Statistical Methods in Image Processing EE-048954

Homework 1: Kernel Density Estimation and Normalizing Flows

Due Date: May 11, 2023

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 [ ]:
```
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 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, shuffl
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)
```

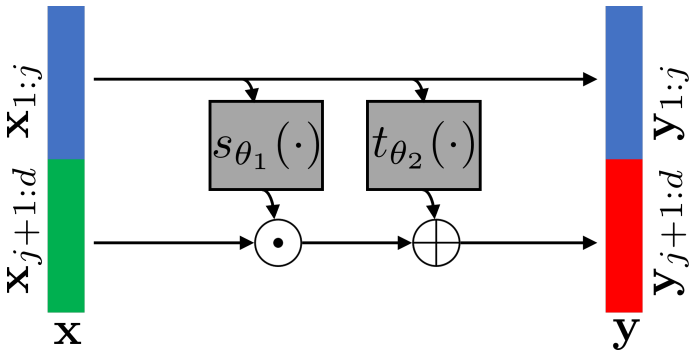
Coupling Layers

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

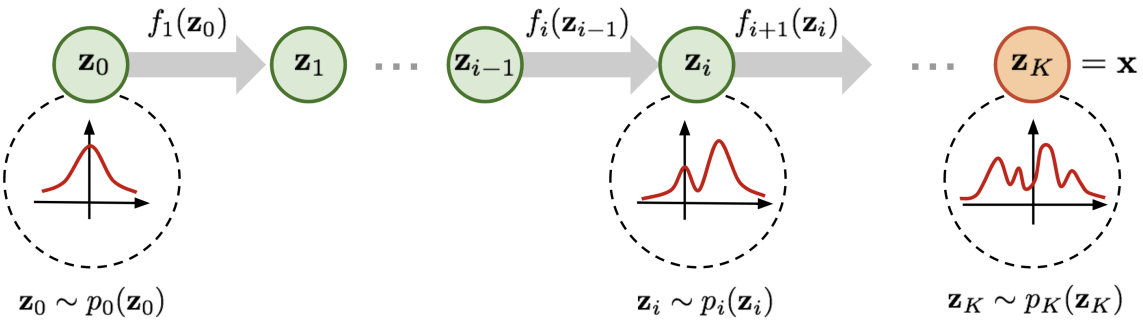
As you might have guessed by now, a coupling layer is powerful yet still limited in its ability to significantly alter the input. This is because it only operates on a chunk of it with element-wise manipulations due to the invertability constraint. We can go on with making our function f more complex. 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 $f_1, f_2, \ldots, f_K$, 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