Leon Tiefenböck

Learning Probabilistic Circuits through sliced score matching

Bachelors's Thesis

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.-Prof. Dipl.-Ing. Dr. techn. Robert Peharz

Graz, October 2024

Abstract

Contents

1. Introduction

2. Background

2.1. Mathematical Notation

When talking about probability, random variables RVs are represented by large letters X where one instance of that RV is represented by small letters x. If x is a multidimensional vector then we will write it bold $\mathbf{x}$ where $x = \{x_1, x_2, ...x_n\}$. When talking about data, a whole dataset is denoted as $X = \{x_1, x_2.., x_n\}$ where N is the number of samples. Note that x_1, x_2 and so on can also be multidimensional. In that case they will be denoted as $\mathbf{x}_1, \mathbf{x}_2$ etc. and X will we be $\mathbf{X}$.

A desnity function of variable x will be denoted as $p(x)$. If the density is parameterized by a parameter θ we will write $p_\theta(x)$. If θ is a vector it will be written bold: $\boldsymbol{\theta}$. If we are talking about the density of some data, we will write $p_{data}(x)$ or simply $p_d(x)$.

2.2. Probabilistic Modelling

At its core, **Machine Learning (ML)** aims to develop algorithms or programs that can learn from data to make informed decisions, predict future outcomes, or perform various inference tasks. One powerful approach to this, especially when there is an inherent uncertainty in the data -which is the case for most real-world scenarios- is **Probabilistic Modelling**. In Probabilistic Modelling, we use probabilities to represent this uncertainty and employ the rules of probability theory to carry out inference tasks [1].

2.2.1. Probability Basics

To clarify these ideas let's say we have two random variables X and Y, that can take instances x and y. If we know how the probabilities are distributed over all possible pairs of outcomes (x, y), we know the *joint probability distribution* $p(x, y)$. For *discrete* random variables, this joint distribution is described by a *probability mass function* (PMF). For *continuous* random variables, it is described by a *probability* *density function* (PDF). Since we commonly deal with continuous variables in ML, we will focus on densities going forward.

For any function to be a valid or proper density it has to fulfill two properties. It has to be non-negative $p(x) \geq 0$ for all x and it has to be normalized so that the total area under the function equals 1, i.e. it has to integrate to 1 over the entire space: $\int_{-\infty}^{\infty} p(x)\,dx = 1$.

Having access to such a valid joint density function, allows us to perform basically all key tasks that we want to accomplish in machine learning. We can sample from the distribution to generate new datapoints, for example to generate images, we can predict by evaluating the density at a specific point and we can compute more complex inference tasks by using rules from probability theory, like for example the marginal probability (MAR) or the conditional probability (CON) [1].

2.2.2. Modelling a Density

In a real-world machine learning scenario, we typically don't have access to any density functions, intstead we are given some data. With probabilistic modelling we assume that the given data $\mathbf{X} = \{x_1, x_2.., x_n\}$ was generated by an unknown density function $p^*(x)$ (p-star). Finding this $p^*(x)$ can be seen as the holy grail in machine learning, because if it was known to us all inference tasks would reduce to simply applying the rules of probability theory [1].

However since we don't know $p^*(x)$, we have to do something different. We create a parameterized model $p_\theta(x)$ and using the data we tune the parameter θ so that $p_\theta(x)$ approximates $p^*(x)$ as closely as possible.

One of the most basic yet widely used methods for learning such a model is Maximum Likelihood Estimation (MLE) [2]. In principle, MLE is straightforward: we tune θ so that the likelihood of the observed data is maximized under the model. Specifically we *maximize the likelihood* (the output of the model) of each datapoint in $\mathbf{X}$ with respect to θ, as expressed in Equation 2.1:

$$\max_\theta \mathcal{L}(\theta) = \prod_{i=1}^{N} \log p_\theta(x_i) \tag{2.1}$$

Where $\mathcal{L}(\theta)$ is usually referred to as the *Likelihood Function.*

It is notworthy that instead of modelling $p_\theta(x)$, most of the time we want to model $\log p_\theta(x)$ because it typically is easier to work with. So instead of maximizing the Likelihood we maximize the Log-Likelihood, turning Equation 2.1 into the following:

$$\max_{\theta} \mathcal{L}(\theta) = \log \prod_{i=1}^{N} p_{\theta}(x_i) = \sum_{i=1}^{N} \log p_{\theta}(x_i) \tag{2.2}$$

Here $\mathcal{L}(\theta)$ is now referred to as the *Log-Likelihood Function*.

After learning a model using MLE we can again perform inference using the rules of probability.

2.2.3. Tractability vs. Expressiveness

Before continuing with different ways of constructing such a model, we want to introduce two key concepts that are often used to discuss these models.

Definition 1 (Expressiveness)**.** Expressiveness refers to the ability of a model to approximate different sets of distributions p^* to a high degree. If model A can approximate a set of distributions well, but model B can also model this set to the same degree plus additional distributions, then B would be more expressive than A. However normally this term is used as in expressive efficiency, so how complex a model must be to be expressive. For instance, if both models A and B can approximate the same set of distributions with equal effectiveness, but model B does so with significantly less complexity, then model B is regarded as more expressive (efficient). [1]

Definition 2 (Tractability)**.** A probabilistic model is called tractable with respect to an inference task, if the model can complete this task exactly and efficiently. Exactly in this case means without relying on approximation and efficiently means in linear time with respect to the model size. [1]

In recent years expressive capability has increased considerably through models based on neural networks like Generative-Adversarial Networks (GANs) [3], Variational Autoencoder (VAE) [4] and many others. However for what these models excel in generating very realistic images or compelling text, they lack in tractability for all except the simplest inference tasks [1].

On the other hand there are many, mostly older, less complex models like Gaussian Mixture Models (GMMs), Hidden Markos Models (HMMs) and so on, that lack in expressiveness and only work well enough for simple data but can compute most if not all inference tasks tractably.

2.3. Gaussian Mixture Model

The already mentioned Gaussian Mixture Model (GMM) [2] is a very foundational yet popular probabilistic model that will serve as good groundwork going forward, in fact a Probabilistic Circuit can fundamentally be seen as a *deep* (gaussian) mixture model.

The basic idea of a GMM is that a single Gaussian density can't model very many distributions, in fact a single Gaussian can only model, well a single Gaussian, but *mixing* many Gaussians together can produce a very capable density estimator.

In a GMM, this mixing refers to calculating a weighted sum of multiple Gaussian components, each representing a different Gaussian distribution. The probability density function of a multivariate GMM is given by Equation 2.3.

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{2.3}$$

In this equation, K is the number of Gaussian components in the mixture, and π_k is the mixture weight associated with the k-th component, where $\sum_{k=1}^{K} \pi_k = 1$, ensuring that the model ouputs a normalized density. $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ represents one single Gaussian component with mean $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$. This component is calculated using the well known multivariate gaussian density function, see Appendix Equation A.1. $\boldsymbol{\theta}$ refers to the parameter vector, where $\boldsymbol{\theta} = \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$

We call $\boldsymbol{\theta}$ the learnable parameters and K the hyperparameter of the model.

