Statistical Methods in Image Processing EE-048954

Homework 2: Langevin Dynamics and Energy-Based Models

Due Date: May 31, 2022

Submission Guidelines

- Submission only in **pairs** on the course website (Moodle).
- Working environment:
  - We encourage you to work in `Jupyter Notebook` online using Google Colab as it does not require any installation.
- You should submit two **separated** files:
  - A `.ipynb` file, with the name: `ee048954_hw2_id1_id2.ipynb` which contains your code implementations.
  - A `.pdf` file, with the name: `ee048954_hw2_id1_id2.pdf` which is your report containing plots, answers, and discussions.
  - **No handwritten submissions** and no other file-types (`.docx` , `.html` , ...) will be accepted.

Mounting your drive for saving/loading stuff

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

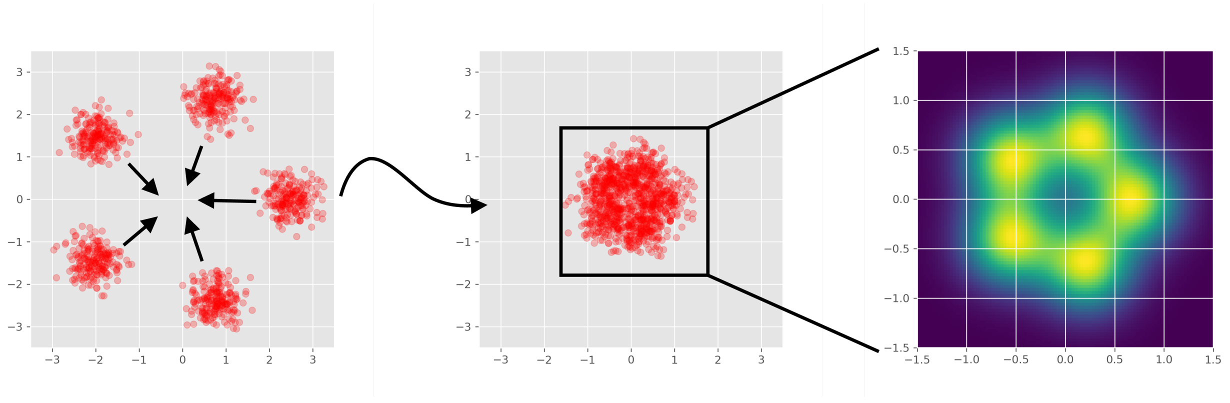
Importing relevant libraries for Part I

In [ ]:

```
## Standard libraries
import os
import math
import time
import numpy as np
import random
import copy

## Scikit-learn built-in dataset generator
from sklearn.datasets import make_blobs

## Progress bar
import tqdm

## Imports for plotting
import matplotlib.pyplot as plt
import matplotlib.animation as animation
%matplotlib inline
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
plt.style.use('ggplot')
```

Part I: Toy 2D Dataset (10 points)

In this homework we will look into stochastic sampling techniques that could be used to sample from energy-based models of images. But, to kick things off, we will first start with the simple toy 2D dataset comprised of 5 rotated and equally-spaced Gaussian Mixture distribution we are familiar with from HW1 with slight adjustments:



More formally, the probability density distribution considered is given by:

$$p(x;\sigma,\{\mu_i\}) = \frac{1}{M}\sum_{m=1}^{M}\frac{1}{2\pi\sigma^2}\exp\left\{-\frac{1}{2\sigma^2}||x-\mu_i||^2\right\},$$

with $M = 5$, $\sigma^2 = 0.1$,

and $\{\mu_m\} = 0.7\cdot\{(1,0)^T, (\cos(\frac{2\pi}{5}),\sin(\frac{2\pi}{5}))^T, (\cos(\frac{4\pi}{5}),\sin(\frac{4\pi}{5}))^T, (\cos(\frac{6\pi}{5}),\sin(\frac{6\pi}{5}))^T, (\cos(\frac{8\pi}{5}),\sin(\frac{8\pi}{5}))^T\}$.

Task 1. Write down the analytical gradient of $\log p(x)$ with respect to x. i.e. $\nabla_x \log p(x)$.

