Deep Generative Models Programming Assignment 1

Ron Ferens, 037222825

Code

nice.py code:

```
"""NICE model
References:
   (1) Paper at arXiv - https://arxiv.org/pdf/1410.8516v6.pdf
   (2) ICLR 2015 lecture - https://www.youtube.com/watch?v=7hKul_tOfsI&t=1s
   (3) Glow: Generative Flow with Invertible 1×1 Convolutions -
https://arxiv.org/pdf/1807.03039v2.pdf
"""
import typing
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from typing import Tuple

"""Additive coupling layer
"""


class AdditiveCoupling(nn.Module):
   def __init__(self, in_out_dim: int, mid_dim: int, hidden: int, mask_config: bool) -> None:
      """Initialize an additive coupling layer.
      Args:
         in_out_dim: input/output dimensions.
         mid_dim: number of units in a hidden layer.
         hidden: number of hidden layers.
         mask_config: 1 if transform odd units, 0 if transform even units.
      """
      super(AdditiveCoupling, self).__init__()

      # Making sure that the input/output dimension can be split by half
      assert in_out_dim % 2 == 0
      self._mask_config = mask_config

      # Defining the coupling model based on the number of requested hidden layers
      self.input_layer = nn.Sequential(nn.Linear(in_out_dim // 2, mid_dim), nn.ReLU())
      self.hidden_layers = nn.ModuleList(
         [nn.Sequential(nn.Linear(mid_dim, mid_dim), nn.ReLU()) for _ in range(hidden)])
      self.output_layer = nn.Linear(mid_dim, in_out_dim // 2)

   def forward(self, x: torch.Tensor, log_det_J: torch.Tensor, reverse: bool = False) ->
Tuple[torch.Tensor, torch.Tensor]:
      """Forward pass.
      Args:
         x: input tensor.
         log_det_J: log determinant of the Jacobian
         reverse: True in inference mode, False in sampling mode.
      Returns:
         transformed tensor and updated log-determinant of Jacobian.
```

```
        """
        # Split the input tensor into to X1 and X2 (two halves)
        x1 = x[:, ::2] if (self._mask_config % 2) == 0 else x[:, 1::2]
        x2 = x[:, 1::2] if (self._mask_config % 2) == 0 else x[:, ::2]

        m_x2 = self.input_layer(x2)
        for i in range(len(self.hidden_layers)):
            m_x2 = self.hidden_layers[i](m_x2)
        m_x2 = self.output_layer(m_x2)
        if reverse:
            x1 = x1 - m_x2
        else:
            x1 = x1 + m_x2

        out = torch.zeros_like(x)
        if (self._mask_config % 2) == 0:
            out[:, ::2] = x1
            out[:, 1::2] = x2
        else:
            out[:, ::2] = x2
            out[:, 1::2] = x1
        return out, log_det_J


"""Affine coupling layer
"""


class AffineCoupling(nn.Module):
    def __init__(self, in_out_dim: int, mid_dim: int, hidden: int, mask_config: bool) -> None:
        """Initialize an affine coupling layer.
        Args:
            in_out_dim: input/output dimensions.
            mid_dim: number of units in a hidden layer.
            hidden: number of hidden layers.
            mask_config: 1 if transform odd units, 0 if transform even units.
        """
        super(AffineCoupling, self).__init__()

        # Making sure that the input/output dimension can be split by half
        assert in_out_dim % 2 == 0
        self._mask_config = mask_config

        # Defining the coupling model based on the number of requested hidden layers
        self.input_layer = nn.Sequential(nn.Linear(in_out_dim // 2, mid_dim), nn.ReLU())
        self.hidden_layers = nn.ModuleList(
            [nn.Sequential(nn.Linear(mid_dim, mid_dim), nn.ReLU()) for _ in range(hidden)])
        self.output_layer = nn.Linear(mid_dim, in_out_dim)

    def forward(self, x: torch.Tensor, log_det_J: torch.Tensor, reverse: bool = False) ->
Tuple[torch.Tensor, torch.Tensor]:
```

```
        """Forward pass.
        Args:
            x: input tensor.
            log_det_J: log determinant of the Jacobian
            reverse: True in inference mode, False in sampling mode.
        Returns:
            transformed tensor and updated log-determinant of Jacobian.
        """
        # Split the input tensor into to X1 and X2 (two halves)
        x1 = x[:, ::2] if (self._mask_config % 2) == 0 else x[:, 1::2]
        x2 = x[:, 1::2] if (self._mask_config % 2) == 0 else x[:, ::2]

        m_x2 = self.input_layer(x2)
        for i in range(len(self.hidden_layers)):
            m_x2 = self.hidden_layers[i](m_x2)
        m_x2 = self.output_layer(m_x2)

        log_s, t = m_x2[:, ::2, ...], m_x2[:, 1::2, ...]
        s = torch.sigmoid(log_s)
        if reverse:
            x1 = (x1 - t) / s
        else:
            x1 = (s * x1) + t

        log_det_J = torch.sum(torch.log(torch.abs(s)))

        out = torch.zeros_like(x)
        if (self._mask_config % 2) == 0:
            out[:, ::2] = x1
            out[:, 1::2] = x2
        else:
            out[:, ::2] = x2
            out[:, 1::2] = x1
        return out, log_det_J


"""Log-scaling layer.
"""


class Scaling(nn.Module):
    def __init__(self, dim: int) -> None:
        """Initialize a (log-)scaling layer.
        Args:
            dim: input/output dimensions.
        """
        super(Scaling, self).__init__()
        self.scale = nn.Parameter(torch.zeros((1, dim)), requires_grad=True)
        self.eps = 1e-5

