Leon Tiefenböck

Training Tractable Probabilistic Models via Score Matching Objectives

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

Dipl.-Ing. Thomas Wedenig

Graz, October 2024

Abstract

Probabilistic Circuits (PCs) are a class of tractable probabilistic models that have seen much interest in recent years. By only introducing complexity in a structured manner, they are expressive and yet able to compute many inference tasks exactly and efficiently at the same time. While PCs have demonstrated success in different scenarios, the methods of training them remain an active research topic. In this thesis, we empirically investigate different methods to train PCs. We focus on Score Matching objectives, which are usually used in Energy Based Models (EBMs), and work with the gradient ("scores") of the log density function instead of the density function itself. We compare the performance of Exact as well as Sliced Score Matching with the current standard to train PCs, namely Maximum Likelihood Estimation (MLE) using either Stochastic Gradient Descent (SGD) or Expectation Maximization (EM).

Contents

1. Introduction

Probabilistic Modelling is an essential tool in machine learning that, through probability theory provides a principled mechanism for decision making under uncertainty. Recent progress, especially through models based on deep neural networks like Generative-Adversarial Networks (GANs) [1], Variational Autoencoders (VAE) [2] and many others, have increased the expressive capability of probabilistic models considerably. However in machine learning we are not only interested in generating realistic text or images, we also want to reason about data and perform various inference tasks. While probability theory conceptually provides an optimal framework to carry out these tasks, it is often computationally very hard, even more so when dealing with highly complex models as the ones mentioned above [3].

A promising research field that addresses this issue are Probabilistic Circuits (PCs). Probabilistic Circuit is an umbrella term for probabilistic models that, by only introducing complexity in a structured manner, are expressive while at the same time being able to compute many inference tasks exactly and efficiently [3].

While PCs have demonstrated success in different scenarios, the methods of training them remain an active research topic. Conventionally PCs are trained through Maximum Likelihood Estimation (MLE), often using Gradient Descent and Expectation Maximization as the optimization algorithms. These well-known methods provide theoretical robustness, but also have their limitations.

A possible direction to explore in this regard is Score Matching (SM) [4], an algorithm usually used with Energy Based Models (EBMs), that works with the gradient ("scores") of the log density instead of the density function itself. There are also variants like Sliced Score Matching (SSM) [5] that reduce the computational complexity of SM, making it more feasible for large-scale and high-dimensional data.

In this thesis we investigate the potential of using Score Matching objectives for training Probabilistic Circuits, by conducting experiments and comparing the results to the traditional MLE-based approaches. We aim to asses the effectiveness of these methods across different types of PCs and datasets, starting with a very simple PC for 2D density estimation and ending with a state-of-the-art framework, to do generative image modelling.

2. Background

2.1. Mathematical Notation

When discussing probabilities, random variables (RVs) are represented by capital letters X. Assignments to this RV are denotes as lowercase x.
Scalar quantities will be written as $x \in \mathbb{R}$, while vectors will be denoted as bold face $\mathbf{x} \in \mathbb{R}^D$. A density 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 data-generating distribution, we will write $p_{\text{data}}(x)$ or simply $p_d(x)$.

When discussing data, a whole dataset is denoted as $\mathcal{X} = \{x_1, ..., x_N\}$ where N is the number of samples. Again scalar quantities will be written as $x \in \mathbb{R}$, and vectors as bold face $\mathbf{x} \in \mathbb{R}^D$.

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 we are uncertain about particular quantities –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 [3].

2.2.1. Probability Basics

To clarify these ideas, consider two random variables X and Y, that can assume values 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 can often be described by a *probability density*

function (PDF). Since we will mostly deal with continuous random variables in this thesis, we will focus on densities going forward.

For any function to be a valid probability 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 a valid joint density function $p(x, y)$, allows us –in principle– to perform basically all key tasks that we want to accomplish in machine learning. [1] We can sample from the distribution to generate new data points, for example to generate images. Moreover, 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 the marginal probability (MAR) or the conditional probability (CON) [3].

The marginal probability function $p(x)$ is the probability of observing x regardless of the value of y. It is computed by integrating over all possible values of y:

$$p(x) = \int p(x, y)\, dy$$

The conditional probability function $p(x|y)$ is the probability of observing x given y. It is computed by dividing the joint probability by the marginal probability of y:

$$p(x|y) = \frac{p(x, y)}{p(y)}$$

2.2.2. Modelling a Density

Assume that we are given data $\mathcal{X} = \{x_1, ..., x_N\}$ that was generated by an unknown density function $p_{\text{data}}(x)$.
In this work, we always assume $x_i \overset{\text{iid}}{\sim} p_{\text{data}}$, which means a sample x_i is *independently* and *identically* distributed (iid) according to p_{data}.

In a real-world scenario, we typically don't have access to $p_{\text{data}}(x)$ and finding it can be seen as the "holy grail" of machine learning, because if it was known to us all inference tasks would reduce to simply applying the rules of probability theory [3]. However since we don't know $p_{\text{data}}(x)$, we create a parameterized model $p_\theta(x)$ and, using the data, we tune the parameter θ so that $p_\theta(x)$ approximates $p_{\text{data}}(x)$ as closely as possible.

[1] Note that wile probabilistic inference is conceptually simple, it is often computationally difficult in practice.

One of the most basic yet widely used methods for learning such a model is Maximum Likelihood Estimation (MLE) [6]. 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 data point in $\mathcal{X}$ with respect to θ, as expressed in Equation 2.1:

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

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

It is noteworthy 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:

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

Here $L(\theta)$ 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.

1. **Expressiveness**
   Expressiveness refers to the ability of a model to approximate different sets of distributions $p_{\text{data}}(x)$ 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) [3].

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 polynomial time with respect to the model size [3].

In recent years expressive capability has increased considerably through models based on neural networks like Generative-Adversarial Networks (GANs) [1], Variational Autoencoder (VAE) [2] and many others. However while these models excel in generating very realistic images or compelling text, they lack in tractability for all except the simplest inference tasks [3].

On the other hand there are many, mostly older, less complex models like Gaussian Mixture Models (GMMs), Hidden Markov Models (HMMs) and so on, that lack in expressiveness and only work well enough for simple data but can compute many inference tasks tractably [6].

2.3. Gaussian Mixture Model

A Gaussian Mixture Model (GMM) [6] is a well-known probabilistic model that will serve as good groundwork going forward.

The basic idea of a GMM is that a single Gaussian density cannot model complex distributions very well. However, a weighted sum of many different Gaussians, a so-called "mixture" can be an expressive density estimator.

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 $\sum_{k=1}^{K} \pi_k = 1$ and $\pi_k \geq 0$, ensuring that the model outputs 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. Furthermore $\boldsymbol{\theta}$ refers to the parameter vector $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$, where $\boldsymbol{\pi} = \{\pi_1, ..., \pi_K\}$, $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_K\}$, $\boldsymbol{\Sigma} = \{\boldsymbol{\Sigma}_1, ..., \boldsymbol{\Sigma}_K\}$.

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

2.4. Probabilistic Circuits

Probabilistic Circuit (PC) [3] is an umbrella term for probabilistic models that can perform many 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 [7], Probabilistic Sentential Decision Diagrams (PSDDs) [8] and others, but we will only be focusing on Sum Product Networks (SPNs) [9], 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 principle, a PC is 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, Binomials, Categoricals and so on, however most of the time we will talk about Gaussians [3]. 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 [3].

Considering this, 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. In fact, PCs overall can fundamentally be seen as *deep* Gaussian Mixture Models. This simple PC can be graphically depicted as in Figure 2.1, where the two nodes at the bottom represent input distributions over the RV X_1 and the top node represents the mixture.

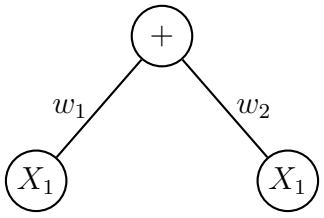


Figure 2.1.: Simplest PC - GMM

Furthermore more complex PCs can be created by hierarchically stacking sum nodes and product nodes. An only slightly more complex PC with a product node and two sum nodes can be seen in Figure 2.2.

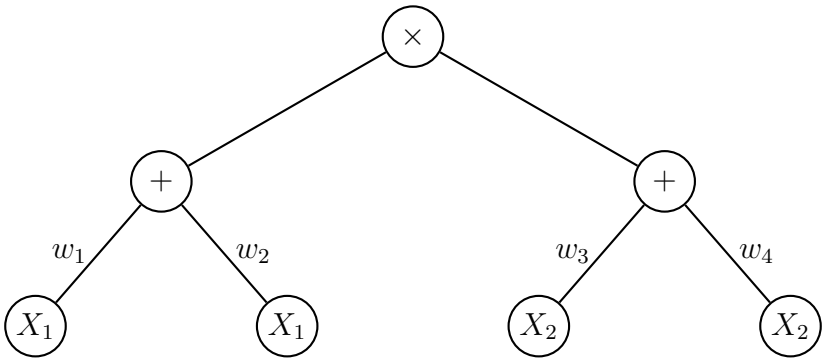


Figure 2.2.: PC: A product of two one-dimensional GMMs

Although 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 1 (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 $\mathbf{X}$ [3].

Definition 2 (Smoothness)**.** A sum node is *smooth* if all its inputs have identical scopes. A PC is smooth if all its sum nodes are smooth [3].

Definition 3 (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 [3].

In simple terms this basically means that sum nodes are only allowed to have inputs over the same variables 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 fulfill, however these two already guarantee tractable computation of the important marginal (MAR) and conditional (CON) queries [3], 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.

We refer the reader to [3] for why these properties guarantee tractable inference and further discussion on other properties.

Using a PC in a real-world scenario then would entail first deciding on a structure, so which properties and leaf distributions to use and how the nodes should be arranged in the graph, and then doing 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 optimization is usually done via Gradient Descent or Expectation-Maximization (EM). More details will follow in Section 3.1.

2.5. Score Matching

Score Matching [4] is a concept that is normally used when dealing with unnormalized models like Energy Based Models (EBMs). The term "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 [4]. Recall that GMMs and PCs can represent normalized densities, nullifying this issue.

The relation between a parameterized density p_θ and the energy E_θ is given by

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

Calculating the normalization constant would mean computing the integral

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

which is the source of the computational infeasibility.

