university of groningen / faculty of science and engineering

# Hybrid Continuous Mixtures of Probabilistic Circuits


| Student: | First supervisor: | Second supervisor: |
|---|---|---|
| Dewi Batista | Prof. M. Grzegorczyk | Prof. H. Jaeger |



**Abstract**

We offer a tutorial-like overview of probabilistic circuits (PCs), a class of generative probabilistic models which offer tractable means of answering evidence and marginal queries exactly. A motivation of their application to tractable probabilistic inference with complete or missing data is then presented. An overview of continuous mixtures of PCs (CMPCs) — appropriately-weighted latent variable mixture models in which latent variables are continuously distributed and PCs are fit to each component — is then given and a review of their application to density estimation and sampling is offered. As of the time of writing, the resources from which one can learn about PCs consists of a series of independent papers whose notation, style of writing and assumed prior knowledge differ significantly. As for CMPCs, there is only one such source: the original paper by Correia et al. [9] in which they were introduced. As such, the tutorial-like overviews of PCs and CMPCs given in this work are two of its intended contributions. We follow by introducing hybrid CMPCs which are CMPCs trained in a manner which encourages both generative and discriminative learning. The extent to which either learning paradigm is encouraged during training is dictated by a mixing hyperparameter $\lambda \in [0, 1]$. Typically, deep learning models are trained under a single paradigm as few model classes facilitate a hybrid learning objective. To investigate the trade-off of generative and discriminative power as the mixing hyperparmeter varies, we train hybrid CMPCs for $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ on Binary MNIST. We evaluate two complementary criteria of each hybrid CMPC: its classification accuracy on incomplete samples of Binary MNIST and the visual quality of samples drawn from the model. Our experiments demonstrate that hybrid CMPCs trained with intermediate values of $\lambda \in [0.4, 0.6]$ yield an effective balance, obtaining high classification accuracies while maintaining reasonable sample quality.


**Master's Thesis**
**Applied Mathematics**

# Contents

# 1 Introduction

In modelling real-world systems, an often-important criterion is the extent to which the considered model class facilitates uncertainty. A well-known example of such a real-world system, whose method of handling uncertainty predates modern methods, is MYCIN [41]: an expert system developed in the 1970s to assist medical practitioners in treating bacterial infections. A common situation in the setting of treating such infections is that the symptoms reported by a patient may be vague, incomplete or entirely missing. Additionally, distinct infections may be present with overlapping symptoms making it difficult to isolate a specific cause. To diagnose such a patient, even in the face of said ambiguities, MYCIN would compute certainty factors: a heuristic rule-of-thumb measure assigning strength to the hypothesis that the patient had a particular infection. While impressive for its time, MYCIN was ultimately never deployed commercially for a myriad of reasons. One such reason was scepticism toward its ad-hoc approach to handling uncertainty. This is in contrast to later methods grounded in well-studied frameworks, such as fuzzy logic approaches based on fuzzy set theory [45] and probabilistic approaches grounded in probability theory. Of these, the approach of particular interest to us is probabilistic modelling, to which probabilistic circuits (PCs) belong.

An important advent of probabilistic modelling was the development of Bayesian networks by Judea Pearl in the mid-1980s. Bayesian networks offer mathematically-grounded means of dealing with uncertainty and, importantly for comparison in our work, are capable of encoding complex distributions. While certainly a significant step in the history of probabilistic modelling, Roth [38] showed in 1996 that answering marginal queries exactly using Bayesian networks is in general #P-hard, demonstrating that their application to probabilistic inference has a clear and unfortunate limit. In line with this, efforts were made to develop probabilistic models capable of both encoding complex distributions sufficiently-well and efficiently answering typically-intractable probabilistic queries such as marginals. A consequence of this effort was the introduction of sum-product networks (SPNs) in 2011 [35] which were renamed to probabilistic circuits (PCs) in 2020 [6].

Since their inception, methods for learning PCs from data have made reasonable progress but have been overshadowed by many rapidly improving approaches in deep learning for common tasks such as sampling, density estimation and classification, to name a few. In attempt to remedy this, many contemporary methods of learning PCs from data involve the incorporation of ideas from deep learning-based approaches, such as PC architectures which in some way include convolutions [4], variational autoencoders [9] or transformers [26]. Of interest to us is work by Correia et al. [9] which details a variational autoencoder-inspired method of learning PCs from data, purely for generative purposes (sampling and density estimation), by utilising Monte Carlo-style approximations. In adapting their work, we seek to learn PCs in a similar manner of Monte Carlo-style approximations but for a hybrid of purposes, encouraging the learning of PCs with generative capabilities, as in the original work, and discriminative capabilities for classification. This is done by utilising a

hybrid loss function during training, inspired by [3], in which both generative and discriminative learning is encouraged. The extent to which either is encouraged is determined by a mixing hyperparameter $\lambda \in [0, 1]$ as in

$$\mathcal{L}_\lambda(\phi) = \lambda \text{CE}(\phi) + (1 - \lambda)\text{NLL}(\phi)$$

where $\phi$ denotes the parameters of the model, $\text{CE}(\phi)$ is a measure of discriminative loss and $\text{NLL}(\phi)$ is a measure of generative loss.

The intended contributions of this work are three-fold: 1) to offer a tutorial-like overview of probabilistic circuits (PCs), 2) to offer an overview of continuous mixtures of probabilistic circuits (CMPCs) and 3) to experiment with the hybrid learning of CMPCs. As of now, there exist no beginner-friendly overviews of probabilistic circuits and so one must piece together their understanding from multiple papers which differ in notation, writing style and assumed prior knowledge. Similarly, the only source one has available to learn about continuous mixtures of probabilistic circuits, by Correia et al. [9], assumes reasonable prior knowledge about probabilistic circuits and areas of deep learning. As such, contributions 1) and 2) seek to remedy this.

In Section 2, a series of preliminaries needed to offer an overview of PCs are given, including the tractability, expressivity and expressive-efficiency of classes of probabilistic models. Following this, PCs are overviewed in Section 3 along with a tutorial-like example of using them to answer standard probabilistic queries. Then, a brief list of the applications of PCs to well-known tasks, such as anomaly and out-of-distribution (OOD) detection, sampling, classification and in-painting is given. In Section 4, we detail the variational autoencoder-inspired method of learning PCs by Correia et al. [9] which yields continuous mixtures of probabilistic circuits. Our adaptations to the work of Correia et al. are given in Section 5, including benchmarks of their application to Binary MNIST. A discussion section and concluding remarks follow.

# 2 Preliminaries

To provide a sufficiently-detailed overview of probabilistic circuits, we first lay out the relevant preliminary theory. We begin by introducing generative probabilistic modelling followed by notions of the tractability, expressivity and expressive-efficiency of a class of probabilistic models.

## 2.1 Generative Probabilistic Modelling

Probabilistic modelling is a framework for representing and reasoning about uncertainty in systems involving random variables and their interplay. It allows one to reason meaningfully in a probabilistic manner in contexts where uncertainty is inherent, data is incomplete or noisy, and complex relationships between variables is to be understood and quantified.

For a formal definition of a generative probabilistic model consider Definition 2.1.

**Note:** To avoid having to write *probability mass/density function* in full throughout this work, we will write *probability function*.

**Definition 2.1** *In line with Choi et al. [6, Subsection 2.1], a generative probabilistic model* $(\mathbf{X}, p_\Theta)$ *consists of a finite set* $\mathbf{X} = \{X_1, \ldots, X_n\}$ *of random variables and a representation* $p_\Theta : \Omega_{\mathbf{X}} \to \mathbb{R}_{\geq 0}$ *of the joint probability function on the sample space* $\Omega_{\mathbf{X}} := \Omega_{X_1} \times \cdots \times \Omega_{X_n}$ *of* $\mathbf{X}$ *parameterised by a parameter set* $\Theta$.

Note that we do not require the parameter set $\Theta$ of a probabilistic model $(\mathbf{X}, p_\Theta)$ to consist only of real numbers as it is often helpful to allow for the model's representation $p_\Theta$ of the underlying joint probability function to correspond to richer structures such as graphs, as in Bayesian networks, Markov random fields and probabilistic circuits. This definition is rather broad but such flexibility proves rewarding.

The holy grail of generative probabilistic modelling is to find an exact, efficiently computed and algebraically manipulable representation of the underlying joint probability function of interest. If obtained, one can reason probabilistically with ease but, as we will see, the purpose, limitations and learning of generative probabilistic models varies heavily. To illustrate this, consider probabilistic graphical models (PGMs) which offer a graph structure in which conditional dependencies between modelled random variables are expressed in a compact and human-interpretable manner. The most well-known class of PGMs are Bayesian networks, as defined in Definition 2.2.

**Definition 2.2** *A Bayesian network* $(\mathbf{X}, p_{(\mathcal{G},\theta)})$ *is a generative probabilistic model whose parameter set consists of a directed acyclic graph (DAG)* $\mathcal{G}$ *in which there is a single node for each random variable in* $\mathbf{X} = \{X_1, \ldots, X_n\}$ *and directed edges represent conditional dependencies between these variables. The parameter set* $\theta$ *specifies the conditional probability distributions associated with each variable given its parents in* $\mathcal{G}$.

As to how a Bayesian network $(\mathbf{X}, p_{(\mathcal{G},\theta)})$ offers a representation of the underlying joint probability function over $\Omega_{\mathbf{X}}$, Bayesian networks satisfy the local Markov property: each random variable $X_i \in \mathbf{X}$ is independent of its non-descendants in $\mathcal{G}$ given its parents. That is, with informal/relaxed notation for brevity, $\mathbb{P}(X_i|\mathbf{X}\backslash\{X_i\}; \mathcal{G}, \theta) = \mathbb{P}(X_i|\mathbf{pa}(X_i); \mathcal{G}, \theta)$ for $i = 1, \ldots, n$ where $\mathbf{pa}(X_i)$ denotes the set of nodes in $\mathcal{G}$ with an outgoing edge to $X_i$, often referred to as the parents of $X_i$. With this in mind, a Bayesian network $(\mathbf{X}, p_{(\mathcal{G},\theta)})$ can be described as a generative probabilistic model in which the parameters are as described above and the representation of the joint probability function $p_{(\mathcal{G},\theta)} : \Omega_{\mathbf{X}} \to \mathbb{R}_{\geq 0}$ is given by

$$p_{(\mathcal{G},\theta)}(\mathbf{x}) = \prod_{i=1}^{n} \mathbb{P}(x_i|\mathbf{pa}(X_i); \mathcal{G}, \theta).$$

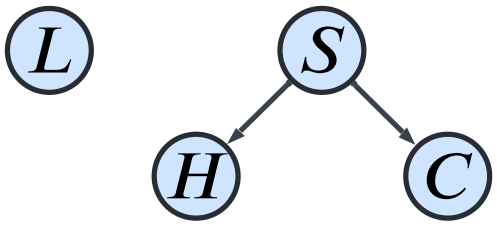


**Figure 1:** A Bayesian network pertaining to patients of a medical practitioner in which modelled variables consist of whether or not the patient is left handed (L), smokes (S), has heart disease (H) and has lung cancer (C).

How one learns a Bayesian network from data — that is, its DAG $\mathcal{G}$ and parameter set $\theta$ — is detailed in [23, chapter 18]. To demonstrate how a Bayesian network might be employed in practice, consider Example 2.1.