2.4. Probabilistic Circuits

Probabilistic Circuit (PC) [1] is an umbrella term for probabilistic models that can perform most inference tasks tractably but can be highly expressive at the same time. In general this is achieved by only introducing complexity in a structured manner. More specifically for a PC to be valid it has to adhere to certain structural properties, but more on that later.

Prominent members of the PC class include Cutset Networks [5], Probabilistic Sentential Decision Diagrams (PSDDs) [6] and others, but we will only be focusing on Sum Product Networks (SPNs) [7], which to our knowledge has seen the widest adoption and a lot of the times actually a SPN is meant when talking about PCs. However continuing on, we will only talk about PCs but often you could use PC and SPN interchangeably.

In General one can think of a PC as a structured neural network that consists of leaf nodes, sum nodes and product nodes. A PC then recursively computes weighted mixtures (sum nodes) and factorizations (product nodes) of simple input distributions (leaf nodes). These inputs can basically be any probability distribution like Gaussians, Bernoullies, Categroical distributions and so on, however most of the time we will talk about Gaussians [1].

Note that if the leaf distributions have normalized density functions and the weights of the PC are normalized correctly then the whole PC also models a normalized density [1].

Considering this, as already hinted at, one could notice that the simplest form of a PC is just a basic Gaussian Mixture Model, where one sum node mixes two leaf nodes, which graphically can be depicted as in Figure 2.1.

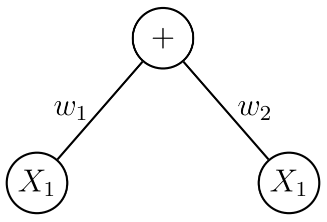


Figure 2.1.: Simplest PC - GMM

Furthermore more complex PCs can be created by introducing mixtures of mixtures and mixtures of mixtures of mixtures and so on, deepening the model stucture. An only slightly more complex PC with a product node and two sum nodes can be seen in Figure 2.2.

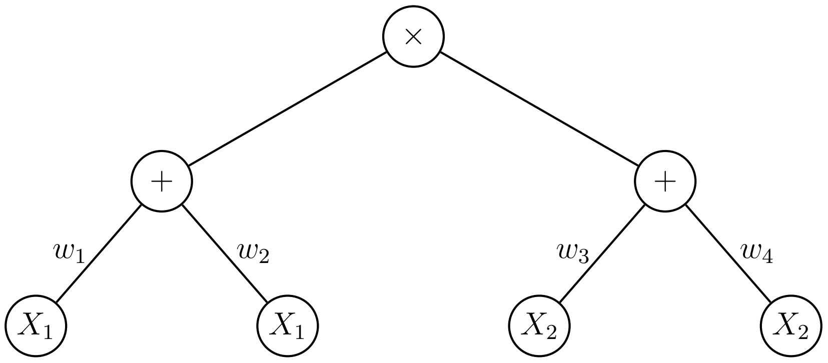


Figure 2.2.: Simple PC

Altough this is still very very basic it is a great starting point to introduce and make sense of two central structural properties that allow PCs to be expressive and tractable.

Definition 3 (Scope)**.** If a PC P models the joint distribution of a set of variables $\mathbf{X}$, each node of P models a distribution over a subset of $\mathbf{X}$. This subset is called the scope of the node. For a leaf node the scope is the input variable to that leaf, for all other nodes the scope is the union of its children's scopes. So the root node always has scope $\mathbf{X}$. [1]

Definition 4 (Smoothness)**.** A sum node is *smooth* if all its inputs have identical scopes. A PC is smooth if all its sum nodes are smooth. [1]

Definition 5 (Decomposability)**.** A product node is *decomposable* if all it's input scopes do not share variables. A PC is decomposable if all of its product nodes are decomposable. [1]

In simple terms this basically means that sum nodes are only allowed to have inputs over the same variable and product nodes are only allowed to have inputs over different variables. The two depicted PCs are smooth and decomposable.

There are more structural properties that a PC can fullfill, however these two already guarantee tractable computation of the central marginal (MAR) and conditional (CON) queries [1], which is already sufficient for many scenarios. Generally the more structure a PC has, which means the more structural properties it must fulfill, the more inference tasks it can compute tractably.

In [1] there is an in depth explanation, why these properties guarantee tractable inference and further discussion on other properties.

Using a PC in a real-world scenario then would bacisally entail first deciding on a structure (which properties and leaf distributions to use and how the nodes should be aranged in the graph) depending on the task at hand and then do a Maximum Likelihood Estimation, where θ is composed of the weights of the sum nodes and the parameters of the leaf distributions (e.g. means and variances for a Gaussian). This omptimization is usually done via Gradient Descent or the Expectation-Maximization (EM) algorithm. More details will follow later in Section 3.1.

2.5. Score Matching

Score Matching [8] is a concept that is normally used when dealing with unnormalized models like Energy Based Models (EBMs). Here Energy typically just refers to an unnormalized log-density $\log p(x)$. Learning these models through Maximum Likelihood Estimation can be difficult because of the computationally infeasible normalization constant, usually called Z_θ [8]. Recall that GMMs and PCs can represent normalized densities, nullifying this issue.

In Equation 2.4 the relation between a parameterized density p_θ and an Energy E_θ is expressed.

$$p_\theta(\mathbf{x}) = \frac{e^{-E_\theta(\mathbf{x})}}{Z_\theta} \tag{2.4}$$

Calculating the normalization constant would mean computing the integral over E_θ:

$$Z_\theta = \int e^{-E_\theta(\mathbf{x})}\, d\mathbf{x}$$

which is the source of the computational infeasability.

Score Matching proposes a workaround by working with the gradient log of the density, instead of $p_\theta(\mathbf{x})$, since calculating $\nabla_x \log p_\theta(\mathbf{x})$ removes Z_θ from the computation:

$$\nabla_x \log p_\theta(\mathbf{x}) = \nabla_x \log \; \frac{e^{-E_\theta(\mathbf{x})}}{Z_\theta} = \nabla_x \left(-E_\theta(\mathbf{x}) - Z_\theta\right) = -\nabla_x E_\theta(\mathbf{x})$$

$\nabla_x \log p_\theta(\mathbf{x})$ is called the score function, giving score matching it's name. Using the score which we will further refer to as s_θ the author of [8] proposes to minimize the Fisher Divergence between the scores of the data and the scores of the model:

$$\mathcal{L}(\theta) = \frac{1}{2}\mathbb{E}_{p_d}\left[\|s_\theta(x) - s_d(x)\|_2^2\right] \tag{2.5}$$

This doesn't depend on the intractable normalization constant, however it introduces another problem since it depends on the scores of the data $s_d(x)$, which is unknown. Furthermore it is shown that by partial integration the expression can be rewritten as [8]:

$$\mathcal{L}(\theta) = \mathbb{E}_{p_d}\left[\operatorname{tr}\left(\nabla_x s_\theta(x)\right) + \frac{1}{2}\left\|s_\theta(x)\right\|^2\right] \tag{2.6}$$

which is equivalent to 2.5 plus some additive constant. Here $tr()$ refers to the trace (sum of the diagonals) of the hessian matrix.