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

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

$\nabla_x \log p_\theta(x)$ is called the score function, giving score matching it's name.
Let $s_\theta(x) \coloneqq \nabla_x \log p_\theta(x)$, prior work [4] proposes to minimize the Fisher Divergence between the scores of the data-generating distribution $s_d(x) \coloneqq \nabla_x \log p_d(x)$ and the scores of the model $s_\theta(x)$:

$$\mathcal{L}(\theta) = \frac{1}{2}\mathbb{E}_{p_d}\left[\|s_\theta(x) - s_d(x)\|_2^2\right] \tag{2.4}$$

Note that $\mathcal{L}(\theta)$ does not depend on the intractable normalization constant, however it introduces another problem since it depends on $s_d(x)$, which is unknown. Furthermore [4] shows that by partial integration the expression can be rewritten as:

$$\mathcal{L}(\theta) = \mathbb{E}_{p_d}\left[\text{tr}\left(\nabla_x s_\theta(x)\right) + \frac{1}{2}\|s_\theta(x)\|^2\right] + C \tag{2.5}$$

which doesn't depend on $s_d(x)$ anymore. Here C is a constant independent of θ and $\text{tr}\left(\nabla_x s_\theta(x)\right)$ refers to the trace (sum of the diagonals) of the Hessian matrix of $\log p(x)$.

To use this final score matching objective to train a model on some data $\mathcal{X} = \{x_1, ..., x_N\}$ we can approximate $\mathcal{L}(\theta)$ via a Monte Carlo estimate $\hat{\mathcal{L}}(\theta)$ [4]:

$$\hat{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(\text{tr}\left(\nabla_{x_i} s_\theta(x_i)\right) + \frac{1}{2}\|s_\theta(x_i)\|^2\right) \tag{2.6}$$

Notice that we also removed the constant C since we seek $\theta^{SM} = \arg\min_\theta \hat{\mathcal{L}}(\theta)$ where C has no impact.

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 2.6 one derivation of $s_\theta(x)$ 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 infeasible when learning higher dimensional data, for instance images.

In Sliced Score Matching (SSM) [5] the basic idea is to project the high dimensional data onto a random direction $\mathbf{v}$ 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 2.4 [5]:

$$\mathcal{L}(\theta; p_v) = \frac{1}{2}\mathbb{E}_{p_\mathbf{v}}\mathbb{E}_{p_d}\left[\left(\mathbf{v}^T s_\theta(x) - \mathbf{v}^T s_d(x)\right)^2\right] \tag{2.7}$$

where $p_\mathbf{v}$ is chosen s.t. $\mathbb{E}\left[\mathbf{v}\mathbf{v}^T\right] \succ 0$.

By using partial integration, similar to what was done in Score Matching, we can get rid of the unknown scores of the data $s_d(x)$:

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

where C is again a constant independent of θ [5].

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 2.8 can be integrated analytically [5], yielding the following objective:

$$\mathcal{L}(\theta; p_\mathbf{v}) = \mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\mathbf{v}^T\nabla_\mathbf{x} s_\theta(\mathbf{x})\mathbf{v} + \frac{1}{2}\|s_\theta(\mathbf{x})\|_2^2\right] \tag{2.9}$$

The authors of [5] 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 Equation 2.8.

To again use this for training with data $\mathcal{X} = \{x_1, ..., x_n\}$ and M random vectors $\mathbf{v}_{ij}$ for each data point, we use the Monte Carlo estimate [5]:

$$\hat{\mathcal{L}(\theta)} = \frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(\mathbf{v}_{ij}^T\nabla_{\mathbf{x}_i} s_\theta(\mathbf{x}_i)\mathbf{v}_{ij} + \frac{1}{2}\|s_\theta(\mathbf{x}_i)\|_2^2\right) \tag{2.10}$$

where $\mathbf{v}_{ij} \overset{\text{iid}}{\sim} p(\mathbf{v})$.

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 conventional 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 Equation 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)** [10] 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 underflow [10]. Overflow is still possible but typically that is less of an issue.

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 Equation 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 Equation 3.3 as just $\log p_{\boldsymbol{\theta}}(\mathbf{x})$.

3.1.1. Maximum Likelihood Estimation

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

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

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 $\text{NLL}(\boldsymbol{\theta})$ for each parameter in $\boldsymbol{\theta}$:

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

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

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} \text{NLL}(\boldsymbol{\theta})$$

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

Algorithm 1 Gradient Descent

Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, η, T

1: **for** $t = 1$ to T **do**
2: $\quad \text{NLL}(\boldsymbol{\theta}) \leftarrow -\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i)$
3: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \text{NLL}(\boldsymbol{\theta})$
4: **end for**
5: **return** $\boldsymbol{\theta}$

We implemented 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** [6].

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

for $t = 1$ to T **do**
  for $i = 1$ to N **do**
    for $k = 1$ to K **do**
      $\gamma_{ik} \leftarrow \frac{\pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$
    end for
  end for
  for $k = 1$ to K **do**
    $\pi_k \leftarrow \frac{1}{N}\sum_{i=1}^{N}\gamma_{ik}$
    $\boldsymbol{\mu}_k \leftarrow \frac{\sum_{i=1}^{N}\gamma_{ik}\mathbf{x}_i}{\sum_{i=1}^{N}\gamma_{ik}}$
    $\boldsymbol{\Sigma}_k \leftarrow \frac{\sum_{i=1}^{N}\gamma_{ik}(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T}{\sum_{i=1}^{N}\gamma_{ik}}$
  end for
end for
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 from Equation 2.6, Section 2.5 and formulate the optimization problem:

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

Then we optimize this objective using Gradient Descent for T Training Iteration and Learning Rate η.

Algorithm 3 Score Matching

Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, η, T

1: **for** $t = 1$ to T **do**
2: $\quad \mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N} \sum_{i=1}^{N}$ ScoreMatchingLoss$(\mathbf{x}_i)$
3: $\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$
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 from Equation 2.10, Section 2.6.
Note that for simplicity we only use one random vector $\mathbf{v}_i$ (one slice) for each data point.

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

Algorithm 4 Sliced Score Matching

Input: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, $\boldsymbol{\theta}$, η, T

for $t = 1$ to T **do**
  $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N} \sum_{i=1}^{N} \textsc{SlicedScoreMatchingLoss}(\mathbf{x}_i)$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$
end for
return $\boldsymbol{\theta}$

Procedure: $\textsc{SlicedScoreMatchingLoss}(\mathbf{x})$

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

We implemented both Algorithms again using PyTorch [11] and `torch.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: $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$

$k \sim \textsc{Categorical}(\boldsymbol{\pi})$
$x \sim \mathcal{N}(\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$
return x

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 two 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 EinsumNetworks we refer to [13]. In general, the same principles as in our GMM apply but due to the deeper structure, the model is much more complex and can approximate more complex distributions. It also has a lot more trainable parameters and thus training takes more time.

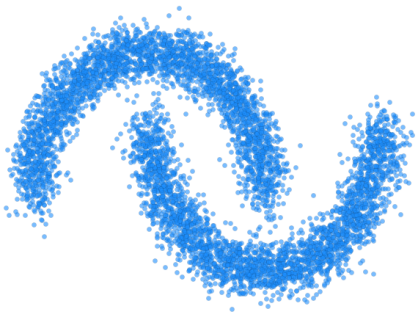
However implementing the training of an EinsumNetwork with our targeted algorithms is basically the same as in the GMM case. Note that training with EM already comes with the framework, so we will leave this out.
For all other algorithms, we can apply them in the exact same way as described in Section 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

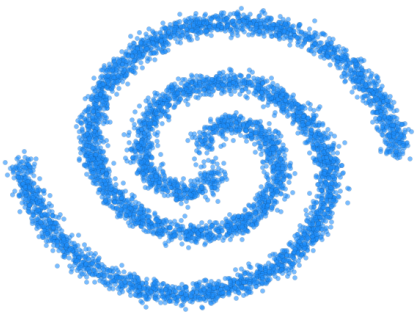
4.1. 2D Density Estimation

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 Sections 3.1.1 to 3.1.2, we used some two dimensional data to perform density estimation.

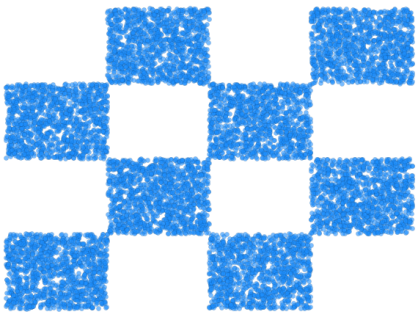
Samples from the three datasets we used can be seen in Figure 4.1.



(a) halfmoons



(b) spirals



(c) board

Figure 4.1.: Samples for all three datasets

4.1.1. Main Experiment

Here the goal was to achieve the best possible results for each algorithm on each dataset. For this we generated 20,000 data points for each dataset 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 identity 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, through some initial 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 diminishing returns and a moderate K in the middle.

For the training process, recall that it is also governed by hyperparameters, namely the number of iterations T for all algorithms and the learning rate η for all the gradient-based algorithms (GD, SM, SSM). To choose these parameters we fixed K to one of the three chosen values and performed 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 can be seen in the following three pages.