**Example 2.1** *Suppose a medical practitioner has amassed a dataset pertaining to her patients including whether or not they smoke* $(S)$*, have heart disease* $(H)$*, have lung cancer* $(C)$ *and are left handed* $(L)$*. She would like to know how these mostly health-related factors influence one another, if at all, and so she constructs a Bayesian network, whose DAG is illustrated in Figure 1, from her dataset. The medical practitioner sees immediately that left handedness has no influence on, and is not influenced by, any of the other factors. More interestingly, the structure would suggest that the medical practitioner's confidence/belief that a patient has lung cancer given that they smoke should be unchanged upon knowing that the patient has heart disease. That is,* $\mathbb{P}(C|S,H) = \mathbb{P}(C|S)$*. Further, the medical practitioner is able to obtain the representation*

$$p_{(\mathcal{G},\theta)}(L,S,H,C) = \mathbb{P}(L)\mathbb{P}(S)\mathbb{P}(H|S)\mathbb{P}(C|S)$$

*of the underlying joint probability function over the random variables* $L, S, H$ *and* $C$*.*

It should be noted that Bayesian networks are not the only example of a prominent generative probabilistic model. The reason that we briefly consider Bayesian networks here is that they serve as a friendly and intuitive introduction to generative probabilistic models. Additionally, the core makeup of a Bayesian network being a directed acyclic graph and a parameter set lends nicely to the core makeup of the class of generative probabilistic models introduced in Section 3: probabilistic circuits (PCs).

Following this, a natural question arises: when is employing the class of probabilistic circuits, as opposed to the class of Bayesian networks, or any other class of probabilistic models, beneficial? In addressing this question, it helps to understand the tractability, expressivity and expressive-efficiency of classes of probabilistic models, which are considered in the remainder of this section.

## 2.2 Tractability

The tractability of a class of probabilistic models is the extent to which it offers computationally efficient means of performing probabilistic inference. Given a probabilistic model $(\mathbf{X}, p_\Theta)$, probabilistic inference is the act of answering probabilistic queries using $p_\Theta$. Exact probabilistic inference is simply probabilistic inference without approximation. A list of common probabilistic queries, which often arise naturally in performing inference on real-world systems, are given in Definitions 2.3, 2.4, 2.5, 2.6 and 2.7.

**Definition 2.3** *A full evidence query* (**evi**) *is the computation of* $p_\Theta(\mathbf{x})$ *for a realisation* $\mathbf{x} \in \Omega_\mathbf{X}$.

**Definition 2.4** *A marginal query* (**marg**) *is the computation of* $p_\Theta(\mathbf{x}')$ *for a partial realisation* $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ *with* $\mathbf{X}' \subset \mathbf{X}$.

**Definition 2.5** *A conditional query* (**cond**) *is the computation of* $p_\Theta(\mathbf{x}'|\mathbf{x}'')$ *for partial realisations* $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ *and* $\mathbf{x}'' \in \Omega_{\mathbf{X}''}$ *with* $\mathbf{X}', \mathbf{X}'' \subset \mathbf{X}$ *such that* $\mathbf{X}' \cap \mathbf{X}'' = \emptyset$. *Note that if a class of probabilistic models is tractable with respect to* **evi** *and* **marg** *queries then it is tractable with respect to* **cond** *queries since* $p_\Theta(\mathbf{x}'|\mathbf{x}'') = p_\Theta(\mathbf{x}', \mathbf{x}'')/p_\Theta(\mathbf{x}'')$ *requires answering a single* **evi** *query and a single* **marg** *query.*

**Definition 2.6** *A maximum a posteriori query* (**MAP**) *is the computation of the partial realisation* $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ *which maximises* $p_\Theta(\mathbf{x}'|\mathbf{x}'')$ *for a partial realisation* $\mathbf{x}'' \in \Omega_{\mathbf{X}''}$ *where* $\mathbf{X}', \mathbf{X}'' \subset \mathbf{X}$ *such that* $\mathbf{X}' \cap \mathbf{X}'' = \emptyset$ *and* $\mathbf{X}' \cup \mathbf{X}'' = \mathbf{X}$. *That is, computing* $\arg\max_{\mathbf{x}' \in \Omega_{\mathbf{X}'}} p_\Theta(\mathbf{x}'|\mathbf{x}'')$.

**Definition 2.7** *A marginal maximum a posteriori query* (**MMAP**) *is the computation of the partial realisation* $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ *which maximises* $p_\Theta(\mathbf{x}'|\mathbf{x}'')$ *for a partial realisation* $\mathbf{x}'' \in \Omega_{\mathbf{X}''}$ *where* $\mathbf{X}', \mathbf{X}'' \subset \mathbf{X}$ *such that* $\mathbf{X}' \cap \mathbf{X}'' = \emptyset$ *and* $\mathbf{X}' \cup \mathbf{X}'' \subset \mathbf{X}$.

With definitions of the probabilistic queries of interest out of the way, we define the tractability of a class of probabilistic models with respect to a class of probabilistic queries in Definition 2.8

**Definition 2.8** *A class of probabilistic models* $\mathcal{M}$ *is tractable with respect to a class of probabilistic queries* $\mathcal{Q}$ *if all queries* $q \in \mathcal{Q}$ *can be answered with complexity* $\mathcal{O}(\text{poly}(|m|))$ *for all models* $m \in \mathcal{M}$.

Here, $|m|$ denotes the 'size' of a model $m \in \mathcal{M}$ and is dependent on the class $\mathcal{M}$ at hand. For example, the size of a Bayesian network $(\mathbf{X}, p_\Theta)$ is determined by the size of its factors [23, Definition 4.1], e.g. the sizes of the conditional probability tables (if over discrete random variables) it encodes. For a probabilistic circuit, which has a computational graph structure, its size is determined by the number of edges in its computational graph. As

will be shown later on, the class of probabilistic circuits is tractable with respect to **evi**, **marg** and **cond** queries but not **MAP** and **MMAP** queries.

Put into perspective, it would be only natural for the medical practitioner in Example 2.1 to want to use a probabilistic model to answer **marg** queries in which left handedness is not present as it has no influence on, and is not influenced by, any other factors. For the sake of illustrating a more sophisticated query, she may be interested in finding the partial realisation which maximises the marginal distribution consisting of $S$, $C$ and $H$ alone as left-handedness is seemingly irrelevant to her interests. While this may be feasible in the case of Example 2.1 due to its small scale, to the medical practitioner's disappointment, it is not computationally feasible in general using Bayesian networks: within the framework of Bayesian networks, **evi** queries are linear in the size of the network but **marg** queries are #P-hard [38]. Similar challenge-inducing computational complexities follow for **cond**, **MAP** and **MMAP** queries. That is, Bayesian networks are not tractable with respect to **marg**, **cond**, **MAP** or **MMAP** queries.

Given these limitations, it is only natural to explore classes of probabilistic models which are tractable with respect to **marg** and **cond** queries, which includes the class of probabilistic circuits (PCs) [35]. For an overview of which (non-)probabilistic models are tractable with respect to the probabilistic queries defined earlier, consider Table 1.

| **Model \Probabilistic query** | **sampling** | **evi** | **marg** | **cond** | **MAP** | **MMAP** |
|---|---|---|---|---|---|---|
| GANs [29] | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Bayesian networks (BNs) [38] | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Probabilistic circuits [6] | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Tree-shaped BNs [23] | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Fully-factorised models | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

**Table 1:** A variety of generative (non-)probabilistic models and their tractability. 'GANs' is an abbreviation of generative adversarial networks, a class of deep generative models.

### 2.3 Expressivity and Expressive-Efficiency

We see in Table 1 that the class of probabilistic circuits is tractable with respect to classes of probabilistic queries which the class of Bayesian networks is not, namely **marg** and **cond** queries. Similarly, the classes of fully-factorised models and tree-structured Bayesian networks are both tractable with respect to **MAP** queries while PCs are not. When shown this, a natural question is why one would not always employ fully-factorised models in favour of probabilistic circuits or Bayesian networks. To answer this, like all good things in machine learning, developing an appropriate model is a balancing act. Roughly put, increased tractability comes with undesirable deficits. In line with this, of interest to our work are classes of generative probabilistic models which strike a balance between tractability and the extent to which they can capture underlying distributions from data,

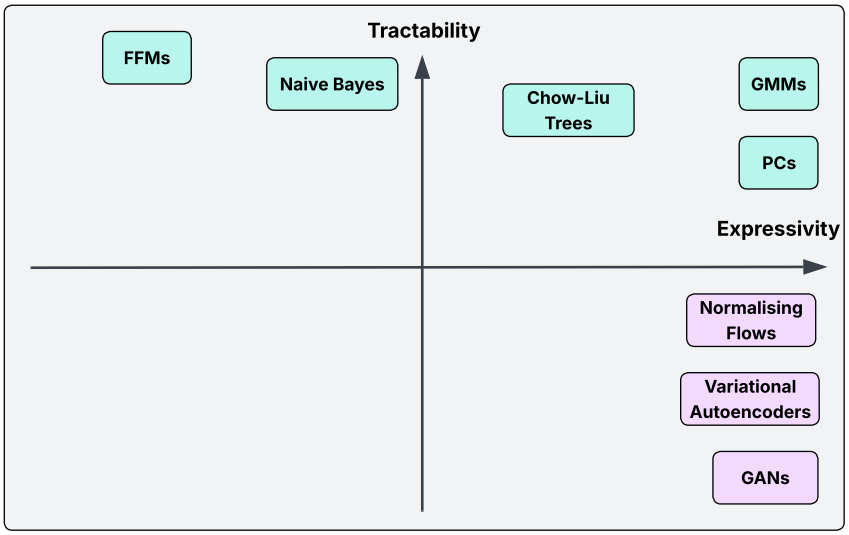


**Figure 2:** The expressivity-tractability spectrum of generative (non-)probabilistic models. Probabilistic models in light blue. Deep generative models in purple.

further referred to as the expressivity of the probabilistic model class in question. For a crude illustration of where some well-known generative (non-)probabilistic models lie on the expressivity-tractability spectrum, consider Figure 2. Note that this illustration is not an analytic illustration of where these models lie: it is purely empirically-motivated.

To see why balancing expressivity and tractability is crucial to generative probabilistic modelling, consider fully-factorised models (FFMs), illustrated in the top left of Figure 2, which assume independence between model variables:

$$p_\Theta(\mathbf{x}) = \prod_{i=1}^{n} p(x_i).$$

Such models are highly tractable — answering **evi**, **marg**, **cond**, **MAP** and **MMAP** queries is straightforward — but their simplifying assumption of independence between model variables is often far too simplistic. Consequently, fully-factorised models trade off high tractability for low expressivity. Toward the other end of the spectrum, variational autoencoders (VAEs), which are not generative probabilistic models in the sense alluded to by Definition 2.1 but are generative models nonetheless, typically do an excellent job of encoding underlying distributions from data given suitably large datasets. That is, they offer excellent expressivity. Their deficits lie in that they offer only tractable sampling and so can only be used for the generative task of sampling. A well-trained VAE can offer very

convincing samples.

The significance of PCs is that they hit a sweet spot on the expressivity-tractability spectrum. They are not quite as expressive as variational autoencoders — and so sampling from them will not yield similarly convincing samples — but they offer far better tractability. Similarly, PCs do not offer the same level of tractability as fully-factorised models — and so their application to probabilistic inference is more limited — but they offer far higher expressivity.

To add to this, tractability and expressivity alone are not enough: the expressive-efficiency of model classes is also important. The expressive-efficiency of model classes can be compared in the number of components in their representation $p_\Theta$ needed to encode an underlying distribution to a given level of precision. For example, it is known that Gaussian mixture models (GMMs) satisfy a universal distribution approximation theorem [18, Page 67]: any continuous density can be approximated arbitrarily well by a GMM with sufficiently many components. As such, GMMs offer perfect expressivity. That said, in practice, they require an infeasible number of components to fit complex distributions to reasonable degrees of precision. That is, their expressivity is perfect but their expressive-efficiency is low in the context of complex distributions. To contrast this, PCs offer the same level of expressivity and tractability but higher expressive-efficiency, making them an attractive probabilistic model class when fitting complex distributions. Admittedly, the expressive-efficiency of PCs is not well-studied and conclusions drawn surrounding their expressive-efficiency are mostly empirically-motivated.