This final score matching objective doesn't depend on the score of the data s_d anymore. To use it to train a model on some data $X = \{x_1, x_2, x_n\}$ we can formulate the following unbiased estimator [8]:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(\operatorname{tr}\left(\nabla_{x_i} s_\theta(x_i)\right) + \frac{1}{2}\left\|s_\theta(x_i)\right\|^2\right) \tag{2.7}$$

2.6. Sliced Score Matching

While Score Matching get's rid of the normalization constant Z_θ, it introduces another term that can become hard to compute. To calculate the trace of the hessian in Equation 2.7 one derivation needs to be computed for each diagonal element of the hessian. This basically means that the number of derivations needed equals the dimension of the data. While this is fine for low dimensional data it quickly becomes unfeasible when learning higher dimensional data, for instance images.

In Sliced Score Matching (SSM) [9] the basic idea is to project the high dimensional data onto a random direction v to reduce the dimensionality and solve a lower dimensional problem. The number of random directions aka. slices can range from 1 upward, but most of the time 1 slice should suffice.

Applying this idea results in the following objective replacing Fisher Divergence from Equation 2.5 [9]:

$$\mathcal{L}(\theta; p_v) = \frac{1}{2}\mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\left(\mathbf{v}^T s_\theta(x) - \mathbf{v}^T s_d(x)\right)^2\right] \tag{2.8}$$

By using partial integration, similar to what was done in Score Matching, we can get rid of the unkown scores of the data s_d:

$$\mathcal{L}(\theta; p_{\mathbf{v}}) = \mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\mathbf{v}^T \nabla_{\mathbf{x}} s_\theta(\mathbf{x})\mathbf{v} + \frac{1}{2}\left(\mathbf{v}^T s_\theta(\mathbf{x})\right)^2\right] \tag{2.9}$$

which is again equivalent to 2.8 plus some additive constant [9].

This objective can be used, however it is worth to point out that when p_v is a multivariate standard normal or multivariate Rademacher distribution, $\mathbb{E}_{p_v}[(\mathbf{v}^T s_\theta(\mathbf{x}))^2] = \|s_\theta(\mathbf{x})\|_2^2$ in which case the second term of 2.9 can be integrated analytically [9], yielding the following objective.

$$\mathcal{L}(\theta; p_{\mathbf{v}}) = \mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\mathbf{v}^T \nabla_{\mathbf{x}} s_\theta(\mathbf{x})\mathbf{v} + \frac{1}{2}\left\|s_\theta(\mathbf{x})\right\|_2^2\right] \tag{2.10}$$

The authors of [9] refer to this objective as Sliced Score Matching with Reduced Variance (SSM-VR), which according to them produces better performance than the standard SSM objective from 2.9.

To again use this for training with data $X = \{x_1, x_2, x_n\}$ and M random vectors $\mathbf{v}_{ij}$ for each datapoint we can formulate an estimator [9]:

$$\mathcal{L}(\theta) = \frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(\mathbf{v}_{ij}^T \nabla_{\mathbf{x}_i} s_\theta(\mathbf{x}_i)\mathbf{v}_{ij} + \frac{1}{2}\left\| s_\theta(\mathbf{x}_i)\right\|_2^2\right) \tag{2.11}$$

3. Methods

In this thesis we try to address the concerns discussed in the introduction by training Probabilistic Circuits (PCs) through novel ways, specifically Score Matching (SM) and Sliced Score Matching (SSM). Then we compare results from these methods with results from convential methods to train PCs, specifically Maximum Likelihood Estimation (MLE) using both Gradient Descent (GD) and Expectation Maximization (EM).

3.1. Creating a simple PC

Recall from section 2.4 that the simplest variant of a PC computes the weighted sum of two input distributions. This is called a Mixture Model and the graph can be seen in Figure 2.1. If we expand this to a mixture of K components and use Gaussian distributions in the leaf nodes then we arrive at the Gaussian Mixture Model discussed in Section 2.3 with the modelled density function from Equation 2.3.

However when implementing this, we want to model the log-density $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ since it is easier to work with. So we take the log of Equation 2.3:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{3.1}$$

We also want to calculate a single weighted Gaussian component in log-space, so inside the sum, we take the exponential of the logarithm, which is valid since it would cancel out. See Appendix A.2 for the expression to calculate the log-density of a Gaussian.

$$\begin{aligned}\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \log \sum_{k=1}^{K} \exp\left(\log\left(\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)\right) \\ &= \log \sum_{k=1}^{K} \exp\left(\log(\pi_k) + \log\left(\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)\right)\end{aligned} \tag{3.2}$$

To compute this exactly as in 3.2, we encounter a challenge where the calculation of exponentials, can lead to instability due to overflow or underflow [10].
To address this, we use the **log-sum-exp (LSE)** trick:

$$\log \sum_{i=k}^{K} \exp(x_k) = C + \log \sum_{i=k}^{K} \exp(x_k - C)$$

Introducing this constant C, where usually $C = \max_k(x_k)$, mitigates the underflow and overflow potential [10].

Let $x_k = \log(\pi_k) + \log\left(\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)$ and let $C = \max_k \left(\log(\pi_k) + \log\left(\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)\right)$ we can rewrite 3.2 to arrive at a final numerically stable expression:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = C + \log \sum_{k=1}^{K} \exp\left(\log(\pi_k) + \log\left(\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right) - C\right) \tag{3.3}$$

We implemented this expression in python and used pytorch's [11] torch.nn.Parameter for all learnable parameters in $\boldsymbol{\theta}$ to use pytorch's optimization framework later.

We also used torch.distributions.MultivariateNormal to compute the log density of a single gaussian component $\mathcal{N}(\mathbf{x}_s|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ and torch.logsumexp for the LSE operation.

Note that for readability we will further refer to the whole density function of the GMM from 3.3 as just $\log p_{\boldsymbol{\theta}}(\mathbf{x})$.

3.1.1. Maximum Likelihood Estimation

Now to train this GMM on some data $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2.., \mathbf{x}_n\}$ in the convential way, we can calculate $\boldsymbol{\theta}$ using the log Maximum Likelihood Estimation (MLE) objective from Equation 2.2 in the Background chapter and include the density function of the GMM from Equation 3.3:

$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i) \tag{3.4}$$

In plain text this means maximizing $\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x})$ with respect to $\boldsymbol{\theta}$, which consists of the weights $\boldsymbol{\pi}$, the means $\boldsymbol{\mu}$ and the covariance matrices $\boldsymbol{\Sigma}$.

Now with the objective formulated we will introduce Gradient Descent (GD) and Expectation Maximization (EM), which are learning algorithms that provide a concrete way on how the best possible $\boldsymbol{\theta}$ can be found.