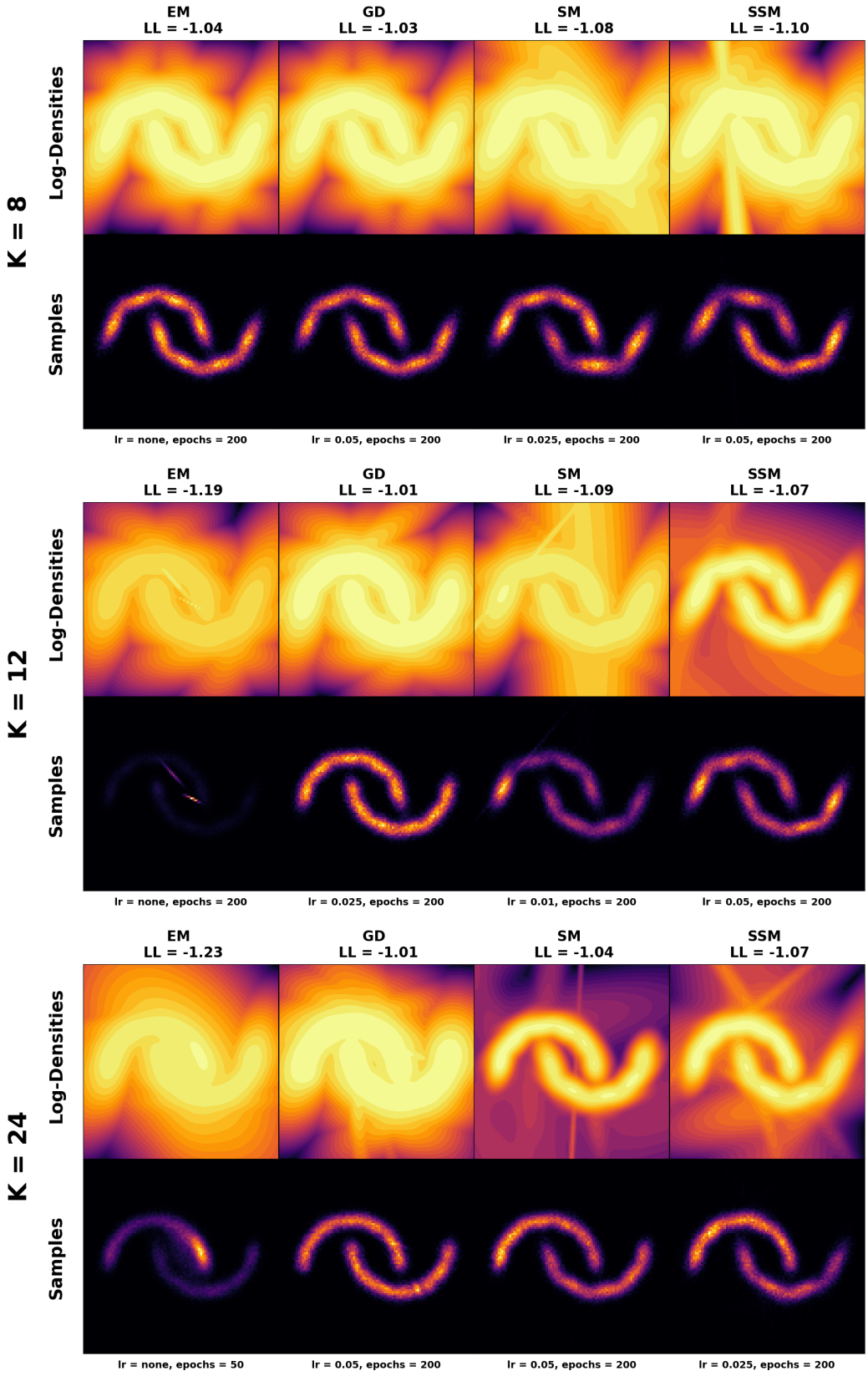


Figure 4.2.: Densities and Samples for the halfmoon dataset

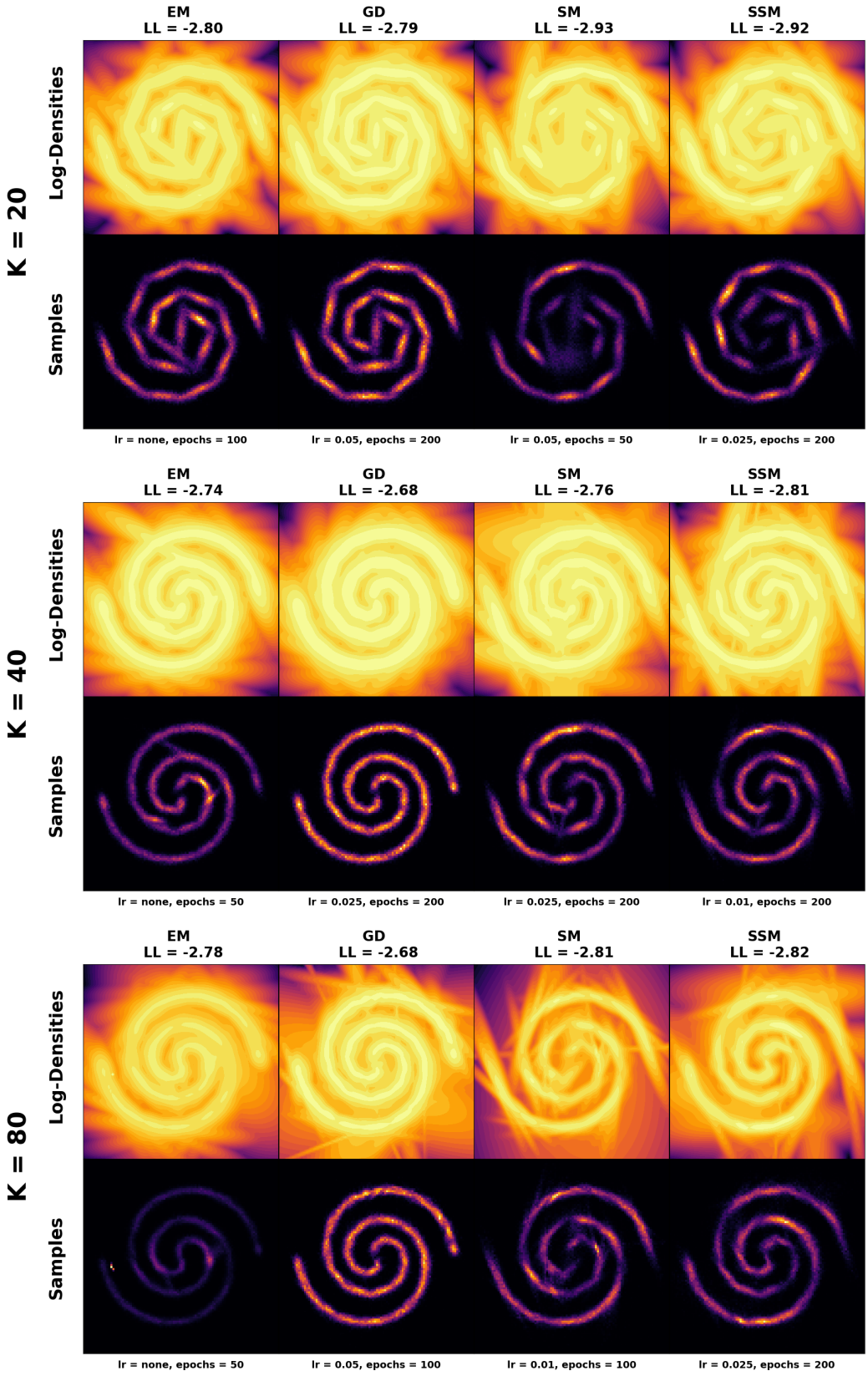


Figure 4.3.: Densities and Samples for the spirals dataset

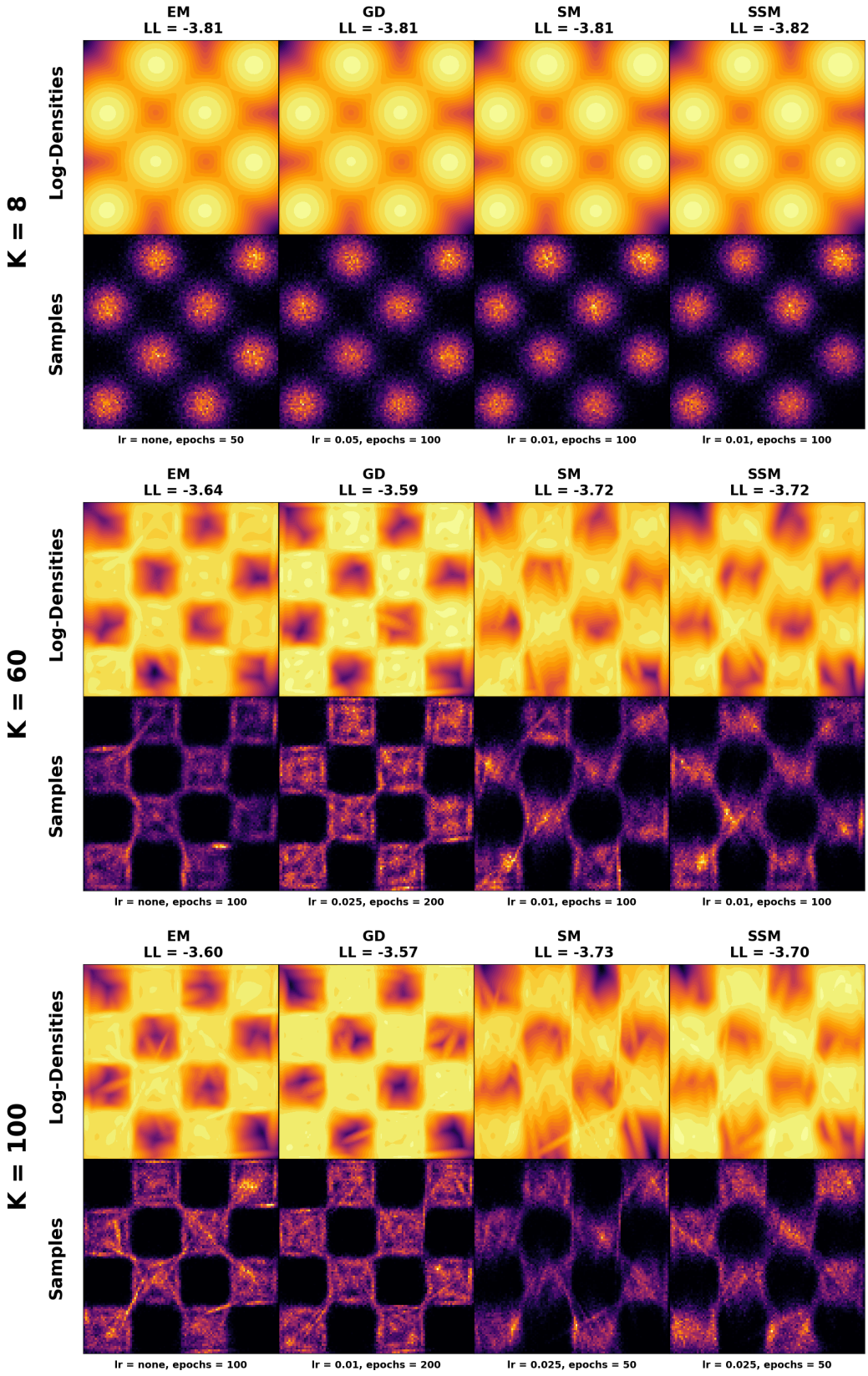


Figure 4.4.: Densities and Samples for the board dataset

Looking at these results, specifically at the LL, we see that GD usually performs the best, EM, depending on the scenario, in second place and SM and SSM, with very similar results, in last place.

Analyzing the estimated densities is more difficult. Generally GD also appears to produce the best results, meaning mostly noise free, evenly distributed densities. EM behaves very similar when having a low K, but with both the halfmoons and spirals having a higher K produces densities that are very dense in a few small points, producing samples that appear empty in most places except these very high density regions. SM and SSM in most cases produce the most noisy, unevenly distributed, densities. However with the halfmoons dataset with K equaling 12 or 24 we see both SM and SSM producing very dense but still correct densities, that set them apart from the other algorithms.

4.1.2. Further Analysis

Training Time

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 analyzed 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. Because all algorithms except Sliced Score Matching are deterministic, we did multiple runs (10) for SSM and plotted the mean value with the standard deviation shaded.

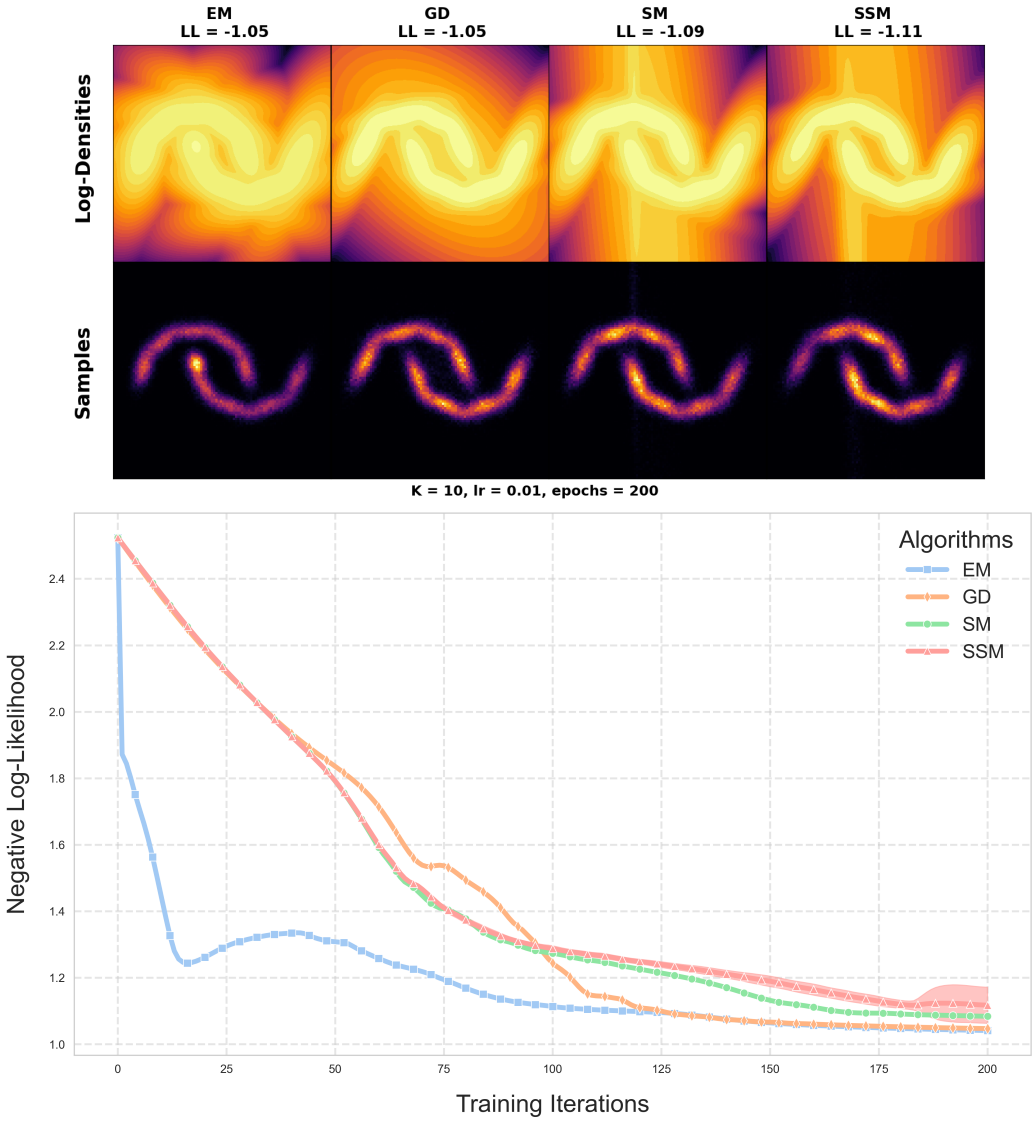


Figure 4.5.: Densities, Samples and NLL over Training Iterations

First, we see that on average SSM performs very similar to SM (usually a little worse), which given that SSM is an approximation of SM, can be expected. Sometimes the stochastic nature of SSM can also by sheer luck yield better results. Note that when increasing the number of slices, which were 1 for all experiments up until now, we saw that SSM increasingly gives a better approximation of SM.

Also clearly noticeable is that EM appears to learn the fastest and all the gradient-based algorithms behave quite similarly. This was similar for all other datasets and hyperparameter configurations as well. However note that in the Figures above, we are only looking at NLL over Epochs, so we are not accounting for the duration of a single

epoch. When running all algorithms for a specific amount of time, the differences increase considerably.

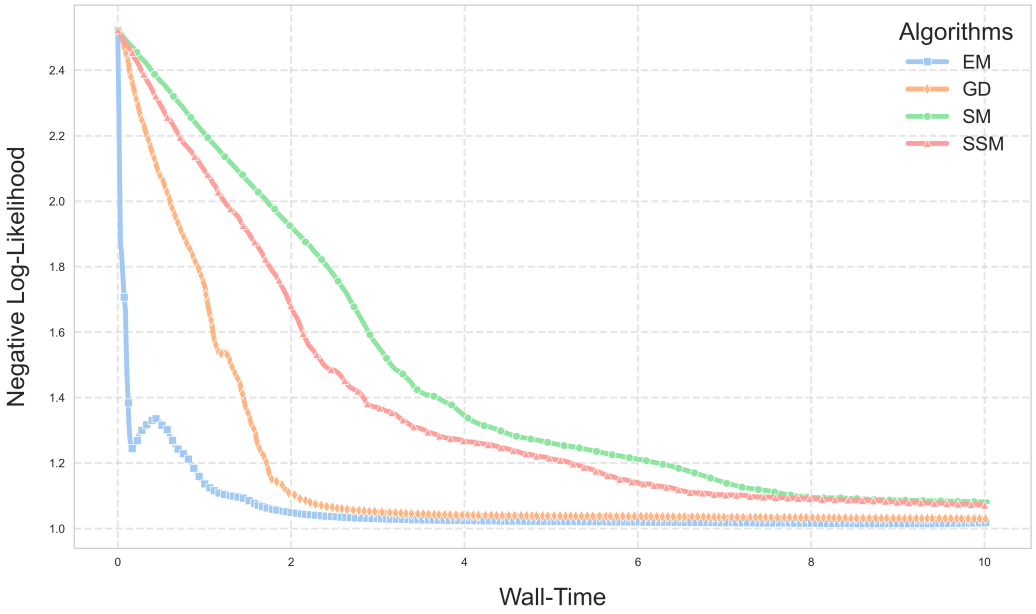


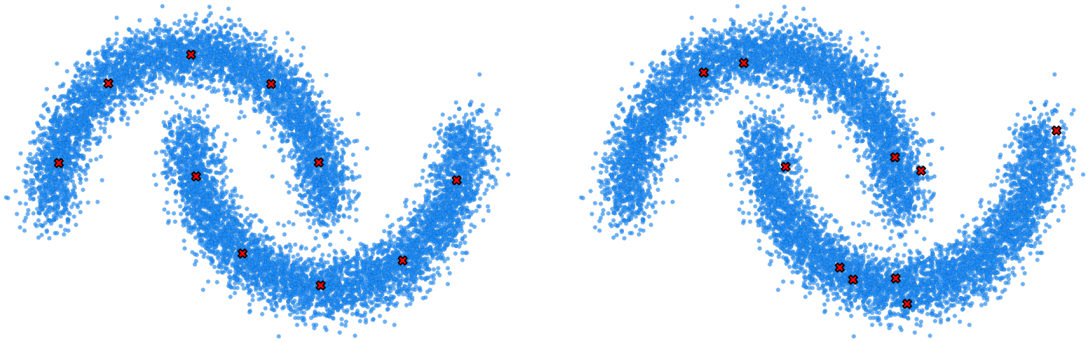
Figure 4.6.: NLL over Time

EM and GD appear considerably faster than their Score Matching alternatives, in fact while GD on average only takes around 1.1 times as long as EM, SSM takes around twice as long and SM around 2.5 times as long.

Initialization

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 conveniently makes a lot of sense with our data, however the initialization of both these parameters can be problematic and has to be done randomly quite often. Therefore we also analyzed how each algorithm behaves when these parameters are initialized randomly.

To initialize the means we draw K random data points 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.



(a) KMeans (b) Random

Figure 4.7.: KMeans vs. Random Initialization of means

In Figure 4.8 the mean Log-Likelihood and standard deviation over 10 runs with random initialization in each run can be seen.

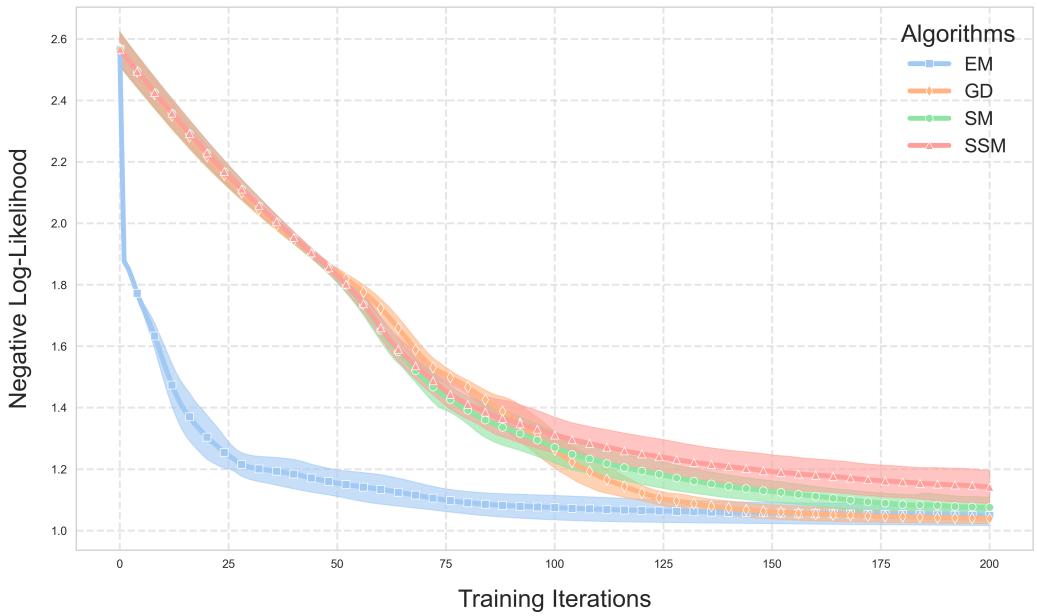


Figure 4.8.: Negative Log Likelihood over Epochs with random parameter initialization

On this simple dataset it appears that random initialization performs relatively similar to the KMeans initialization when comparing with the NLL over Epochs curve from Figure 4.5. For some concrete results (Densities and Samples) refer to the Appendix B.2. We repeated Experiments 4.1.2 and 4.1.2 for the spirals dataset, results can be found in Appendix B.1.

4.2. Generative Image Modelling

To train a before mentioned EinsumNetwork [13] with our targeted algorithms, we used the provided demo-file for the MNIST [15] dataset (`demo_mnist.py`) and included our algorithms. MNIST is a dataset of handwritten digits (from 0 to 9) and quite popular for image modelling tasks. Note that, since training already took quite long, we left out exact Score Matching and only focused on Sliced Score Matching. In all following results we also ever only used one slice, since having more did not significantly change the results and increased training time.

Figure 4.9.: Samples of MNIST

4.2.1. Single Class

Here the goal is to produce the best possible samples on a single class, meaning one type of digit, of MNIST.

We left the model related hyperparameters as they were provided in the framework and only tuned the training specific hyperparameters (iterations and learning rate for gradient based algorithms and only iterations for EM) to find the best samples for each algorithm.

We found that, while EM gave quite good samples after only around 10 iterations, SGD and SSM required much longer to converge. More precisely, we saw that training with SSM or SGD, using an already moderately high learning rate of 0.01, needed around 200 iterations to produce good results, which of course even increases more if we set a lower learning rate. We provide a graph of samples at specific epochs on MNIST class 7 for all algorithms in Figure 4.10.

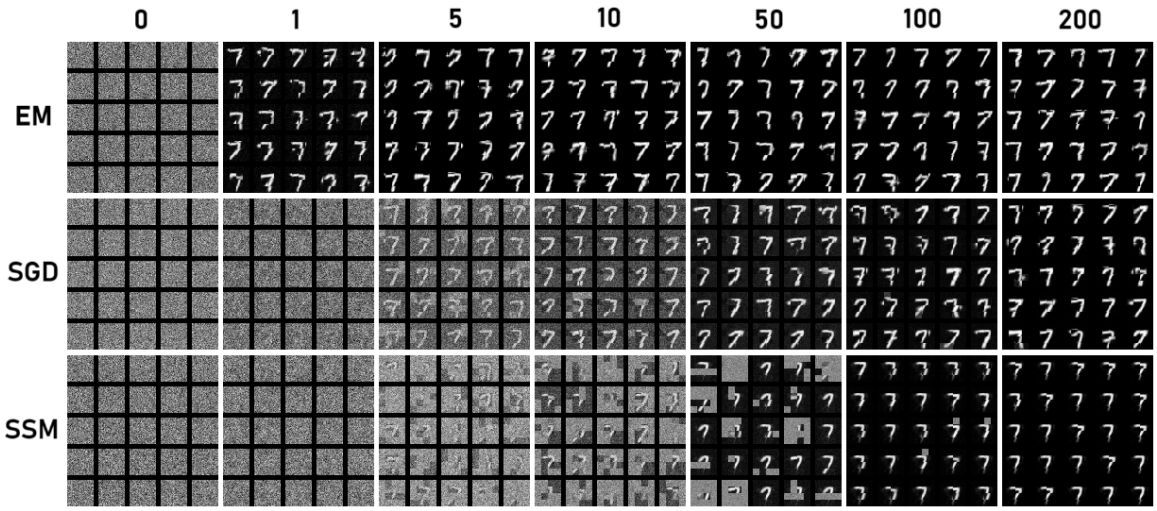


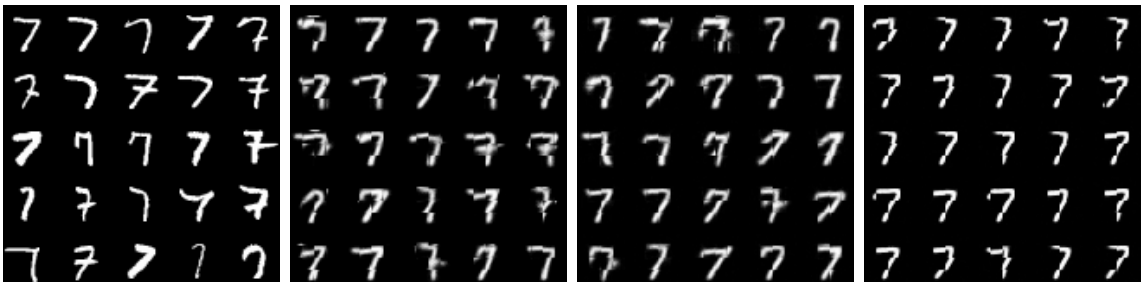
Figure 4.10.: Samples over epochs for different training algorithms

Although increasing the learning rate further mostly worked for SGD, SSM saw vanishing or exploding gradients after some iterations and thus was not able to properly finish. Even applying gradient clipping did not really solve this.

Therefore, the best way to train an EinsumNetwork using SSM, we could find, is to start with a relatively high learning rate of 0.2 or 0.1 and use learning rate decay to decrease it over time. Although this could further be specifically tuned for each different MNIST digit, we find that training for 20 iterations using an initial learning rate of 0.2 and decaying it with a factor of 0.5 every 5 epochs while additionally using gradient clipping with a threshold of 1.0 was the most practical approach for all classes. This approach also improved the SGD results further.

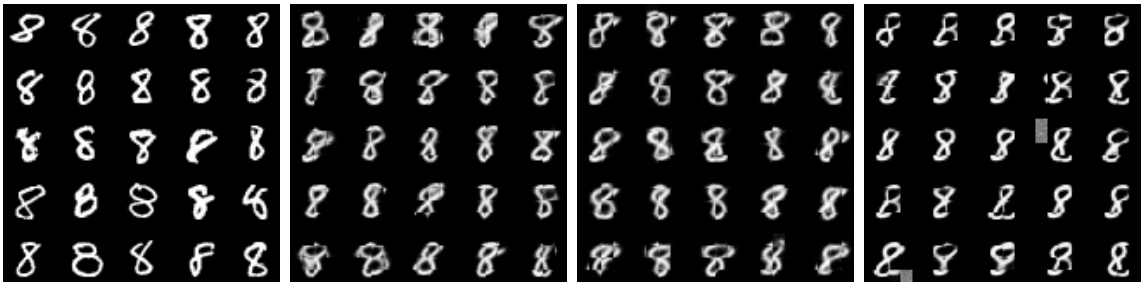
We trained with each algorithm for 20 iterations using the described learning rate schedule for SGD and SSM on three different classes of MNIST. Log Likelihood (LL) and samples can be seen in the figures below. For results on FashionMNIST we refer to to Appendix B.3.

On a side note, this setup resulted in 204146 learnable parameters in the EinsumNetwork. For comparison the most complex 2D model we used, with 100 mixture components, had 700 learnable parameters.



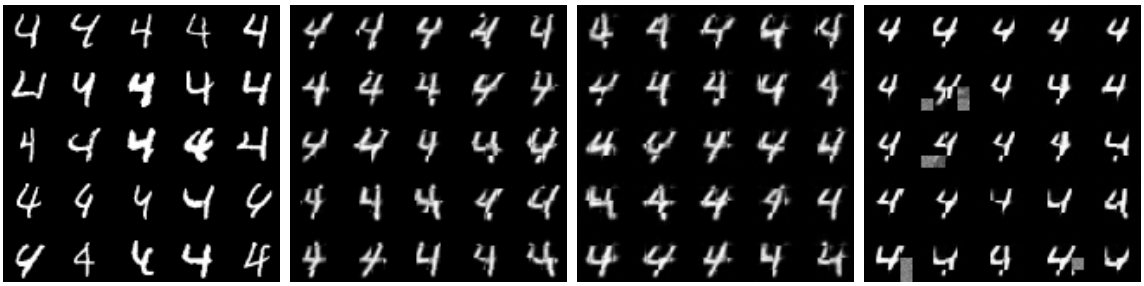
(a) Ground Truth (b) EM (LL: -11576) (c) SGD (LL: -12972) (d) SSM (LL: -16307)

Figure 4.11.: Samples and Log-Likelihood of MNIST-class 7



(a) Ground Truth (b) EM (LL: -15945) (c) SGD (LL: -17796) (d) SSM (LL: -22654)

Figure 4.12.: Samples and Log-Likelihood of MNIST-class 8



(a) Ground Truth (b) EM (LL: -13301) (c) SGD (LL: -15626) (d) SSM (LL: -20309)

Figure 4.13.: Samples and Log-Likelihood of MNIST-class 4

Looking at these figures, specifically the LL, we see a quite similar picture as in the 2D Density Estimation emerge. EM and SGD performed on par while SSM lagged a little behind. Also the time needed for training reflected the 2D counterpart, though the gap increased even more. While EM and SGD roughly took the same amount of time, SSM needed approximately four times longer to finish the 20 iterations.

When directly looking at the samples the results appear to be a bit different. While EM and SGD provided more or less the same samples, with SSM, they seemed to

be the least "blurry" or most "sharp", however with the caveat that sometimes noisy artifacts appear, best scene with class 8. We also found that when introducing a weight decay in the optimizer, the number of these artifacts increase. Furthermore it often seemed that one particular type of the provided digit dominated, giving all samples a very similar look, best scene with class 7.

Finally we also performed image reconstructions, where a certain region of the image is masked out and then reconstructed using the trained model. The masked out images as well as the reconstructions for the same MNIST digits as used before can be seen below.

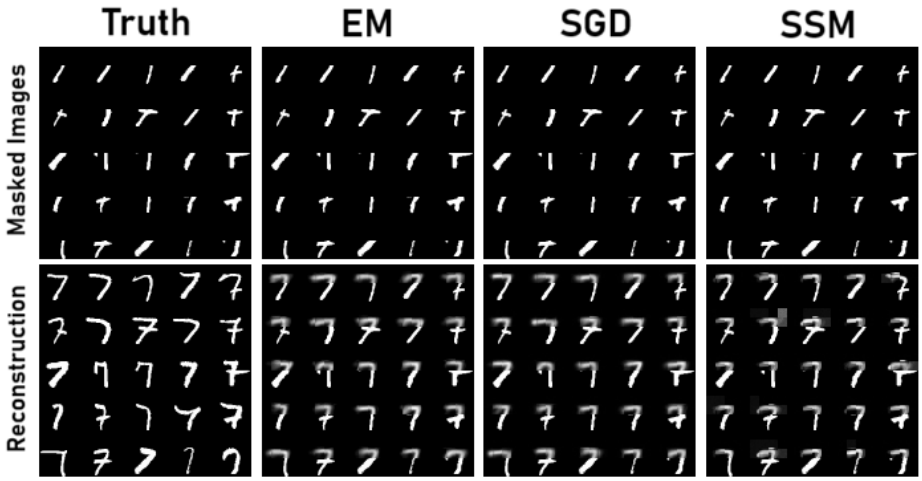


Figure 4.14.: Masked images and reconstructions of MNIST-class 7

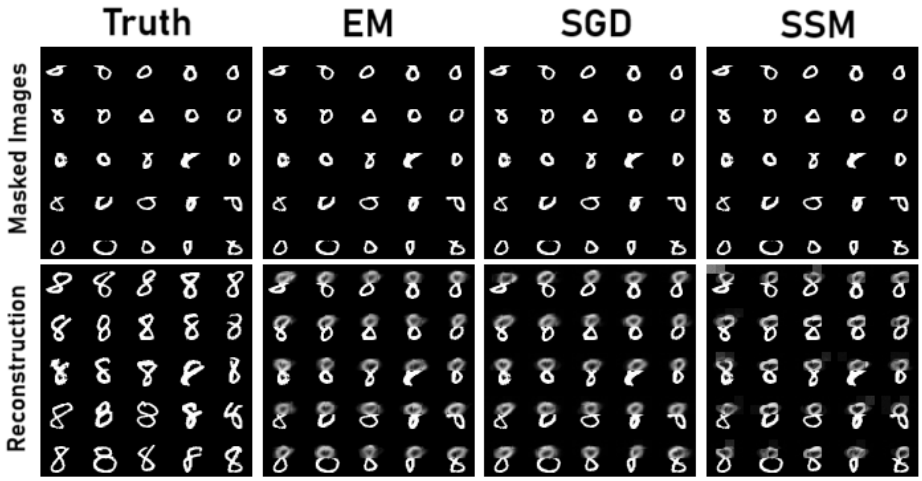


Figure 4.15.: Masked images and reconstructions of MNIST-class 8

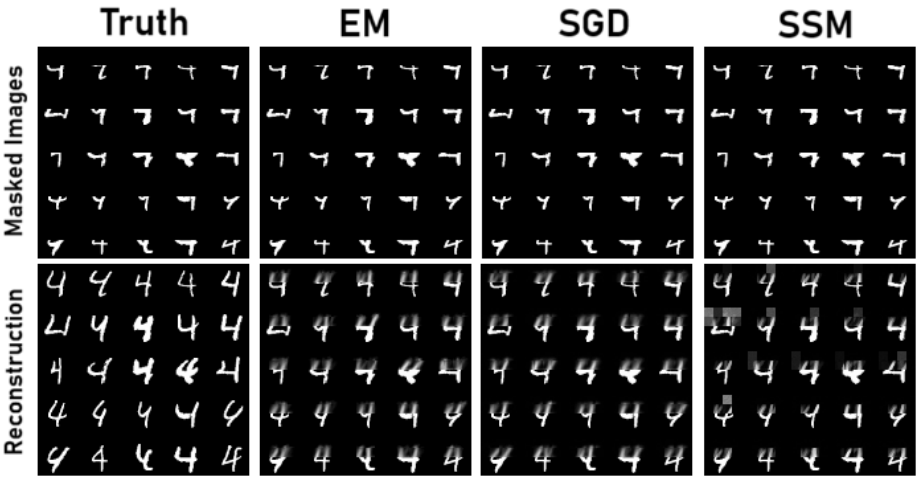


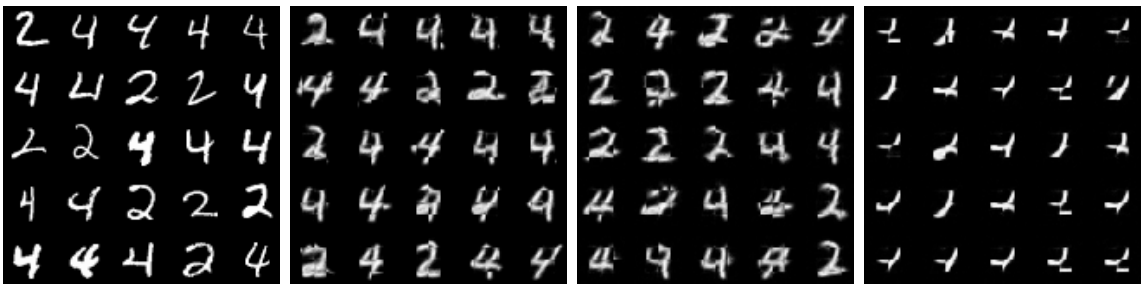
Figure 4.16.: Masked images and reconstructions of MNIST-class 4

The results here are quite on par with the samples, not revealing too much new differences between the algorithms. All algorithms are able to fill out the masked parts reasonably well. One thing of note is that is seems SSM is loosing some of the sharpness, that was present with the samples, in this scenario.

4.2.2. Multi Class

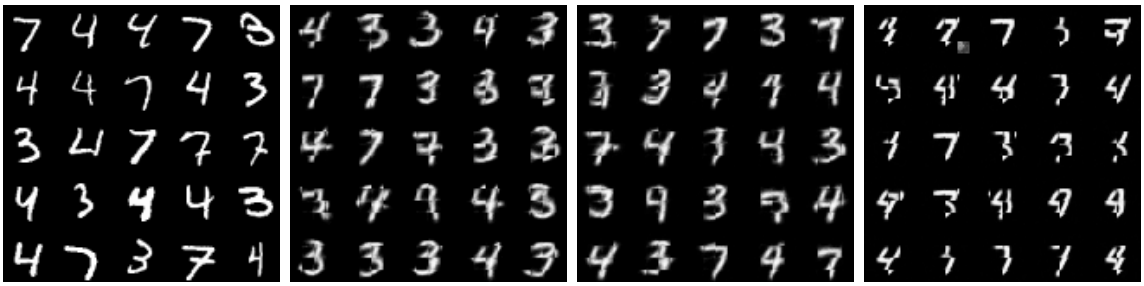
Since in real world scenarios it is seldom the case that we only want to model one specific class of the data, we also trained an EinsumNetwork on multiple classes of MNIST at the same time.

We again trained for 20 iterations with each algorithm, though we had to adjust the initial learning rate for SSM to 0.1 in all scenarios, and the decay to 0.5 every 2 epochs when training on all classes. Results, meaning samples and LL, for three different combinations can be seen below.



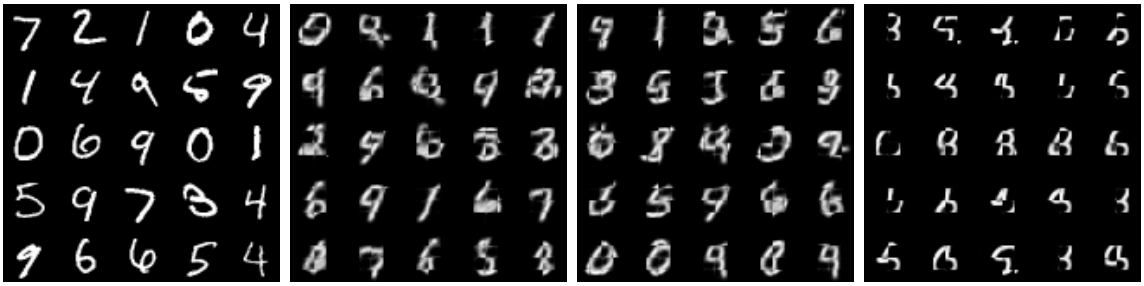
(a) Ground Truth (b) EM (LL: -17219) (c) SGD (LL: -19385) (d) SSM (LL: -25558)

Figure 4.17.: Samples and Log-Likelihood of MNIST-classes 2 and 4



(a) Ground Truth (b) EM (LL: -16571) (c) SGD (LL: -17800) (d) SSM (LL: -22736)

Figure 4.18.: Samples and Log-Likelihood of MNIST-classes 3, 4 and 7



(a) Ground Truth (b) EM (LL: -17322) (c) SGD (LL: -18596) (d) SSM (LL: -25221)

Figure 4.19.: Samples and Log-Likelihood of all MNIST-classes

The NLLs, although being higher overall, are relatively also very similar to before. EM and SGD samples are quite similar again and SSM samples also appear to be the most "clear", however this time the samples are clearly worse. While EM and SGD do not produce perfect samples, different classes are distinguishable in most cases. This is not the case for SSM. Here the samples appear to be more of a superposition of the different classes, which is not what we want.

5. Discussion

5.1. Interpretation of Experimental Results

5.1.1. 2D Density Estimation

Looking again at the main experiment from the 2D Density Estimation in Section 4.1.1, we can see that Maximum Likelihood Estimation with Gradient Descent or EM provided the highest Log-Likelihood values for all scenarios –except where EM failed and produced very small high density regions–. This could be somewhat expected, since here we are directly maximizing the Likelihood, whereas with the Score Matching algorithms, the Likelihood is only indirectly optimized by minimizing the Score Matching Objective, that is derived from Fisher Divergence. However, when looking at the density and sample plots, Gradient Descent also objectively produced the nicest results, meaning the least noisy and the most accurately distributed, which wasn't necessarily expected. Also not expected was that EM in many cases struggled with large K, finding very small high density regions. Still considering these plots it was interesting to see, that Sliced Score Matching with only 1 slice was able to approximate exact Score Matching so well.

When looking at the Analysis in Section 4.1.2 the main takeaways are that EM is the fastest epoch-wise and also overall time-wise, though GD is only slightly slower. As expected, the many additional calculations for the Score Matching Algorithms provide a huge performance penalty. Another thing to reiterate here is the time-wise difference between SM and SSM. In our measurements SSM was only very slightly faster than SM, which does not suggest a huge improvement. However remember that SM scales with the dimensionality of the data, so in our 2D scenario, with 1-slice SSM, basically only one additional calculation is needed to be exact. In higher dimensions this difference would be much more pronounced.

For the initialization analysis from Section 4.1.2 it was somewhat surprising how similar all algorithms performed when compared to their KMeans counterpart. The Log-Likelihood values were usually only slightly worse and the same goes for the density and sample plots. However we noticed that for the more complex spirals dataset the difference was already larger than for the simple halfmoon dataset. So we expect that the more complex the dataset is, the worse random initialization performs.

Overall we can conclude that for 2D Density Estimation, Gradient Descent provided the best results, had the best robustness to different hyperparameter settings and was nearly tied as the fastest algorithm. While the Score Matching algorithms were not much worse Log-likelihood wise and provided some interesting and unique samples, the overall performance and also longer training time make them less attractive for this task.

5.1.2. Image Modelling

Now considering the image modelling using MNIST from Section 4.2 we can see similar patterns emerge. EM and SGD perform very similarly time-wise and result-wise –when focusing on the measured Log-Likelihoods–, while SSM lags behind in both aspects.

The main findings we want to highlight however are: In the single-class scenario, while EM and SGD both produce reasonable and diverse samples, where different "types" of the same digit are clearly recognizable, they are quite blurry and have very blurry artifacts. On the other hand, SSM produces very sharp samples, but they are not very diverse, meaning most sampled digits appear to be of the same type. Additionally they also sometimes exhibit noisy artifacts. One explanation for this could be that with SSM, one or a few weights in the mixture dominate, growing quite large while the others remain at 0, thus only ever producing the same samples.

When looking at the multi-class scenario we see the same behavior for EM and SGD, meaning they produce distinct and recognizably different samples, though now even more blurry and with more artifacts. For SSM, we see that the samples still are the most sharp, with the least artifacts and now also much more diverse compared to before, but they appear to be different superpositions of the classes they are trained on, meaning the different classes are not really recognizable. This was best seen with classes 2 and 4. Considering this, it could be possible that a superposition is also learned in the single-class scenarios, giving another explanation for the similar samples.

Concluding we can say that for image modelling, still EM or SGD are the best choices, though here SSM provides even more interesting samples –especially the superior sharpness– that show a lot of potential.

6. Conclusions and Future Work

In this thesis we experimentally tested different methods to train Probabilistic Circuits (PCs), specifically we wanted to examine how recent developments in training Energy or Score based Models, namely Sliced Score Matching would transfer to PCs. We achieved this by implementing the algorithms and conducting experiments on 2D Density Estimation and Image Modelling. In all relevant experiments we compared performance by producing samples (and density plots) and measuring the Log-Likelihood.

Concluding we can say, we did not recognize any obvious benefits to use Score Matching, wether it being Sliced or Exact, over the conventional Maximum Likelihood Estimation (MLE) using either Gradient Descent or Expectation Maximization, when just looking at the Log-Likelihood measurements or the time it took to train a model. When considering the samples, especially for the Image Modelling task, we saw that Sliced Score Matching produced very sharp and less noisy samples compared to the MLE counterparts. These huge improvements in image quality, however came with the downside that samples were less diverse in the single-class scenario and in the multi-class scenario the different classes were not even recognizable.

For future work we suggest to further investigate the reasons for the increased sharpness and the lack of diversity as well as the superposition of classes in SSM, with the goal of finding an algorithm that combines diversity and sample quality. For now to provide a possible solution, at least when trying to train on an entire dataset with multiple classes, one could use Sliced Score Matching to train a model on each class individually and then create a mixture to combine them. Though this does not solve the problem in essence.

Another path of possible future research is to use exact Score Matching in combination with PCs. We hypothesize that, due to the nature of PCs, exact Score Matching could be implemented in a more efficient way than in general Energy Based Models, reducing the computational overhead and making it more feasible to use in practice.

Appendix

Appendix A.

Density Functions

Multivariate Gaussian

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

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

B.1. Spirals Analysis

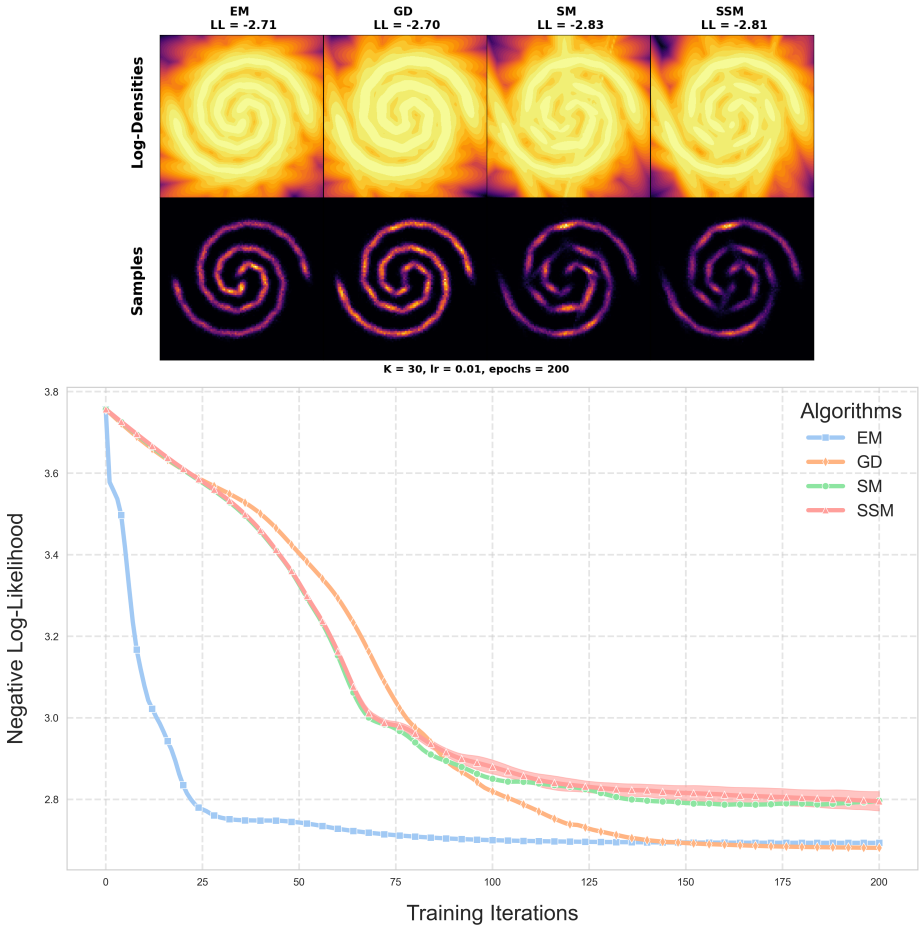


Figure B.1.: Densities, Samples and NLL over Training Iterations

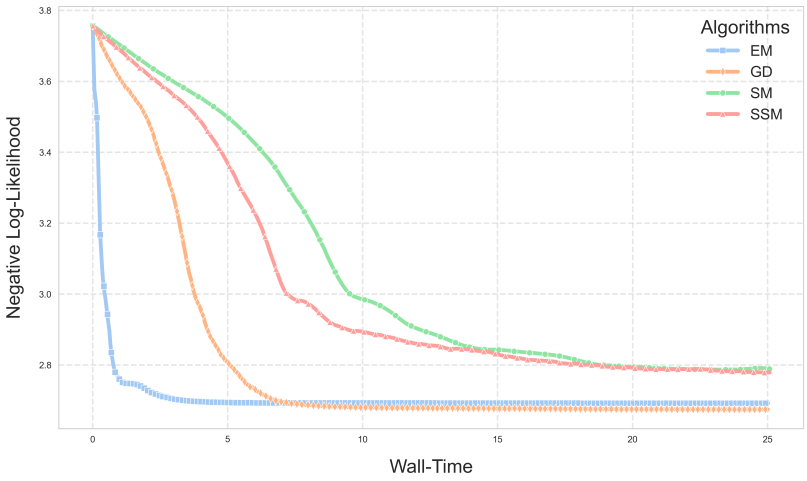
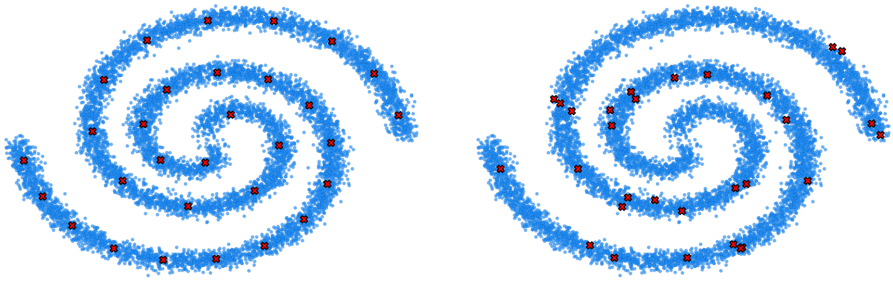


Figure B.2.: NLL over Time



(a) KMeans (b) Random

Figure B.3.: KMeans vs. Random Initialization of means

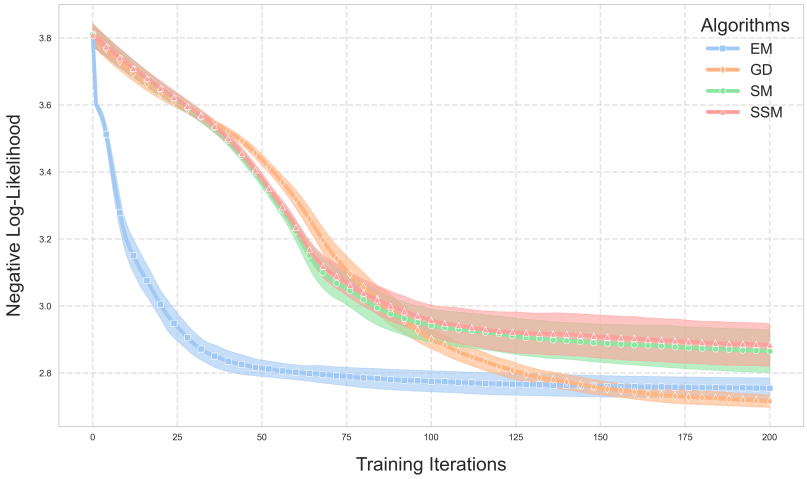


Figure B.4.: Negative Log Likelihood over Epochs with random parameter initialization

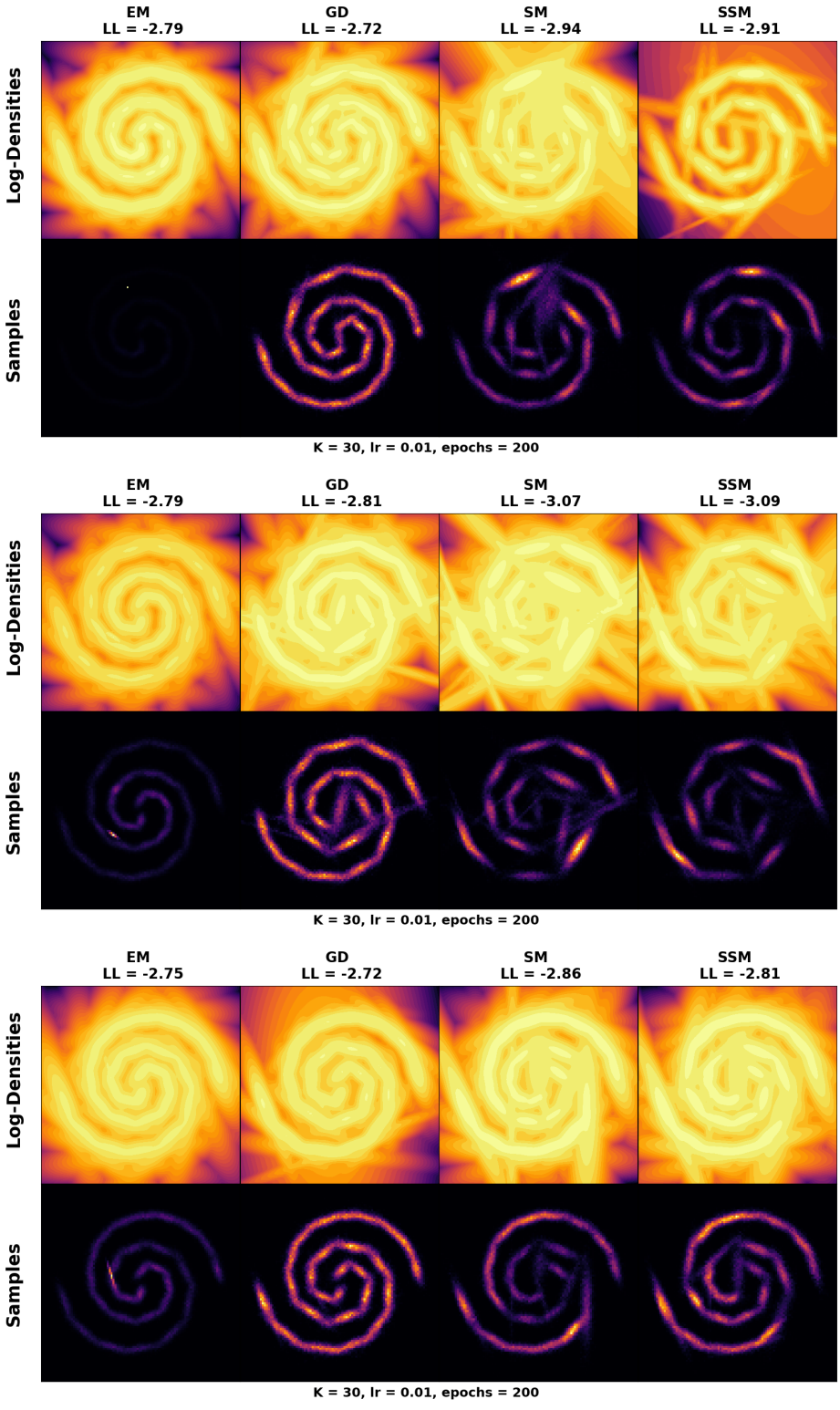


Figure B.5.: Densities and Samples for the spirals dataset

B.2. Halfmoons Random Initialization Results

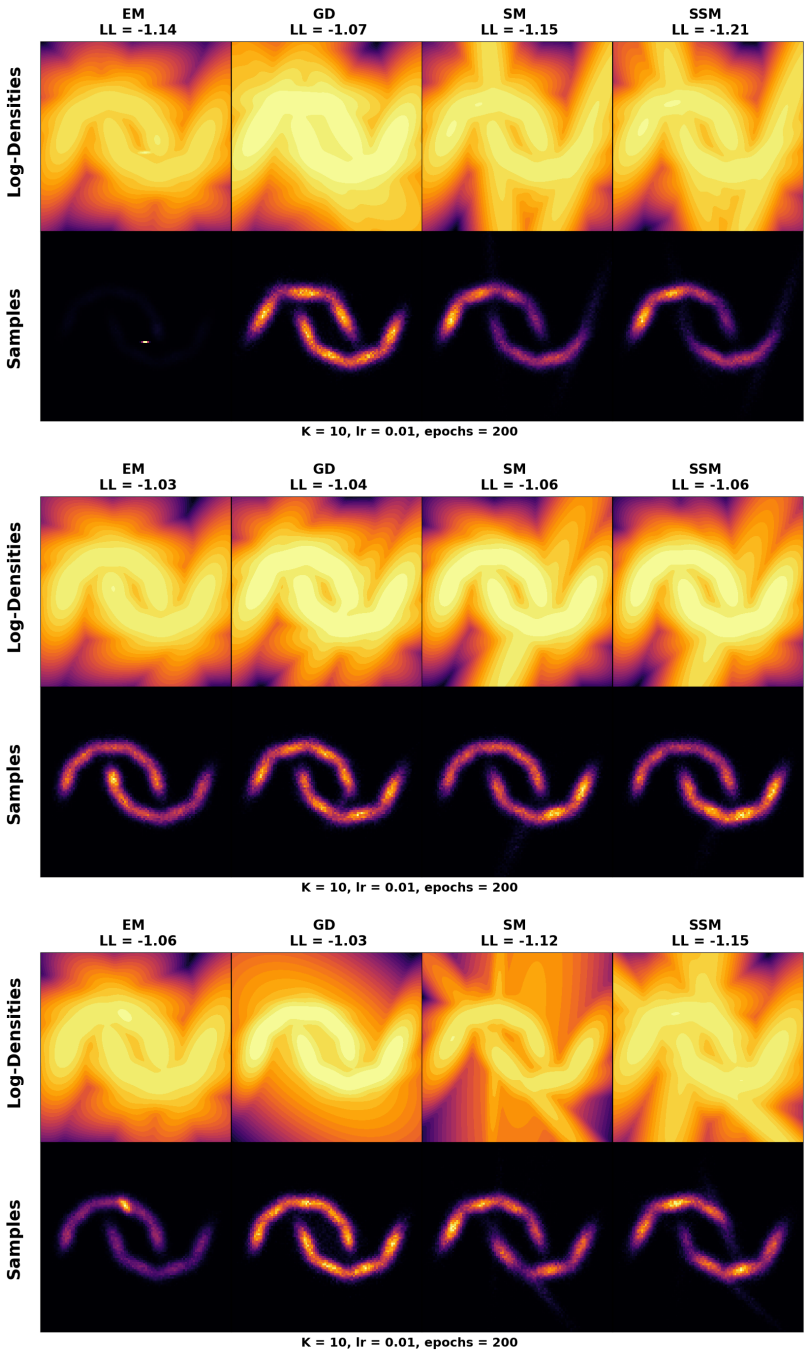
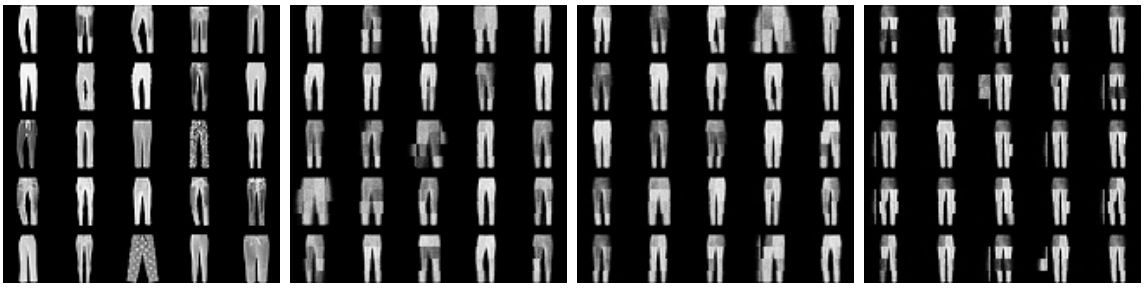


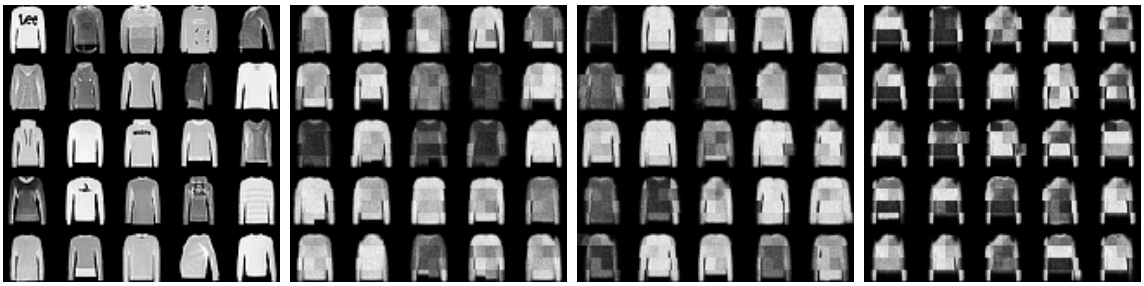
Figure B.6.: Densities and Samples for the halfmoon dataset

B.3. FashionMNIST



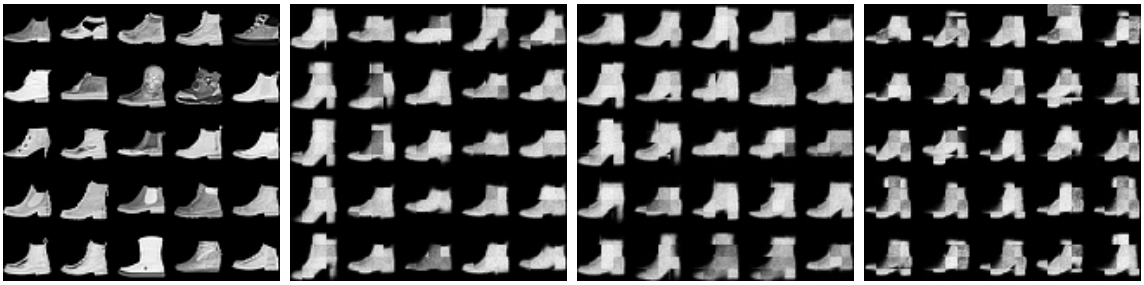
(a) Ground Truth (b) EM (LL: -8230) (c) SGD (LL: -8142) (d) SSM (LL: -10797)

Figure B.7.: Samples and Log-Likelihood of FashionMNIST-class 1



(a) Ground Truth (b) EM (LL: -11973) (c) SGD (LL: -12909) (d) SSM (LL: -17594)

Figure B.8.: Samples and Log-Likelihood of FashionMNIST-class 2



(a) Ground Truth (b) EM (LL: -11173) (c) SGD (LL: -11632) (d) SSM (LL: -15864)

Figure B.9.: Samples and Log-Likelihood of FashionMNIST-class 9

Bibliography

[1] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML]. URL: https://arxiv.org/abs/1406.2661 (cit. on pp. 1, 5).

[2] 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 pp. 1, 5).

[3] 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. 1–7).

[4] 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. 1, 8, 9).

[5] 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. 1, 9, 10).

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (cit. on pp. 4, 5, 14).

[7] 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. 6).

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

[9] 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. 6).

[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 pp. 17, 27).

[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. 27).