# 3 Probabilistic Circuits (PCs)

Probabilistic circuits (PCs) [35] belong to the class of generative probabilistic models that are tractable with respect to sampling, **evi**, **marg** and **cond** queries. Additionally, they are highly expressive and so, in an informal sense, lie in a sweet spot on the expressivity-tractability spectrum, as illustrated in Figure 2. In this section, we offer an overview of PCs, provide a tutorial-like example of their use in answering **evi**, **marg** and **cond** queries and detail some of their applications.

## 3.1 Overview of PCs

A PC $(\mathbf{X}, p_{(\mathcal{G},\mathbf{w},\theta)})$ is a generative probabilistic model whose parameter set consists of a weighted computational graph $\mathcal{G}$, a tuple of weights $\mathbf{w}$ pertaining to edges in $\mathcal{G}$ and a tuple of parameters $\theta$ pertaining to the associated functions of the leaf nodes (inputs) of the PC. The computational graph $\mathcal{G}$ of a PC (often referred to as the PC itself for convenience) is a rooted finite weighted directed acyclic graph with a topological ordering (nodes are ordered bottom-up) whose weights are non-negative and nodes belong to one of three types: sum, product and leaf. The computational graph $\mathcal{G}$ can be illustrated in a layer-by-layer manner in which the first layer consists only of leaf nodes and each preceding layer alternates

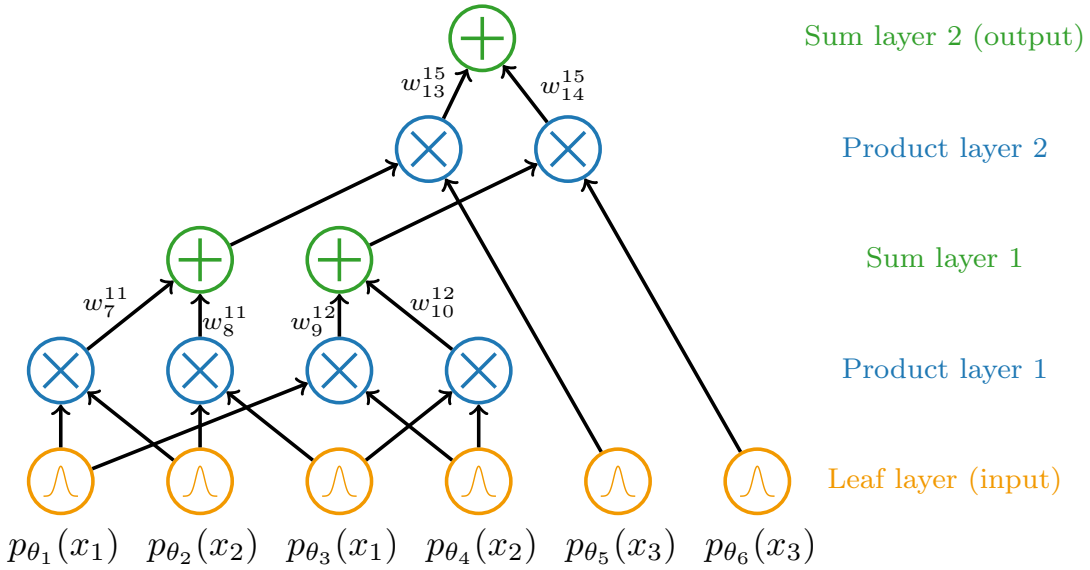


**Figure 3:** A PC pertaining to model variables $\mathbf{X} = \{X_1, X_2, X_3\}$ with indicated layers and weights indexed according to the left-right bottom-up enumeration of the nodes. Leaf nodes are coloured in orange, product nodes in blue and sum nodes in green.

between only product nodes and only sum nodes. While the root node of $\mathcal{G}$ (the output node of the PC) is the only node with no children, the leaf nodes of $\mathcal{G}$ (the leaf nodes of the PC) are precisely those with no parents and each corresponds to a distribution over the sample space of one random variable in $\mathbf{X} = \{X_1, \ldots, X_n\}$, illustrated in Figure 3 in orange.

Before describing the purpose of each node type, we define the restriction of a realisation $\mathbf{x} \in \Omega_{\mathbf{X}}$ to a subset of $\mathbf{X}$ in Definition 3.1 as well as the scope of a node in the computational graph of a PC in Definition 3.2, which can be thought of as the set of random variables in $\mathbf{X}$ belonging to its ancestry in $\mathcal{G}$. Note that the potential ambiguity of the sample space $\Omega_{\mathbf{X}'}$ of a subset of model variables $\mathbf{X}' \subset \mathbf{X}$ is resolved in Appendix A.

**Definition 3.1** *The restriction of a realisation $\mathbf{x} \in \Omega_{\mathbf{X}}$ to a subset of random variables $\mathbf{X}' \subset \mathbf{X}$ is the realisation $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ such that the individual realisations of $\mathbf{x}'$ align with the corresponding individual realisations of $\mathbf{x}$.*

**Example 3.1** *If $\mathbf{X} = \{X_1, X_2, X_3\}$ and $\mathbf{X}' = \{X_1, X_3\}$ then the restriction of the realisation $(x_1, x_2, x_3) \in \Omega_{\mathbf{X}}$ to $\mathbf{X}'$ is $(x_1, x_3) \in \Omega_{\mathbf{X}'}$.*

**Definition 3.2** *Defined recursively, the scope $sc(n)$ of a sum or product node $n$ in the computational graph of a PC is the union of the scopes of its parents. The scope of a leaf node is the singleton set containing its associated random variable in $\mathbf{X}$.*

**Example 3.2** *The scopes of both sum nodes in the third layer of the PC in Figure 3 are* $\{X_1, X_2\}$*. The scopes of both product nodes in the fourth layer are* $\{X_1, X_2, X_3\}$ *since both have two parents whose scopes are* $\{X_1, X_2\}$ *and* $\{X_3\}$*. The scope of the root, illustrated in the fifth layer, is* $\{X_1, X_2, X_3\} = \mathbf{X}$*.*

We are now able to discuss the purpose of each node type (sum, product and leaf) along with how the computational graph of a PC, as well as its edge weights $\mathbf{w}$ and parameter tuple $\theta$, yield a representation of a joint probability function over $\mathbf{X}$. For ease of notation, enumerate the nodes of the computational graph of a PC $(\mathbf{X}, p_{(\mathcal{G},\mathbf{w},\theta)})$ as $1, \ldots, N$, where $N$ is the number of nodes in $\mathcal{G}$, according to a topological ordering of $\mathcal{G}$. Let $n_i$ denote the $i^{\text{th}}$ node and let $\Omega_{\text{sc}(n_i)}$ denote the sample space of the scope of $n_i$. We define the associated function $p_{n_i} : \Omega_{\text{sc}(n_i)} \to \mathbb{R}_{\geq 0}$ of $n_i$ as

$$p_{n_i}(\mathbf{x}) := \begin{cases} \sum\limits_{n_j \in \mathbf{pa}(n_i)} w_j^i p_{n_j}(\mathbf{x}_{n_j}), & \text{if } n_i \text{ is a sum node} \\ \prod\limits_{n_j \in \mathbf{pa}(n_i)} p_{n_j}(\mathbf{x}_{n_j}), & \text{if } n_i \text{ is a product node} \\ p_{\theta_i}(\mathbf{x}), & \text{if } n_i \text{ is a leaf node} \end{cases} \tag{1}$$

where $w_j^i$ is the weight associated with the edge pointing from from $n_j$ to $n_i$, $\mathbf{x}_{n_j}$ is the restriction of $\mathbf{x}$ to $\text{sc}(n_j)$ and $p_{\theta_i}$ is the probability density/mass function corresponding to $n_i$ (if it is a leaf node). We additionally require that the weights pertaining to the parents of a given sum node sum to 1.

At this point, we would like to define the representation $p_{(\mathcal{G},\mathbf{w},\theta)}$ of the joint probability function over $\mathbf{X}$ of a PC as the associated function of the root node of $\mathcal{G}$ but our overview is missing two key ingredients. These ingredients come in the form of structural constraints on its computational graph $\mathcal{G}$, smoothness and decomposability, which ensure that the associated function of any node in $\mathcal{G}$ is a probability function over its scope. That is, for all nodes $n$ in the computational graph $\mathcal{G}$ of a PC, we would like $p_n : \Omega_{\text{sc}(n)} \to \mathbb{R}_{\geq 0}$ to be non-negative, which is already satisfied by our overview up to now, and integrate (or sum) to unity.

### 3.1.1 Smoothness

An important detail in Equation 1 is the presence of the restriction $\mathbf{x}_{n_j}$ of the realisation $\mathbf{x} \in \Omega_{\text{sc}(n_i)}$ to the scope of each parent node $n_j \in \mathbf{pa}(n_i)$ when $n_i$ is a sum node. This detail is necessary as we have not yet imposed the restriction that the associated functions of the parents of a given sum node share the same domain. In line with this, for a given sum node, its associated function is not simply the weighted sum of the associated functions of its parents. For an illustration of this, consider Example 3.3.

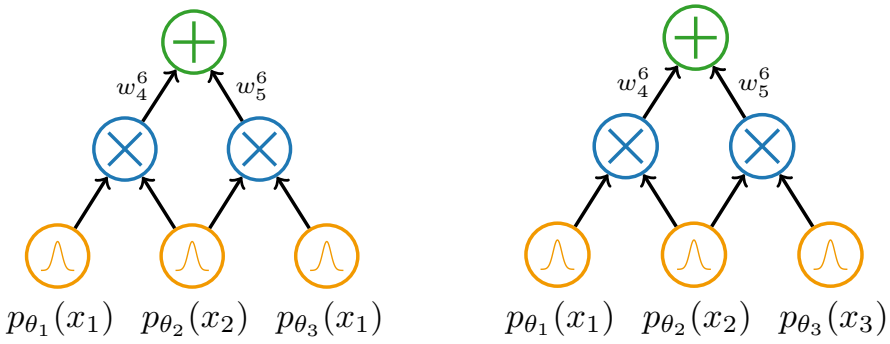


**Figure 4:** The PC on the left is smooth as the scopes of the parents of the only sum node are both $\{X_1, X_2\}$. The PC on the right is not smooth as the scopes of the parents of the only sum node are $\{X_1, X_2\}$ and $\{X_2, X_3\}$, which are not equal.

**Example 3.3** *Consider the computational graph on the right of Figure 4. If $w_4^6$ and $w_5^6$ are non-negative and sum to 1, say $w_4^6 = 1/4$ and $w_5^6 = 3/4$, then it is satisfies all conditions imposed on the computational graph of a PC up to now. That said, the scopes of the two product nodes in its second layer are $\{X_1, X_2\}$ and $\{X_2, X_3\}$ respectively. As such, the domains of the associated functions of said product nodes are $\Omega_{X_1} \times \Omega_{X_2}$ and $\Omega_{X_2} \times \Omega_{X_3}$ respectively, which are not equal. Since the domains of the parents of the sum node in the third layer differ, its associated function is not the weighted sum of the associated functions of its parents. This is not inconsistent with how the associated function of a sum node is defined in Equation 1 as the inputs of the associated functions of the parents of said sum node are restricted to their scope. For example, in this case, with a topological enumeration of the nodes of the computational graph (left-right bottom-up), the associated function $p_{n_6} : \Omega_{X_1} \times \Omega_{X_2} \times \Omega_{X_3} \to \mathbb{R}_{\geq 0}$ of the root node $n_6$ is given by*

$$\begin{aligned} p_{n_6}(x_1, x_2, x_3) &= \frac{1}{4} p_{n_4}(x_1, x_2) + \frac{3}{4} p_{n_5}(x_2, x_3) \\ &= \frac{1}{4} p_{\theta_1}(x_1) p_{\theta_2}(x_2) + \frac{3}{4} p_{\theta_2}(x_2) p_{\theta_3}(x_3). \end{aligned}$$

*Note that $p_{n_6}$ is not a probability function over its scope: it is non-negative but does not integrate (or sum) to unity.*

To alleviate what is illustrated in Example 3.3, we seek to ensure that all sum nodes in a PC constitute a mixture over their scope. To ensure this, we impose the structural constraint of smoothness on the computational graph of a PC as defined in Definition 3.3.