Gradient Descent

Gradient Descent (GD) is an iterative optimization algorithm used to minimize a given objective function, by computing the gradients of the function with respect to its parameters and updating them iteratively [12].

Recall that however when training a GMM, the goal is to maximize the log-likelihood function defined in Equation 3.4. So to do this using GD, we instead **minimize** the **negative** log-likelihood (NLL), which is equivalent.

The main steps in applying GD to our GMM objective are:

1. **Computing the gradients** of the negative Log-Likelihood function $\mathcal{L}(\boldsymbol{\theta})$ for each parameter in $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}(-\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}))$$

2. **Updating the parameters** using the computed gradients multiplied by a learning rate η:

$$\theta^{(t+1)} = \theta^{(t)} + \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$$

We repeat this for T training iterations, also often called Epochs.
Note that for the first iteration the prameters $\boldsymbol{\theta}$ have to be initialized in some manner, e.g. randomly.

Algorithm 1 Gradient Descent

```
Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, $\eta$, $T$
1: for $t = 1$ to $T$ do
2:     $\mathcal{L}(\boldsymbol{\theta}) \leftarrow -\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i)$
3:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
4: end for
5: return $\boldsymbol{\theta}$
```

We implemeted this using pytorch's [11] autograd capabilities to compute the gradients.

Expectation Maximization

Expectation Maximization (EM) is another method used for iterative optimization, especially popular with Gaussian Mixture Models. EM achieves this by alternating between the **Expectation (E) step** and the **Maximization (M) step** [2].

1. **Expectation Step (E-Step):** In the E-step, we compute the posterior probability (also called responsibilities) that a given data point $\mathbf{x}_i$ belongs to each Gaussian component k:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

2. **Maximization Step (M-Step):** In the M-step, we update the parameters π_k, $\boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}_k$ based on the posterior probabilities from the E-step:

$$\pi_k = \frac{1}{N}\sum_{i=1}^{N} \gamma_{ik}, \quad \boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i}{\sum_{i=1}^{N} \gamma_{ik}}, \quad \boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_{ik}(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T}{\sum_{i=1}^{N} \gamma_{ik}}$$

We again repeat this for T training iterations, and initialize $\boldsymbol{\theta}$ for the first one.

Algorithm 2 Expectation Maximization

```
Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, $T$
 1: for $t = 1$ to $T$ do
 2:     for $i = 1$ to $N$ do
 3:         for $k = 1$ to $K$ do
 4:             $\gamma_{ik} \leftarrow \frac{\pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)}$
 5:         end for
 6:     end for
 7:     for $k = 1$ to $K$ do
 8:         $\pi_k \leftarrow \frac{1}{N}\sum_{i=1}^{N} \gamma_{ik}$
 9:         $\boldsymbol{\mu}_k \leftarrow \frac{\sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i}{\sum_{i=1}^{N} \gamma_{ik}}$
10:         $\boldsymbol{\Sigma}_k \leftarrow \frac{\sum_{i=1}^{N} \gamma_{ik}(\mathbf{x}_i-\boldsymbol{\mu}_k)(\mathbf{x}_i-\boldsymbol{\mu}_k)^T}{\sum_{i=1}^{N} \gamma_{ik}}$
11:     end for
12: end for
13: return $\boldsymbol{\theta}$
```

3.1.2. Score Matching

Training the GMM with Score Matching is actually not very different from the MLE with Gradient Descent from above, we just need to switch the objective function.

Exact Score Matching

We take the Exact Score Matching objective function 2.7 from Section 2.5 and formulate the optimization problem:

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left(\operatorname{tr}\left(\nabla_{\mathbf{x}_i} s_{\boldsymbol{\theta}}(\mathbf{x}_i) \right) + \frac{1}{2} \left\| s_{\boldsymbol{\theta}}(\mathbf{x}_i) \right\|^2 \right) \tag{3.5}$$

The we optimze this objective using Gradient Descent for T Training Iteration and Learning Rate η.

Algorithm 3 Score Matching

```
Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, $\eta$, $T$
1: for $t = 1$ to $T$ do
2:     $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N} \sum_{i=1}^{N}$ ScoreMatchingLoss($\mathbf{x}_i$)
3:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$
4: end for
5: return $\boldsymbol{\theta}$

Procedure: ScoreMatchingLoss($\mathbf{x}$)
1: $s_{\boldsymbol{\theta}}(\mathbf{x}) \leftarrow \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$
2: $\mathcal{L}(\boldsymbol{\theta}) \leftarrow tr(\nabla_{\mathbf{x}} s_{\boldsymbol{\theta}}(\mathbf{x})) + \frac{1}{2} \left\| s_{\boldsymbol{\theta}}(\mathbf{x}) \right\|^2$
3: return $\mathcal{L}(\boldsymbol{\theta})$
```

Sliced Score Matching

To use Sliced instead of Exact Score Matching we can do everything in the same way as above, we only need to replace the objective function with the Sliced Score Matching objective 2.11 from Section 2.6.
Note that for simplicity we only use one random vector v_i (one slice) for each datapoint.

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{v}_i^T \nabla_{\mathbf{x}_i} s_{\boldsymbol{\theta}}(\mathbf{x}_i) \mathbf{v}_i + \frac{1}{2} \left\| s_{\boldsymbol{\theta}}(\mathbf{x}_i) \right\|_2^2 \right) \tag{3.6}$$

Algorithm 4 Sliced Score Matching

```
Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, $\eta$, $T$
1: for $t = 1$ to $T$ do
2:     $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N}\sum_{i=1}^{N} \textsc{SlicedScoreMatchingLoss}(\mathbf{x}_i)$
3:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
4: end for
5: return $\boldsymbol{\theta}$
```

```
Procedure: $\textsc{SlicedScoreMatchingLoss}(\mathbf{x})$
1: $\mathbf{v} \leftarrow \mathcal{N}(0,1)$
2: $s_{\boldsymbol{\theta}}(\mathbf{x}) \leftarrow \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$
3: $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \mathbf{v}^T \nabla_{\mathbf{x}_i} s_{\boldsymbol{\theta}}(\mathbf{x}_i)\mathbf{v} + \frac{1}{2}\left\|s_{\boldsymbol{\theta}}(\mathbf{x}_i)\right\|_2^2$
4: return $\mathcal{L}(\boldsymbol{\theta})$
```

We implented both Algorithms again using PyTorch [11] and troch.autograd.

3.1.3. Sampling

Once the Gaussian Mixture Model (GMM) is trained, we can generate new samples from the learned distribution. Sampling from a GMM involves two steps:

1. **Component Selection**: First, we sample a component index k from a categorical distribution with the mixture weights $\boldsymbol{\pi}$ as probabilities.

2. **Gaussian Sampling**: After selecting a component k, we sample from the corresponding Gaussian with parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$.

Algorithm 5 GMM Sampling

```
Input: $N$, $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$
1: samples $\leftarrow 0$
2: for $i = 1$ to $N$ do
3:     $k \leftarrow \textsc{Categorical}(\boldsymbol{\pi})$
4:     samples $\leftarrow$ samples $+ \mathcal{N}(\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$
5: end for
6: return samples
```

We implemented this in PyTorch [11], using `torch.distributions.MultivariateNormal` for generating samples, and `torch.Categorical` for component selection.