While this distribution can be sampeld trivially using standard techniques, here we will sample from it using Langevin Dynamics.

Task 2. Implement Langevin Dynamics for sampling from $p(x)$:

- Initialize 1000 random 2D points x i.i.d distributed according to $U[-3.0, 3.0]$.
- Update the points according to the Langevin Dynamics update step:

$$x^{k+1} = x^k + \varepsilon\nabla\log p(x^k) + \sqrt{2\varepsilon}N^k.$$

  Use $\sqrt{2\varepsilon} = \frac{10}{256}$ and $N \sim \mathcal{N}(0, I)$.

- Repeat the previous step for $K = 5000$ iterations.

Task 3. Draw $N = 1000$ samples x_i from $p(x)$ using the provided code lines below and compare them visually to the samples drawn using Langevin Dynamics. Present both sample types and discuss the results.

In [ ]:
```
num_samples, seed, = 1000 , 0
np.random.seed(seed)
angles = np.linspace(0, 2 * np.pi, 5, endpoint=False)
centers = np.stack([0.7 * np.array([np.cos(angle), np.sin(angle)]) for angle in angles])
real_samples = make_blobs(num_samples, centers=centers, cluster_std=np.sqrt(.1), random_state=seed, shuff
```

Task 4. Repeat Tasks 2 and 3, this time with $\sqrt{2\varepsilon} = \left\{\frac{1}{256}, \frac{200}{256}\right\}$. Compare the results to Task 3, and explain the effect of ε on the resulting samples. Tip: To have a clear visual examination of the phenomenon, plot the path a sampled point goes through throughout the dynamics for different values of ε.

Importing additional relevant libraries for Parts II-IV

In [ ]:
```
## Useful for creating GIFs
import imageio
```

```python
## PyTorch
import torch
import torchvision

# Function for setting the seed
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
set_seed(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# device to be used for Parts II-IV is preferably a GPU
# try to change the runtime type to GPU if you can in Google Colab
device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Using device", device)
```

Part II: Langevin Dynamics (30 points)

General Introduction

In the remainder of this exercise we will focus on sampling from an Energy Based Model (EBM) that was trained to fit the distribution $p(x)$ of the MNIST digits dataset (no need to download the dataset). The EBM is given in the form of a Convolutional Neural Network (CNN), and defined by

$$p_\theta(x) = \frac{1}{Z(\theta)} e^{-E_\theta(x)}.$$

Specifically, the model gets an image x and returns $E_\theta(x)$, where θ are the trained model parameters. We will use this model in order to sample new digits from the MNIST distribution. In order to do so, we will use MCMC with Langevin Dynamics.

Model Architecture

The provided class below `ResNet` implements the neural network approximating $E(x)$ with a parametric function $E_\theta(x)$, and is based on the architecture from the paper wide residual networks. Note that for our purposes in this exercise you **don't** need to understand this class thoroughly. You can treat this network as a black-box that accepts an image x as input and returns $E_\theta(x) \approx E(x)$ as output, and can also provide us with the gradient of $E(x)$ with respect to its input using automatic differentiation, i.e. $\nabla_x E(x)$.

