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!
```

```python
# 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
```

```python
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
```