Deep Generative Models

Programming Exersice 2

Ron Ferens, 037222825

Code

Implementation of the VAE.py file:

```
"""NICE model
"""
import torch
import torch.nn as nn
import torch.nn.functional as F
import typing


class Model(nn.Module):
    def __init__(self, latent_dim: int, device: str):
        """
        Initialize a VAE
        Args:
            latent_dim: dimension of embedding
            device: run on cpu or gpu
        """
        # ----------------------------
        # Encoder
        # ----------------------------
        super(Model, self).__init__()
        self.device = device
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 4, 1, 2),  # B,  32, 28, 28
            nn.ReLU(True),
            nn.Conv2d(32, 32, 4, 2, 1),  # B,  32, 14, 14
            nn.ReLU(True),
            nn.Conv2d(32, 64, 4, 2, 1),  # B,  64,  7, 7
        )
        self.mu = nn.Linear(64 * 7 * 7, latent_dim)
        self.log_var = nn.Linear(64 * 7 * 7, latent_dim)
        # ----------------------------
        # Decoder
        # ----------------------------
        self.upsample = nn.Linear(latent_dim, 64 * 7 * 7)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 4, 2, 1),  # B,  64,  14,  14
            nn.ReLU(True),
            nn.ConvTranspose2d(32, 32, 4, 2, 1, 1),  # B,  32, 28, 28
            nn.ReLU(True),
            nn.ConvTranspose2d(32, 1, 4, 1, 2),  # B, 1, 28, 28
            nn.Sigmoid()
        )

    def sample(self, sample_size: int, mu: float = None, log_var=None) -> typing.List:
        """
        Sampled images from the model
        :param sample_size: Number of samples
        :param mu: z mean, None for prior (init with zeros)
        :param log_var: z log-STD, None for prior (init with zeros)
        :return:
        """
        if mu is None:
            mu = torch.zeros((sample_size, self.latent_dim)).to(self.device)
        if log_var is None:
```

```python
            log_var = torch.zeros((sample_size, self.latent_dim)).to(self.device)
        # Making sure the model is not trained while sampling and no gradients are
accumulated
        samples = []
        with torch.no_grad():
            z = self.z_sample(mu=mu, log_var=log_var)
            output = self.decoder(self.upsample(z).view(-1, 64, 7, 7))
            for i in range(sample_size):
                samples.append(output[i].squeeze().data.cpu().numpy())
        return samples

    def z_sample(self, mu: torch.Tensor, log_var: torch.Tensor) -> torch.Tensor:
        """
        Applying the reparameterization trick to generate Z
        :param mu: Z mean
        :param log_var: z log-STD
        :return: samples of Z
        """
        # Getting the STD from log−variance input
        std = torch.exp(0.5 * log_var)
        # Sampling epsilon from Normal (0, 1)
        eps = torch.rand_like(std).to(self.device)
        # Applying the reparameterization trick for Gaussian
        z = mu + eps * std
        return z

    @staticmethod
    def loss(x: torch.Tensor, recon: torch.Tensor, mu: float, log_var: float):
        """
        Calculating the model's loss composed of BCE and KL elements
        :param x: input data
        :param recon: reconstructed data
        :param mu: Z mean
        :param log_var: Z log-STD
        :return:
        """
        # Calculating the binary cross entropy loss
        bce_loss = F.binary_cross_entropy(recon, x, reduction='sum')
        # Calculating the KL Divergence loss
        kl_loss = torch.sum(1 + log_var - torch.exp(log_var) - torch.pow(mu, 2)) / 2.0
        # Calculating the ELBO(p, q, x)
        elbo = bce_loss - kl_loss
        return elbo

    def forward(self, x: torch.Tensor) -> typing.Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
        """"
        Running the model's forward pass
        :param x: input data
        """
        # Encoder forward pass
        x = self.encoder(x)
        x = x.reshape(-1, 64 * 7 * 7)
        # Generating the mean and variance values
        mu = self.mu(x)
        log_var = self.log_var(x)
        # Decoder forward pass
        z = self.z_sample(mu=mu, log_var=log_var)
        output = self.decoder(self.upsample(z).reshape(-1, 64, 7, 7))
```

```python
        return output, mu, log_var
```

