# Assignment 3

## Machine Learning 2

Lea Bogensperger, lea.bogensperger@icg.tugraz.at
Benedikt Kantz, benedikt.kantz@student.tugraz.at

June 4, 2024

**Deadline:** June 25, 2024 at 23:55h.
**Submission:** Upload your report (`report.pdf`), your implementation (`main.py`) and your figure file (`figures.pdf`) to the TeachCenter. Please do not zip your files. Use the provided file containing the code framework for your implementation.

## Generative Modeling

In this assignment you will work with a generative model to estimate the density $p(\mathbf{x})$ of a data set where you have a access to a finite training data set $\{\mathbf{x}_1, \dots, \mathbf{x}_S\}_{s=1}^S$ with $\mathbf{x}_s \in \mathbb{R}^2$. To achieve this, you will use a popular technique called denoising score matching (DSM) to estimate the score (the gradient of the log) of the underlying data distribution using a small neural network. Then you can use Markov Chain Monte Carlo (MCMC) sampling to generate new data samples from the data distribution.

Note: for the training of your neural network, you can use `pytorch` in this example to ease the learning process using the provided automatic differentiation.

## 1 Data Set (2P)

Create a data set comprised of $K = 3$ Gaussians (i.e. a gaussian mixture model (GMM)) with weights $\pi_k = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. You should sample $S = 10000$ altogether, i.e. make sure you sample correctly from each Gaussian according to the given proportion. The parameters for the mean vectors are

$$\boldsymbol{\mu}_1 = \tfrac{1}{4}(1,1)^T, \quad \boldsymbol{\mu}_2 = \tfrac{1}{4}(3,1)^T, \quad \boldsymbol{\mu}_3 = \tfrac{1}{4}(2,3)^T,$$

and the covariance matrices are

$$\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma}_3 = \begin{pmatrix} 0.01 & 0.0 \\ 0.0 & 0.01 \end{pmatrix}$$

As a reminder, our GMM has the following form

$$p(\mathbf{x}) = \prod_{s=1}^{S} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{1}$$

where each individual multivariate Gaussian is naturally given by

$$\mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{2\pi |\boldsymbol{\Sigma}_k|^{1/2}} \exp\big( -\tfrac{1}{2}(\mathbf{x}_s - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_s - \boldsymbol{\mu}_k) \big).$$

**Tasks**

1. Generate the 2D data as given above and plot them in a 2D histogram using `matplotlib.pyplot.hist2d` with 128 bins.

## 2 Learning the Data Distribution with Score Matching (5P)

Ideally, you would directly try to learn the data distribution $p(\mathbf{x})$ using maximum likelihood learning, where you directly model the probability density function (pdf). To model this, we can use a function $f_\theta$ (a.k.a. neural network) parameterized by learnable parameters $\theta$ that constitute the pdf by

$$p_\theta(\mathbf{x}) = \frac{e^{-f_\theta(\mathbf{x})}}{Z_\theta}, \tag{2}$$

where $Z_\theta > 0$ ensures the proper normalization of the pdf such that it actually fulfills the properties required for a valid pdf. The function $f_\theta(\mathbf{x})$ is often referred to as **energy-based model**, as it assigns to a configuration $\mathbf{x}$ a scalar-valued energy quantity indicating how likely the respective state is. To put it into practical terms for our 2D points, we have

$$f_\theta : \mathbb{R}^2 \to \mathbb{R}. \tag{3}$$

In maximum likelihood, we would directly maximize the log. of (2), however, an alternative is to model the score function of a distribution

$$\nabla_{\mathbf{x}} \log p(\mathbf{x}), \tag{4}$$

by a model $s_\theta(\mathbf{x}) \approx \nabla_{\mathbf{x}} \log p(\mathbf{x})$. While direct modeling of this is not straightforward as we do not have access to $p(\mathbf{x})$ in practice, there are some options to approximate the score of the data distribution, one of them being DSM [1], which is illustrated in Figure 1. The idea is to perturb the data distribution with Gaussian noise $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ scaled by noise levels $\sigma_1 < \sigma_2 < \cdots < \sigma_L$, such that we obtain noisy data samples $\bar{\mathbf{x}}$

$$\bar{\mathbf{x}} = \mathbf{x} + \sigma_i \mathbf{z}, \quad \Rightarrow \text{this follows the distribution } p_{\sigma_i}(\bar{\mathbf{x}}|\mathbf{x})$$

i.e. we obtain $p_{\sigma_i}(\bar{\mathbf{x}}|\mathbf{x})$. The reason why this is actually helpful is due to Tweedie's formula [2], which gives us an estimate for the expectation of a denoised sample $\mathbf{x}$ given a noisy sample $\bar{\mathbf{x}}$ by

$$\mathbb{E}_{\mathbf{x}|\bar{\mathbf{x}}}[\mathbf{x}] = \bar{\mathbf{x}} + \sigma_i^2 \nabla_{\bar{\mathbf{x}}} \log p_{\sigma_i}(\bar{\mathbf{x}}|\mathbf{x}) \tag{5}$$

