean Squared Error (MSE) loss function as `criterion` to measure the difference between the original and reconstructed images.
- Use Adam as the `optimizer`.
- During training, add noise to the input images by adding Gaussian noise with a standard deviation of 0.5 to the input.
- After adding the Gaussian noise, it is important to clip the resulting tensor in the range [0, 1], this ensures all the values lie between the given range and do not generate weird images due to the noise.

In the architecture mentioned above, the bottleneck layer is the layer that produces the compressed representation of the input image. This layer is usually located in the middle of the encoder network, and its size determines the level of compression of the input image. In this assignment, you should set the size of the bottleneck layer to be smaller than the size of the input image, but large enough to capture the essential features of the input.

However, you should be careful not to set the bottleneck layer size too small, as this can result in information loss and poor reconstruction quality. Conversely, setting the bottleneck layer size too large can result in a less compact representation and slower training. Therefore, it's important to experiment with different bottleneck layer sizes and find the optimal size that balances the trade-off between compression and reconstruction quality.

Deliverables for the PDF report: Your architecture should not exceed 15k trainable parameters and achieve a loss of less than 0.025 within 20 epochs. You will need to submit:

- The training loss.
- Snip of your model summary provided by the `torchsummary.summary`
- As in the results section in the notebook, use the trained denoising autoencoder to remove noise from a sample image from the testing set, and plot the noisy image and the denoised image together. An example image is provided in Fig. 5.

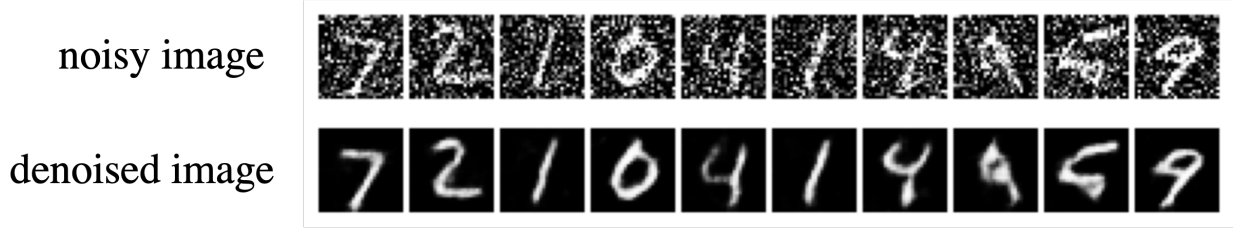


Figure 5: Example of denoised images

4.3 Deliverable 3: Fully Connected Variational Autoencoder (FC-VAE)

From the previous deliverables, you would have got a good understanding of what an encoder is capable of. But, we have no knowledge of the coding produced by our network using a general autoencoder. Although, we could take a look at and compare different encoded objects, but it's doubtful that we would be able to fully understand what is happening. Because we don't know what the inputs should look like, we won't be able to utilize our decoder to generate new images.

A variational autoencoder specifies the desired distribution of the latent vectors to the network, in contrast to a general autoencoder, which makes assumptions about the distribution of latent vectors. The network is typically forced to generate latent vectors that follow the unit normal distribution. To generate new data, we can sample values from this distribution, feed them to the decoder, and obtain new objects that resemble the ones used to train the network.

In this deliverable, you will be implementing a fully-connected VAE network `myVAE`, which comprises of `Encoder and Decoder models`, `reparametrization trick`, and the `Loss function` in the `VAE_Starter_Notebook.ipynb`.

Encoder
The encoder will take the images as input and pass them through Linear + Activation layers. Specifically:

- This network will take a batch of images as input `(N, 1, H, W)` and produces a batch of hidden feature vectors as output `(N, H_d)`, where `H_d` is the hidden dimension size that you can come up with.
- The `mu_layer` and `logvar_layer` give the mean and log-variance of the latent variables and are represented by separate linear layers that take the output of the encoder as input and produce vectors of shape `(N, Z)`
- Use `nn.Sequential` to define the encoder layers.

Decoder
This network takes a batch of latent vectors `(N, Z)` as inputs and produces predicted images as outputs `(N, 1, H, W)`. Remember to use a sigmoid activation at the end to ensure that the output values are in the range of 0 and 1. Also use `nn.Sequential` to define the decoder layers.

Weiyu: the following paragraph seems copied from this link. Need to be edited

Loss function
To be specific, the overall loss function of VAE is

$$-\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] + D_{KL}(q_\phi(z|x)||p(z)),$$

which composed of two loss terms: **reconstruction loss term** (left) and **KL divergence term** (right).