In [ ]:

```python
# the next lines define the architecture of the model and its functionality in forward path
# (e.g how it operates when inputting an image)
class ResNet(torch.nn.Module):
    def __init__(self, n_channels):
        super().__init__()

        levels_params = [
            {'n_channels': 16, 'n_blocks': 2, 'downsample': False},
            {'n_channels': 32, 'n_blocks': 2, 'downsample': True},
            {'n_channels': 64, 'n_blocks': 2, 'downsample': True},
            {'n_channels': 64, 'n_blocks': 2, 'downsample': True},
            ]

        self._el = torch.nn.ModuleDict()

        self._el['in_conv'] = torch.nn.Conv2d(n_channels, 16, kernel_size=3, padding=3)
        n_channels = 16
        levels = torch.nn.ModuleList()
        for level_params in levels_params:
            level = torch.nn.ModuleDict()
            res_blocks = torch.nn.ModuleList()

            ## The first residual block in the level
            res_block = torch.nn.ModuleDict()
            n_channels_out = level_params['n_channels']
            if level_params['downsample']:
                res_block['shortcut_conv'] = torch.nn.Conv2d(n_channels, n_channels_out, kernel_size=2, s
            else:
                res_block['shortcut_conv'] = torch.nn.Conv2d(n_channels, n_channels_out, kernel_size=1)
            res_block['conv_1'] = torch.nn.Conv2d(n_channels, n_channels_out, kernel_size=3, padding=1)
            n_channels = n_channels_out
            if level_params['downsample']:
                res_block['conv_2'] = torch.nn.Conv2d(n_channels, n_channels, kernel_size=4, stride=2, pa
            else:
                res_block['conv_2'] = torch.nn.Conv2d(n_channels, n_channels, kernel_size=3, stride=1, pa
            res_blocks.append(res_block)

            ## The rest of the residual blocks in the level
            for _ in range(level_params['n_blocks'] - 1):
                res_block = torch.nn.ModuleDict()
                res_block['conv_1'] = torch.nn.Conv2d(n_channels, n_channels, kernel_size=3, padding=1)
                res_block['conv_2'] = torch.nn.Conv2d(n_channels, n_channels, kernel_size=3, padding=1)
                res_blocks.append(res_block)
            level['res_blocks'] = res_blocks
            levels.append(level)
        self._el['levels'] = levels

        self._el['out_fc'] = torch.nn.Linear(n_channels, 1, bias=False)

        for module in self.modules():
            if isinstance(module, torch.nn.Conv2d):
                torch.nn.init.xavier_uniform_(module.weight, gain=2 ** 0.5)
                if module.bias is not None:
                    module.bias.data.zero_()

    # functionality when inputting image x to the model:
    def forward(self, x):
        x = self._el['in_conv'](x)

        for level in self._el['levels']:
            for res_block in level['res_blocks']:
                shortcut = x
                x = torch.nn.functional.leaky_relu(x, 0.2)
                x = res_block['conv_1'](x)
                x = torch.nn.functional.leaky_relu(x, 0.2)
                x = res_block['conv_2'](x)
                if 'shortcut_conv' in res_block:
                    shortcut = res_block['shortcut_conv'](shortcut)
                x = x + shortcut

        x = torch.nn.functional.leaky_relu(x, 0.2)
        x = x.view(x.shape[0], x.shape[1], -1).sum(dim=2)
        x = self._el['out_fc'](x)

        return x[:, 0]
```

For your convinience, we provide pre-trained model weights $\theta^\star$ on the MNIST dataset, Courtesy of Mr. Omer Yair. To instantiate the model and load the pretrained weights $\theta^\star$ from the attached file `checkpoint.pt`, you can use the following:

In [ ]:

```
# instantiate the class above for images with 1 channel and load it to the device (CPU/GPU)
ebm = ResNet(n_channels=1).to(device)

# transfer the model to evaluation mode (as we don't want to train it, just to use it)
ebm.eval()
```

```
# load the trained model weights/parameters from the checkpoint file
checkpoint_path = 'checkpoint.pt'
ebm.load_state_dict(torch.load(checkpoint_path, map_location=device))
```

After loading the trained weights $\theta^{\star}$, we can feed the model with images x and get their approximated energy $E_{\theta^{\star}}(x)$ by a simple forward pass:

In [ ]:
```
# number of images to generate
n_imgs = 30

# randomly initialized 28x28x1 images with i.i.d pixels ~U[0,1]
imgs = torch.rand((n_imgs, 1, 28, 28), device=device)

# set the images to have a gradient graph so we could calculate the gradient of the model
imgs.requires_grad_(True)

# run the model: input the images x, getting as output their estimated energy E(x)
energy = ebm(imgs)
```

To calculate the gradient of the model output $E(x)$ with respect to the input images x, we can use automatic differentiation because we previously set the `requires_grad_` property of the input images to `True`:

In [ ]:
```
# calculates the gradient of the model: grad(E(x)) with respect to x.
grad = torch.autograd.grad(energy.sum(), imgs)[0]
```

Task 5. Run Langevin Dynamics for the provided EBM:

- Initialize 30 random images of size 28×28 distributed i.i.d according to $U[0,1]$.
- Update the images according to the Langevin Dynamics update step:

$$x^{k+1} = x^k + \varepsilon \nabla \log p(x^k) + \sqrt{2\varepsilon} N^k.$$

  Use $\sqrt{2\varepsilon} = \frac{2}{256}$ and $N \sim \mathcal{N}(0, I)$. Note that although we do not have an explicit expression for $p(x)$, but only for $E(x)$, we are still able to perform the update step, how? For calculating the gradient you can use automatic differentiation as described above.