Implementation of the train.py file:

```python
"""Training procedure for NICE.
"""
import argparse
import torch, torchvision
from torchvision import transforms
import numpy as np
from VAE import Model
import matplotlib.pyplot as plt
import torch.nn as nn

PLOT_DIM = 5


def train(vae: nn.Module, trainloader: torch.utils.data.DataLoader, optimizer: torch.optim) ->
float:
    # set to training mode
    vae.train()
    # Setting an accumulated epoch loss
    running_loss = 0
    num_iterations = 0
    # Going over the training dataset (single epoch)
    for inputs, _ in trainloader:
        # Loading the batch input
        inputs = inputs.to(vae.device)
        # Model forward pass
        model_res, mu, log_var = vae(inputs)
        # Calculating the loss criteria
        optimizer.zero_grad()
        loss = vae.loss(x=inputs, recon=model_res, mu=mu, log_var=log_var)
        running_loss += loss
        num_iterations += 1
        # Backprop and optimization
        loss.backward()
        optimizer.step()
    epoch_loss = running_loss / num_iterations
    return epoch_loss.item()


def test(vae: nn.Module, testloader: torch.utils.data.DataLoader) -> float:
    vae.eval()  # set to inference mode
    with torch.no_grad():
        running_loss = 0
        for batch_idx, (inputs, _) in enumerate(testloader):
            # Loading the batch input
            inputs = inputs.to(vae.device)
            # Model forward pass
            model_res, mu, log_var = vae(inputs)
            # Calculating the loss criteria
            loss = vae.loss(inputs, model_res, mu=mu, log_var=log_var)
            running_loss += loss.item()
    test_loss = running_loss / (batch_idx + 1)
    return test_loss
```

