Statistical Methods in Image Processing EE-048954

Homework 1: Kernel Density Estimation and Normalizing Flows

Due Date: May 08, 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_hw1_id1_id2.ipynb` which contains your code implementations.
  - A `.pdf` file, with the name: `ee048954_hw1_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 [ ]:

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

Importing relevant libraries for Part I

In [ ]:

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

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

Part I: Kernel Density Estimation (30 points)

The multivariate kernel density estimate of a density $f(\mathbf{x})$ given a set of samples $\{\mathbf{x}_i\}$ is given by

$$\hat{f}(\mathbf{x}) = \frac{1}{N}\frac{1}{|H|}\sum_{i=1}^{N} K(H^{-1}(\mathbf{x}_i - \mathbf{x})),$$

where H is a bandwidth matrix, $K(\cdot)$ is the kernel and $\{\mathbf{x}_i\}_{i=1}^{N}$ are $i.i.d.$ samples drawn from $f(\mathbf{x})$.

Task 1. Consider the following density functions:

- Gaussian Mixture:

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

with $M = 4$, $\sigma = \frac{1}{2}$, and $\{\mu_m\} = \{(0,0)^T, (0,2)^T, (2,0)^T, (2,2)^T\}$.

- Gaussian Mixture with $M = 4$, $\sigma = 1$, and $\{\mu_m\} = \{(0,0)^T, (0,2)^T, (2,0)^T, (2,2)^T\}$.
- Spiral with $\theta \sim \mathcal{U}[0, 4\pi]$ and $r|\theta \sim \mathcal{N}(\frac{\theta}{2}, 0.25)$.

For each of the three distributions above, implement a function that draws $N = 10000$ samples $\mathbf{x}_i$ from $f(\mathbf{x})$. Display the drawn samples for each distribution separately.

Task 2. Implement a function that accepts samples $\{\mathbf{x}_i\}$ and a bandwidth matrix H and estimates $\hat{f}(\mathbf{x})$ using multivariate kernel density estimation. Use the two-dimension separable kernel $K(u) = k(u_1)k(u_2)$ where $k(u) = \frac{1}{\sqrt{2\pi}}\exp\{\frac{-u^2}{2}\}$.

Task 3. For **each** distribution, compare between $f(\mathbf{x})$ and $\hat{f}(\mathbf{x})$ using the bandwidth matrices $H = \begin{pmatrix} h & 0 \\ 0 & h \end{pmatrix}$ with $h = 0.1, 0.5, 1.$ and display the estimation. Discuss the trade-off of the choice of h.

Task 4. Which of the distributions was the easiest/hardest to estimate? Why?

Importing additional relevant libraries for Part II

In [ ]:

```
## Scikit-Learn built-in dataset generators
from sklearn.datasets import make_moons, make_circles, make_blobs

## Progress bar
import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, random_split
from torch.optim import Adam
from torch.distributions.multivariate_normal import MultivariateNormal

# Traning will be done on CPU for this homework.
# For K=4, N=1500, epochs=1000 takes < 3 mins.
device = torch.device("cpu")
print("Using device", device)
```

Part II: Invertible Neural Networks (70 points)

In this part, we will take a closer look at invertible neural networks, otherwise known as *Normalizing Flows*. The most popular, current application of normalizing flows is to model datasets of images. In this part, we will implement a type of flows called *Coupling Flows* for a simplified problem of sampling from toy 2D datasets, although similar concepts (with deeper models + tricks) can be used to model images.

General Concept

Recall that given a random vector Z with a density $p_Z(\mathbf{z})$ (e.g. Gaussian) and an invertible function f, the density $p_X(\mathbf{x})$ of $X = f(Z)$ is given by:

$$\log p_X(\mathbf{x}) = \log p_Z(f^{-1}(\mathbf{x})) + \log \left|\det \frac{df^{-1}(\mathbf{x})}{d\mathbf{x}}\right|$$

Toy Datasets

The provided function `sample_2d_datasets` samples from 4 different toy datasets that we will use for this part to experiment with NFs. The supported options in this function are `{'Circles', 'Moons', 'GaussiansGrid',`

'GaussiansRot'} , where for the last distribution 'GaussiansRot' , the function supports a varying number of gaussians using the parameter num_gaussians .

In [ ]:

```
def sample_2d_datasets(dist_type, num_samples=1000, seed=0, num_gaussians=5):
  """
  function samples from simple pre-defined distributions in 2D.
  Inputs:
    - dist_type: str specifying the distribution to be chosen from:
      {'Circles', 'Moons', 'GaussiansGrid', 'GaussiansRot'}
    - num_samples: Number of samples to draw from dist_type (int).
    - seed: Random seed integer.
    - num_gaussians: Number of rotated gaussians if dist_type='GaussiansRot'.
      (relevant only for dist_type='GaussiansRot', should be a keyword argument)
  Outputs:
    - data (np.array): array of num_samplesx2 samples from dist_type
  """
  np.random.seed(seed)
  if dist_type == 'Circles':
    data = make_circles(num_samples, noise=.1, factor=.8, random_state=seed, shuffle=True)[0]
  elif dist_type == 'Moons':
    data = make_moons(num_samples, noise=.1, random_state=seed, shuffle=True)[0]
  elif dist_type == 'GaussiansGrid':
    centers = np.array([[0,0],[0,2],[2,0],[2,2]])
    data = make_blobs(num_samples, centers=centers, cluster_std=.5, random_state=seed, shuffle=True)[0]
  elif dist_type == 'GaussiansRot':
    angles = np.linspace(0, 2 * np.pi, num_gaussians, endpoint=False)
    centers = np.stack([2.5 * np.array([np.cos(angle), np.sin(angle)]) for angle in angles])
    data = make_blobs(num_samples, centers=centers, cluster_std=np.sqrt(.1), random_state=seed, shuffle=T
  else:
    raise NotImplementedError
  return data
```

Task 1. To get acquainted with this function, for each of the 4 distributions above, draw $N = 1000$ samples x_i. Display the drawn samples for each distribution in a separate plot.

For convience purposes, we wrap the function sample_2d_datasets with a torch.utils.data.Dataset class, and implement the methods __len__ and __getitem__ to sample batches afterward with dataloaders when we train our models.

In [ ]:

```
class ToyDataset(Dataset):
  def __init__(self, dist_type, num_samples=1000, seed=0, num_gaussians=5):
      """
      Wrapper around the function "sample_2d_datasets" to allow iterating
      batches using a datalaoder when training our normalizing flow model.
      """
      self.data = sample_2d_datasets(dist_type, num_samples, seed, num_gaussians)
      self.num_samples = num_samples

  def __len__(self):
      return self.num_samples

  def __getitem__(self, index):
      return torch.from_numpy(self.data[index]).type(torch.FloatTensor)
```

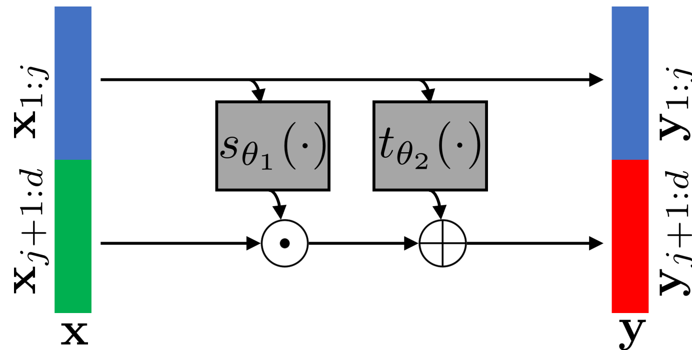
Coupling Layers

Next, we look at possible transformations to apply inside the flow, focusing on the simplest and most efficient one. A recent popular flow layer, which works well in combination with deep neural networks, is the coupling layer introduced by Dinh et al.. The input $\mathbf{x}$ is arbitrarily split into two parts, $\mathbf{x}_{1:j}$ and $\mathbf{x}_{j+1:d}$, of which the first remains unchanged by the flow. Yet, $\mathbf{x}_{1:j}$ is used to parameterize the transformation for the second part, $\mathbf{x}_{j+1:d}$. In this coupling layer, we apply an affine transformation by scaling the input by $\mathbf{s}$ and shifting it by $\mathbf{t}$. In other words, our transformation $\mathbf{y} = f(\mathbf{x})$ looks as follows:

$$\mathbf{y}_{1:j} = \mathbf{x}_{1:j}$$

$$\mathbf{y}_{j+1:d} = \mathbf{s}_{\theta_1}(\mathbf{x}_{1:j}) \odot \mathbf{x}_{j+1:d} + \mathbf{t}_{\theta_2}(\mathbf{x}_{1:j})$$