**Definition 3.3** *A PC is smooth if for any sum node in its computational graph, the scopes of its parents are equal. That is, for all sum nodes $n_i$ we have $sc(n_j) = sc(n_k)$ for all $n_j, n_k \in \mathbf{pa}(n_i)$.*

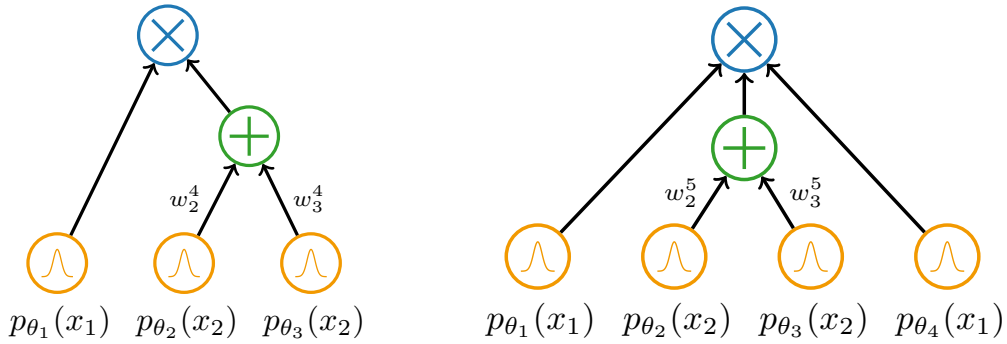


**Figure 5:** The PC on the left is smooth and decomposable as the scopes of the parents of the only product node are $\{X_1\}$ and $\{X_2\}$ which are disjoint. The PC on the right is smooth but not decomposable as two of the parents of the only product node contain $X_1$ in their scope. That is, the scopes of the parents of the product not are not disjoint.

**Example 3.4** *Consider the computational graph on the left of Figure 4 and suppose $w_4^6$ and $w_5^6$ are non-negative and sum to 1. The only sum node of the PC is at its root and the scopes of its parents, the two product nodes in the second layer, are both $\{X_1, X_2\}$ and so the PC is smooth.*

### 3.1.2 Decomposability

The second structural constraint imposed on the computational graphs of PCs, alongside smoothness, is decomposability. Before defining decomposability, consider Example 3.5, which illustrates that even with smoothness we are not guaranteed that the associated function of a given node in a PC integrates (or sums) to unity.

**Example 3.5** *Consider the computational graph on the right of Figure 5 and suppose $w_4^6$ and $w_5^6$ are non-negative and sum to 1, say $w_2^5 = 1/4$ and $w_3^5 = 3/4$. The only sum node of the PC is in the second layer and the scopes of its parents are both $\{X_2\}$, so the PC is smooth. The associated function of its root node, $p_{n_6} : \Omega_{X_1} \times \Omega_{X_2} \to \mathbb{R}_{\geq 0}$, is given by*

$$\begin{aligned} p_{n_6}(x_1, x_2) &= p_{\theta_1}(x_1) \left( \frac{1}{4} p_{\theta_2}(x_2) + \frac{3}{4} p_{\theta_3}(x_2) \right) p_{\theta_4}(x_1) \\ &= \frac{1}{4} p_{\theta_1}(x_1) p_{\theta_2}(x_2) p_{\theta_4}(x_1) + \frac{3}{4} p_{\theta_1}(x_1) p_{\theta_3}(x_2) p_{\theta_4}(x_1). \end{aligned}$$

*Upon inspection, $p_{n_6}$ is not a probability function over its scope: it is non-negative but does not necessarily integrate (or sum) to unity.*

To alleviate what is illustrated in Example 3.5, we seek to ensure that all product nodes in a PC constitute a factorisation over their scope. That is, for any product node, we would

like for any given random variable in its scope to belong to the scope of just one of its parents. To ensure this, we impose the structural constraint of decomposability on the computational graph of a PC as defined in Definition 3.4.

**Definition 3.4** *A PC is decomposable if for any given product node in its computational graph, the scopes of its parents are disjoint. That is, for all product nodes $n_i$ we have $sc(n_j) \cap sc(n_k) = \emptyset$ for all $n_j, n_k \in \mathbf{pa}(n_i)$ such that $n_j \neq n_k$.*

**Example 3.6** *Consider the computational graph on the left of Figure 5 and suppose $w_2^4$ and $w_3^4$ are non-negative and sum to 1. The only product node of the PC is at its root and the scopes of its parents are $\{X_1\}$ and $\{X_2\}$ respectively, which are disjoint, and so the PC is decomposable.*

With the imposition of smoothness and decomposability on the computational graphs of PCs, our overview is complete. That is, smoothness and decomposability ensure that the associated function of the root node, and in fact any node, of a PC is a probability function over its scope: it is non-negative and integrates (or sums) to unity, the latter of which is shown in Lemma 3.1.

**Lemma 3.1** *The associated function of a node $n_i$ in a PC $(\mathbf{X}, (\mathcal{G}, \mathbf{w}, \theta))$ is a probability function over its scope. That is, $p_{n_i}$ is non-negative and integrates (or sums) to 1.*

*Proof.* If $n_i$ is a leaf node then its associated function is, by definition, a probability function over its scope (a single random variable in $\mathbf{X}$). What remains is to show that the statement is true for all sum and product nodes.

Assume, for ease of notation, that the random variables in $\mathbf{X}$ are continuously distributed. It suffices to show that the integral of the associated function of a sum or product node with respect to a random variable in its scope can be expressed as the sum or product of integrals over the associated functions of a subset of its parents. In an informal but visual sense, one can see this as the integral being passed from a node to some subset of its parents. To see that this is indeed the case, note that for all $\mathbf{x} \in \Omega_{\mathrm{sc}(n_i)}$ and $X_k \in \mathrm{sc}(n_i)$ we have

$$\int p_{n_i}(\mathbf{x}) dx_k = \begin{cases} \sum\limits_{n_j \in \mathbf{pa}(n_i)} w_j^i \int p_{n_j}(\mathbf{x}) dx_k, & \text{if } n_i \text{ is a sum node} \\ \int p_{n_z}(\mathbf{x}_{n_z}) dx_k \cdot \prod\limits_{n_j \in \mathbf{pa}(n_i) \setminus \{n_z\}} p_{n_j}(\mathbf{x}_{n_j}), & \text{if } n_i \text{ is a product node} \\ 1, & \text{if } n_i \text{ is a leaf node} \end{cases}$$

where $n_z$ is the parent node of $n_i$ whose scope contains $X_k$ (when $n_i$ is a product node). Note that precisely one parent of $n_i$ contains $X_k$ in its scope, when $n_i$ is a product node,

due to decomposability. Additionally, smoothness ensures that we do not need to restrict $\mathbf{x}$ to the scopes of the parents of $n_i$, if it is a sum node, as was needed in Equation 1 before smoothness was imposed.

As all weights are non-negative and the weights of parents of a given sum node sum to 1, it follows that $p_{n_i}$ integrates to unity over its scope. ■

**Example 3.7** *Consider the computational graph on the left of Figure 5. Upon inspection it is both smooth and decomposable and so if $w_2^4$ and $w_3^4$ are non-negative and sum to 1 then it is a PC with $\mathbf{X} = \{X_1, X_2\}$, $\mathbf{w} = (w_2^4, w_3^4)$ and $\theta = (\theta_1, \theta_2, \theta_3)$.*

*Enumerate the nodes in $\mathcal{G}$ according to its left-right bottom-up topological ordering and let $w_2^4 = 1/4$ and $w_3^4 = 3/4$. The associated function $p_{n_5} : \Omega_{X_1} \times \Omega_{X_2} \to \mathbb{R}_{\geq 0}$ of its root node is given by*

$$\begin{aligned} p_{n_5}(x_1, x_2) &= p_{\theta_1}(x_1)\left(\frac{1}{4}p_{\theta_2}(x_2) + \frac{3}{4}p_{\theta_3}(x_2)\right) \\ &= \frac{1}{4}p_{\theta_1}(x_1)p_{\theta_2}(x_2) + \frac{3}{4}p_{\theta_1}(x_1)p_{\theta_3}(x_2). \end{aligned}$$

*Since $p_{\theta_1}$ is a probability function over $X_1$ and $p_{\theta_2}$ and $p_{\theta_3}$ are probability functions over $X_2$, it follows that $p_{n_5}$ is non-negative. Further, we see that*

$$\begin{aligned} \iint p_{n_5}(x_1, x_2)dx_1dx_2 &= \iint \frac{1}{4}p_{\theta_1}(x_1)p_{\theta_2}(x_2) + \frac{3}{4}p_{\theta_1}(x_1)p_{\theta_3}(x_2)dx_1dx_2 \\ &= \frac{1}{4}\int p_{\theta_1}(x_1)dx_1 \int p_{\theta_2}(x_2)dx_2 + \frac{3}{4}\int p_{\theta_1}(x_1)dx_1 \int p_{\theta_3}(x_2)dx_2 \\ &= \frac{1}{4} + \frac{3}{4} \\ &= 1. \end{aligned}$$

## 3.2 Tractability of PCs

With our overview complete, we are now able to show that PCs are tractable with respect to **evi** and **marg** queries. Recall that a PC $(\mathbf{X}, p_{(\mathcal{G},\mathbf{w},\theta)})$ is a generative probabilistic model for which the representation $p_{(\mathcal{G},\mathbf{w},\theta)}$ of the joint probability function over $\mathbf{X}$ is given by the associated function of the root node of $\mathcal{G}$. As such, answering an **evi** query reduces to evaluating the associated function of the root node of our PC for the relevant realisation $\mathbf{x} \in \Omega_{\mathbf{X}}$. This can be done by evaluating the probability functions of the univariate distributions at the leaf nodes (in the first layer of the PC) and traversing the computational graph in a forward-pass (or upward-pass), computing products or sums where relevant until reaching the root. This corresponds to a series of multiplications and additions whose number grows linearly in the number of edges in $\mathcal{G}$ (as precisely one operation is executed for each edge). As such, in line with Definition 2.8, PCs are tractable

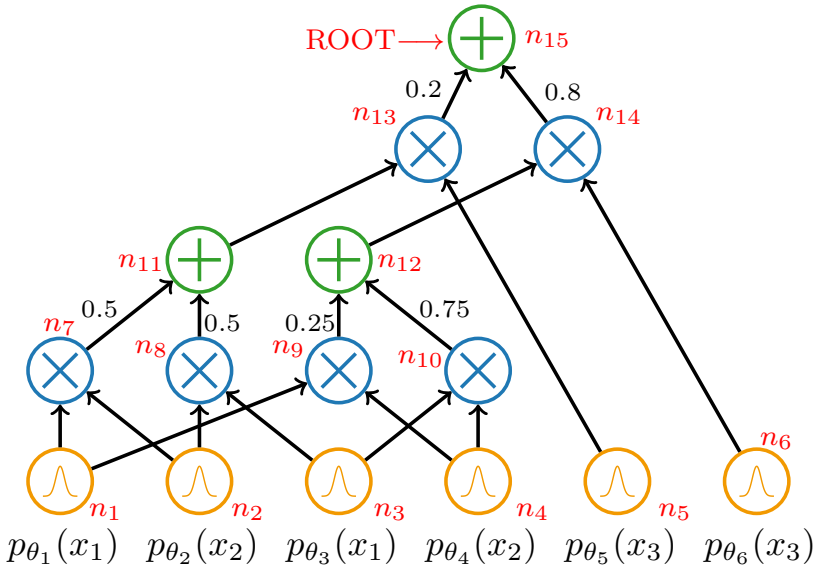


**Figure 6:** Our PC enumerated in red according to its topological ordering.

with respect to **evi** queries and their complexity is linear in the number of edges of $\mathcal{G}$. This is often stated as the complexity being linear in the 'size' of the PC. A demonstration of answering an **evi** query using a PC is given in Subsection 3.3.