3.2. Using more complex PCs

While the model proposed in Section 3.1 works well for relatively simple tasks like 2 dimensional density estimation it doesn't perform very well with complex higher dimensional data, e.g. images.

So to get a wider variety of results we decided to also use a state-of-the-art framework for Probabilistic Circuits called EinsumNetworks [13] for modelling more complex higher dimensional datasets.

For more details on how EinsumNetworks work please refer to [13] but in general the same principles as in our GMM apply but due to the deeper structure the model is much more complex and thus much larger.

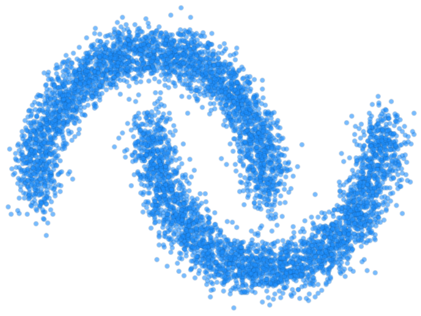
However implementing the training of an EinsumNetwork with our targeted algorithms is basically the same as in the GMM. Note that training with EM already comes with the framework, so we will leave this out.
For all other algorithms, we can apply them in the exact same way as described in Section 3.1 above, only needing to switch out the calculation of $\log p_{\boldsymbol{\theta}(\mathbf{x})}$ to the log-density function the EinsumNetwork framework provides.

4. Experimental Results

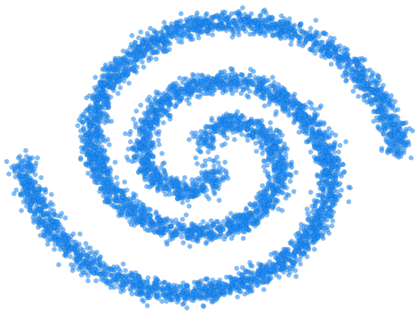
4.1. 2D Density Esitmation

To test the simple model from Section 3.1 with the discussed algorithms, so Expectation Maximization (EM), Gradient Descent (GD), Score Matching (SM) and Sliced Score Matching (SSM) from subsections 3.1.1 to 3.1.2, we used some two dimensional data to perform density estimation.

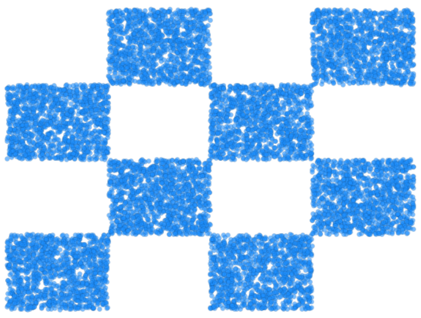
Samples from the three datasets we used can be seen in Figure 4.1.



(a) halfmoons



(b) spirals



(c) board

Figure 4.1.: Samples for all three datasets

4.1.1. Experiment 1

In the basic first set of experiments we wanted to see the best possible results for each algorithm on each dataset.

For each dataset we generated 20,000 datapoints and split them evenly for training and validation. However before the training process can start, recall that our model is governed by the learnable parameters $\boldsymbol{\pi}$ (mixture weights), $\boldsymbol{\mu}$ (means of components) and $\boldsymbol{\Sigma}$ (covariance matrices of components) that need to be initialized in some manner and the hyperparameter K (mixture count) that needs to be chosen.

As for the learnable parameters, we initialized the mixture weights $\boldsymbol{\pi}$ uniformly

$$\boldsymbol{\pi}_k = \frac{1}{K}, \quad k = 1, 2, \ldots, K$$

the covariance matrices $\boldsymbol{\Sigma}$ as identy matrices of size $D \times D$

$$\boldsymbol{\Sigma}_k = \mathbf{I}_D, \quad k = 1, 2, \ldots, K$$

where D is the dimensionality of the data and the means $\boldsymbol{\mu}$ by computing cluster centers of the data with the sklearn [14] implementation of KMeans.

As for K, trough some intitial testing we chose three different values, namely a minimal K, that is needed for producing reasonable results, a very large K from which onward there are dimishing returns and a moderate K in the middle between the minimal and the large K.

Now for the training process also recall that it is governed by the hyperparameters epochs (number of training iterations) for all algorithms and a learning rate (step-size for Gradient Descent) for all the gradient-based algorithms (GD, SM, SSM). To choose these parameters we fixed K to one of the three chosen values and did cross-validation by computing the validation set Log-Likelihood of the models with all possible remaining hyperparameter combinations and choosing the combination with the highest Log-Likelihood.

Results, more specifically Log-Likelihood, estimated densities and samples for all datasets and all values of K with the best possible training-specific hyperparameters can be seen in the following three pages. Note that when setting the same random seed all of these results should be reproducable.

(I noticed that with higher Ks 40 and up SM and SSM didtn consistenly provide same results, maube becaus of autograd? EM and GD had always exact results .. not sure if i should include this)