- Repeat the previous step for $K = 2000$ iterations. Present the final samples and discuss the results.

Task 6. Repeat the previous task for $K = 2000$ iterations, this time with $\sqrt{2\varepsilon} = \frac{3}{256}$. Present the final samples and discuss the results comparing to the previous section.

Part III: MALA (30 points)

We will now expand Part II into the Metropolis-Adjusted Langevin Algorithm (MALA):

- Use the same initialization scheme as in Langevin Dynamics
- use the same update step as before

$$x^{k+1} = x^k + \varepsilon \nabla \log p(x^k) + \sqrt{2\varepsilon} N^k,$$

  with $\sqrt{2\varepsilon} = \frac{2}{256}$ and $N \sim \mathcal{N}(0, I)$.
- **Acceptance step**: accept the sample x^{k+1} according to the acceptance rule:
  - If x^{k+1} is more probable than x^k then accept x^{k+1}
  - else, replace x^{k+1} with x^k with probability α, where

$$\alpha \triangleq \frac{p(x^{k+1}) q(x^k \mid x^{k+1})}{p(x^k) q(x^{k+1} \mid x^k)}$$

and

$$q(x' \mid x) \propto \exp\left(-\frac{1}{4\varepsilon}\|x' - x - \varepsilon\nabla \log p(x)\|_2^2\right).$$

Task 7. In the acceptance step of the MALA algorithm we can use $E(x)$ without explicitly knowing $p(x)$. Why?

Task 8. Apply the MALA algorithm for $K = 2000$ iterations. Present the final samples and discuss the results.

Task 9. Apply the MALA algorithm for $K = 2000$ iterations, with $\sqrt{2\varepsilon} = \frac{3}{256}$. Present the final samples and discuss the results.

Task 10. Apply the MALA algorithm for $K = 20,000$ iterations (it might take a few minutes), with $\sqrt{2\varepsilon} = \frac{3}{256}$. **Instead of random initialization**, run Langevin dynamics as in Part II with $\sqrt{2\varepsilon} = \frac{2}{256}$ for $K = 500$ iterations and use these images as your initialization for the MALA algorithm. Present the final samples and examples for samples in intermediate iterations. Discuss the results.

Part IV: Perceptual and MMSE Denoising (30 points)

Given a noisy image $y = x + n$, where x is a clean image and $n \sim \mathcal{N}(0, \sigma^2 I)$, we would like to estimate x using Langevin Dynamics and the EBM model trained to estimate $p(x)$. We will perform denoising by drawing samples from $p(x|y)$.

Task 11. Write an explicit expression for $\nabla_x \log p(x|y)$ in terms of $p_x(x)$ and $p_n(n)$.

Task 12. How should the update step in Langevin Dynamics (Part II) be changed in order to draw samples from $p(x|y)$ instead of $p(x)$? We call such samples *perceptual* denoising results.

Task 13. The attachment of this exercise includes 30 noisy digits with $\sigma = \left\{\frac{50}{256}, \frac{100}{256}\right\}$ (you can load the images using the command `torch.load()`). Perform perceptual denoising with Langevin Dynamics with the parameters $K = 2000$ and $\sqrt{2\varepsilon} = \frac{2}{256}$. Present and discuss the results.

Task 14. Minimum MSE denoising can be obtained by averaging over the perceptual denoising results, since the conditional expectation $E[x|y]$ can be approximated by averaging over samples from $p(x|y)$ (namely, $E[x|y] \approx \frac{1}{N}\sum_{n=1}^{N} x_n$, where $\{x_n\}$ are samples from $p(x|y)$). Present the MMSE results averages over 10 perceptual samples. Why do you think the task of *perceptual denoising* is called that way?