$\mathbf{y}$ is the output of the flow layer, the functions $\mathbf{s}$ and $\mathbf{t}$ are implemented as neural networks, and the sum and multiplication are performed element-wise. Here's a block diagram that visualize the coupling layer in the form of a computation graph:



For convience, since we train using the log-density, it is common practice to apply an exponential on the predicted scaling factor prior to multiplication with $\mathbf{x}_{j+1:d}$, as this simplifies the calculation of the log-determinant of the Jacobian that we will derive shortly. Hence, the implemented transformation in practice is usually:

$$\mathbf{y}_{1:j} = \mathbf{x}_{1:j}$$

$$\mathbf{y}_{j+1:d} = \exp(\mathbf{s}_{\theta_1}(\mathbf{x}_{1:j})) \odot \mathbf{x}_{j+1:d} + \mathbf{t}_{\theta_2}(\mathbf{x}_{1:j})$$

Task 2. Write down the inverse of this layer $\mathbf{x} = f^{-1}(\mathbf{y})$ for some $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$. Draw the inverse function $f^{-1}(\mathbf{y})$ as a computational graph that has $\mathbf{y}$ as input and $\mathbf{x}$ as output.

Task 3. Write down the Jacobian of this layer $\frac{d\mathbf{y}}{d\mathbf{x}}$. Do you recognize a special structure in this matrix?

Task 4. Write down the explicit expression for the *log*-determinant of the Jacobian matrix from the previous section.

Task 5. Write down the explicit expression for the *log*-determinant of the Jacobian matrix of the *inverse* function $\frac{d\mathbf{x}}{d\mathbf{y}}$.

In our implementation, we will realize the splitting of variables as masking. The variables to be transformed, $\mathbf{x}_{j+1:d}$, are masked when passing $\mathbf{x}$ to the networks to predict the transformation parameters $\mathbf{s}_{\theta_1}(\mathbf{x}_{1:j})$ and $\mathbf{t}_{\theta_2}(\mathbf{x}_{1:j})$. Also, afterward when applying the transformation (don't forget to exponentiate the scaling!), we mask the parameters for $\mathbf{x}_{1:j}$ so that we have an identity operation for those variables.

For predicting the shifting and scaling parameters for our toy datasets we will be using neural networks with 3 Fully Connected layers with LeakyReLU activations in between. Additionally, for stabilization purposes, we multiply the scaling output $\mathbf{s}_{\theta_1}(\mathbf{x}_{1:j})$ prior to exponentiation with a learnable parameter per dimension `scale_factor` initialized to 0. Meaning, our scaling is initialized to 1 as $\exp(0) = 1$. This prevents sudden large scaling values that can destabilize training (especially in the beginning).

The functions $\mathbf{s}_{\theta_1}(\cdot)$, $\mathbf{t}_{\theta_2}(\cdot)$, and `scale_factor` are already implemented in the provided class `CouplingLayer` below for your convinience.

Task 6. Implement missing `forward` and `inverse` methods in the class `CouplingLayer`:

- The `forward` method should take in `x` and use the mask `self.mask` and the learnable functions `self.s_func`, `self.scale_factor` and `self.t_func` to predict the transformation parameters and compute `y`. This method should also return the parameter `log_det_jac` which is the log-determinant of the Jacobian.

- The `inverse` method goes in the other direction. It takes in `y` and uses `self.mask`, `self.s_func`, `self.scale_factor` and `self.t_func` to compute `x`. This method should also return the parameter `inv_log_det_jac` which is the log-determinant of the Jacobian of $f^{-1}(\cdot)$.

In [ ]:

```
class CouplingLayer(nn.Module):
  def __init__(self, mask):
    super(CouplingLayer, self).__init__()

    # mask for splitting (fixed not learnable)
    self.mask = nn.Parameter(mask, requires_grad=False)

    # scaling function and stabilizing scale_factor init. to 0
    self.s_func = nn.Sequential(nn.Linear(in_features=2, out_features=32),
                                nn.LeakyReLU(),
                                nn.Linear(in_features=32, out_features=32),
                                nn.LeakyReLU(),
                                nn.Linear(in_features=32, out_features=2))
    self.scale_factor = nn.Parameter(torch.Tensor(2).fill_(0.0))

    # shifting function
    self.t_func = nn.Sequential(nn.Linear(in_features=2, out_features=32),
                                nn.LeakyReLU(),
                                nn.Linear(in_features=32, out_features=32),
                                nn.LeakyReLU(),
                                nn.Linear(in_features=32, out_features=2))

  def forward(self, x):
    """
    TODO: replace y and log_det_jac with your code.
    """
    y, log_det_jac = None, None
    return y, log_det_jac

  def inverse(self, y):
    """
    TODO: replace x and inv_log_det_jac with your code.
    """
    x, inv_log_det_jac = None, None
    return x, inv_log_det_jac
```

Coupling Flows

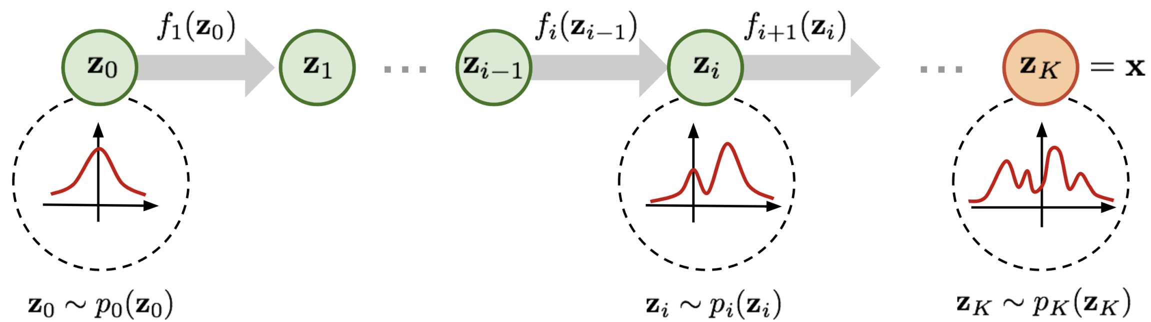
As you might have guessed by now, a coupling layer is powerful yet still limited in its ability to significantly alter the input. This is because it only operates on a chunk of it with element-wise manipulations due to the invertability constraint. We can go on with making our function f more complex. How can we implement more complex invertible functions? The answer is: invertible function composition. We can stack multiple invertible functions $f_1, \ldots, f_K$ (e.g. Coupling Layers) after each other, as all together, they still represent a single, invertible function. Specifically, if $\mathbf{y} = f_1(\mathbf{z})$ and $\mathbf{x} = f_2(\mathbf{y})$ are invertible functions, then $\mathbf{x} = f_2 \circ f_1(\mathbf{z})$ is an invertible function and its inverse is given by $f_1^{-1} \circ f_2^{-1}$. More importantly, the calculation of the log-determinant of the Jacobian in this case is simple using the chain rule.

Task 7. Assuming $\mathbf{y} = f_1(\mathbf{z})$ and $\mathbf{x} = f_2(\mathbf{y})$ are coupling layers, calculate the log-determinant of the Jacobian $\frac{d\mathbf{x}}{d\mathbf{z}}$. How is it related to the log-determinant of the Jacobians $\frac{d\mathbf{y}}{d\mathbf{z}}$ and $\frac{d\mathbf{x}}{d\mathbf{y}}$?

Coupling layers generalize to any masking technique we could think of. However, the most common approach is to split the input $\mathbf{x}$ in half using the mask. For our toy 2D datasets comprised of samples $\mathbf{x} = (p_x, p_y) \in \mathbb{R}^{d=2}$, this means that either $\{\mathbf{x}_{1:j} = p_x, \mathbf{x}_{j+1:d} = p_y\}$, or $\{\mathbf{x}_{1:j} = p_y, \mathbf{x}_{j+1:d} = p_x\}$. These correspond to masks $[1,0]^T$ and $[0,1]^T$ respectively. Note that when we apply multiple coupling layers, we invert the masking every other layer so that each variable is transformed a similar amount of times.