The reconstruction loss is an expected negative log-likelihood of a sample data x, parameterized by latent variable z given decoder parameters θ. It can be computed by simply using the binary cross entropy loss between the original input pixels and the output pixels of our decoder with `nn.functional.binary_cross_entropy`.

On the other hand, the KL divergence aims to ensure the latent variable z, when estimated by $q_\phi(z|x)$ given encoder parameter ϕ and sample data x, can match the prior gaussian distribution $p(z)$.

To be precise, we first assume that $p(z)$ is a unit-norm Gaussian $p(z) = \mathcal{N}(0, I)$, and each dimension of $q_\phi(z|x)$ also follows a Gaussian distribution $\mathcal{N}(\mu_j, \sigma_j^2)$, then as pointed out in Appendix B in the original VAE paper, their KL divergence can be written as:

$$D_{KL}(q_\phi(z|x)||p(z))) = -\frac{1}{2}\sum_{j=1}^{Z}(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2),$$

where j is the dimension index in vector z.

Implement the overall loss in the `loss_function` in `myVAE`. While working with batches, you should average the loss across samples in the batch. Also, make sure your Z is not too low. An ideal range would be around 12-18.

Reparameterization trick

To learn μ and σ, we need to make $\mathcal{N}(\mu_j, \sigma_j^2)$ differentiable w.r.t ϕ. Thus, as in Fig. 6, we need can apply a reparameterization trick: instead of directly sampling $z_i \sim \mathcal{N}(\mu_j, \sigma_j^2)$, we can first sample an auxiliary variable $\epsilon \sim \mathcal{N}(0, 1)$ and rewrite $z_j = \mu_j + \sigma_j * \epsilon$. Implement this trick in the `reparameterize` function in `myVAE`.

Training
After implementing `myVAE` class, train the model by:

- Define the `hidden_dim, latent_dim, batch_size, n_epochs` in the initialization. You do not need to deal with the `conditional_vec_dim` in this deliverable.

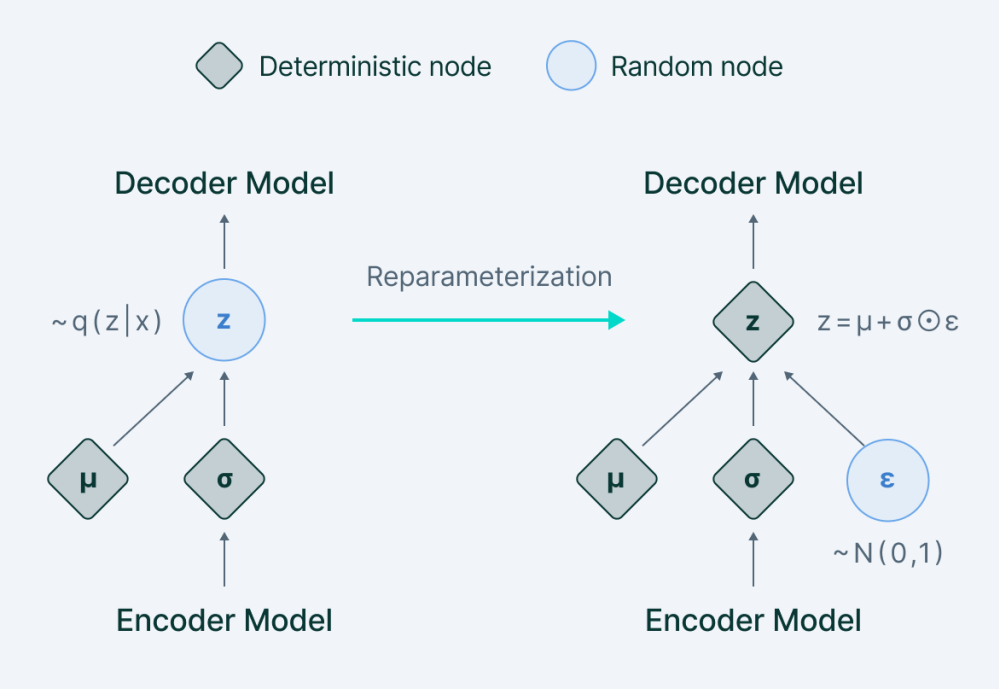


Figure 6: Reparameterization trick Source

- Use Adam as the `optimizer`.