    def forward(self, x: torch.Tensor, reverse: bool = False) -> Tuple[torch.Tensor, torch.Tensor]:
```

```
        """Forward pass.
        Args:
            x: input tensor.
            reverse: True in inference mode, False in sampling mode.
        Returns:
            transformed tensor and log-determinant of Jacobian.
        """
        # Calculating the scale factor
        scale = torch.exp(self.scale) + self.eps
        if reverse:
            # h_reverse = h_prev / exp(s)
            x = x / scale
        else:
            # h = h_prev * exp(s)
            x = x * scale

        # Calculating the log-determinant of Jacobian
        log_det_J = torch.sum(self.scale) + self.eps
        return x, log_det_J


"""Standard logistic distribution.
"""


class StandardLogistic(torch.distributions.Distribution):
    def __init__(self):
        super(StandardLogistic, self).__init__()

    @staticmethod
    def log_prob(x: torch.Tensor) -> torch.Tensor:
        """Computes data log-likelihood.
        Args:
            x: input tensor.
        Returns:
            log-likelihood.
        """
        return -(F.softplus(x) + F.softplus(-x))

    @staticmethod
    def sample(size: typing.Tuple) -> torch.Tensor:
        """Samples from the distribution.
        Args:
            size: number of samples to generate.
        Returns:
            samples.
        """
        z = torch.distributions.Uniform(0., 1.).sample(size).cuda()
        return torch.log(z) - torch.log(1. - z)
```

```
"""NICE main model.
"""


class NICE(nn.Module):
  def __init__(self, prior: str, coupling: int, coupling_type: str, in_out_dim: int, mid_dim: int,
hidden: int,
         device: str):
    """Initialize a NICE.
    Args:
      coupling_type: 'additive' or 'adaptive'
      coupling: number of coupling layers.
      in_out_dim: input/output dimensions.
      mid_dim: number of units in a hidden layer.
      hidden: number of hidden layers.
      device: run on cpu or gpu
    """
    super(NICE, self).__init__()
    self.device = device
    if prior == 'gaussian':
      self.prior = torch.distributions.Normal(torch.tensor(0.).to(device),
torch.tensor(1.).to(device))
    elif prior == 'logistic':
      self.prior = StandardLogistic()
    else:
      raise ValueError('Prior not implemented.')

    self.in_out_dim = in_out_dim
    self.num_coupling_layers = coupling
    self.coupling_type = coupling_type
    mask_config = 1

    if self.coupling_type.lower() == 'additive':
      self.coupling_layers = nn.ModuleList([AdditiveCoupling(in_out_dim=in_out_dim,
                                   mid_dim=mid_dim,
                                   hidden=hidden,
                                   mask_config=(mask_config + i) % 2) for i in
                         range(coupling)])
    elif self.coupling_type.lower() == 'affine':
      self.coupling_layers = nn.ModuleList([AffineCoupling(in_out_dim=in_out_dim,
                                  mid_dim=mid_dim,
                                  hidden=hidden,
                                  mask_config=(mask_config + i) % 2) for i in
                         range(coupling)])
    else:
      raise ValueError('coupling_type not implemented.')

    self.scaling = Scaling(in_out_dim)

  def f_inverse(self, z: torch.Tensor) -> torch.Tensor:
    """Transformation g: Z -> X (inverse of f).
```

```
    Args:
      z: tensor in latent space Z.
    Returns:
      transformed tensor in data space X.
    """
    x, _ = self.scaling(z, reverse=True)
    for i in reversed(range(self.num_coupling_layers)):
      x, _ = self.coupling_layers[i](x, 0, reverse=True)
    return x

  def f(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    """Transformation f: X -> Z (inverse of g).

    Args:
      x: tensor in data space X.
    Returns:
      transformed tensor in latent space Z and log determinant Jacobian
    """
    log_det_J = 0.

    # Forward pass over the coupling layers
    for i in range(self.num_coupling_layers):
      x, log_det_j_layer = self.coupling_layers[i](x, log_det_J)
      log_det_J += log_det_j_layer

    # Scaling the latent tensors and Jacobian
    x, log_det_j_scale = self.scaling(x)
    log_det_J += log_det_j_scale
    return x, log_det_J

  def log_prob(self, x: torch.Tensor) -> torch.Tensor:
    """Computes data log-likelihood.

    (See Section 3.3 in the NICE paper.)

    Args:
      x: input minibatch.
    Returns:
      log-likelihood of input.
    """
    z, log_det_J = self.f(x)

    #  Log det for rescaling from [0.256] (after dequantization) to [0,1]
    log_det_J -= np.log(256) * self.in_out_dim

    log_ll = torch.sum(self.prior.log_prob(z), dim=1)
    return log_ll + log_det_J

  def sample(self, size: int) -> torch.Tensor:
    """Generates samples.
```

```
        Args:
            size: number of samples to generate.
        Returns:
            samples from the data space X.
        """
        z = self.prior.sample((size, self.in_out_dim)).to(self.device)
        x = self.f_inverse(z)
        return x

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward pass.

        Args:
            x: input minibatch.
        Returns:
            log-likelihood of input.
        """
        return self.log_prob(x)
```