Intuition in 1D

Intuitively, using multiple, learnable invertible functions, a normalizing flow attempts to transform $p_z(z)$ slowly into a more complex distribution which should finally be $p_x(x)$. We visualize the idea below (figure credit - Lilian Weng):



Starting from z_0, which follows the prior Gaussian distribution, we sequentially apply the invertible functions $f_1, f_2, \dots, f_K$, until z_K represents x.

Implementation

Using the `CouplingLayer` class from above, here we will implement a class named `CouplingFlow` that is comprised of stacked coupling layers. This class will have the following attributes:

- `num_layers` - a scalar passed at initialization that will determine the number of coupling layers to stack, refered to later as K.
- `self.layers` - a Module list comprised of K stacked coupling layers each with its own mask. Note that the masks are fixed and non-learnable therefore we set their `requires_grad` property to `False` inside `CouplingLayer`.
- `self.prior` - This is the prior/base distribution. Here we will use a standard Gaussian distribution with a unit variance per dimension implemented using the `torch.distributions` module.

Task 8. Implement the following 3 methods of this class:

- `log_probability` - method that takes in a batch of samples `x` and returns `log_prob` which is their log-probability $\log p(\mathbf{x})$. For convinience we will assume the overall function $f = f_K \circ f_{K-1} \cdots \circ f_1$ satisfies $\mathbf{x} = f(\mathbf{z})$. Hence, to calculate the log-probability you should employ the inverse functions f_i^{-1} starting from the last layer f_K^{-1} (assuming K layers).
- `sample_x` - method that takes in a parameter `num_samples` and returns samples `x` from $p(\mathbf{x})$ alongside their log-probability values `log_prob`.
- `sample_x_each_step` - method takes in a parameter `num_samples` and returns a list of samples `samples` after each intermediate coupling layer in `self.layers`. The first element of this list is samples from the prior distribution $p(\mathbf{z})$ and the last element is samples from $p(\mathbf{x})$.

In [ ]:

```
class CouplingFlow(nn.Module):
  def __init__(self, num_layers):
      super(CouplingFlow, self).__init__()

      # concatenate coupling layers with alternating masks
      masks = F.one_hot(torch.tensor([i % 2 for i in range(num_layers)])).float()
      self.layers = nn.ModuleList([CouplingLayer(mask) for mask in masks])

      # define prior distribution to be z~N(0,I)
      self.prior = MultivariateNormal(torch.zeros(2),torch.eye(2))

  def log_probability(self, x):
    """
    TODO: replace log_prob with your code.
    """
    log_prob = None
    return log_prob
```

```
  def sample_x(self, num_samples):
    """
    TODO: replace x and log_prob with your code.
    """
    x, log_prob = None, None
    return x, log_prob

  def sample_x_each_step(self, num_samples):
    """
    TODO: replace samples with your code.
    """
    samples = None
    return samples
```

Training Coupling Flows

Now that we have finished implementing the flow model, we can start training it. Provided below is an already implemented function `train` to train your `CouplingFlow` models. The function recieves 4 arguments:

- `model` - an instance of the class `CouplingFlow`.
- `data` - an instance of the class `ToyDataset`.
- `epochs` - number of epochs to train the model (int).
- `batch_size` - the batch size to use in training (int).

In [ ]:

```
# detach tensor and transfer to numpy
def to_np(x):
  return x.detach().numpy()

# Simple training function
def train(model, data, epochs = 100, batch_size = 64):

  # move model into the device
  model = model.to(device)

  # split into training and validation, and create the loaders
  lengths = [int(len(data)*0.9), len(data) - int(len(data)*0.9)]
  train_set, valid_set = random_split(data, lengths)
  train_loader = DataLoader(train_set, batch_size=batch_size)
  valid_loader = DataLoader(valid_set, batch_size=batch_size)

  # define the optimizer and scheduler
  optimizer = Adam(model.parameters(), lr=1e-3)

  # train the model
  train_losses, valid_losses, min_valid_loss = [], [], np.Inf
  with tqdm.tqdm(range(epochs), unit=' Epoch') as tepoch:
    for epoch in tepoch:

      # training loop
      epoch_loss = 0
      model.train(True)
      for batch_index, training_sample in enumerate(train_loader):
          log_prob = model.log_probability(training_sample)
          loss = - log_prob.mean(0)
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()
          epoch_loss += loss
      epoch_loss /= len(train_loader)
      train_losses.append(np.copy(to_np(epoch_loss)))

      # validation loop
      epoch_loss_valid = 0
      model.train(False)
      for batch_index, valid_sample in enumerate(valid_loader):
        log_prob = model.log_probability(valid_sample)
        loss_valid = - log_prob.mean(0)
```

```
        epoch_loss_valid += loss_valid

      epoch_loss_valid /= len(valid_loader)
      valid_losses.append(np.copy(to_np(epoch_loss_valid)))

      # save best model based off validation loss
      if epoch_loss_valid < min_valid_loss:
        model_best = copy.deepcopy(model)
        min_valid_loss = epoch_loss_valid
        epoch_min = epoch

      # report progress with tqdm pbar
      tepoch.set_postfix(train_loss=to_np(epoch_loss), valid_loss=to_np(epoch_loss_valid))

  # report best model on val.
  print('\n Best Model achieved {:.4f} validation loss at epoch {} \n'.
        format(min_valid_loss, epoch_min))

  # if the number of samples is too low take the final weights regardless of
  # valdiation loss due to weak statistics (overfitting avoided by early stopping)
  if lengths[1] < 500:
    model_best = model

  return model_best, train_losses, valid_losses
```

Here is an example snippet for using this function to train a flow model with $K = 4$ layers on a dataset of $N = 1500$ samples $\mathbf{x}_i$ from the `'Moons'` distribution for 1000 epochs:

In [ ]:

```
# seeds to ensure reproducibility
torch.manual_seed(8)
np.random.seed(0)

# dataset
num_samples = 1500
data = ToyDataset('Moons', num_samples=num_samples)

# learning hyper-parameters
K = 4
nepochs = 1000

# instantiate model and optimize the parameters
Flow_model = CouplingFlow(num_layers=K)
moon_model, train_loss, valid_loss = train(Flow_model, data, epochs=nepochs)
```

Task 9. For each of the 4 provided distributions (use the default value of `num_gaussians=5` for `'GaussiansRot'`), use a similar snippet to repeat the following:

- Create a `ToyDataset` instance with $N = 1500$ samples from the distribution.
- Learn a `CouplingFlow` model with $K = 4$ layers, by training for 500 epochs. For the `'Moons'` dataset train for 1000 epochs.
- Plot the estimated density $p(\mathbf{x})$ in $\mathbb{R}^2$. To achieve this, use the method `log_probability` to calculate the log-probability $\log p(\mathbf{x})$ at a pre-determined grid of points $\mathbf{x} \in [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \subset \mathbb{R}^2$ (e.g. using `np.meshgrid`). Sample each coordinate with at least 100 points (i.e. a grid of 100×100 positions in 2D). Plot the resulting 2D distribution $p(\mathbf{x}) = \exp(\log p(\mathbf{x}))$ as an image where the value in each pixel is $p(\mathbf{x})$.
- Plot samples from intermediate flow layers, including the prior $p(\mathbf{z})$ and the modelled $p(\mathbf{x})$. To achieve this, use the method `sample_x_each_layer` with $N = 1000$ samples. Plot the resulting samples from each layer in the **same** axis limits to visualize the transformation of each coupling layer separately. You can use `plt.subplot(..., sharex=True, sharey=True)` to achieve link the axis of all subplots.

Implementation tips:

- Use the same seeds as the example snippet for reproducibility. This will also ensure a non-diverging erroneous behavior with other seeds.
- To make sure the model is not overfitting you can look at the training and the valdiation loss outputs of the provided function `train`. Do not include these in your report, use them just for sanity check.
- Estimate the log-probability in a reasonable vicinity of the training domain. For far away coordinates from the training data, the estimation could be poor locally and might bias the dynamic range of your plot.
- To avoid code duplication, you are encouraged to implement two plotting functions: one for plotting the density, and one for plotting the transformations across layers.