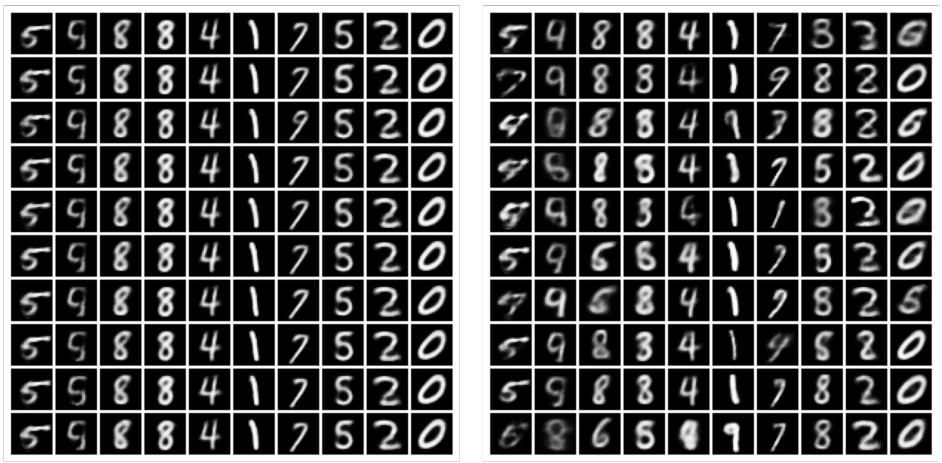
Deliverables for the PDF report: Your architecture should not cross 1.5M parameters and achieve a training loss of less than 120 within 10 epochs. To be specific, show the following:

- The training loss.
- Snip of your model summary provided by the `torchsummary.summary`
- The results generated under the results section in the notebook. To be specific, as in Fig. 7 (a), if we forward the same image multiple times, we would observe slightly different images because of the sampling. We could also enforce a larger variance σ^2 to increase the variation of the image as in Fig. 7 (b).

4.4 Deliverable 4: Conditional Variational Autoencoder (CVAE)

You might have noticed in the previous task that we don't have control over what kind of data the decoder generates. You implemented and trained a VAE with the MNIST data set and tried to generate images by feeding $\epsilon \sim \mathcal{N}(0, 1)$ into the decoder, the output can change appearance, but you still have no control over what digit the decoder will produce. In this case, you can not directly tell the VAE to produce an image of digit '2'.

Hence to get control over what digit the decoder should generate, you will now extend your implementation of VAE to Conditional VAE with a small modification. Instead of directly learn $q_\phi(z|x)$ and $p_\theta(x|z)$, now you learn $q_\phi(z|x, c)$ and $p_\theta(x|z, c)$ conditioned on the image label c, and make you allow to control the output image by changing c.



(a) (b)

Figure 7: Example of images with variations. (a) Sampled image with original variance. (b) Sampled image with larger (20x) variance.

Encoder
Continue on using `myVAE`, but this time allow it to optionally take in not only the flattened input image but also an extra one-hot label vector. Specifically:

- inputs a batch of input shape `(N, H*W + C)`, where `C` is the conditional vector size (*number of classes in this case*) into a batch of hidden features of shape `(N, H_d)`.
- the `mean_layer` and `logvar_layer` will remain the same.

Decoder
Continue on using `myVAE`, but modify it to decode from the latent space + one-hot vector. Specifically:

- Inputs a batch of latent vectors of shape `(N, Z+C)` through the hidden layers and output an image of dimension `(N, 1, H, W)`.
- You will also need to allow the forward pass to combine the flattened input image with the one-hot vectors (`torch.cat`) before passing them to the encoder and combining the latent space with the one-hot vectors (`torch.cat`) before passing them to the decoder.

Training
The same as the previous deliverable, expect that you should use the `one_hot` function to generate the conditioned input. Select the `conditional_vec_dim (C)` properly. Note that your implementation should not affect your previous deliverable if `C = 0`.

Deliverables for the PDF report: As in Fig. 8, show the result of CVAE output conditioned on one-hot label vectors from 0 to 9.

4.5 Deliverable 5: Structured Output Prediction by CVAE

As suggested in the original paper, instead of conditioning on class c in the previous deliverable. the CVAE architecture can also be used to reconstruct the digit conditioned on the part of the

Figure 8: Example of CVAE output conditioned on label.

image. This allows you to predict the full digit with only part of it available.