gehört zusammen

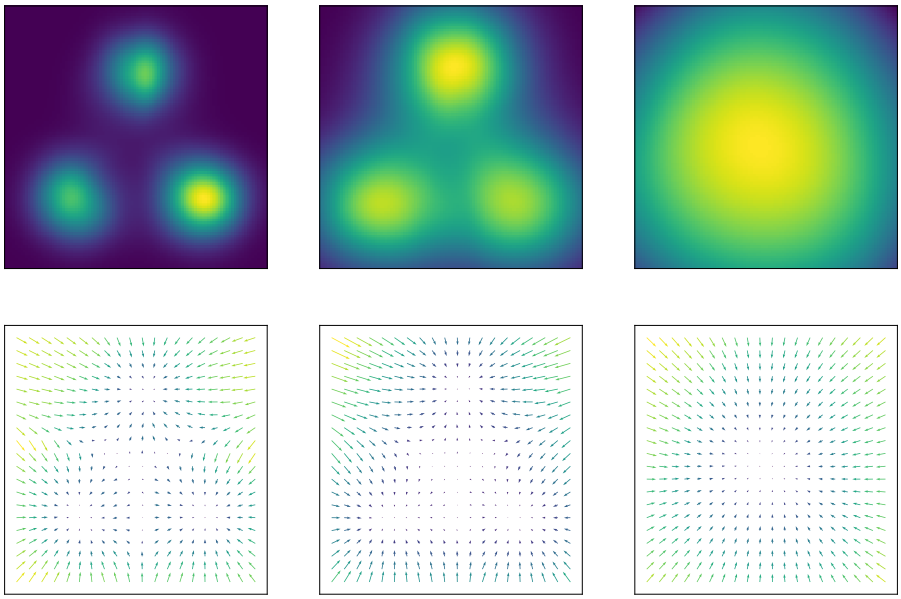

Figure 1: Idea of DSM. For small noise levels (left), the original data distribution is well approximated, but the scores bear little information in low-density regions. By using a higher noise level (center to right), the scores can be useful to yield a rough estimate of the directions of modes of the data distribution with the tradeoff of a blurred version of the data distribution.

Sinn vom score: je größer man den noise level macht, desto kleiner werden die low density areas kleiner, abere die vrteilung sieht eben auch immer blurrier aus, und nicht wie die originale verteilung

Using these insights, we can finally estimate the score of the perturbed data distribution

$$\arg\min_\theta \frac{1}{S} \sum_{s=1}^{S} \sigma_i^2 \mathbb{E}_{p_{\sigma_i}(\bar{\mathbf{x}}|\mathbf{x}_s)} \left[ \| \nabla_{\bar{\mathbf{x}}} \log p_{\sigma_i}(\bar{\mathbf{x}}|\mathbf{x}_s) - s_\theta(\bar{\mathbf{x}}, \sigma_i) \|_2^2 \right], \tag{6}$$

gehört zusammen

mit dem score berechnet man nicht den gradienten selbst, sondern nähert ihn durch neuronales netzwrt an(dass man trianiert um den gradienten anzunähern) -> vor allem hilfreich, weil die Berechnung bei höherdimensionalen daten immer reechnerisch aufwendiger wird

where *for each* data sample $\mathbf{x}_s$ we sample a random noise level $\sigma_i \in \{\sigma_1, \ldots, \sigma_L\}$.

Note that we therefore have to feed the additional information of the noise level $\sigma_i$ into the score network $s_\theta(\bar{\mathbf{x}}, \sigma_i)$ or the energy-based network $f_\theta(\bar{\mathbf{x}}, \sigma_i)$ , respectively, such that we can train the network to denoise our sample $\bar{\mathbf{x}}$ based on the current noise level $\sigma_i$.

**Tasks**

1. Try to think of a reason why direct optimization via maximum-likelihood learning in (2) might be difficult to achieve in practice.

2. Compute the score (the gradient of the log) as given in (4) of the sought pdf in (2). You should reach a form where the score depends only on the energy-based neural network $f_\theta(\mathbf{x})$. Relate the obtained result to your discussion in the previous point.

3. Rewrite (6) in its final form using Tweedie's estimator from (5) and your computed score in the form of the energy-based neural network $f_\theta$.

# 3 Denoising Score Matching in Practice (14P)

Your task is now to implement the above idea in practice. This means that you should implement a simple feedforward neural network composed of only linear layers and activation functions that are twice differentiable (e.g. you can use the exponential linear unit function `elu` here). Further, you have to ensure that you condition your network on the current noise level that was used to generate the noisy data, by stacking it with the 2D input point, i.e. $(\bar{\mathbf{x}}_s, \sigma_i)^T$. Thus, your network will map as follows (compare (3), where the noise level condition is not yet included):

-> can be implemented without the theory

$$f_\theta : \mathbb{R}^3 \to \mathbb{R}.$$

This network should then be trained such that you can estimate the scores of the noisy data distribution. Therefore, as we are using DSM, your task is to decide for a minimum and a maximum noise level, where it is important to ensure that $p_{\sigma_1}(\bar{\mathbf{x}}|\mathbf{x}) \approx p(\mathbf{x})$ and $p_{\sigma_L}(\bar{\mathbf{x}}|\mathbf{x}) \approx \mathcal{N}(\mathbf{0}, \mathbf{I})$

-> the higher the noise level, the closer you get to standard normal dist., rather than your actual distribution

**Tasks**

1. Experiment with different noise levels $\sigma_i$ to perturb your data distribution. Choose $\sigma_1$ and $\sigma_L$ and give an intuition on your choice. Choose a number of noise scales $L$ (state it) and interpolate all $L$ noise scales from $[\sigma_1, \sigma_L]$ geometrically (this function is provided in the code for you).