Analyzing Coupling Layers

Task 10. Train two flow models with a varying number of coupling layers $K = \{2, 4\}$ for 250 epochs, using $N = 1500$ samples from the `'GaussiansRot'` dataset with `num_gaussians=5`. Compare the resulting estimated density $p(\mathbf{x})$ (using a grid of 100×100 points) and the intermediate distributions of $N = 1000$ samples throughout the coupling layers of the model. Which model fits $p(\mathbf{x})$ better? What do you conclude regarding the effect of model depth? explain the result in your report and attach the resulting plots.

Task 11. Train two flow models with $K = 4$ layers for 400 epochs, using a varying number of $N = \{1500, 3000\}$ samples from the `'GaussiansRot'` dataset with `num_gaussians=5`. Compare the resulting estimated density $p(\mathbf{x})$ (using a grid of 100×100 points) and the intermediate distributions of $N = 1000$ samples throughout the coupling layers of the model. Which model fits $p(\mathbf{x})$ better? What do you conclude regarding the effect of training set size? explain the result in your report and attach the resulting plots.

Task 12. Train two flow models with $K = 4$ layers for 250 epochs, using $N = 1500$ samples from the `'GaussiansRot'` dataset with a varying number of Gaussians `num_gaussians` $= \{3, 7\}$. Compare the resulting estimated density $p(\mathbf{x})$ (using a grid of 100×100 points) and the intermediate distributions of $N = 1000$ samples throughout the coupling layers of the model. Which distribution $p(\mathbf{x})$ is fitted better by the model? What do you conclude regarding the effect of data complexity? explain the result in your report and attach the resulting plots.

Conclusion

In conclusion, we have seen how to implement our own normalizing flow on toy 2D datasets. However, as mentioned in the beginning of Part II, similar models with significantly more layers and additional tricks can be used to model images (e.g. Glow). The most common flow element, the coupling layer, is simple to implement, and yet effective. Normalizing flows are an interesting generative model compared to GANs, as they allow an exact likelihood estimate in continuous space, and we have the guarantee that every possible input x has a corresponding latent vector z. Recent advances in Neural ODEs allow a flow with infinite number of layers, called Continuous Normalizing Flows, whose potential is yet to fully explore.

References and Credits

[1] Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). “Density estimation using Real NVP,” In: 5th International Conference on Learning Representations, ICLR 2017. Link

[2] Kingma, D. P., and Dhariwal, P. (2018). “Glow: Generative Flow with Invertible 1x1 Convolutions,” In: Advances in Neural Information Processing Systems, vol. 31, pp. 10215--10224. Link

[3] University of Amsterdam, Deep Learning 1, Tutorial 11. Link

[4] Technical University of Munich, Machine Learning for Graphs and Sequential Data, Generative Models. Link

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')
```

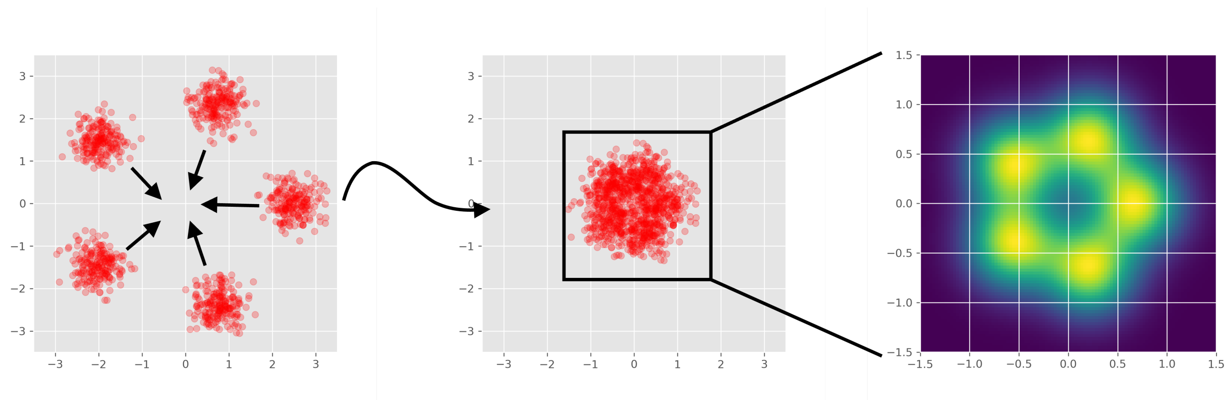
Importing relevant libraries for Part I

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
```

```
## 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.determinstic = 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 [ ]:

```
# 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, st
            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, pad
            else:
                res_block['conv_2'] = torch.nn.Conv2d(n_channels, n_channels, kernel_size=3, stride=1, pad
            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?

Statistical Methods in Image Processing EE-048954

Homework 3: Contrastive Divergence and Noise Contrastive Estimation

Due Date: June 16, 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_hw3_id1_id2.ipynb` which contains your code implementations.
  - A `.pdf` file, with the name: `ee048954_hw3_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

In [6]:

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

## Scipy optimization routines
from scipy.optimize import minimize

## 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: Contrastive Divergence (50 points)

Consider the following Gaussian Mixture Model (GMM) distribution

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

where $x, \mu_i \in \mathbb{R}^2$. We will use $N = 4$, $\sigma = 1$, and $\{\mu_i\} = \{(0,0)^T, (0,3)^T, (3,0)^T, (3,3)^T\}$.

Sampling from GMM

Task 1. Direct sampling: Use your function from HW1 that accepts $\{\mu_i\}$, and returns a sample x from $p(x;\{\mu_i\})$. Draw $J = 1000$ samples $\{x\}$ from the distribution $p(x;\{\mu_i\})$ using this function. These will be our **real samples**.

Task 2. Sampling with MCMC: implement the MALA algorithm to draw samples from $p(x;\{\mu_i\})$. The function will get an initial guess $\{\hat{x}_i\}$ and will generate chains of length L. Use $\sqrt{2\varepsilon} = 0.1$ and $N \sim \mathcal{N}(0, I)$.

From now on, we will refer to $\{\mu_i\}$ **as unknowns** and we will estimate them using different algorithms.

Estimation of $\{\mu_i\}$

Task 3. Implement Maximum likelihood (ML) estimation of $\{\mu_i\}$ using direct sampling:

- Step 1: Randomly initialize $\{\tilde{\mu}_i\}$ from $U([0,3]^2)$.
- Step 2: Use your function from Task 1 to draw 100 samples $\tilde{x}$ from $p(x;\{\mu_i\})$ using $\{\tilde{\mu}_i\}$.
- Step 3: Update $\{\tilde{\mu}_i\}$ using the ML gradient descent step:

  $$\tilde{\mu}_i^{k+1} = \tilde{\mu}_i^{k} + \eta\left(\langle\nabla_{\mu_i}\log p(x;\{\mu_i\})\rangle_x - \langle\nabla_{\mu_i}\log p(x;\{\mu_i\})\rangle_{\tilde{x}}\right),$$

  where $\langle\cdot\rangle_x$ denotes averaging over the real samples from Section A and $\langle\cdot\rangle_{\tilde{x}}$ denotes averaging over the synthetically generated samples from Step 2. Use $\eta = 1$.
- Repeat Step 2 and Step 3 until convergence.

Task 4. Implement Maximum likelihood (ML) estimation of $\{\mu_i\}$ using MCMC:

- Step 1: Randomly initialize $\{\tilde{\mu}_i\}$ from $U([0,3]^2)$.
- Step 2: Use your function from Task 2 to draw 100 samples $\tilde{x}$ from $p(x;\{\mu_i\})$ using $\{\tilde{\mu}_i\}$. Initialize the chains with $\hat{x}_i \sim \mathcal{N}(1.5, 2)$ and use chains length of $L = 1000$.
- Step 3: Update $\{\tilde{\mu}_i\}$ using the ML gradient descent step:

$$\tilde{\mu}_i^{k+1} = \tilde{\mu}_i^k + \eta \left(\langle \nabla_{\mu_i} \log p(x; \{\mu_i\}) \rangle_x - \langle \nabla_{\mu_i} \log p(x; \{\mu_i\}) \rangle_{\tilde{x}} \right) ,$$

where $\langle \cdot \rangle_x$ denotes averaging over the real samples from Section A and $\langle \cdot \rangle_{\tilde{x}}$ denotes averaging over the synthetically generated samples from Step 2. Use $\eta = 1$.