To see that PCs are tractable with respect to **marg** queries, once again consider Lemma 3.1 which shows that integrating the associated function of a sum or product node in the computational graph of a PC with respect to a random variable in its scope reduces to integrating the associated functions of some subset of its parents with respect to the same random variable. In turn, this act of the integral being 'passed down' to the relevant parents results exclusively in integrals over individual leaf nodes corresponding to said random variable. Such integrals simply evaluate to 1 as the associated function of leaf nodes are probability functions over a given random variable in $\mathbf{X}$. This makes answering **marg** queries using a PC straightforward and the process is detailed in Corollary 3.1.

**Corollary 3.1** *Given a PC* $(\mathbf{X}, p_{(\mathcal{G},\mathbf{w},\theta)})$*, answering the* **marg** *query for the partial realisation* $\mathbf{x}' \in \Omega_{\mathbf{X}'}$ *for* $\mathbf{X}' \subset \mathbf{X}$ *can be done by setting the values of leaf nodes corresponding to the marginalised variables* $\mathbf{X} \backslash \mathbf{X}'$ *to* 1 *and evaluating an* **evi** *query.*

A demonstration of the approach alluded to in Corollary 3.1 is given in Subsection 3.3.

### 3.3 Tutorial-like Example of PCs

Consider the PC $(\mathbf{X}, (\mathcal{G}, \mathbf{w}, \theta))$, whose computational graph $\mathcal{G}$ and enumeration is illustrated in Figure 6, for which $\mathbf{X} = \{X_1, X_2, X_3\}$,

$$\mathbf{w} = (w_7^{11}, w_8^{11}, w_9^{12}, w_{10}^{12}, w_{13}^{15}, w_{14}^{15}) = (0.5, 0.5, 0.25, 0.75, 0.2, 0.8)$$

and

$$\theta = (\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = (0.5, 0.1, 0.9, 0.8, (0.4, 0.3, 0.3), (0.1, 0.7, 0.2)).$$

Upon inspection, $\mathcal{G}$ is both smooth and decomposable. Suppose the leaf nodes $n_1, n_2, n_3$ and $n_4$ encode Bernoulli distributions, i.e. $p_{\theta_i} : \{0, 1\} \to [0, 1]$ with

$$p_{\theta_i}(x) = \theta_i^x (1 - \theta_i)^{1 - x_i}$$

for $i = 1, \ldots, 4$. The parameters $\theta_1, \ldots, \theta_4$ are given above. Further, suppose the leaf nodes $n_5$ and $n_6$, which corresponds to $X_3$, encode categorical distributions pertaining to three categories, i.e. $p_{\theta_i} : \{0, 1, 2\} \to [0, 1]$ with

$$p_{\theta_i}(x) = \sum_{k=0}^{2} \mathbf{1}(x = k)\theta_{i,k}$$

for $i = 5$ and $i = 6$ where $(\theta_{5,0}, \theta_{5,1}, \theta_{5,2}) = (0.4, 0.3, 0.3)$ and $(\theta_{6,0}, \theta_{6,1}, \theta_{6,2}) = (0.1, 0.7, 0.2)$. Note that since $n_1, \ldots, n_6$ are leaf nodes, we have $\mathrm{sc}(n_1), \ldots, \mathrm{sc}(n_6) \in \{X_1, X_2, X_3\}$.

#### 3.3.1 Answering evi queries

To answer the **evi** query $p_{(\mathcal{G}, \mathbf{w}, \theta)}(x_1, x_2, x_3)$ for the realisation $(x_1, x_2, x_3) = (0, 1, 2)$, we begin by computing the associated values of the leaf nodes, which corresponds to evaluating univariate probability mass functions for our realisation,

$$\begin{aligned} p_{\theta_1}(0) = 0.5 \quad p_{\theta_2}(1) = 0.1 \quad p_{\theta_3}(0) = 0.1 \\ p_{\theta_4}(1) = 0.8 \quad p_{\theta_5}(2) = 0.3 \quad p_{\theta_6}(2) = 0.2 \end{aligned}$$

and follow the structure of the computational graph, computing products and weighted sums where indicated. This is illustrated entirely in Figure 7. For example, the associated value of $n_7$ (the leftmost product node in the second layer) is the product of the associated values of leaf nodes $n_1$ and $n_2$, as the representation of the computational graph would suggest, and so

$$p_{n_7}(0, 1) = p_{n_1}(0) \cdot p_{n_2}(1) = 0.5 \cdot 0.1 = 0.05.$$

Following this process fully yields the evaluation of the associated function of the root node for our realisation, i.e. the answer to the given **evi** query according to our model, which is 0.0274.

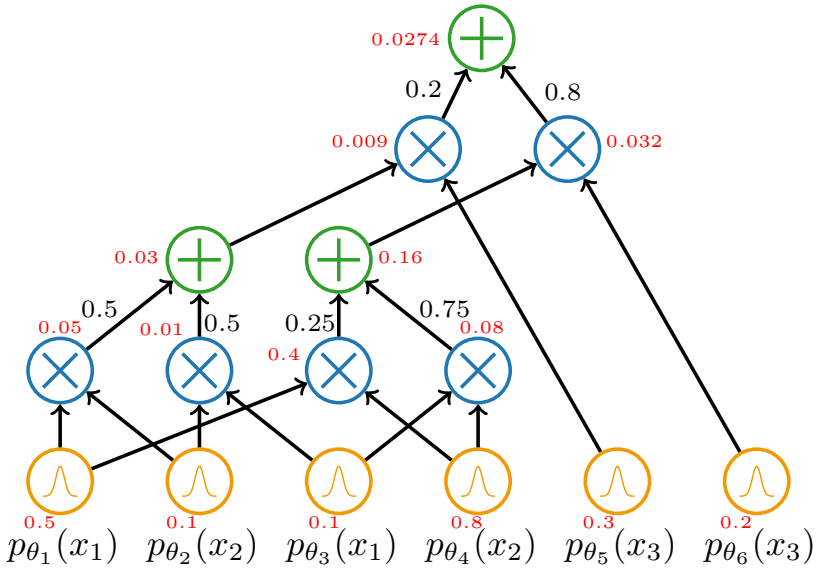


**Figure 7:** Our PC in which the associated value of each node is labelled in red for the realisation $(x_1, x_2, x_3) = (0, 1, 2)$.

Alternatively, one could express the joint probability function encoded by the PC explicitly from the PC's computational graph and compute $p_{(\mathcal{G},\mathbf{w},\theta)}(0, 1, 2)$ directly. In our case, said explicit representation can be obtained through a top-down traversal of $\mathcal{G}$ and is given by

$$
\begin{aligned}
p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_2, x_3) &= p_{n_{15}}(x_1, x_2, x_3) \\
&= \frac{1}{5} p_{n_{13}}(x_1, x_2, x_3) + \frac{4}{5} p_{n_{14}}(x_1, x_2, x_3) \\
&= \frac{1}{5} p_{n_{11}}(x_1, x_2) p_{n_5}(x_3) + \frac{4}{5} p_{n_{12}}(x_1, x_2) p_{n_6}(x_3) \\
&= \frac{1}{5} \left( \frac{1}{2} p_{n_7}(x_1, x_2) + \frac{1}{2} p_{n_8}(x_1, x_2) \right) p_{n_5}(x_3) \\
&+ \frac{4}{5} \left( \frac{1}{4} p_{n_9}(x_1, x_2) + \frac{3}{4} p_{n_{10}}(x_1, x_2) \right) p_{n_6}(x_3) \\
&= \frac{1}{10} p_{n_1}(x_1) p_{n_2}(x_2) p_{n_5}(x_3) + \frac{1}{10} p_{n_3}(x_1) p_{n_2}(x_2) p_{n_5}(x_3) \\
&+ \frac{2}{10} p_{n_1}(x_1) p_{n_4}(x_2) p_{n_6}(x_3) + \frac{6}{10} p_{n_3}(x_1) p_{n_4}(x_2) p_{n_6}(x_3)
\end{aligned}
$$

Directly answering the query by evaluating the explicit representation for our realisation $(x_1, x_2, x_3) = (0, 1, 2)$ yields 0.0274, matching the value obtained earlier via an upward-pass

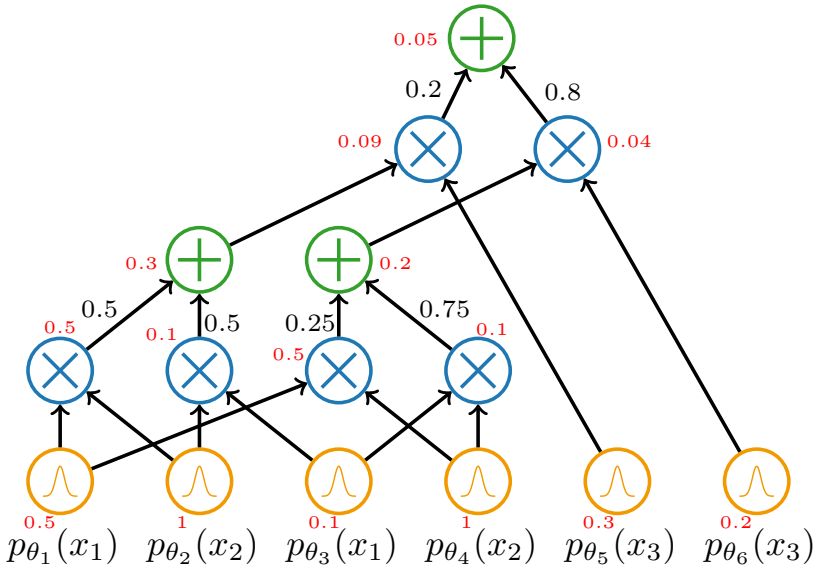


**Figure 8:** Our PC labelled in red according to the output of the associated function of each node for the realisation $(x_1, x_3) = (0, 2)$, i.e. with $X_2$ marginalised out.

of $\mathcal{G}$. Note that writing out such an explicit representation like this is typically far from feasible as the computational graphs of PCs learned from data are often very deep.

### 3.3.2 Answering marg queries

In continuing our example, let us answer the **marg** query $p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_3)$ for the partial realisation $(x_1, x_3) = (0, 2)$. We will do this using an upward-pass of the computational graph while utilising Corollary 3.1 and then a second time using the explicit representation of the joint probability function derived in the previous example.