train.py code:

```
"""Training procedure for NICE.
"""
import argparse
import torch
import torchvision
from torchvision import transforms
import matplotlib.pyplot as plt
import nice
from os import getcwd, mkdir
from os.path import join, exists


def train(flow: torch.nn.Module,
        trainloader: torch.utils.data.DataLoader,
        optimizer: torch.optim) -> float:
    # set to training mode
    flow.train()

    # Setting an accumulated epoch loss
    running_loss = 0
    num_iterations = 0

    # Going over the training dataset (single epoch)
    for inputs, _ in trainloader:
        # change  shape from BxCxHxW to Bx(C*H*W)
        inputs = inputs.view(inputs.shape[0], inputs.shape[1] * inputs.shape[2] *
inputs.shape[3]).to(flow.device)

        # Model forward pass
        model_res = flow(inputs)

        # Calculating the loss criteria
        optimizer.zero_grad()
        loss = -model_res.mean()
        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(flow: torch.nn.Module,
        testloader: torch.utils.data.DataLoader,
        filename: str,
        epoch: int,
        sample_shape: int) -> float:
```

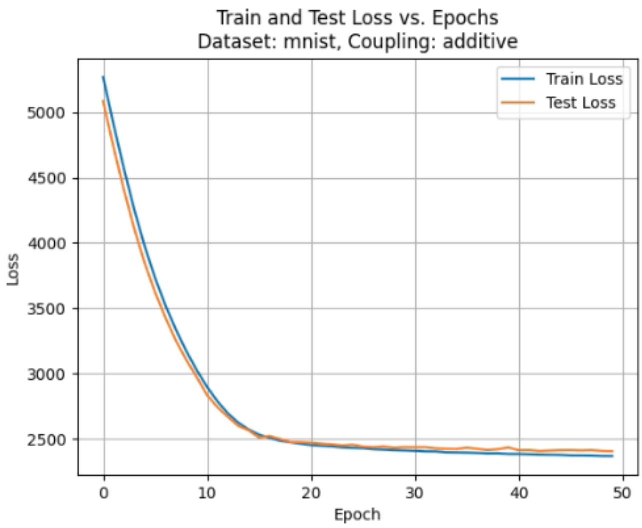
```
    # set to inference mode
    flow.eval()

    # Setting an accumulated test loss
    running_loss = 0
    num_iterations = 0

    with torch.no_grad():
        samples = flow.sample(100).cpu()
        a, b = samples.min(), samples.max()
        samples = (samples - a) / (b - a + 1e-10)
        samples = samples.view(-1, sample_shape[0], sample_shape[1], sample_shape[2])
        torchvision.utils.save_image(torchvision.utils.make_grid(samples), filename + 'epoch%d.png'
% epoch)

        # Going over the test dataset (single epoch)
        for inputs, _ in testloader:
            # change  shape from BxCxHxW to Bx(C*H*W)
            inputs = inputs.view(inputs.shape[0], inputs.shape[1] * inputs.shape[2] *
inputs.shape[3]).to(flow.device)

            # Model forward pass
            model_res = flow(inputs)

            # Calculating the loss criteria
            loss = -model_res.mean()
            running_loss += loss
            num_iterations += 1

    test_loss = running_loss / num_iterations
    return test_loss.item()


def main(args):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    sample_shape = [1, 28, 28]
    full_dim = 1 * 28 * 28

    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (1.,)),
        transforms.Lambda(lambda x: x + torch.zeros_like(x).uniform_(0., 1. / 256.))  # dequantization
    ])

    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')

model_save_filename = '%s_' % args.dataset \
            + 'batch%d_' % args.batch_size \
            + 'coupling%d_' % args.coupling \