- Repeat Step 2 and Step 3 until convergence.

Task 5. Implement Contrastive Divergence (CD) estimation of $\{\mu_i\}$ using MCMC sampling:

- Step 1: Randomly initialize $\{\tilde{\mu}_i\}$ from $U([0,3]^2)$.
- Step 2: Use your function from Task 2 to draw 100 samples $\tilde{x}$ from $p(x; \{\mu_i\})$ using $\{\tilde{\mu}_i\}$. Initialize the chains with **100 samples randomly chosen from the real set of examples from Task 1**, and use only $L = 10$ update steps.
- Step 3: Update $\{\tilde{\mu}_i\}$ using the CD gradient descent step:

$$\tilde{\mu}_i^{k+1} = \tilde{\mu}_i^k + \eta \left(\langle \nabla_{\mu_i} \log p(x; \{\mu_i\}) \rangle_x - \langle \nabla_{\mu_i} \log p(x; \{\mu_i\}) \rangle_{\tilde{x}} \right) ,$$

where $\langle \cdot \rangle_x$ denotes averaging over the 100 real samples used for initialization of the chains in Step 2 and $\langle \cdot \rangle_{\tilde{x}}$ denotes averaging over the MCMC generated samples from Step 3. Use $\eta = 1$.

- Repeat Step 2 and Step 3 until convergence.

Task 6. Present the estimated $\{\mu_i\}$ and the final random samples $\{\tilde{x}_i\}$ generated with each of the three algorithms in Tasks 3-5. Discuss the differences in convergence.

Part II: Noise Contrastive Estimation (50 points)

Consider the distribution

$$p_m(x; \{\mu_i\}) = \frac{1}{Z} \sum_{i=1}^{N} \exp\left\{ -\frac{1}{2\sigma^2} ||x - \mu_i||^2 \right\},$$

where $Z \in \mathbb{R}$ is a normalization constant, and $x, \mu_i \in \mathbb{R}^2$.

Sampling from GMM

Task 7. What is the value of Z?

Task 8. Use $N = 4$, $\sigma = 1$, and $\{\mu_i\} = \{(0,0)^T, (0,3)^T, (3,0)^T, (3,3)^T\}$. Draw $J = 1000$ samples $\{x_j\}$ from the distribution $p_m(x; \{\mu_i\})$ using the function from Task 1.

From now on, we will refer to $\{\mu_i\}$ **as unknowns** and we will estimate them using the Noise Contrastive Estimation method.

Estimation of $\{\mu_i\}$

Task 9. Implement Noise Contrastive Estimation of $\{\mu_i\}$:

- Step 1: Generating the artificial data-set of noise: Draw $J = 1000$ samples $\{y_j\}$ from

$$p_n(y;\mu_n) = \frac{1}{2\pi\sigma_n^2}\exp\left\{-\frac{1}{2\sigma_n^2}||y-\mu_n||^2\right\}$$

  using $\mu_n = (1,1)^T$ and $\sigma_n = 2$.
- Step 2: Randomly select an initial guess for the model means $\{\tilde{\mu}_i\}$ from $U([0,3]^2)$.
- Step 3: Update $\{\tilde{\mu}_i\}$ by **maximizing**:

$$\{\tilde{\mu}_i\} = \arg\max_{\{\mu_i\}} \sum_{j=1}^{J} \left[\ln(h(x_j;\{\mu_i\})) + \ln(1-h(y_j;\{\mu_i\}))\right],$$

  where

$$h(u;\{\mu_i\}) = \frac{p_m(u;\{\mu_i\})}{p_m(u;\{\mu_i\}) + p_n(u;\mu_n)}.$$

  Implementation Tip: This step can be executed using the function `scipy.optimize.minimize` which finds the **minimum** of an (unconstrained) optimization problem (e.g. using the `'BFGS'` method), given a function that calculates the objective and an initial guess (see scipy documentation for more details). In our case, for **maximization**, implement a function that calculates the **minus** of the objective above.

We will now regard both $\{\mu_i\}$ **and the normalization constant** Z as unknowns, and will estimate them using Noise Contrastive Estimation.

Task 10. Implement Noise Contrastive Estimation with an **un-normalized** probability model:

- Step 1: Generating the artificial data-set of noise: Draw $J = 1000$ samples $\{y_j\}$ from

$$p_n(y;\mu_n) = \frac{1}{2\pi\sigma_n^2}\exp\left\{-\frac{1}{2\sigma_n^2}||y-\mu_n||^2\right\}$$

  using $\mu_n = (1,1)^T$ and $\sigma_n = 2$.
- Step 2: Randomly select an initial guess for the model means $\{\tilde{\mu}_i\}$ from $U([0,3]^2)$, and for the normalization constant Z from $U([0.1,1])$
- Step 3: Update $\{\tilde{\mu}_i\}$ and Z by **maximizing**:

$$\{\tilde{\mu}_i\}, Z = \arg\max_{\{\mu_i\},Z} \sum_{j=1}^{J} \left[\ln(h(x_j;\{\mu_i\},Z)) + \ln(1-h(y_j;\{\mu_i\},Z))\right],$$

  where

$$h(u;\{\mu_i\},Z) = \frac{p_m(u;\{\mu_i\},Z)}{p_m(u;\{\mu_i\},Z) + p_n(u;\mu_n)}.$$

Evaluating the Results

Task 11. Visually: plot the estimates of $\{\mu_i\}$ of Tasks 9 and 10 (two separate figures). Include

the model samples, the noise samples, the initial guess for the model means, and the final estimates of $\{\tilde{\mu}_i\}$.

Task 12. Quantitatively: repeat Tasks 9 and 10, this time with $J = 100 \times [1, 5, 10, 20, 30, 50]$. For each value of J repeat the estimation process for 50 times, each time with different realizations for $\{x_j\}$ and $\{y_j\}$ and initial guesses for the estimands ($\{\mu_i\}$ in Task 9 and $\{\mu_i\}, Z$ in Task 10.

For each value of J, calculate the MSE between the true parameter values and their estimates (the mean will be taken over the different realizations). Note that for the model means, the MSE should be calculated to the closest true μ_i for each estimation. If at the same run two estimated μ_is pick the same true μ_i, then this run should be declared as a failure and should be disregarded. Report the number of failure runs.

Task 13. Discussion: How does the number of samples J affect the accuracy of the estimation? How does the addition of Z as an unknown affect the accuracy?

Statistical Methods in Image Processing EE-048954

Homework 4: Score-Based Generative Modelling

Due Date: July 01, 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_hw4_id1_id2.ipynb` which contains your code implementations.
  - A `.pdf` file, with the name: `ee048954_hw4_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

In [ ]:

```
## Standard libraries
import numpy as np
import random

## KDE estimation
from sklearn.neighbors import KernelDensity

## Progress bar
import tqdm.notebook as 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')

## Pytorch
import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd

# device to be used (preferably a GPU, change Colab runtime type if needed)
device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print("Using device", device)
```

Introduction

Given a probablity density function $p(\mathbf{x})$, we define the *score* as $\nabla_{\mathbf{x}} \log p(\mathbf{x})$. As you might guess, score-based generative models are trained to estimate $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.

Unlike likelihood-based models such as normalizing flows from HW1, score-based models do not have to be normalized and are easier to parameterize. For example, recall the non-normalized statistical model from HW2 and HW3 $p_\theta(\mathbf{x}) = \frac{e^{-E_\theta(\mathbf{x})}}{Z_\theta}$, where $E_\theta(\mathbf{x}) \in \mathbb{R}$ is called the energy function and Z_θ is an unknown normalizing constant that makes $p_\theta(\mathbf{x})$ a proper probability density function. We saw that $E_\theta(\mathbf{x})$ can be parameterized by a flexible neural network, and computing the score $\nabla_\mathbf{x} \log p_\theta(\mathbf{x}) = -\nabla_\mathbf{x} E_\theta(\mathbf{x})$ didn't require computing the normalizing constant Z_θ. Similarly, in Score Matching, any neural network that maps an input vector $\mathbf{x} \in \mathbb{R}^d$ to an output vector $\mathbf{y} \in \mathbb{R}^d$ can be used as a score-based model, as long as the output and input have the same dimensionality. This yields huge flexibility in choosing model architectures.