In line with Corollary 3.1 and illustrated in Figure 8, we set the associated values of the leaf nodes corresponding to $X_2$ (the random variable being marginalised out) to 1 and proceed by completing an upward-pass of the computational graph for the partial realisation $(x_1, x_3) = (0, 2)$. This process yields an associated value of 0.05 for the root node. Instead using the explicit representation of $p_{(\mathcal{G},\mathbf{w},\theta)}$ derived in the previous example, we compute $p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_3)$ in the usual way of summing $p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_2, x_3)$ over each state

of $X_2$ as in

$$
\begin{aligned}
p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_3) &= \sum_{x_2 \in \{0,1\}} p_{(\mathcal{G},\mathbf{w},\theta)}(x_1, x_2, x_3) \\
&= \sum_{x_2 \in \{0,1\}} \Bigg[ \frac{1}{10} p_{n_1}(x_1) p_{n_2}(x_2) p_{n_5}(x_3) + \frac{1}{10} p_{n_3}(x_1) p_{n_2}(x_2) p_{n_5}(x_3) \\
&+ \frac{2}{10} p_{n_1}(x_1) p_{n_4}(x_2) p_{n_6}(x_3) + \frac{6}{10} p_{n_3}(x_1) p_{n_4}(x_2) p_{n_6}(x_3) \Bigg] \\
&= \frac{1}{10} p_{n_1}(x_1) p_{n_5}(x_3) \sum_{x_2 \in \{0,1\}} p_{n_2}(x_2) + \frac{1}{10} p_{n_3}(x_1) p_{n_5}(x_3) \sum_{x_2 \in \{0,1\}} p_{n_2}(x_2) \\
&+ \frac{2}{10} p_{n_1}(x_1) p_{n_6}(x_3) \sum_{x_2 \in \{0,1\}} p_{n_4}(x_2) + \frac{6}{10} p_{n_3}(x_1) p_{n_6}(x_3) \sum_{x_2 \in \{0,1\}} p_{n_4}(x_2) \\
&= \frac{1}{10} p_{n_1}(x_1) p_{n_5}(x_3) + \frac{1}{10} p_{n_3}(x_1) p_{n_5}(x_3) + \frac{2}{10} p_{n_1}(x_1) p_{n_6}(x_3) + \frac{6}{10} p_{n_3}(x_1) p_{n_6}(x_3)
\end{aligned}
$$

and computing $p_{(\mathcal{G},\mathbf{w},\theta)}(0, 2) = 0.05$ directly. We see in this example how integrals/sums pass down from the root node (or any other node in a PC for that matter) to integrals/sums over the leaf nodes corresponding to the random variable(s) being marginalised out, effectively illustrating Corollary 3.1.

## 3.4 Applications of PCs

We briefly detail selected applications of PCs: anomaly and out-of-distribution detection, sampling, classification and in-painting.

### 3.4.1 Anomaly and out-of-distribution (OOD) detection

Anomaly detection and out-of-distribution (OOD) detection are related tasks in which one seeks to flag samples which deviate significantly from the distribution learned by a model. OOD detection seeks to flag samples which do not at all belong to the distribution learned during training (e.g. flagging Fashion-MNIST samples when trained on regular MNIST), whereas anomaly detection seeks to flag sufficiently unlikely in-distribution samples (e.g. the MNIST samples illustrated in Figure 9). In both cases, one seeks to distinguish samples which do not align with the learned distribution sufficiently-well. In its most basic form, this is done by answering the **evi** query corresponding to the given sample. That is, computing its probability. If the answer of the **evi** query is below some pre-determined threshold then the sample is flagged.

Since PCs are tractable with respect to **evi** queries, they can be applied to both anomaly and OOD detection. Further, since PCs are tractable with respect to **marg** queries, one

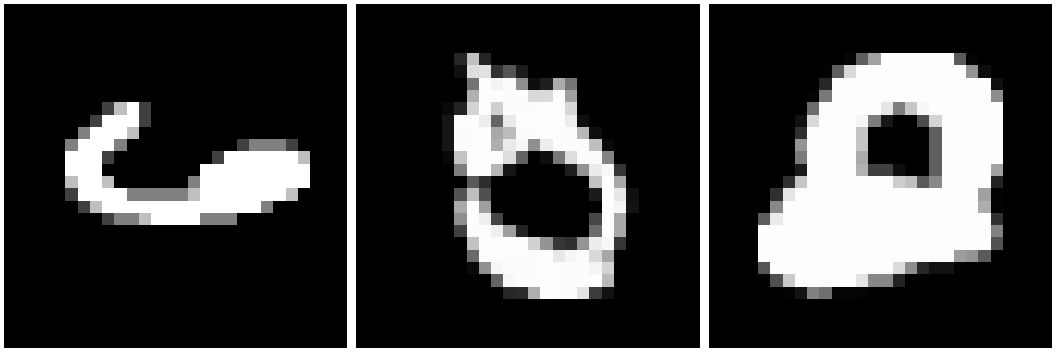

**Figure 9:** MNIST samples whose class labels are 6, 0 and 2 respectively.

can seamlessly perform anomaly and OOD detection for incomplete samples. As for how PCs perform empirically on these tasks, Peharz et al. [34] developed large-scale PCs (∼90M parameters) which, when trained on MNIST, assign near-zero probabilities to non-MNIST samples and achieve effectively no overlap between in-distribution and out-of-distribution probabilities. Pevný & Smídl [28] utilised the tractability of PCs with respect to **marg** queries in order to efficiently evaluate the contribution of various feature subsets yielding a scalable method of subspace-based outlier detection by identifying feature combinations which minimise the marginal density, i.e. maximising the attributed outlier score, without having to re-train the model for each subset of features.

### 3.4.2 Sampling

Sampling from the distribution encoded by a PC corresponds to sampling from the distribution encoded by its root node. This involves a backwards traversal of its computational graph from the root node to precisely one corresponding leaf node for each random variable in $\mathbf{X}$. Due to their hierarchical mixture model structure, this backwards traversal process can be described in a way that is analogous to how one would sample from traditional mixture models and partially-factorised models.

To sample from a mixture model

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\mathbf{Z} = k)$$

over $\mathbf{X} = \{X_1, \ldots, X_n\}$ where $\mathbf{Z} \sim \text{Cat}(\pi_1, \ldots, \pi_K)$, one can sample an integer $k \sim p(\mathbf{z})$ and then sample $\mathbf{x}$ from the corresponding mixture component $p(\mathbf{x}|\mathbf{Z} = k)$. How this corresponds to sampling from the distribution learned by a PC is due to its sum nodes, each of which define a mixture over their scope. In line with this, during the backwards traversal process, when at a sum node, choose a parent node according to the categorical distribution whose probabilities are the weights of the edges from the parents to said sum

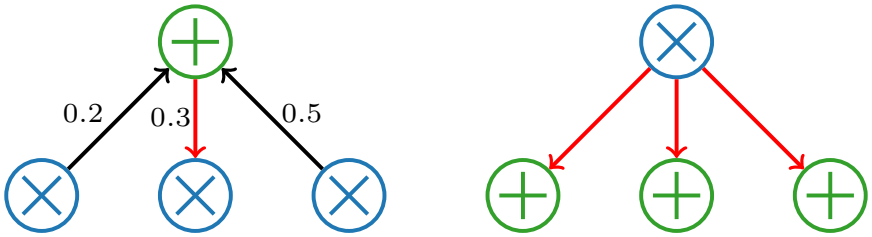


**Figure 10:** Left: sampling from the distribution encoded by a sum node — instead sample from the distribution encoded by one of its parents. Red edges point to the parent(s) chosen during the backwards sampling process. Right: sampling from the distribution encoded by a product node — sample from the distributions encoded by each of its parents.

node. Then, sample from the distribution encoded by said parent node. This is illustrated on the left of Figure 10.

To sample from a partially-factorised model

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(y_k)$$

over $\mathbf{X} = \{X_1, \ldots, X_n\}$ where $Y_k \subset \mathbf{X}$ with $\cup_{k=1}^{K} Y_k = \mathbf{X}$ and $Y_i \cap Y_j = \emptyset$ for all $i, j \in \{1, \ldots, K\}$ such that $i \neq j$, one simply samples from each component distribution $p(y_1), \ldots, p(y_K)$. How this corresponds to sampling from the distribution learned by a PC is in its product nodes, each of which define a partially-factorised model over their scope. In line with this, during the backwards traversal process, when at a product node, simply sample from each of the distributions encoded by its parent nodes. This is illustrated on the right of Figure 10.

Once our backwards traversal process reaches a leaf node we simply sample from the univariate distribution attributed to it, often a Gaussian or a categorical distribution. For an illustration of the entire backwards traversal process, consider Figure 11, in which we sample from the distribution encoded by the PC used in our tutorial-like example earlier. Additionally, for an idea of the sample quality that can be achieved by PCs, consider the MNIST samples illustrated in Figure 12.

### 3.4.3 Classification with full or missing data

As with any generative model which is tractable with respect to **evi** queries, PCs are capable of classification via maximum a posteriori. That is, by assigning the label $y \in \Omega_Y$ which maximises the conditional probability $p(y|\mathbf{x})$ where $\mathbf{x} \in \Omega_{\mathbf{X}}$ is the sample we would like to classify. This amounts to answering an **evi** query for each class label in $\Omega_Y$, i.e.

$$\arg\max_{y \in \Omega_Y} p(y|\mathbf{x}) = \arg\max_{y \in \Omega_Y} \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \arg\max_{y \in \Omega_Y} p(\mathbf{x}, y).$$

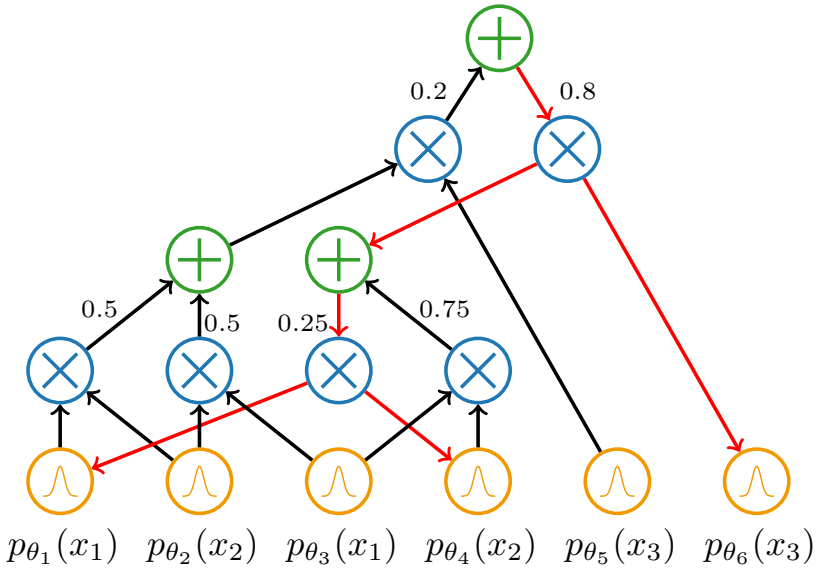


**Figure 11:** Sampling from the distribution learned by a PC via a backward-pass. Red edges indicate the backwards traversal process. At the leaf layer, sample from the univariate distributions $x'_1 \sim p_{\theta_1}(x_1)$, $x'_2 \sim p_{\theta_4}(x_2)$ and $x'_3 \sim p_{\theta_6}(x_3)$ to obtain the sample $(x'_1, x'_2, x'_3) \in \Omega_{\mathbf{X}}$.

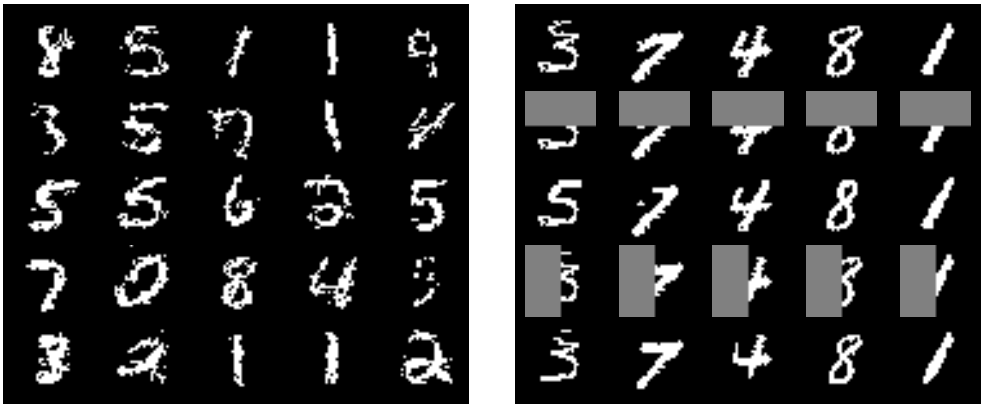

**Figure 12:** Left: samples drawn from a PC trained on MNIST. Right: incomplete MNIST samples in-painted using a PC trained on MNIST. Original samples in first row. Incomplete samples in second and fourth rows. In-painted equivalents in third and fifth rows. Taken from [44, Figure 3].