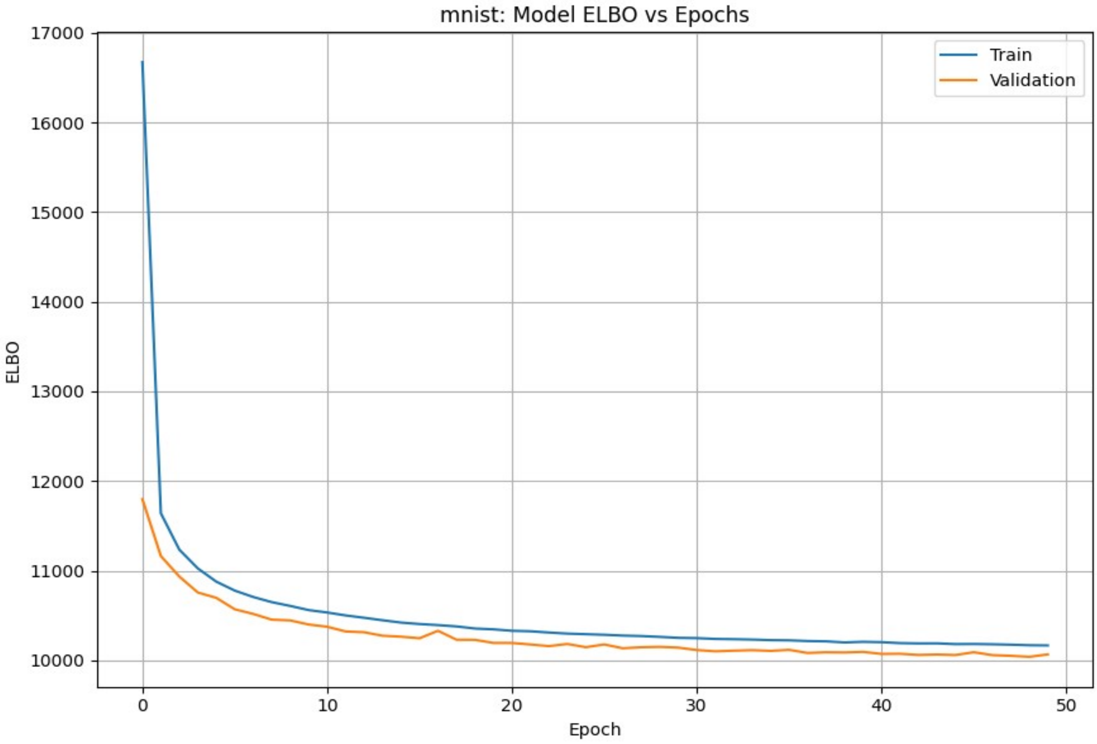
```
def main(args):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x + torch.zeros_like(x).uniform_(0., 1. / 256.)),  #
dequantization
        transforms.Normalize((0.,), (257. / 256.,)),  # rescales to [0,1]
    ])

    if args.dataset == 'mnist':
        trainset = torchvision.datasets.MNIST(root='./data/MNIST',
                                              train=True, download=True,
transform=transform)
        trainloader = torch.utils.data.DataLoader(trainset,
                                                  batch_size=args.batch_size,
shuffle=True, num_workers=2)
        testset = torchvision.datasets.MNIST(root='./data/MNIST',
                                             train=False, download=True,
transform=transform)
        testloader = torch.utils.data.DataLoader(testset,
                                                 batch_size=args.batch_size,
shuffle=False, num_workers=2)
    elif args.dataset == 'fashion-mnist':
        trainset = torchvision.datasets.FashionMNIST(root='~/torch/data/FashionMNIST',
                                                     train=True, download=True,
transform=transform)
        trainloader = torch.utils.data.DataLoader(trainset,
                                                  batch_size=args.batch_size,
shuffle=True, num_workers=2)
        testset = torchvision.datasets.FashionMNIST(root='./data/FashionMNIST',
                                                    train=False, download=True,
transform=transform)
        testloader = torch.utils.data.DataLoader(testset,
                                                 batch_size=args.batch_size,
shuffle=False, num_workers=2)
    else:
        raise ValueError('Dataset not implemented')
    # Creating an instance of the VEA model
    vae = Model(latent_dim=args.latent_dim, device=device).to(device)
    # Defining an Adam optimizer
    optimizer = torch.optim.Adam(vae.parameters(), lr=args.lr)

    # Resetting the ELBO arrays for train and validation
    elbo_train = []
    elbo_val = []

    for epoch in range(args.epochs):
        # Running a single training epoch
        epoch_loss = train(vae=vae, trainloader=trainloader, optimizer=optimizer)
        elbo_train.append(epoch_loss)
        # Running model validation
        val_loss = test(vae=vae, testloader=testloader)
        elbo_val.append(val_loss)
        print(f'Epoch {epoch}: Train Loss={epoch_loss}, Validation Loss={val_loss}')

    # Plotting the model's train and validation ELBO
    plt.figure()
    plt.plot(elbo_train, label='Train')
    plt.plot(elbo_val, label='Validation')
    plt.title(f"{args.dataset}: Model ELBO vs Epochs")
```

```python
    plt.xlabel("Epoch")
    plt.ylabel("ELBO")
    plt.legend()
    plt.grid(True)
    plt.show()

    # Saving samples generated by the trained model
    samples = vae.sample(sample_size=args.sample_size, mu=None, log_var=None)
    fig, ax = plt.subplots(nrows=np.ceil(args.sample_size / PLOT_DIM).astype(np.uint8),
                           ncols=PLOT_DIM)
    for i in range(args.sample_size):
        ax[(i // PLOT_DIM), (i % PLOT_DIM)].imshow(samples[i])
    [axi.set_axis_off() for axi in ax.ravel()]
    fig.suptitle(f"{args.dataset}: Trained Model - Output Samples")
    plt.show()


if __name__ == '__main__':
    parser = argparse.ArgumentParser('')
    parser.add_argument('--dataset',
                        help='dataset to be modeled.',
                        type=str,
                        default='fashion-mnist')
    parser.add_argument('--batch_size',
                        help='number of images in a mini-batch.',
                        type=int,
                        default=128)
    parser.add_argument('--epochs',
                        help='maximum number of iterations.',
                        type=int,
                        default=50)
    parser.add_argument('--sample_size',
                        help='number of images to generate.',
                        type=int,
                        default=25)
    parser.add_argument('--latent-dim',
                        help='.',
                        type=int,
                        default=100)
    parser.add_argument('--lr',
                        help='initial learning rate.',
                        type=float,
                        default=1e-3)
    args = parser.parse_args()
    main(args)
```