Recently, variants of these models (a.k.a. Diffusion Models) have obtained high-quality samples comparable to/better than GANs (e.g. this paper) without requiring adversarial training, and are considered to be the current state-of-the-art. For the sake of simplicity, as usual, we will work with a toy example to enhance our understanding of the problems that arise when trying to learn and sample scores $\nabla_\mathbf{x} \log p(\mathbf{x})$.

Part I: Annealed Langevin Dynamics (40 points)

Consider the following Gaussian Mixture Model (GMM) distribution

$$p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\}) = \pi \mathcal{N}\left(\mathbf{x}; \mu_1, \sigma^2 I\right) + (1-\pi)\mathcal{N}\left(\mathbf{x}; \mu_2, \sigma^2 I\right),$$

where $\pi = \frac{1}{5}$, $\mathbf{x}, \mu_1, \mu_2 \in \mathbb{R}^2$ with $\mu_1 = (-5, -5)^T$, $\mu_2 = (5, 5)^T$, $\sigma^2 = 1$, and $I \in \mathbb{R}^{2\times 2}$ is the identity matrix. For your convenience, we provide below the class `GMMDist`.

In [ ]:

```
class GMMDist(object):
    def __init__(self):
        self.mix_probs = torch.tensor([0.2, 0.8])
        self.means = torch.stack([-torch.ones(2)*5, torch.ones(2)*5], dim=0)
        self.sigma = 1.

    def sample(self, n, sigma=1):
        """
        TODO: replace samples with your code
        """
        samples = None
        return samples

    def log_prob(self, samples, sigma=1):
        """
        TODO: replace logp with your code
        """
        logp = None
        return logp

    def score(self, samples, sigma=1):
        """
        TODO: replace grad_logp with your code
        """
        grad_logp = None
        return grad_logp
```

Task 1. How can we sample from $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ considering $\pi = 0.2$? Implement the missing method `sample` that accepts the number of samples `n` and the standard deviation `sigma` and returns a tensor of $n \times 2$ samples $\{\mathbf{x_i}\}$. Plot $J = 1000$ i.i.d. samples $\{\mathbf{x_i}\}$ from $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$.

Task 2. Write down the analytical expression for $\log p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ and implement the missing method `log_prob`. The method accepts as input two arguments: `samples` - the points at which we wish to evaluate the expression, `sigma` - the standard deviation, and returns an $n \times 1$ tensor `logp` (scalar per sample). Using this method, plot $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ at a 2D grid of 100×100 points in the domain $[-8, 8] \times [-8, 8]$.

Task 3. Write down the analytical expression for $\nabla_{\mathbf{x}} \log p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ and implement the missing method `score`. The method accepts as input two arguments: `samples` - the points at which we wish to evaluate the expression, `sigma` - the standard deviation, and returns an $n \times 2$ tensor `grad_logp` (2D vector per sample). Using this method, plot the vector field $\nabla_{\mathbf{x}} \log p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ at a 2D grid of 20×20 points in the domain $[-8, 8] \times [-8, 8]$ using a quiver plot.

Next, we would like to sample from $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ assuming we **only** have access to the scores $\nabla_{\mathbf{x}} \log p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$.

Task 4. Implement a function that samples $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ using Langevin dynamics:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \epsilon \nabla_{\mathbf{x}} \log p(\mathbf{x}^k) + \sqrt{2\epsilon}\mathbf{z}^k,$$

with $\mathbf{z}^k \sim \mathcal{N}(0, I)$. This function accepts four inputs:

- `score` - a function that accepts an $n \times 2$ tensor of `samples` and a standard deviation `sigma` and returns an $n \times 2$ tensor of `scores`.
- `init` - $n \times 2$ tensor of initial positions initialized uniformly in $[-8, 8] \times [-8, 8]$ (i.e. $\mathbf{u_i} \sim \mathcal{U}[-8, 8]^2$).
- `epsilon` - the step size of the Langevin update.
- `T` - the number of Langevin steps.

Show the resulting samples for $n = 1280, \epsilon = 0.05, T = 1000$. Discuss the results in terms of the relative mode weight. Do you notice anything strange? What is the reason for this behavior?

Let $\{\sigma_i\}_{i=1}^L$ be a positive geometric sequence that satisfies $\frac{\sigma_1}{\sigma_2} = \cdots = \frac{\sigma_{L-1}}{\sigma_L} > 1$. The **Annealed** Langevin dynamics algorithm, proposes to sample from $p(x)$ using L successive applications of Langevin dynamics with a twist: In the ith application of Langevin dynamics for T steps, we initialize the samples with the output of the $(i-1)^{\text{th}}$ run, update the step size according to $\epsilon_i = \epsilon \frac{\sigma_i^2}{\sigma_L^2}$, and use the scores of a noisy version of our original samples $\{\tilde{\mathbf{x}}\}$ with a probability density of $q_{\sigma_i}(\tilde{\mathbf{x}}) \triangleq \int p(\mathbf{x}) \mathcal{N}(\tilde{\mathbf{x}} | \mathbf{x}, \sigma_i^2 I) d\mathbf{x}$. The sequence $\{\sigma_i\}_{i=1}^L$ is chosen such that σ_1 is large enough to mitigate the difficulties encountered in Task 4, and σ_L is small enough to minimize the effect on the data. Note that for our Toy dataset adding Gaussian noise with σ_i will result in the same GMM distribution with a different effective standard deviation of $\tilde{\sigma}_i = \sqrt{\sigma^2 + \sigma_i^2}$. Therefore, for simplicity of implementation, we will consider the effective standard deviations $\{\tilde{\sigma}\}_{i=1}^L$ directly and implement the scores of the distribution $q_{\sigma_i}(\tilde{\mathbf{x}})$ using the class method `GMMDist.score()`. In our implementation, we will use a sequence of $L = 10$ effective standard deviations $\{\tilde{\sigma}\}_{i=1}^L$ spaced linearly in log-space between $\tilde{\sigma}_1 = 20$ and $\tilde{\sigma}_L = 1$. Meaning, the resulting sequence $\{\tilde{\sigma}_i\}_{i=1}^L$ can be implemented using `np.linspace()` like so: `sigmas=np.exp(np.linspace(np.log(20.),np.log(1.),10))`.

Task 5. Implement a function that samples $p(\mathbf{x}; \{\mu_1, \mu_2, \pi, \sigma^2\})$ using **Annealed** Langevin dynamics summarized as follows:

- Initialize with $\mathbf{x}^0 \sim \mathcal{U}[-8, 8]^2$.
- For $i = 1, 2, \ldots, L$:
  - Update the step size: $\epsilon_i = \epsilon \frac{\tilde{\sigma}_i^2}{\tilde{\sigma}_L^2}$
  - For $t = 1, 2, \ldots, T$:
    - Draw $\mathbf{z}^t \sim \mathcal{N}(0, I)$
    - $\mathbf{x}^t = \mathbf{x}^{t-1} + \epsilon \nabla_{\mathbf{x}} \log p(\mathbf{x}^{t-1}; \tilde{\sigma}_i^2) + \sqrt{2\epsilon}\mathbf{z}^t$
  - Update initialization $\mathbf{x}^0 = \mathbf{x}^T$

This function accepts five inputs:

- `score` - a function that accepts an $n \times 2$ tensor of `samples` and a standard deviation `sigma` and returns an $n \times 2$ tensor of `scores`.

- `init` - $n \times 2$ tensor of initial positions initialized uniformly in $[-8,8] \times [-8,8]$ (i.e. $\mathbf{u_i} \sim \mathcal{U}[-8,8]^2$).
- `sigmas` - $L \times 1$ tensor of effective standard deviations in a decreasing order (i.e. `sigmas[0]` is $\tilde{\sigma}_1$).
- `epsilon` - the step size of the Langevin update.
- `T` - the number of Langevin steps.