            + 'coupling_type%s_' % args.coupling_type \
            + 'mid%d_' % args.mid_dim \
            + 'hidden%d_' % args.hidden \
            + '.pt'

# Creating an output folder for the selected dataset
dataset_samples_folder = f'{args.dataset}_samples'
if not exists(dataset_samples_folder):
    mkdir(dataset_samples_folder)
dataset_samples_folder = join(dataset_samples_folder, args.coupling_type)
if not exists(dataset_samples_folder):
    mkdir(dataset_samples_folder)

# Creating the NICE model
flow = nice.NICE(prior=args.prior,
        coupling=args.coupling,
        coupling_type=args.coupling_type,
        in_out_dim=full_dim,
        mid_dim=args.mid_dim,
        hidden=args.hidden,
        device=device).to(device)

optimizer = torch.optim.Adam(flow.parameters(), lr=args.lr)

train_loss = []
test_loss = []
for epoch in range(args.epochs):
    # Running training
    train_loss.append(train(flow, trainloader, optimizer))

    # Running test on single epoch
    test_loss.append(test(flow=flow,
            testloader=testloader,
```

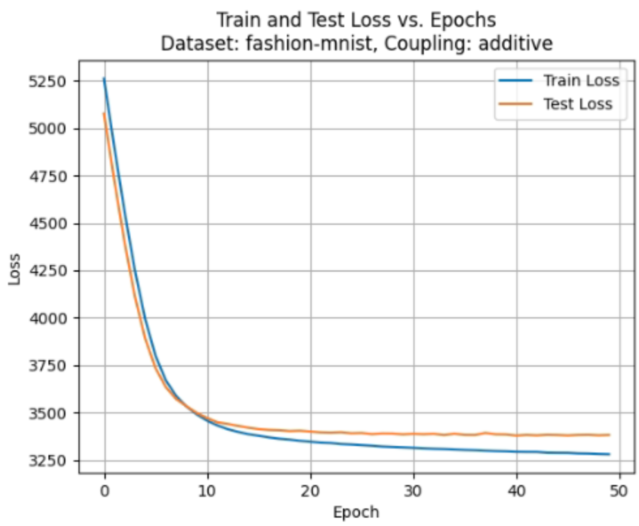
```
                        filename=join(dataset_samples_folder, f"{args.dataset}_sampled_"),
                        epoch=epoch,
                        sample_shape=sample_shape))

        # Printing the training and test losses' values
        print(f"Epoch {epoch}:  train loss={train_loss[-1]}, test loss={test_loss[-1]}")

        # Saving the model
        if epoch % 10 == 0:
            torch.save(flow.state_dict(), "./models/" + model_save_filename)

    # Ploting the train and test loss over the training process (for each epoch)
    plt.figure()
    plt.plot(train_loss, label='Train Loss')
    plt.plot(test_loss, label='Test Loss')
    plt.title("Train and Test Loss vs. Epochs\n" + f'Dataset: {args.dataset}, Coupling:
{args.coupling_type}')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid(True)
    plt.savefig(join(getcwd(), f"loss_{args.dataset}_loss_coupling_{args.coupling_type}.png"))


if __name__ == '__main__':
    parser = argparse.ArgumentParser('')
    parser.add_argument('--dataset', help='dataset to be modeled.', type=str, default='mnist')
    parser.add_argument('--prior', help='latent distribution.', type=str, default='logistic')
    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=64)
    parser.add_argument('--coupling_type', help='.', type=str, default='affine')
    parser.add_argument('--coupling', help='.', type=int, default=4)
    parser.add_argument('--mid-dim', help='.', type=int, default=1000)
    parser.add_argument('--hidden', help='.', type=int, default=5)
    parser.add_argument('--lr', help='initial learning rate.', type=float, default=1e-3)
    input_args = parser.parse_args()
    main(input_args)
```