Training
If you implement your previous deliverable correctly, you should have no need to modify anything to your `myVAE` class. However, instead of using one-hot vectors, now condition on the flattened lower left part of the image, which has a dimension of `C = H*W/4`.

Deliverables for the PDF report: Show the result of CVAE output conditioned on a quarter of the image. For comparison, you also need to implement a `BaselineNN` with the same capacity as your decoder. As in Fig. 9, you should find that unlike `BaselineNN` can only generate one deterministic prediction, `CVAE` can generate multiple outputs and could possibly generate images similar to the original input.

NOTE: You will have to implement certain parts of the result section to generate the results for this deliverable.

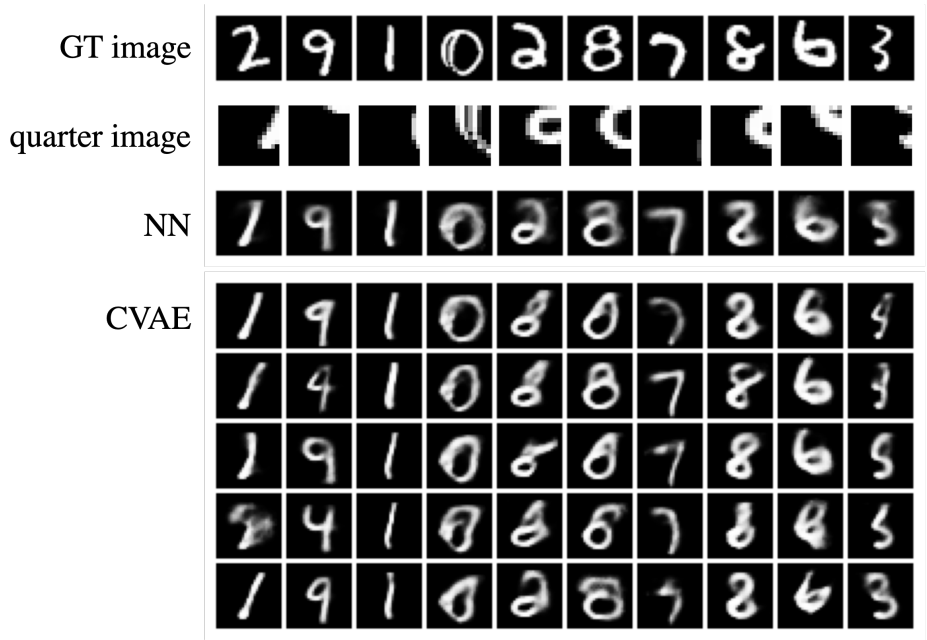


Figure 9: Example of CVAE output conditioned on quarter images.

4.6 Deliverable 6 (Bonus): Generation with Vector Quantized VAE

In this deliverable, we will use VQ-VAEs on the CIFAR dataset for image generation. The given notebook, `VQ-VAE Notebook.ipynb`, (adapted from https://colab.research.google.com/github/zalandoresearch/pytorch-vq-vae/blob/master/vq-vae.ipynb) can be *nearly* run from

start to finish as is. However, feel free to play around with the hyperparameters. For this deliverable, you'll need to tweak/add a few lines in the implementation and answer questions about the implementation.

Deliverables for the PDF report:

- How does the Vector Quantizer module create a discrete latent space representation?
- Explain the role of the hyperparameters *num hiddens*, *num residual hiddens* and *num residual layers*.
- Submit plots of the reconstruction loss while training and projection of the learned codebook (you can find this in the last cell of the notebook).
- The VQ-VAE training loss has 3 components: reconstruction loss, codebook loss and commitment loss.
  $\mathcal{L} = \underbrace{\mathbb{E}_{z\sim q(z|x)}[-\log p(x|z)]}_{\text{Reconstruction Loss}} + \underbrace{\lambda \cdot \mathbb{E}_{z\sim q(z|x)}[\|z - e(z)\|^2]}_{\text{Commitment Loss}} + \underbrace{\gamma \cdot \mathbb{E}_{z\sim q(z|x)}[\|e(z) - \text{sg}(z)\|^2]}_{\text{Vector Quantization Loss}}$ Using the provided hyperparamters, train the model without the commitment loss and submit plots of the reconstruction loss while training and projection of the learned codebook.
- What effect does removing the commitment loss have on the training? Why do you think this is happening? *(Hint: Compare the range of values of embeddings produced by the encoder in both cases, training with the commitmennt loss and without)*