Show the resulting samples for $n = 1280, \epsilon = 0.05, T = 100$. Discuss the results in terms of the relative mode weight compared to Task 4. Plot the resulting samples for each annealing step $i = 1, \ldots, L$ (you're encouraged to create an animation), and explain the effect of using multiple standard deviations.

Task 6. To enforce your conclusions from Task 5, plot the estimated probability density function from Tasks 4 and 5 at a 2D grid of 100×100 points in the domain $[-8,8] \times [-8,8]$ and compare them to the result in Task 2. Your comparison should be performed both visually in 2D and along the line $x = y$ in 1D. To approximate the estimated density $\hat{p}(\mathbf{x})$ in Tasks 4 and 5, apply kernel density estimation to the respective samples using the function `sklearn.neighbors.KernelDensity()` with `bandwidth=0.5`.

Part II: Score Matching (60 points)

So far, we assumed we have access to the underlying ground truth scores. However, in reality we only have access to $i.i.d.$ samples $\mathbf{x}_i \sim p(\mathbf{x})$ from which we need to learn these scores. To learn the scores, we would like to minimize the Fisher Divergence between the score model $s_\theta(\mathbf{x})$ and the true data scores $\nabla_\mathbf{x} \log p(\mathbf{x})$:

$$\frac{1}{2}\mathbb{E}_{p(\mathbf{x})}\left[\|s_\theta(\mathbf{x}) - \nabla_\mathbf{x} \log p(\mathbf{x})\|_2^2\right]$$

However, we don't know the ground truth scores $\nabla_\mathbf{x} \log p(\mathbf{x})$, and therefore we can't compute the Fisher Divergence directly. Luckily, this objective can be shown equivalent to the following up to a constant:

$$\mathbb{E}_{p(\mathbf{x})}\left[\operatorname{tr}\left(\nabla_\mathbf{x} s_\theta(\mathbf{x})\right) + \frac{1}{2}\|s_\theta(\mathbf{x})\|_2^2\right]$$

where $\operatorname{tr}(\nabla_\mathbf{x} s_\theta(\mathbf{x}))$ is the trace of the Jacobian of $s_\theta(\mathbf{x})$. In deep networks calculating the trace of the Jacobian is very expensive computationally, and therefore there are two popular methods to circumvent its calculation. In class we saw one method for avoiding the costly Jacobian computation, called Denoising Score Matching (DSM). DSM is a variant of score matching that completely circumvents $\operatorname{tr}(\nabla_\mathbf{x} s_\theta(\mathbf{x}))$. It first perturbs the data point $\mathbf{x}$ with a pre-specified noise distribution $q_\sigma(\tilde{\mathbf{x}}|\mathbf{x})$ and then employs score matching to estimate the score of the perturbed data distribution $q_\sigma(\tilde{\mathbf{x}}) \triangleq \int p(\mathbf{x})\mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}, \sigma^2 I)d\mathbf{x}$. The DSM objective was proved equivalent to the following:

$$\mathbb{E}_{q_\sigma(\tilde{\mathbf{x}}|\mathbf{x})p(\mathbf{x})}\left[\|s_\theta(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \log q_\sigma(\tilde{\mathbf{x}}|\mathbf{x})\|_2^2\right],$$

where the optimal minimizer satisfies $s_{\theta^*}(\tilde{\mathbf{x}}) = \nabla_{\tilde{\mathbf{x}}} \log q_\sigma(\tilde{\mathbf{x}}) \approx \nabla_\mathbf{x} \log p(\mathbf{x})$, which is roughly true when the noise is small enough such that $q_\sigma(\tilde{\mathbf{x}}) \approx p(\mathbf{x})$.

Here we will see a different method, called Sliced Score Matching (SSM). SSM uses random projections to approximate $\operatorname{tr}(\nabla_\mathbf{x} s_\theta(\mathbf{x}))$. The objective is given by:

$$\mathbb{E}_{p(\mathbf{x})}\left[\mathbb{E}_{p(\mathbf{v})}\left[\mathbf{v}^T \nabla_\mathbf{x} s_\theta(\mathbf{x})\, \mathbf{v} + \frac{1}{2}\|s_\theta(\mathbf{x})\|_2^2\right]\right]$$

where $p_\mathbf{v}$ is a simple distribution of random vectors, e.g. the multivariate standard normal. Note that the term $\mathbf{v}^T \nabla_\mathbf{x} s_\theta(\mathbf{x})\, \mathbf{v}$ can be efficiently computed by forward mode auto-differentiation. Meaning, assuming we have a random vector `v`, and a score model `score` that accepts `x` as input and outputs the score `s_x` as output, we can compute the term $\nabla_\mathbf{x} s_\theta(\mathbf{x})\, \mathbf{v}$ using:

```
x.requires_grad_(True)
s_x = score(x)
grad_sx_v = autograd.grad(torch.sum(s_x*v), x, create_graph=True)[0]
```

Task 7. Implement a function that calculates the sliced score matching objective. Your function should accept three inputs:

- `score_net` - the score model that accepts samples and estimate the scores.
- `samples` - the current batch of $B \times 2$ training samples $\mathbf{x}_i$.
- `nv` - the number of random vectors to approximate $\mathbb{E}_{p(\mathbf{v})}$ using samples.

Tip: to enable forward mode auto-differentiation of the score model `score_net` w.r.t. the input samples `samples`, you need to set their `.requires_grad_()` property to `True`.

In [ ]:
```
def sliced_score_matching(score_net, samples, nv=1):
    """
    TODO: replace loss with your code
    """
    loss = None
    return loss
```

Now that we have an objective quantifying how good we can estimate the underlying scores, we can train a neural network. For the purpose of this exercise we will use a simple 3-layer fully connected network with the smooth "Softplus" activation function:

In [ ]:
```
score_net = nn.Sequential(nn.Linear(2, 128),
                          nn.Softplus(),
                          nn.Linear(128, 128),
                          nn.Softplus(),
                          nn.Linear(128, 2))
```

Task 8. Train a score model using the following training function with the default arguments:

In [ ]:
```
# training function: model, optimizer and loss
def train(score_net, batch_size=128, nsteps=10000, lr=0.001):

    # initialize the sampler, the optimizer and the loss list
    teacher = GMMDist()
    optimizer = optim.Adam(score_net.parameters(), lr=lr)
    losses = []

    # train until stagnation (can't overfit due to infinite data)
    tqdm_steps = tqdm.trange(nsteps)
    for step in tqdm_steps:

        # sample x_i~p(x) and calculate the slice-score-matching objective
        samples = teacher.sample(batch_size)
        loss = sliced_score_matching(score_net, samples, nv=1)

        # backprop to update the score model
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # print and save current loss
        losses.append(loss.detach().item())
        if step % 50 == 0:
            tqdm_steps.set_description('step: {}, loss: {:.5f}'.format(step, loss.item()))

    return losses, score_net
```

This function internally uses the class method `GMMDist.sample` from Task 1 as an infinite data sampler, and calculates the sliced score matching objective using the implemented function from Task 7. After training is done, the function outputs the loss progression during training `losses` and the optimized score model `score_net`. Plot the resulting loss curve for the first 1000 steps for sanity check. After training is done, use the resulting score model `score_net` to

estimate the scores at a 2D grid of 20×20 points in the domain $[-8, 8] \times [-8, 8]$. Plot the estimated scores as a vector field using a quiver plot, and compare the result to the real data scores from Task 3. Are the estimated scores uniformly accurate across the considered domain? If not, point out the most accurate areas, and explain the result.

Task 9. Now that we have a trained model estimating the score, we can sample data samples from $p(x)$ using Langevin dynamics as we did in Task 4. Repeat Task 4 with the estimated score model from Task 8, and plot the resulting samples. Compare the samples obtained using the estimated vs the real scores. In addition, compare the estimated probability distribution $\hat{p}\left(\mathbf{x}\right)$ at a 2D grid of 100×100 points in the domain $[-8, 8] \times [-8, 8]$ with the result of Task 2. To estimate $\hat{p}\left(\mathbf{x}\right)$ of the score model use kernel density estimation implemented with `sklearn.neighbors.KernelDensity()` with `bandwidth=0.5` on the samples from Langevin dynamics. Discuss the results.