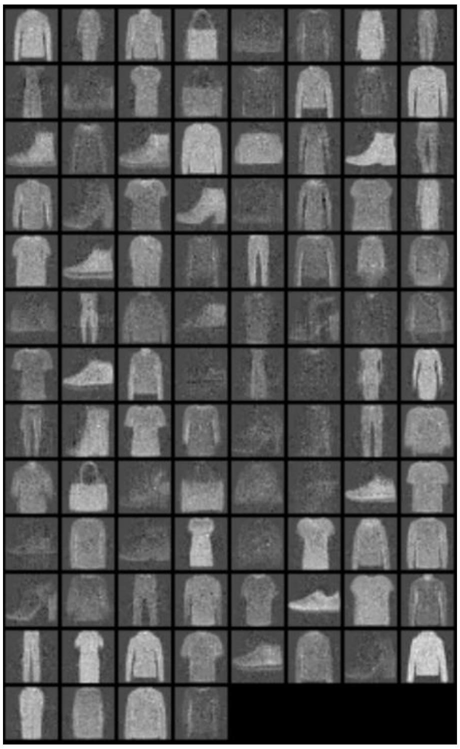
Results

1. **MNIST** with **Additive** coupling layers

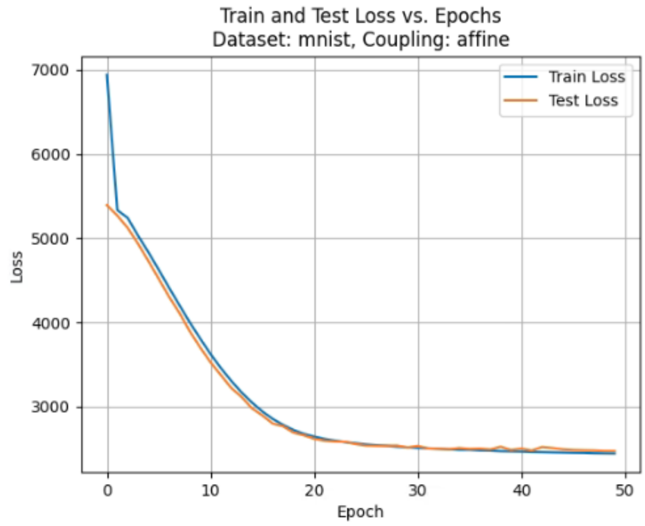


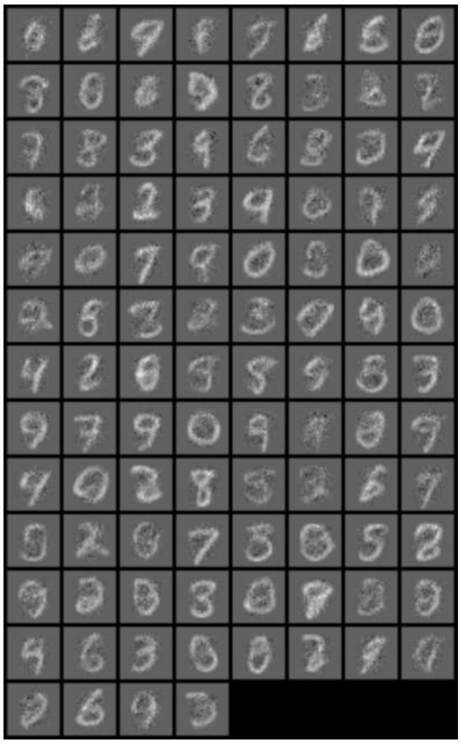
2. **Fashion-MNIST** with **Additive** coupling layers





3. **MNIST** with **Affine** coupling layers





4. **Fashion-MNIST** with **Affine** coupling layers