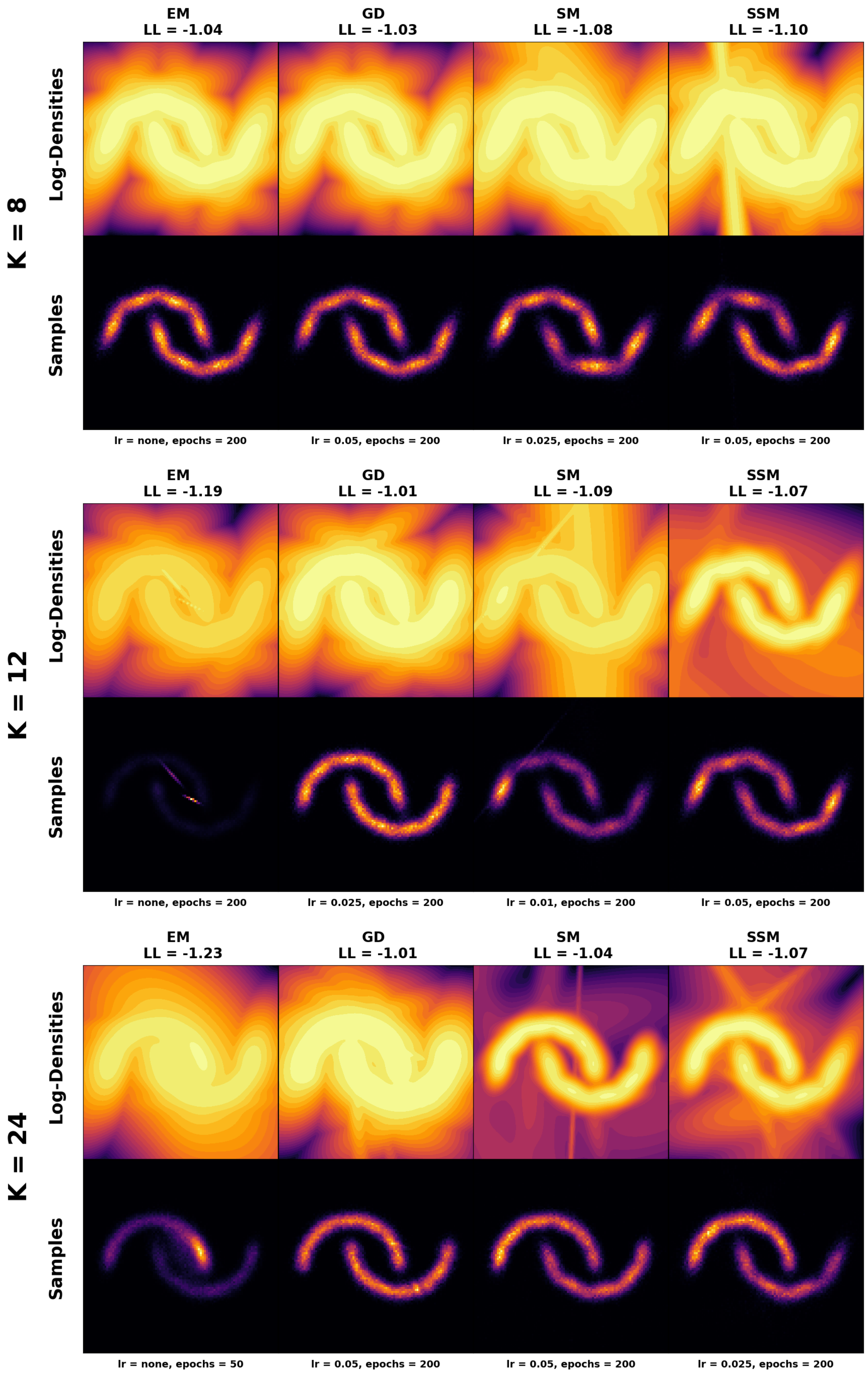


Figure 4.2.: Densities and Samples for the halfmoon dataset

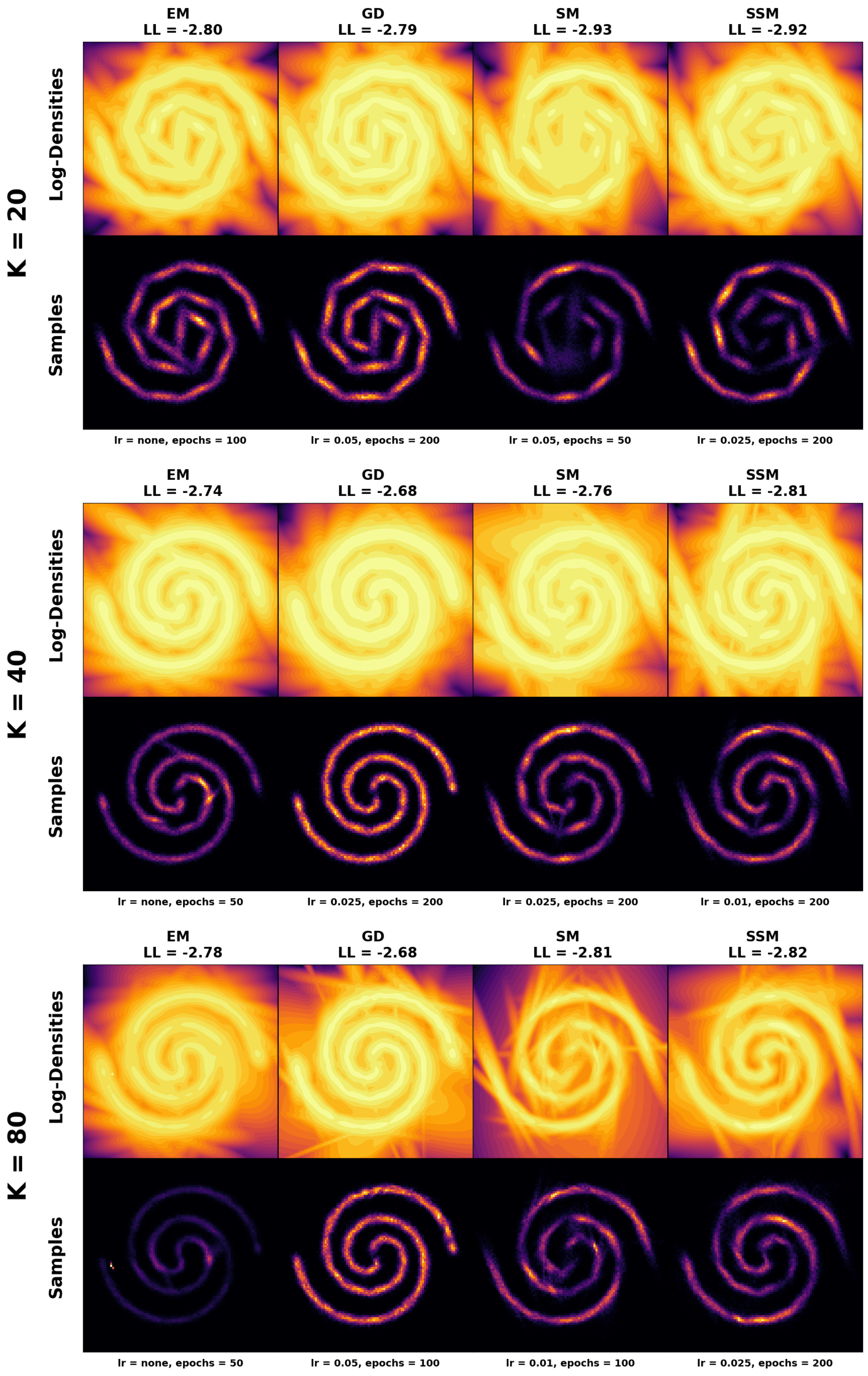


Figure 4.3.: Densities and Samples for the spirals dataset

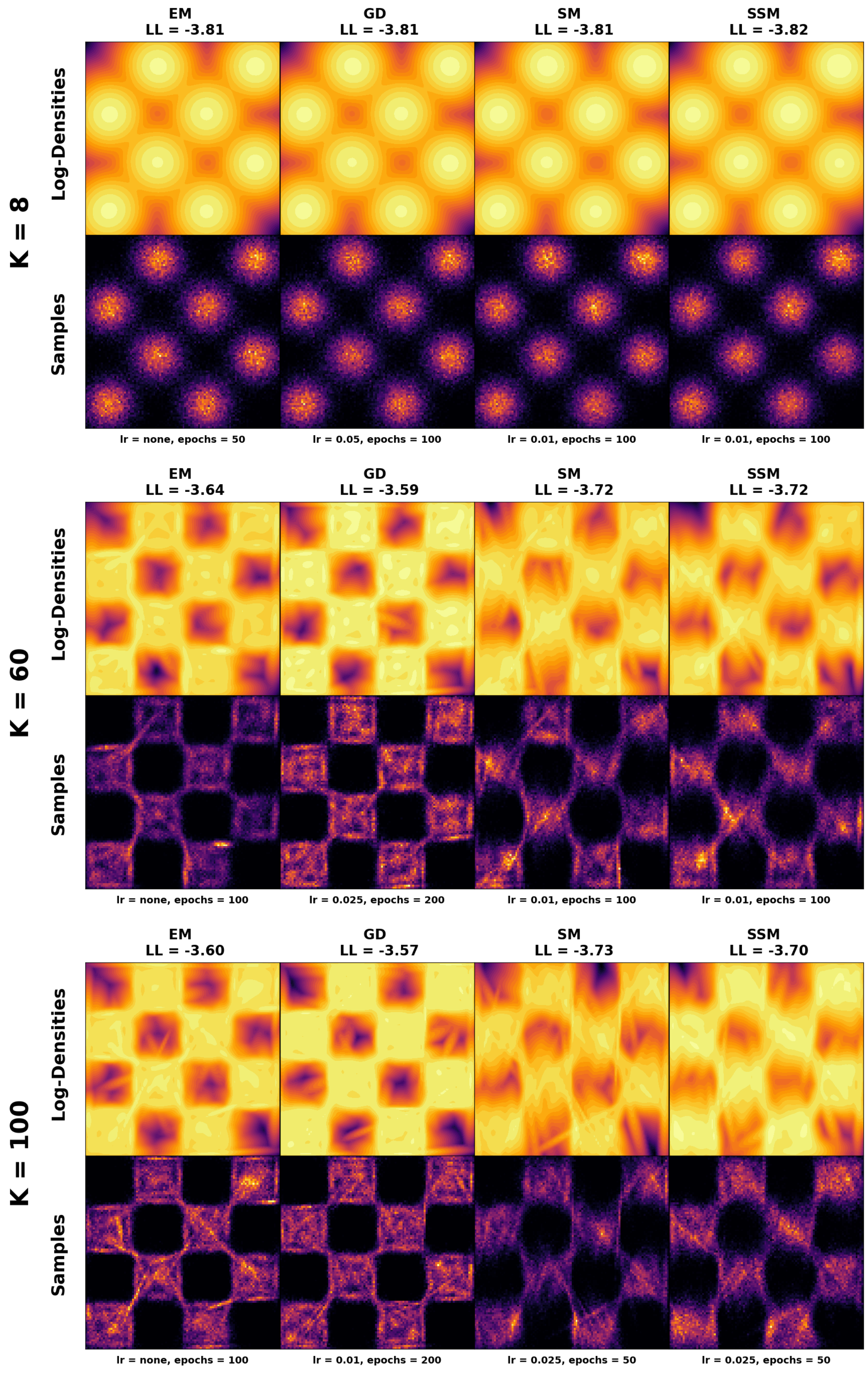


Figure 4.4.: Densities and Samples for the board dataset

4.1.2. Experiment 2

To gain further insight in how the learning process differs for each algorithm we chose a dataset and a set of hyperparameters where all algorithms perform somewhat similar and analized the Negative Log-Likelihood (NLL) over Training Iterations. Estimated density and samples and now also the mentioned training curve can be seen in Figure 4.5.

Note that because all algorithms except Sliced Score Matching are deterministic, we did multiple runs (10) for SSM and ploted the mean value with the standard deviation shaded.

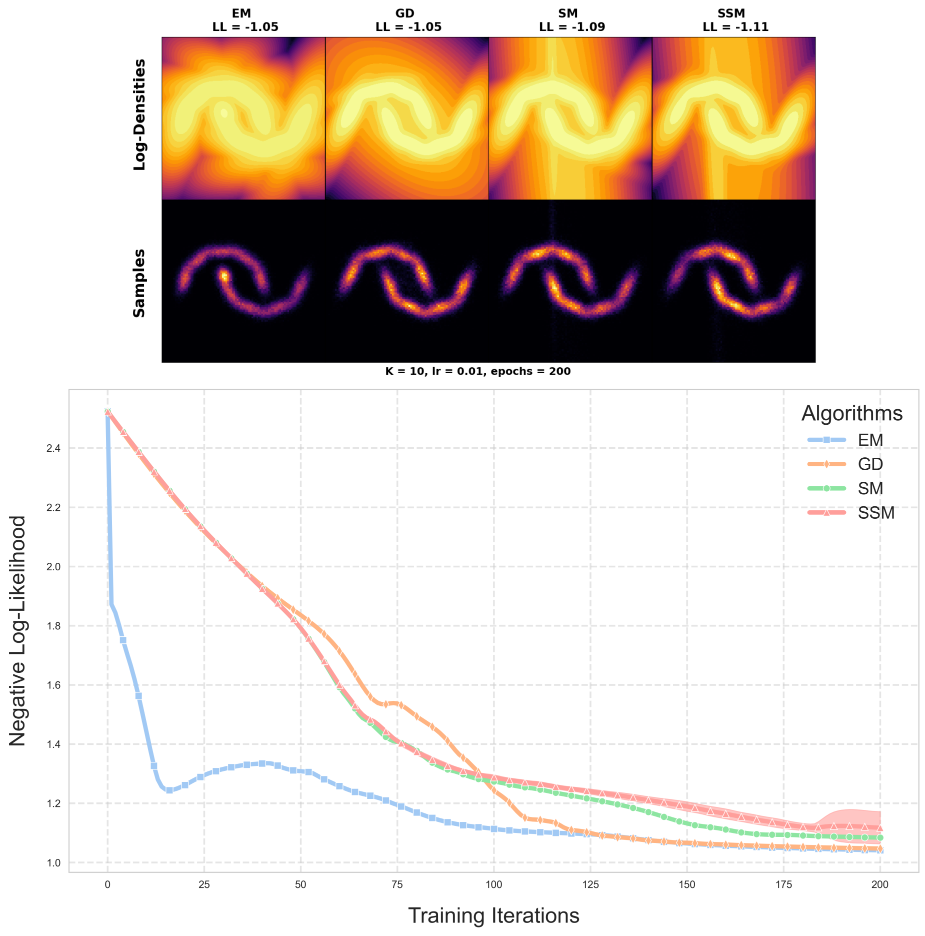


Figure 4.5.: Densities, Samples and NLL over Training Iterations

On average SSM performs somewhere in the ballpark of SM but usually a little worse, which makes sense since SSM is only an approximation of SM. Sometimes the stochastic nature of SSM can also by luck perform better than SM but this is only happens very rarely.

4.1.3. Experiment 3

Another point we wanted to examine was the initialization of the learnable parameters. With our data it is quite straightforward to initialize the means, which up to now has been done with KMeans, but this is not always the case. Also initializing the weights uniformly convieniently makes a lot of sense with our data, however the initialization of both of these learnable parameters can be problematic and a lot of times this has to be done randomly. Therefore we also analized how each algorithm behaves when these parameters are initialized randomly. To intitialize the means we draw K random datapoints from our training data and to initialize the weights we just draw random numbers from 0 to 1 and normalize them so they sum to one.