As we saw earlier in Task 5, even with the perfect scores Langevin dynamics can have practical pitfalls when sampling our Toy distribution. One solution in such cases is using **annealing** with multiple standard deviations $\{\sigma_i\}_{i=1}^{L}$ to improve the results. However, that requires us to know the scores of all L distributions $q_{\sigma_i}(\tilde{\mathbf{x}}) = \int p(\mathbf{x})\mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}, \sigma_i^2 I)d\mathbf{x}$. While in theory we can learn L separate score models to approximate the desired sequence of scores $\nabla_{\tilde{\mathbf{x}}} \log q_{\sigma_i}(\tilde{\mathbf{x}})$, in practice this is too heavy and computationally inefficient. Therefore, To alleviate this issue, we can learn a single score model that is conditioned on the standard deviation σ_i^2 of the added noise to our samples, i.e. $s_\theta\left(\mathbf{x}, \sigma_i^2\right)$. As noted in this paper, a simple reparameterization of the conditional score model $s_\theta\left(\mathbf{x}, \sigma_i^2\right)$ into $\frac{s_\theta(\mathbf{x})}{\sigma_i}$ could suffice in practice. However, we still need to train this model with L noisy versions of our dataset $\{\tilde{\mathbf{x}}_j\}_i$, where the samples in the ith dataset are simulated according to $\tilde{\mathbf{x}}_j \sim \mathcal{N}\left(\tilde{\mathbf{x}}_j; \mathbf{x}_j, \sigma_i^2\right)$. This can be achieved by replacing our sliced score matching objective from Task 7 with an annealed version like so:

$$\mathcal{L}\left(\theta; \{\sigma_i\}_{i=1}^{L}\right) = \frac{1}{L}\sum_{i=1}^{L} \lambda\left(\sigma_i\right) \ell\left(\theta; \sigma_i\right),$$

where $\lambda\left(\sigma_i\right) > 0$ is a coefficient function depending on σ_i, and $\ell\left(\theta; \sigma_i\right) = \mathbb{E}_{p(\mathbf{x})}\left[\mathbb{E}_{p(\mathbf{v})}\left[\mathbf{v}^T \nabla_{\mathbf{x}} s_\theta\left(\mathbf{x}\right)\mathbf{v} + \frac{1}{2}\|s_\theta\left(\mathbf{x}\right)\|_2^2\right]\right]$ is the sliced score matching objective from Task 7. Specifically, in our implementation we will use $\lambda\left(\sigma\right) = \sigma^2$ which was shown to be a good choice to make the values of $\lambda\left(\sigma_i\right)\ell\left(\theta; \sigma_i\right)$ for all $\{\sigma_i\}_{i=1}^{L}$ roughly of the same order.

Task 10. Implement the **Annealed** version of the sliced score matching objective. Your function should accept four inputs:

- `cond_score_net` - the conditional score model that accepts `samples` and `used_sigmas`, and estimate the `scores`.
- `perturbed_samples` - the current batch of $B \times 2$ perturbed training samples $\tilde{\mathbf{x}}_i$.
- `used_sigmas` - $B \times 1$ vector indicating the used standard deviation `sigma` corresponding to each sample in the batch.
- `nv` - the number of random vectors to approximate $\mathbb{E}_{p(\mathbf{v})}$ using samples.

In [ ]:

```
def annealed_sliced_score_matching(cond_score_net, perturbed_samples, used_sigmas, nv=1):
    """
    TODO: replace loss with your code
    """
    loss = None
    return loss
```

Now that we have an annealed version of our sliced score matching objective, we can train a **noise-conditional** score model to estimate the desired sequence of scores $\{\nabla_{\tilde{\mathbf{x}}} \log q_{\sigma_i}(\tilde{\mathbf{x}})\}_{i=1}^{L}$. For the purpose of this excercise, we will use the following simple conditional score model (both taking σ_i as input and normalizing the output):

In [ ]:

```
# simple conditional score model
class CondScoreNet(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        self.cond_score_net = nn.Sequential(nn.Linear(3, 128),
                                            nn.Softplus(),
                                            nn.Linear(128, 128),
                                            nn.Softplus(),
                                            nn.Linear(128, 2))

    def forward(self, x, used_sigmas):
        x = torch.cat((x, used_sigmas), axis=1)
        return self.cond_score_net(x)/used_sigmas
cond_score_net = CondScoreNet()
```

Task 11. Train a **noise-conditional** score model using the following training function with the default arguments:

In [ ]:

```
# training function: model, optimizer and loss
def train_annealed(cond_score_net, sigmas, batch_size=256, nsteps=10000, lr=0.001):

    # initialize the sampler, the optimizer and the loss list
    teacher = GMMDist()
    optimizer = optim.Adam(cond_score_net.parameters(), lr=lr)
    losses = []

    # train until stagnation (can't overfit due to infinite data)
    tqdm_steps = tqdm.trange(nsteps)
    for step in tqdm_steps:

        # sample x_i~p(x)
        samples = teacher.sample(batch_size)

        # sample noise stds and perturb the samples accorindgly
        labels = torch.randint(0, len(sigmas), (batch_size,))
        used_sigmas = sigmas[labels].view(samples.shape[0], 1)
        perturbed_samples = samples + torch.randn_like(samples) * used_sigmas

        # calculate the annealed slice-score-matching objective
        loss = annealed_sliced_score_matching(cond_score_net, perturbed_samples, used_sigmas, nv=1)

        # backprop to update the score model
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # print and save current loss
        losses.append(loss.detach().item())
        if step % 50 == 0:
            tqdm_steps.set_description('step: {}, loss: {:.5f}'.format(step, loss.item()))

    return losses, cond_score_net
```

Use `sigmas=np.exp(np.linspace(np.log(10.),np.log(.1),10))` for the sequence of $L=10$ standard deviations $\{\sigma_i\}_{i=1}^{L}$. After training is done, the function outputs the loss progression during training `losses` and the optimized noise-conditional score model `cond_score_net`. Plot the resulting loss curve for the first 1000 steps for sanity check. Use the resulting score model `cond_score_net` to estimate the scores at a 2D grid of 20×20 points in the domain $[-8,8]\times[-8,8]$ for all noise levels. Plot the estimated scores as a vector field using a quiver plot (each in a separate plot), and discuss the extreme cases: σ_L and σ_1 vs the real data scores from Task 3 with an equivalent standard deviation (i.e. $\tilde{\sigma}_i=\sqrt{\sigma_i^2+\sigma^2}$). To guide your intuition, plot the respective $p\left(\mathbf{x};\tilde{\sigma}_i\right)$ at a 2D grid of 100×100 points in the domain $[-8,8]\times[-8,8]$ using the method from Task 2.

Task 12. Now that we have a noise-conditional score model approximating the sequence of scores $\{\nabla_{\tilde{\mathbf{x}}}\log q_{\sigma_i}(\tilde{\mathbf{x}})\}_{i=1}^{L}$ we can use the **Annealed** version of Langevin dynamics to produce samples from $p\left(\mathbf{x}\right)$. Repeat Task 5 using the estimated model from Task 11. Show the resulting samples for $n=1280,\epsilon=0.0005,T=100$. Discuss the

results in terms of the relative mode weight compared to Task 9. Plot the resulting samples for each annealing step $i = 1, \ldots, L$ (you're encouraged to create an animation), and explain the effect of using multiple standard deviations.

Conclusion

To conclude, in this HW we have seen the Annealed version of Langevin dynamics that can circumnvent some of the challenges when dealing with complex multi-modal distributions. We have also seen a popular approximation of the score matching objective in Sliced Score Matching, and implemented an annealed version of it to learn a noise-conditional score model. As mentioned in the beginning of this HW, similar models (much deeper + tricks) are currently leading to state-of-the-art performance in image synthesis, and recently also in solving linear inverse problems. For those of you interested in playing with such models, a lot of great resources are publicly available online, for example this tutorial by Yang Song who is the author of some key papers in this field.

References

[1] Song, Y., et al. "Generative modeling by estimating gradients of the data distribution." NeuroIPS (2019). Link

[2] Song, Y., et al. "Sliced score matching: A scalable approach to density and score estimation." UAI (2020). Link

[3] Song, Y., et al. "Improved techniques for training score-based generative models." NeuroIPS (2020). Link

[4] Song, Y., et al. "Score-based generative modeling through stochastic differential equations." ICLR (2021). Link