A well-known paper analysing the application of generative and discriminative models to classification is one by Andrew Ng and Michael Jordan [30] in which logistic regression and naive Bayes classifiers were considered. In this work, it was concluded that logistic regression was unequivocally preferred. In line with this, the application of generative models to classification is often less common than discriminative models. Regardless, classification is a task that PCs are often applied to in their benchmarking. An argument in the case of PCs is that one would ideally have a model that is capable of all common tasks 'out of the box': sampling, classification, anomaly detection, etc. and due to their tractability with respect to **evi** and **marg** queries, they are a suitable candidate for such a widely-capable class of probabilistic models.

The empirical classification performance of PCs for well-known datasets, such as MNIST, Fashion-MNIST, CIFAR-10, etc., is mixed as the focus of developments in the learning of large-scale PCs is mostly generative, i.e. sample quality and density estimation. That said, there have been cases of impressive classification performance for PCs (for the time of publication) beginning with an accuracy of 83.96% on CIFAR-10 in 2012 [15]. At the time, the state-of-the-art CNN-based approach by Dan et al. [8] achieved an accuracy of 88.79%. From 2012 to 2020, deep learning-based approaches outpaced PC-based approaches with glimpses of hope for PC-based approaches along the way. For example, in 2020 when random and tensorised sum-product networks (RAT-SPNs) [34], an approach to learning large scale PCs ($\sim$90M parameters), achieved an accuracy of 98.29% on MNIST. While RAT-SPN offered a glimpse into how one might learn large-scale PCs, the performance of deep learning-based approaches to classification have since effectively eclipsed PCs. This is primarily attributed to the difficulty currently faced in scaling PCs to the extent to which deep learning models have been successfully scaled, e.g. CNNs and transformer-based architectures. That said, progress in alleviating this issue of scaling PCs by employing methods from deep learning is promising [26].

### 3.4.4 In-painting incomplete samples

In-painting is the task of completing an incomplete sample. For example, if one is given the bottom half of an MNIST sample, they may want to complete it to obtain a full sample. A common approach to in-painting, when the used model allows for it, is to treat the problem as answering an **MMAP** query, as defined in Definition 2.7. That is, computing the mode of the conditional distribution of the missing portion conditioned on the present portion. An example of this is illustrated on the right of Figure 12 in which the second and fourth rows show variations of incomplete samples produced by removing a portion of the samples present in the first row. In the rows below the second and fourth, one can see the PC's attempt at in-painting.

It is worth noting that, as illustrated in Table 1, PCs are not tractable with respect to **MMAP** queries in general. In line with this, it is only a particular sub-class of PCs which are tractable with respect to **MMAP** (and so in-painting): deterministic PCs. Determin-

ism is a structural constraint one can impose on the computational graphs learned during training (if the graph structure is learned) alongside smoothness and decomposability, to ensure that at most one of the associated values of the parents of a sum node is non-zero for any realisation $\mathbf{x} \in \Omega_{\mathbf{X}}$. While far improved tractability is rewarding, determinism is a heavily restrictive structural constraint and so the class of deterministic PCs is notably less expressive than the class of PCs.

The intended purpose of including the brief description above is to demonstrate a common occurrence in literature surrounding PCs: one can trade expressivity for tractability by imposing stricter-than-usual constraints on the computational graphs of their PCs, and vice-versa.

# 4 Continuous Mixtures of Probabilistic Circuits (CMPCs)

Before overviewing continuous mixtures of probabilistic circuits (CMPCs), as introduced by Correia et al. [9], we first consider their motivation. In doing so, we detail autoencoders and variational autoencoders (VAEs) [22], the latter of which is a class of deep generative models that heavily inspired the development of CMPCs. Following this, we complete our overview of CMPCs and briefly detail the results obtained by Correia et al. in benchmarking them.

## 4.1 Motivating Continuous Mixtures

At the time of their work (around 2022), Correia et al. noted that methods used to scale PCs had yet to deliver PCs with per-parameter-performance comparable to deep generative models. To illustrate this, the most highly parameterised PC at the time consisted of 90 million parameters and attained a mean negative log-likelihood (a principled measure of a generative model's capability for which a smaller value is better) of 100 nats on Binary MNIST [9]. At the same time, a minimally-tweaked variational autoencoder consisting of around three million parameters had attained a mean negative log-likelihood (NLL) of 90 nats [42]: far better than the PC whose parameter count was around 30 times higher. Here, the mean negative log-likelihood (NLL) attained by a generative model $(\mathbf{X}, p_\Theta)$ over a dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \subset \Omega_{\mathbf{X}}$ is given by

$$-\frac{1}{N}\sum_{i=1}^{N} \log(p_\Theta(\mathbf{x}_i))$$

and the corresponding unit of measurement is nats if log denotes the natural logarithm and bits if it denotes $\log_2$. For further reference, Table 2 illustrates the mean NLLs attained by various model classes on Binary MNIST.

The poor per-parameter-performance of PCs when compared to VAEs begs a very natural question: what is it about VAEs, which are detailed in Subsection 4.2, that allows them to learn so much more effectively than PCs? In trying to answer this, the authors

| Model | Parameter count | Mean NLL (nats) |
|---|---|---|
| Chow-Liu Trees (CLTs) [5] | 2K | 176 |
| Mixture of Bernoullis [40] | 390K | 134 |
| Vanilla VAE [42] | 3M | 90 |
| PixelCNN [21] | 46M | 78 |
| PCs [9] | 90M | 100 |

**Table 2:** Mean negative log-likelihoods achieved by various model classes on Binary MNIST measured in nats (lower is better). Each is rounded to the nearest integer.

noted an important distinction between PCs and VAEs: the latter (VAEs) are latent variable models in which latent variables (or latents for brevity) are continuously distributed while the former (PCs) have a latent variable model interpretation in which latents are discretely distributed. That is, in training a VAE one seeks to model the joint distribution over model variables $\mathbf{X}$ as in

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$$

where $\mathbf{Z}$ is continuously distributed and denotes the latent variables pertaining to the dataset from which the PC is learned. As for PCs, their latent variable interpretation is analogous to that of regular mixture models. The probability function associated with a sum node $n_i$ in a PC is given by

$$p_{n_i}(\mathbf{x}) = \sum_{n_j \in \mathbf{pa}(n_i)} w_j^i p_{n_j}(\mathbf{x})$$

which can be interpreted as some latent being marginalised out [32]. That is,

$$p_{n_i}(\mathbf{x}) = \sum_{\mathbf{z} \in \Omega_{\mathbf{Z}}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$$

in which $\mathbf{Z}$ is categorically distributed and is parameterised according to the weights of the edges pointing to $n_i$ from its parents. As such, if $n_i$ has $m$ parents then for $j \in \{1, \ldots, m\}$ one has $p(\mathbf{Z} = j) = w_j^i$ and $p(\mathbf{x}|\mathbf{Z} = j) = p_{n_j}(\mathbf{x})$. In this interpretation, any sum node in a PC (and the overall PC itself, assuming its root is a sum node) can be seen as a discrete latent variable mixture model over its scope.

**Example 4.1** *Suppose a dataset is compiled of samples pertaining to whether or not weather was good at the location of recording and whether or not an earthquake occurred at the location of recording. Suppose further that 75% of samples are recorded in Lagos, Portugal and the remaining 25% in Groningen, The Netherlands. The location in which each sample was recorded is not made explicit in the dataset. In learning a PC from this dataset, the PC illustrated in Figure 13 may be realised. In this PC, each parent of the*

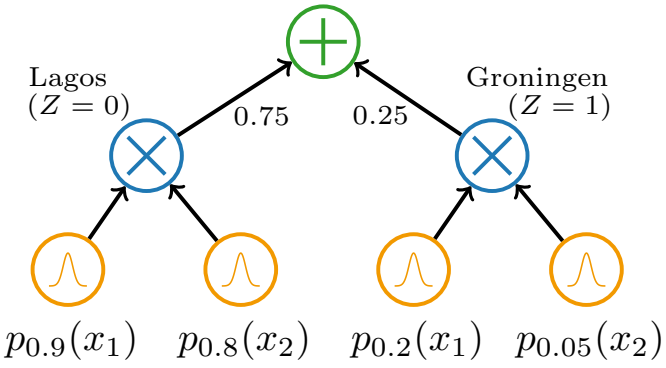


**Figure 13:** A PC in which $X_1$ pertains to whether the weather is good on a given day and $X_2$ pertains whether or not an earthquake will occur within the next year. The latent variable $Z$, pertaining to the location at the time of recording, is marginalised out.

*single sum node at the root pertains to one of the two locations of recording. As such, the sum node can be seen as marginalising out the latent variable pertaining to the location of where the data was recorded.*

In line with this distinction between VAEs and PCs, the authors sought methods of learning PCs in which latents are continuously distributed in hope that it would improve per-parameter-performance. But how might we learn PCs in a way that allows for continuously distributed latents? In answering this, we begin by noting that the joint distribution over model variables $\mathbf{X}$ can be seen as the marginal joint over $\mathbf{X}$ and latent variables $\mathbf{Z}$ in which the latents are marginalised out. That is,

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} = \mathbb{E}_{\mathbf{Z}\sim p(\mathbf{z})}[p(\mathbf{x}|\mathbf{Z})].$$

In practice, after assigning a prior distribution to $\mathbf{Z}$ and fitting $p(\mathbf{x}|\mathbf{z})$ in some way, the evaluation of this integral remains intractable and so the authors proceed by numerical approximation as in

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} \approx \sum_{i=1}^{N} w(\mathbf{z}_i)p(\mathbf{x}|\mathbf{z}_i)$$

where $w : \Omega_{\mathbf{Z}} \to \mathbb{R}_{\geq 0}$ is a weighting function and $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$ are latent samples according to some continuous prior attributed to $\mathbf{Z}$. As $N \to \infty$, this approximation becomes exact as long as $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$ are i.i.d. and the weights normalise the estimator. An approximation of $p(\mathbf{x})$ using numerical integration, in which $\mathbf{Z}$ is continuously distributed (e.g. $p(\mathbf{z}) = \mathcal{N}(0, I_d)$, the $d-$dimensional standard Gaussian), is what the authors refer to as a continuous mixture.

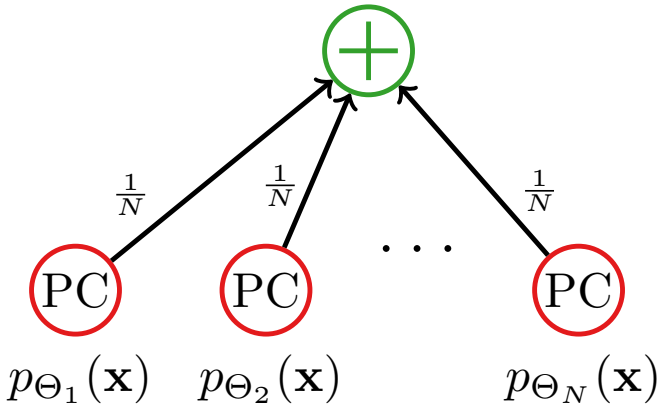


**Figure 14:** The convex sum of $N$ PCs represented as a PC. It is smooth and decomposable.

While a number of well-studied numerical integration methods exist, Correia et al. chose Monte Carlo (MC) integration in which $w(\mathbf{z}) = 1/N$ for all $\mathbf{z} \in \Omega_{\mathbf{Z}}$. Conveniently, the law of large numbers ensures that such an estimator converges almost surely

$$\frac{1}{N}\sum_{i=1}^{N} p(\mathbf{x}|\mathbf{z}_i) \xrightarrow[a.s.]{N\to\infty} \mathbb{E}_{\mathbf{Z}\sim p(\mathbf{z})}[p(\mathbf{x}|\mathbf{z})] = p(\mathbf{x}).$$

The benefits of this numerical integration method lie predominantly in its ease of implementation, interpretation and the agnosticism of its convergence rate $O(N^{-1/2})$ to the latent dimension $d$ of $\Omega_{\mathbf{Z}}$. This final point contrasts other methods of numerical integration which makes MC integration particularly attractive for learning continuous mixtures with high latent dimensions.