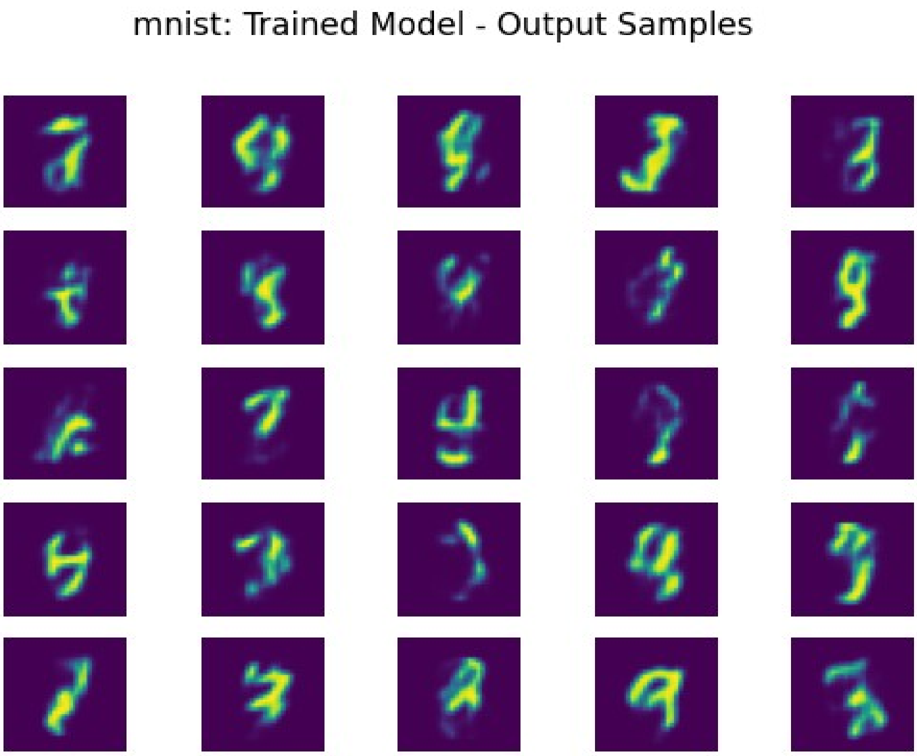
Results

1. Dataset - MNIST

The following plot show the train and test ELBO values per epochs:

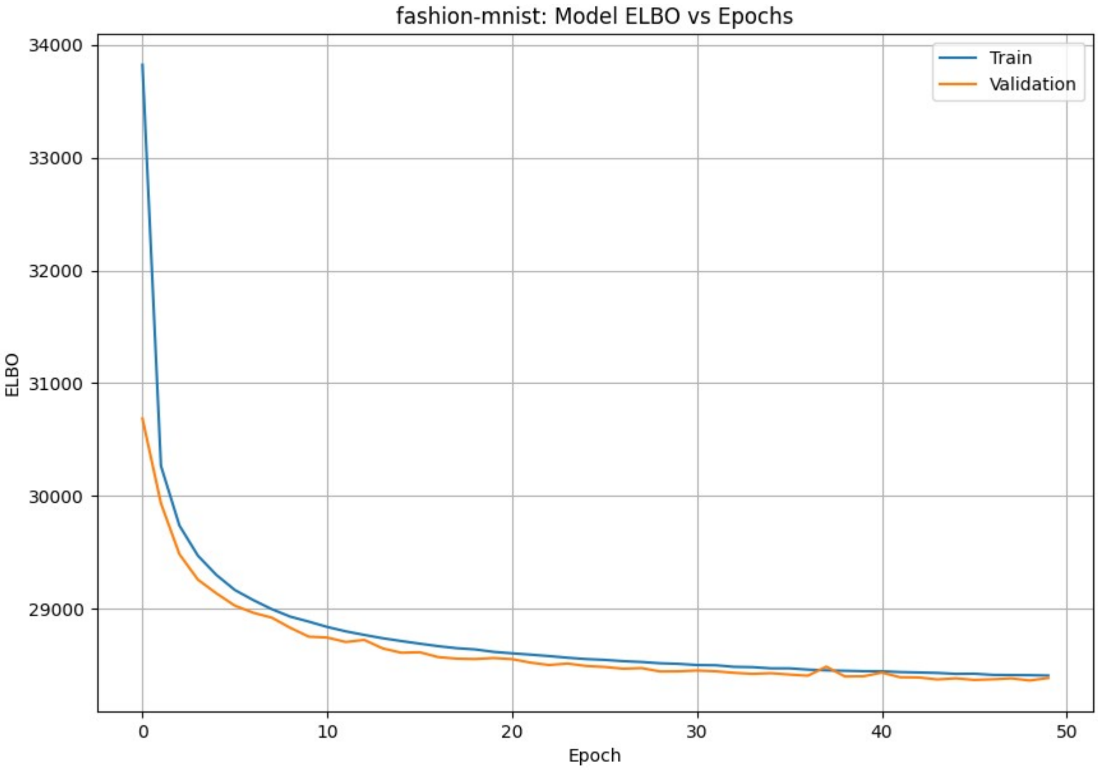


The following plot show 25 sampled images from the trained model:



2. Dataset - Fashion-MNIST

The following plot show the train and test ELBO values per epochs:



The following plot show 25 sampled images from the trained model: