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

Probabilistic Circuits (PCs) have shown to be a promising approach to probabilistic modelling, due to their many effiecient and exact inference opportunities while still being complex enough to model arbitrary distibutions. However they still are difficult to train and lack in results compared to other state of the art approaches in probabilistic modelling, especially with large high dimensional datasets. (Researchers fear that during the gradient based Maximum Likelihood optimization the algorithm gets stuck in local minima.) In this thesis I try to experimentally see if by using other learning objectives, namely sliced score matching, which recently gained prominence in learning energy based models (EBMs), we can work around these issues and achieve better results with PCs. I do this by training PCs using different algorithms on 2D and also high dimensional image data and comparing results.

Contents

Abstract **iii**

1 Introduction **1**

2 Background **2**
2.1 Probabilistic Modelling 2
2.2 Probabilistic Circuits 3
2.3 Score Matching 5
2.4 Sliced Score Matching 7

3 Methods **8**
3.1 Creating a simple PC 8
3.2 Training via Score Matching 10
3.3 Training via Sliced Score Matching 10
3.4 Using more complex PCs 11

4 Experimental Results **12**
4.1 2D Density Esitmation 12
4.2 Images Density Estimation 14

5 Discussion **15**
5.1 Interpretation of Experimental Results 15

6 Conclusions and Future Work **16**

Bibliography **18**

1 Introduction

2 Background

2.1 Probabilistic Modelling

At it's very core Machine Learning (ML) aims to create algorithms/programs that learn from data in order to reason about it, make future predictions or perform all kinds of different inference tasks. One possible way to achieve 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 express this uncertainty and the rules of probability theory for inference.

To make this more clear let's say we have some two dimensional data, from variables X and Y and assume that this data was drawn randomly from some unknown distribution. We would call this distribution over all possible variables the joint distribution $P^*(X, Y)$. If the distribution $P^*(X, Y)$ was known to us all inference tasks would boil down to applying the basic rules of probability theory. The sum rule to compute marginals, the product rule to compute conditionals and more complex rules, derived from these two, to perform harder inference tasks.

However since we don't know $P^*(X, Y)$, we have to do something different. Through machine learning we create a framework that *models* $P(X, Y)$, approximating the unknown true distribution $P^*(X, Y)$, of which we only have limited samples. We would call a framework, that models the joint distribution a Generative Probabilistic Model. After *learning* this model trough the data, which is usually done via Maximum Likelihood Estimation, we can again perform inference using the rules of probability theory. And if the model approximates the true distribution close enough we can expect reliable results. [1].

TODO: explain MLE shortly

Before continuing with different approaches to probabilistic modelling, I want to introduce two key concepts that are often used to discuss these different methods.

TODO: expressiveness as in expressive efficiency

Definition 1: Expressiveness - a probabilistic model is called expressive if the learned distribution approximates the original distribution to a high degree. For instance this can be measured with Kullback-Leibler (KL) divergence [2], which measures how similar two probability distributions are. Two identical distributions would then have a KL-divergence of 0. In the Probabilistic Modelling case a KL-divergence of 0 would mean that the model is most exxpressive.

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 e.g. Monte-Carlo Simulation and efficiently means in linear time with respect to the model size.

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 in language modelling, they lack in tractability for all except the most 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 very simple data but can compute most if not all inference tasks tractably.

2.2 Probabilistic Circuits

Probabilistic Circuit (PC) 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], but I will only be focusing on Sum Product Networks (SPNs) [7], which to my knowledge has seen the widest adoption.

In General one can think of a SPN as a structured neural network that consists of leaf nodes, sum nodes and product nodes. A SPN 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.

Considering this, one could notice that the simplest form of a SPN is just a basic Gaussian Mixture Model (GMM), where one sum node mixes two leaf nodes, as depicted in Figure 2.1.

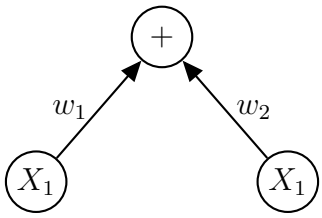

Figure 2.1: Simplest SPN

Furthermore an only slightly more complex SPN with a product node and two sum node can be seen in Figure 2.2.

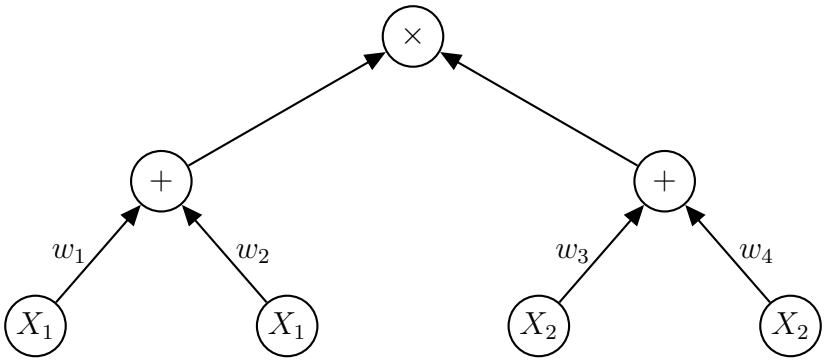

Figure 2.2: Simple SPN

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

In the following Definitions PC and SPN can be used interchangebly.

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 $\mathbf{X}$. [1]

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. [1]

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. [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 SPNs are smooth and decomposable.

There are more structural properties that a SPN or PC can fullfill, however these two already guarantee tractable computation of marginals (MAR) and conditionals (CON)[1], which many other models cannot do and which already is quite enough in many scenarios. Generally the more structure a PC has, which means 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 SPN 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 with 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, which is also very popular with Gaussian Mixture Models.

It is also noteworthy here that a SPN is a normalized model, which means it models a proper density (that integrates to 1 over the entire real space). This makes the omptimization with Maximum Likelihood very straightforward as the model directly outputs the Likelihood when evaluated at one datapoint. Therefore we can just minimize the negative of this output, in turn maximizing the Likelihood. Though normally, as with most models, we model the log-likelihood for numerical stability.

2.3 Score Matching

TODO: change notatioon of p and s

Score Matching [8] is a concept that is normally used when dealing with unnormalized models like Energy Based Models (EBMs). Here Energy typically just refurs to an unnormalized density. It is noteworthy, that most models represent unnormalized densities, because usually there is not a straightforward way to ensure that the model outputs a proper density. Learning these models through Maximum Likelihood Estimation can be difficult because of the computationally infeasible normalization constant, usually called Z_θ.

In Equation2.1 the relation between a parameterized density p_θ and an Energy E_θ is expressed. Notice that calculating the normalization constant would mean computing the integral over E_θ.

$$p_\theta(\mathbf{x}) = \frac{e^{-E_\theta(\mathbf{x})}}{Z_\theta} \tag{2.1}$$

Score Matching proposes a workaround by working with the log gradient $\nabla_x \log\ p_\theta(\mathbf{x})$ of the density, instead of $p_\theta(\mathbf{x})$.

Calculating $\nabla_x \log p_\theta(\mathbf{x})$ results in $-\nabla_x E_\theta(\mathbf{x})$ since

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

thus removing Z_θ.

$\nabla_x \log p_\theta(\mathbf{x})$ is called the score function giving score matching it's name. Using the score which I 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, defined as

$$J(\theta) = \frac{1}{2}\mathbb{E}_{p_d(x)}\left[\|s_\theta(x) - s_d(x)\|_2^2\right] \tag{2.2}$$

as the objective function. Here p_d and s_d refer to the density of the data and the score of the data respectively.

This objective doesn't depend on the intractable normalization constant, however it depends on the score function of the data $s_d(x)$ which is unknown. Furthermore in [8] it is shown that by partial integration this objective function can be expressed as

$$\begin{aligned}\mathcal{J}(\theta) &= \mathbb{E}_{p_d}\left[\operatorname{tr}\left(\nabla_x s_\theta(x)\right) + \frac{1}{2}\|s_\theta(x)\|^2\right] \\ &= \mathbb{E}_{p_d(x)}\left[\operatorname{tr}\left(\nabla_x^2 \log p_\theta(x)\right) + \frac{1}{2}\|\nabla_x \log p_\theta(x)\|^2\right]\end{aligned} \tag{2.3}$$

which is equivalent to 2.2 plus some additive constant. This final score matching objective doesn't depend on the score of the data s_d anymore and can be used for learning. Here tr() refers to the trace (sum of the diagonals) of the hessian matrix.

2.4 Sliced Score Matching

TODO: change notatioon of p and s

While Score Matching get's rid of the normalization constant Z_θ as seen above, it introduces another term that can become hard to compute. To calculate the trace of the hessian in Equation 2.3 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 like images.

In Sliced Score Matching (SSM) [9] the basic idea is to project the high dimensional data onto some 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 2.2:

$$\mathcal{J}(\theta; p_v) = \frac{1}{2}\mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\left(\mathbf{v}^T s_\theta(x) - \mathbf{v}^T s_d(x)\right)^2\right] \tag{2.4}$$

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:

$$\mathcal{J}(\theta; p_\mathbf{v}) = \mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\mathbf{v}^T \nabla_\mathbf{x} s_\theta(\mathbf{x})\mathbf{v} + \frac{1}{2}\left(\mathbf{v}^T s_\theta(\mathbf{x})\right)^2\right] \tag{2.5}$$

which is again equivalent to 2.4 plus some additive constant [9]. This objective can be used for learning, 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 2.5 can be integrated analytically [9], yielding the following objective.

$$\mathcal{J}(\theta; p_\mathbf{v}) = \mathbb{E}_{p_v}\mathbb{E}_{p_d}\left[\mathbf{v}^T \nabla_\mathbf{x} s_\theta(\mathbf{x})\mathbf{v} + \frac{1}{2}\|s_\theta(\mathbf{x})\|_2^2\right] \tag{2.6}$$

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 as in 2.5.

3 Methods

In this thesis I 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 I compare results from these methods with results from convential methods to train PCs.

3.1 Creating a simple PC

Recall from section 2.2 that the simplest variant a PC computes the weighted sum of arbitrary many input distributions. This is called a Mixture Model and the graph of an example with just two input distributions is shown in Figure 2.1.

If the input distributions are Gaussian then it would be called a Gaussian Mixture Model and the expression to calculate the modelled density would be the following:

$$p(\mathbf{x}) = \prod_{s=1}^{S} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{3.1}$$

Where S is the number of samples, K is the number of components (distributions), π_k the mixture weights and $\mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ a Gaussian component with mean $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$.

However since we need to model the log-density $\log p(\mathbf{x})$ for numerical stability we need to take the log of expression 3.1:

$$\begin{aligned} \log p(\mathbf{x}) &= \log \left(\prod_{s=1}^{S} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right) \\ &= \sum_{s=1}^{S} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_s | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right) \end{aligned} \tag{3.2}$$

Here the problem arises that calculating $\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_s|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ can produce very small values and when taking the logarithm this can lead to numerical instability but it can be mitigated with the LogSumExp trick. First we take the exponential of the logarithm of the term inside the second sum.

$$\begin{aligned}&\sum_{k=1}^{K} \exp\left(\log\left(\pi_k \mathcal{N}(\mathbf{x}_s|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)\right) = \\ &\sum_{k=1}^{K} \exp\left(\log(\pi_k) + \log(\mathcal{N}(\mathbf{x}_s|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))\right)\end{aligned} \tag{3.3}$$

This is valid, because $\exp(\log(x)) = x$, since exponential and logarithm cancel each other out. Plugging 3.3 back into Equation 3.2 yields the final numerically stable log density for our model.

$$\log p(\mathbf{x}) = \sum_{s=1}^{S} \log \sum_{k=1}^{K} \exp\left(\log \pi_k + \log \mathcal{N}(\mathbf{x}_s|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right) \tag{3.4}$$

I implemented this expression in python and used pytorch's 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 to calculate $\log \sum_{k=1}^{K} \exp()$.

Now to train this model on some data we need to formulate a objective function and the corresponding optimization problem. In the conventional way of Maximum Likelihood Estimation (MLE) this is quite straight forward, since the objective function is exactly the density or in our case log density function $\log p(\mathbf{x})$ and the optimisation problem is formulated as the following:

$$\max_{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}} \log p(\mathbf{x}) = \sum_{s=1}^{S} \log \sum_{k=1}^{K} \exp\left(\log \pi_k + \log \mathcal{N}(\mathbf{x}_s; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right) \tag{3.5}$$

In plain text this means maximizing $\log p(\mathbf{x})$ with respect to the weights $\boldsymbol{\pi}$, the means $\boldsymbol{\mu}$ and the covariance matrices $\boldsymbol{\Sigma}$.

I implemented the optimization of this objective with Expectation Maximization (EM) and Gradient Descent using pytorch's automatic differentiation capabilities. Note that for Gradient Descent to work we instead need to minimize the negative log likelihood $-\log p(\mathbf{x})$, but this yields the same results as maximizing the log likelihood.

3.2 Training via Score Matching

To train the simple PC from above using Score Matching recall the objective function 2.3 from Section 2.3. With it we can formulate the optimization problem:

$$\max_{\boldsymbol{\theta}} \mathcal{J}(\theta) = \mathbb{E}_{p_d(x)} \left[\operatorname{tr}\left(\nabla_x^2 \log p_\theta(x) \right) + \frac{1}{2} \left\| \nabla_x \log p_\theta(x) \right\|^2 \right] \tag{3.6}$$

Here $\log p_m(.;\theta)$ is of course the log density of our model as described in 3.4.
I used pytorch for the optimization with Gradient Descent.

The exact step-by-step way I calculated $\mathcal{J}(\theta)$ can be seen in Algorithm 1.

Algorithm 1 Score Matching

Input: $\log p_m(.;\theta), \mathbf{x}$

1: $s_m(\mathbf{x};\theta) \leftarrow \nabla_x \log p_m(\mathbf{x};\theta), \mathbf{x}$
2: $\mathcal{J} \leftarrow x$
3: **return** $\mathcal{J}$

To calculate the gradients as in $\operatorname{grad}(\log p_m(\mathbf{x};\theta), \mathbf{x})$ I used torch.autograd.grad.

3.3 Training via Sliced Score Matching

Similar to Section 3.2 we take the objective function 2.6 from Section 2.4 and formulate the optimization problem:

$$\min_{\boldsymbol{\theta}} \mathcal{J}(\theta; p_{\mathbf{v}}) = \mathbb{E}_{p_v} \mathbb{E}_{p_d} \left[\mathbf{v}^T \nabla_{\mathbf{x}} s_\theta(\mathbf{x}) \mathbf{v} + \frac{1}{2} \left\| s_\theta(\mathbf{x}) \right\|_2^2 \right] \tag{3.7}$$

I again used pytorch for optimization with Gradient Descent and the exact computation of $\mathcal{J}(\theta; p_{\mathbf{v}})$ can be seen in Algorithm 2.

Algorithm 2 Sliced Score Matching

Input: $\log p_m(.;\theta), \mathbf{x}$

1: $s_m(\mathbf{x};\theta) \leftarrow \nabla_x \log p_m(\mathbf{x};\theta), \mathbf{x}$
2: $\mathcal{J} \leftarrow x$
3: **return** $\mathcal{J}$

3.4 Using more complex PCs

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

So to get a wider variety of results I decided to also use a state-of-the-art framework for Probabilistic Circuits called EinsumNetworks [10] for modelling more complex higher dimensional datasets, especially images.

For more details on how EinsumNetworks work please refer to the [10] but in principle

4 Experimental Results

4.1 2D Density Esitmation

For the GMM I used three functions that generate two dimensional data in some specific pattern with some added random noise. These can be seen as the true data generating distribution and the goal is to model this distribution using the GMM with the discussed algorithms.

Using these I generated 20,000 datapoints and split them evenly into a train and a test dataset. Note that usually the test dataset is smaller than the training dataset due to limited data availability but since we can basically generate unlimited samples if we know the data generating distribution, I decided to split them in this way. The training samples can be seen in Figure 4.1.

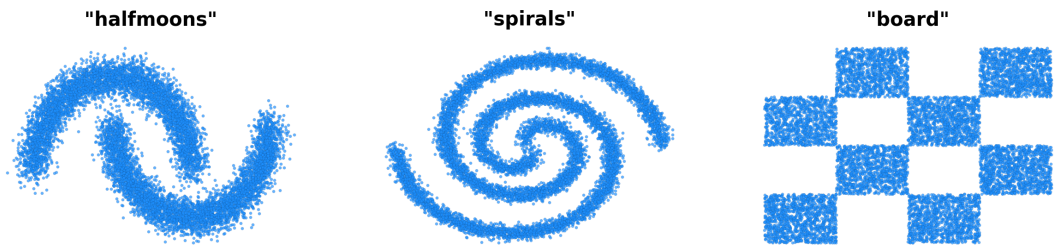

Figure 4.1: 2D Samples for all three distributions

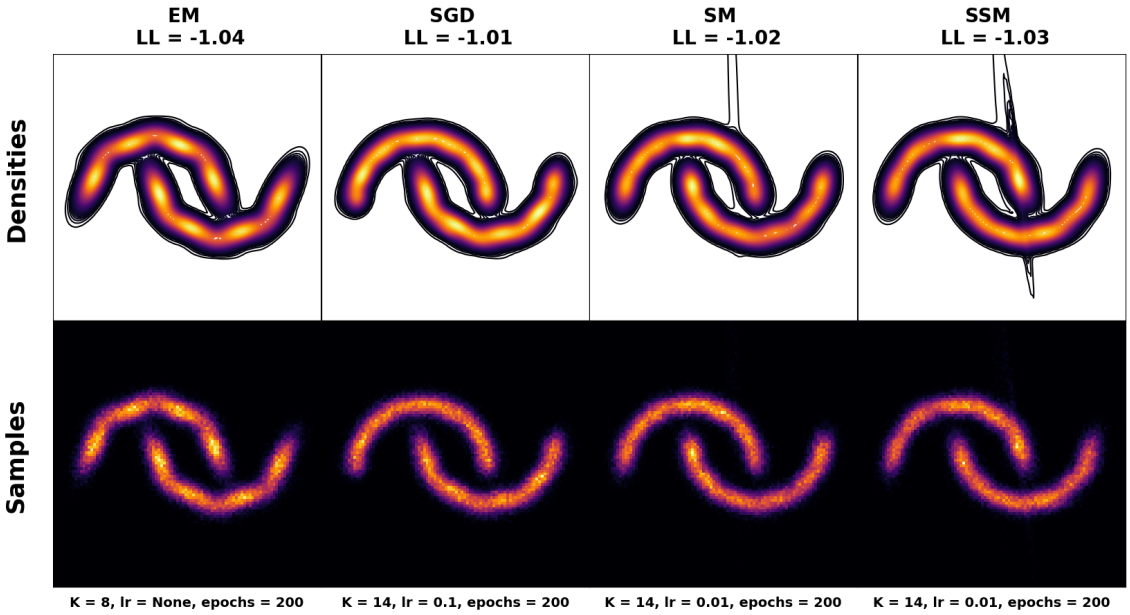

Figure 4.2: Densities and Samples for "Halfmoons" with best hyperparameters for each algorithm

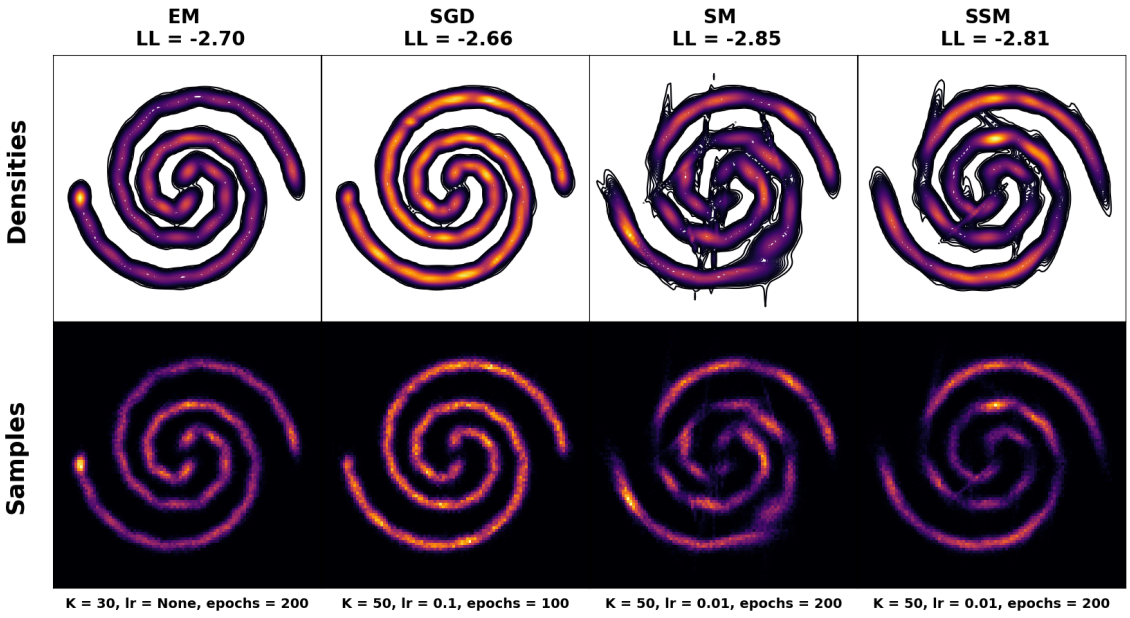

Figure 4.3: Densities and samples for "Spirals" with best hyperparameters for each algorithm

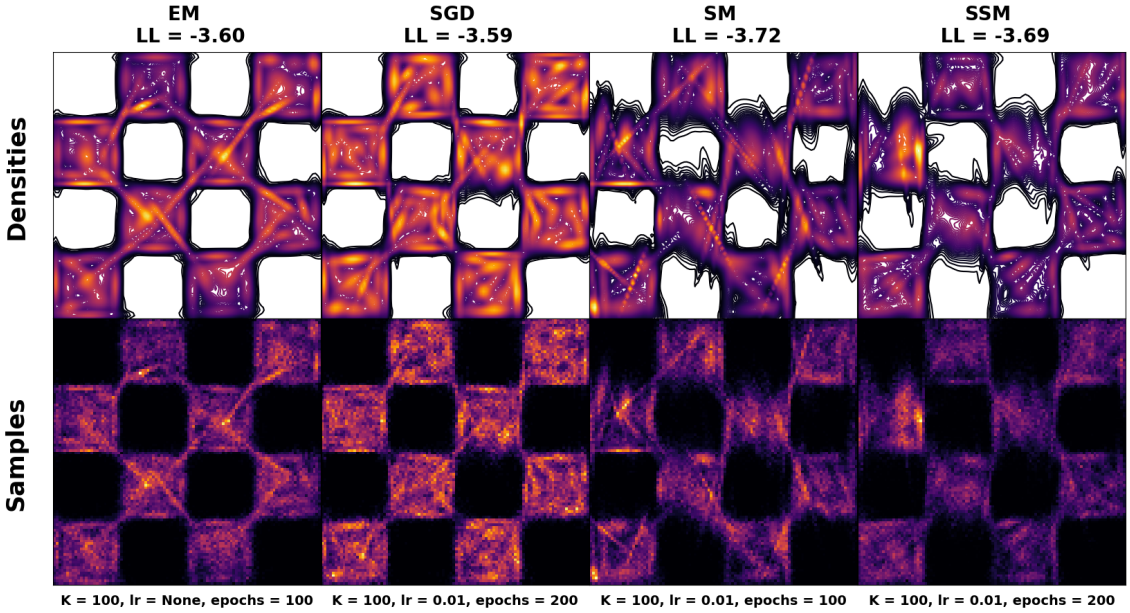

Figure 4.4: Densities and samples for "Board" with best hyperparameters for each algorithm

4.2 Images Density Estimation

I used the MNIST Dataset [11]

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

Figure 4.5: MNIST Samples

5 Discussion

5.1 Interpretation of Experimental Results

6 Conclusions and Future Work

Appendix

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. 2–5).

[2] S. Kullback and R. A. Leibler. "On Information and Sufficiency." In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. ISSN: 00034851. URL: http://www.jstor.org/stable/2236703 (visited on 09/12/2024) (cit. on p. 3).

[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. 3).

[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. 3).

[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. 3).

[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. 3).

[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. 3).

[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. 5, 6).

[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 p. 7).

[10] 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. 11).

[11] 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. 14).