   Plot the perturbed data distributions $p_{\sigma_1}(\bar{\mathbf{x}}|\mathbf{x})$ and $p_{\sigma_L}(\bar{\mathbf{x}}|\mathbf{x})$ along with your original data samples again using `matplotlib.pyplot.hist2d`. -> similar to plots in tutorial 5

2. Decide for a very small and simple network architecture and implement it. State the number of learnable parameters of the neural network in your report. Why do your activation functions need to be twice differentiable?

3. Choose a number of iterations, a learning rate and select a suitable optimizer which is implemented readily in `pytorch` – state your choices in the report. Train your network by sampling random noise levels $\sigma_i$ for each data point in each iteration by minimizing (6). Plot the loss function.

4. Analytically compute the scores of the GMM and implement it in your code. Choose $\sigma_1$, $\sigma_L$ and an intermediate noise level and plot the density function and the scores for the 3 noise levels over the discretized data range in 2D. Use the magnitude of the scores to denote their color when plotting them (i.e. it should look similar to Figure 1). If you perturb a Gaussian with a noise level of $\sigma_i$, you can simply adapt its covariance matrix accordingly:

$$\begin{pmatrix} \Sigma_{11} + \sigma_i^2 & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} + \sigma_i^2 \end{pmatrix}$$

5. Additionally, check whether your learned energy-based model makes sense wrt. its analytic counterpart. Therefore, evaluate the energy for your discretized data range and compute the (unnormalized) density from it (state how to get the unnormalized density from the energy). Moreover, compute the scores. Compare it to the plot of the analytic density function and the scores. Discuss your findings and in what way they might differ.

**Implementation Details**

1. To obtain the gradient of a scalar-valued energy function such as $\nabla_{\mathbf{x}} f_\theta$ in `pytorch`, you can use `torch.autograd.grad` to rely on automatic differentiation.
2. See the provided toy example on how to update a neural network's parameters conveniently in `pytorch`.
3. The `Adam` optimizer could be a good choice for learning the network parameters.
4. To plot the density/scores, generate a 2D grid using `numpy.meshgrid` where you evaluate the quantities of interest for each point of the grid.
5. You can plot a vector field using `matplotlib.pyplot.quiver`. Use `numpy.hypot` to color the scores when creating a `quiver` plot.

# 4 Sampling from the Learned Distribution (4P)

Now that you have a trained score network, you can use it to sample from the data distribution with Langevin dynamics in the form of

$$\bar{\mathbf{x}}_t = \bar{\mathbf{x}}_{t-1} + \frac{\tau}{2} \nabla_{\mathbf{x}} \log p(\mathbf{x}_{t-1}) + \sqrt{\tau} \mathbf{z}_{t-1},$$

where $\tau \in \mathbb{R}$ is a small step size. Here is why we can equally use score matching to learn the score instead of learning the density $p(\mathbf{x})$ directly: for Langevin dynamics we only require access to the score. In order to deal with the different noise levels, we use annealed Langevin [1] as shown in Algorithm 1.

-> aus Inverse problems

**Algorithm 1:** Annealed Langevin sampling algorithm.

Set $\epsilon$, number of Langevin steps $T$ per noise level, use $\{\sigma_i\}_{i=1}^L$
Initialize $\bar{\mathbf{x}}_0$
**for** $i \leftarrow L, \dots, 1$ **do**
  $\alpha_i \leftarrow \varepsilon \cdot \sigma_i^2 / \sigma_1^2$
  **for** $t \leftarrow 1, \dots, T$ **do**
    $\mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\bar{\mathbf{x}}_t \leftarrow \bar{\mathbf{x}}_{t-1} - \frac{\alpha_i}{2} \nabla_{\mathbf{x}} f_\theta(\bar{\mathbf{x}}_{t-1}, \sigma_i) + \sqrt{\alpha_i} \mathbf{z}_t$
  **end**
  $\bar{\mathbf{x}}_0 = \bar{\mathbf{x}}_T$
**end**
Return $\bar{\mathbf{x}}_T$

**Tasks**

1. Choose the hyperparameters $\varepsilon$ and $T$ in Algorithm 1 and implement it.
2. Generate 5000 samples and plot a 2D histogram using `matplotlib.pyplot.hist2d` and compare it with the one from the original data samples.

# References


[1] Song, Y., & Ermon, S. (2019). Generative modeling by estimating gradients of the data distribution. Advances in neural information processing systems, 32.

[2] Efron, B. (2011). Tweedie's formula and selection bias. Journal of the American Statistical Association, 106(496), 1602-1614.