In Figure 4.6 the mean Log-Likelihood and standard deviation over 10 runs with random initialization in each run can be seen. On this simple dataset it appears that random initialization performs relatively similar to the KMeans initialization when comparing to the NLL over Epochs curve from Figure 4.5. For some concrete results (Densities and Samples) refer to the Appendix.

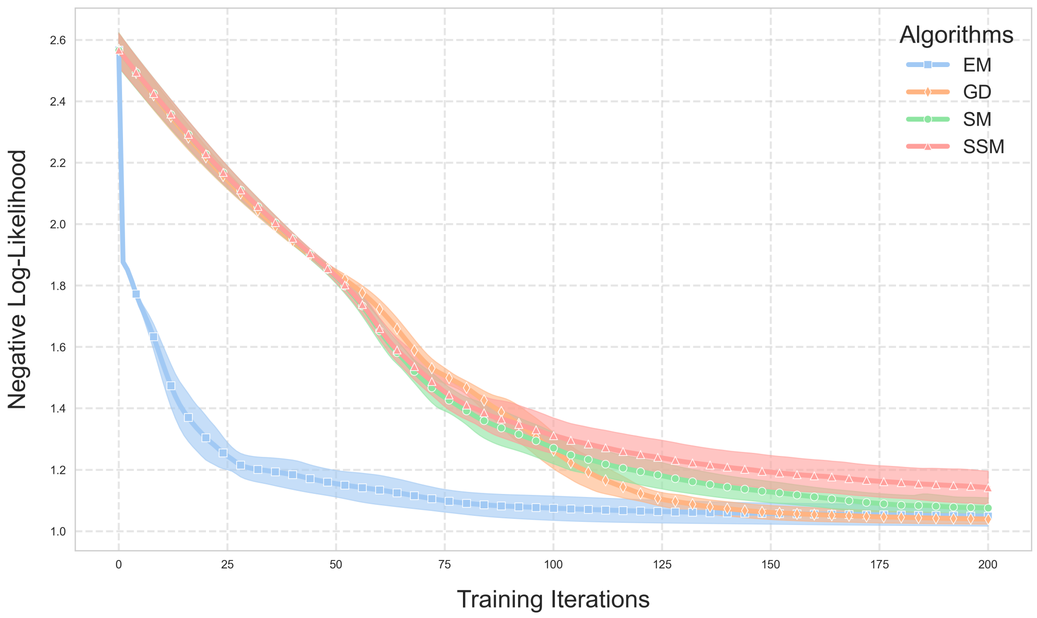


Figure 4.6.: Negative Log Likelihood over Epochs with random parameter initialization

We repeated this Experiment for a more complex dataset where all algorithms performed somewhat worse to when doing KMeans initialization but overall the difference between the algorithms, in which we are mostly intrested, remained basically the same. For those interested these results can also be found in the Appendix.

4.1.4. Experiment 4

Before continuing with the high dimensional experiments, we wanted to compare SSM and SM more extensively since in the high dimensional case only SSM is usable.

4.2. Images Density Estimation

I used the MNIST Dataset [15]

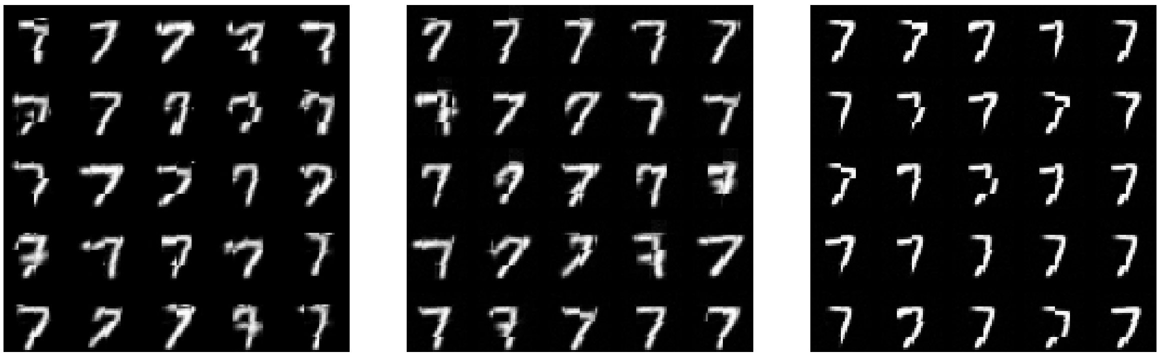

(a) EM (b) SGD (c) SSM

Figure 4.7.: MNIST Samples

5. Discussion

5.1. Interpretation of Experimental Results

6. Conclusions and Future Work

Appendix

Appendix A.

Density Functions

A.1. Multivariate Gaussian

$$p(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{A.1}$$

A.2. Log Multivariate Gaussian

$$\log p(\mathbf{x}) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{A.2}$$

Appendix B.

Additional Experiment Results

Bibliography

[1] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. “Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models.” In: (Oct. 2020). URL: `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf` (cit. on pp. 2–7).

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (cit. on pp. 3, 5, 13).

[3] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`. URL: `https://arxiv.org/abs/1406.2661` (cit. on p. 4).

[4] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: `1312.6114 [stat.ML]`. URL: `https://arxiv.org/abs/1312.6114` (cit. on p. 4).

[5] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. “Cutset Networks: A Simple, Tractable, and Scalable Approach for Improving the Accuracy of Chow-Liu Trees.” In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Toon Calders et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 630–645. ISBN: 978-3-662-44851-9 (cit. on p. 5).

[6] Doga Kisa et al. “Probabilistic sentential decision diagrams.” In: *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2014 (cit. on p. 5).

[7] Hoifung Poon and Pedro Domingos. *Sum-Product Networks: A New Deep Architecture*. 2012. arXiv: `1202.3732 [cs.LG]`. URL: `https://arxiv.org/abs/1202.3732` (cit. on p. 5).

[8] Aapo Hyvärinen and Peter Dayan. “Estimation of non-normalized statistical models by score matching.” In: *Journal of Machine Learning Research* 6.4 (2005) (cit. on pp. 7, 8).

[9] Yang Song et al. “Sliced Score Matching: A Scalable Approach to Density and Score Estimation.” In: (2019). arXiv: `1905.07088 [cs.LG]`. URL: `https://arxiv.org/abs/1905.07088` (cit. on pp. 9, 10).

[10] Pierre Blanchard, Desmond J. Higham, and Nicholas J. Higham. *Accurate Computation of the Log-Sum-Exp and Softmax Functions*. 2019. arXiv: `1909.03469 [math.NA]`. URL: `https://arxiv.org/abs/1909.03469` (cit. on p. 12).

[11] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703 (cit. on pp. 12, 13, 16).

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 13).

[13] Robert Peharz et al. *Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits.* 2020. arXiv: 2004.06231 [cs.LG]. URL: https://arxiv.org/abs/2004.06231 (cit. on p. 17).

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 19).

[15] Yann LeCun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 26).