With the notion of a continuous mixture out of the way, we now consider how one learns a continuous mixture in a way that yields a PC. The key lies in that a convex sum of PCs, illustrated in Figure 14, is itself a PC. With this in mind, if one has $N$ latent samples $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$ and fits a PC to each component $p(\mathbf{x}|\mathbf{z}_1), \ldots, p(\mathbf{x}|\mathbf{z}_N)$ then the MC approximation of the continuous mixture

$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} p(\mathbf{x}|\mathbf{z}_i)$$

is itself a PC. In line with this, continuous mixtures of probabilistic circuits (CMPCs) are defined in Definition 4.1.

**Definition 4.1** *A continuous mixture of probabilistic circuits (CMPC) is a discrete mixture of probabilistic circuits*

$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}|\mathbf{z}_i)$$

*in which the parameters of $i^{th}$ component PC, $p_\phi(\mathbf{x}|\mathbf{z}_i)$, are given by $\phi(\mathbf{z}_i)$ where $\mathbf{z}_1, \ldots, \mathbf{z}_N$ are latent samples drawn from a continuous distribution $p(\mathbf{z})$ and $\phi$ is a continuous transformation whose domain is $\Omega_{\mathbf{Z}}$.*

**Note:** In line with Definition 4.1, the class of CMPCs is at most as expressive as the broader class of discrete mixtures of PCs. This is straightforward to see once one notes that in a regular discrete mixture of PCs, the parameters of each component PC are not dependent through some shared transformation, and so the class of CMPCs is a sub-class of the class of discrete mixtures of PCs. Shown this, a natural question is why one would employ specifically the class of CMPCs over the broader class of discrete mixtures of PCs. This question is answered in Subsection 4.4.

### 4.1.1 Fully-factorised models (FFMs) and Chow-Liu trees (CLTs)

Before detailing CMPCs further, we first consider the subclasses of PCs used in the work of Correia et al. to fit the $N$ component distributions $p(\mathbf{x}|\mathbf{z}_1), \ldots, p(\mathbf{x}|\mathbf{z}_N)$ of their CMPCs. Two subclasses of PCs are considered: fully-factorised models (FFMs) and Chow-Liu trees (CLTs) [7].

FFMs can be seen as Bayesian networks in which there are no edges between nodes. That is, in fitting an FFM, independence between all model variables $X_1, \ldots, X_n \in \mathbf{X}$ is assumed. Thus, the joint probability function pertaining to an FFM over $\mathbf{X}$ is of the form

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i).$$

An FFM is straightforwardly represented by a PC as illustrated in Figure 15. After attributing distributions to $X_1, \ldots, X_n$, to learn an FFM over $\mathbf{X}$ we need only fit the individual distributions $p(x_1), \ldots, p(x_n)$. For example, if all model variables are binary then learning an FFM over $\mathbf{X}$ amounts to learning the $n$ parameters of the Bernoulli distributions attributed to $X_1, \ldots, X_n$.

The second subclass of PCs employed by Correia et al. are Chow-Liu trees (CLTs) which are Bayesian networks in which each node has at most one parent, i.e. tree-shaped Bayesian networks. As such, the joint probability function pertaining to a CLT over $\mathbf{X}$, whose directed acyclic graph (DAG) is denoted by $\mathcal{G}$, is of the form

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{pa}(X_i); \mathcal{G})$$

where $\mathbf{pa}(X_i) \subset \mathbf{X}$ contains at most one element in $\mathbf{X}$. As with FFMs, to learn a CLT we need only learn the parameters of the distributions attributed to $X_1|\mathbf{pa}(X_1), \ldots, X_n|\mathbf{pa}(X_n)$. To learn the DAG $\mathcal{G}$ of a CLT from data, Chow and Liu [7] proposed computing the empirical mutual information between all pairs of model variables from data and then computing

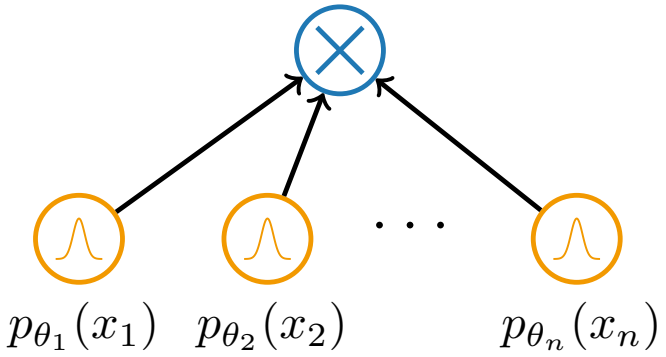


**Figure 15:** A fully-factorised model (FFM) represented as a PC.

the maximum spanning tree in which weights between nodes correspond to their mutual information. For further detail regarding CLTs, consider the original formulation by Chow and Liu [7]. Methods of efficiently compiling CLTs into PCs are known [12, 13] which makes their application to fitting the $N$ components of CMPCs straightforward. Note that CMPCs whose component distributions are fit using CLTs are expected to perform notably better than CMPCs whose components were fit using FFMs. This is due to the fact that CLTs are far more expressive as they make fewer assumptions about the to-be-fit distributions. That said, the implementation and additional compute needed to learn CMPCs with CLTs fit to the component distributions is notably higher.

Alternatively, one can consider the PCs fit to the $N$ component distributions of a CMPC as leaf nodes. This requires an extension of the definition of PCs given earlier to allow for leaf nodes which correspond to more than one random variable in $\mathbf{X}$ but this extension is not particularly cumbersome. With this in mind, CMPCs with CLTs fit to their leaf nodes need not be compiled into PCs. As for whether or not more sophisticated subclasses of PCs should be employed for fitting the $N$ component distributions of a CMPC, Correia et al. note that future work could consider hidden Chow-Liu trees (HCLTs) [25]. Alternatively, as Bayesian networks can be compiled to PCs with complexity exponential in their tree-width [47, Subsection 4.3], a natural choice is to fit Bayesian networks with sufficiently small tree-width to the $N$ component distributions. That said, again, the time needed for implementation and the computational resources needed for training naturally increase.

**Example 4.2** *Suppose one would like to learn a CMPC consisting of $N = 16$ components with $\mathbf{Z} \sim \mathcal{N}(0, I_4)$, i.e. $\mathbf{Z}$ is distributed according to the 4-dimensional standard normal distribution. Further, suppose we fit a fully-factorised model to each component distribution of the CMPC. That is, the component distribution corresponding to the sampled latent $\mathbf{z}' \sim p(\mathbf{z})$ is given by*

$$p(\mathbf{x}|\mathbf{z}') = p(x_1, \ldots, x_n|\mathbf{z}') = \prod_{j=1}^{n} p(x_j|\mathbf{z}').$$

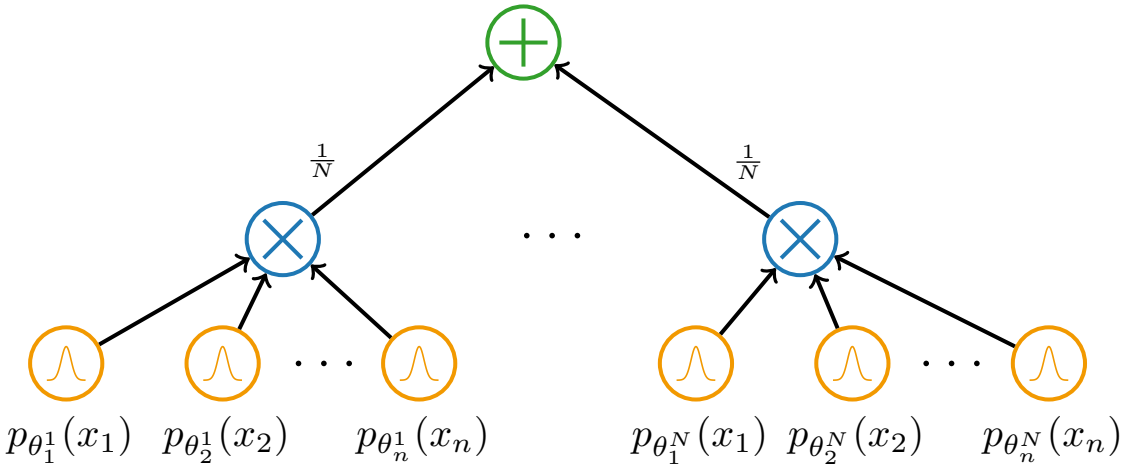


**Figure 16:** An $N-$component CMPC in which the component distributions are fit using FFMs (which are PCs themselves). The notation introduced earlier for each leaf node's parameter set ($p_{\theta_k}$ with no superscript) is deviated from here for figure-brevity.

*Our CMPC of* $N = 16$ *components is thus of the form*

$$p(\mathbf{x}) = \frac{1}{16}\sum_{i=1}^{16} p_\phi(\mathbf{x}|\mathbf{z}_i) = \frac{1}{16}\sum_{i=1}^{16}\prod_{j=1}^{n} p_\phi(x_j|\mathbf{z}_i).$$

*The architecture of an* $N$*-component CMPC with FFMs fit to the component distributions is illustrated in Figure 16. Upon inspection, it is both smooth and decomposable.*

*As stated earlier, the parameters of each component, in this case each FFM, are given by a continuous transinformation* $\phi$ *whose domain is* $\Omega_{\mathbf{Z}}$*. We do not specify a transformation in this example. Further detail on such transformations is given in Subsection 4.3.*

Note that the computational graph $\mathcal{G}$ of CMPCs with FFMs fit to the $N$ component PCs, as in Figure 16, are very simple in structure. That said, as arbitrarily complex PC structures can be used to fit the $N$ component PCs, the computational graphs of CMPCs can become arbitrarily complex.

To complete our overview of CMPCs, we must consider which continuous distribution to assign to $\mathbf{Z}$ (including what latent dimension $d$ of $\Omega_{\mathbf{Z}}$ is appropriate) and how to fit FFMs or CLTs to the components $p_\phi(\mathbf{x}|\mathbf{z}_1), \ldots, p_\phi(\mathbf{x}|\mathbf{z}_N)$ given sampled latents $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$. In their work, Correia et al. take $\mathbf{Z} \sim \mathcal{N}(0, I_d)$ with latent dimension $d \in \{2, \ldots, 16\}$. In fitting a CMPC to more complex distributions, higher latent dimensions are considered. As for fitting FFMs/CLTs to the $N$ component distributions, Correia et al. compute each component's parameters from its sampled latent $\mathbf{z}_i \in \Omega_{\mathbf{Z}}$ using a decoder: a learned transformation from the latent space $\Omega_{\mathbf{Z}}$ to the parameter space corresponding to either

FFMs or CLTs. This decoder-style approach was inspired by variational autoencoders (VAEs), which we now consider, and their application to CMPCs is detailed further in Subsection 4.3.

## 4.2 (Variational) Autoencoders

As PCs are generative probabilistic models, methods of learning them are often benchmarked using generative tasks such as density estimation and sampling. In particular, such models are often fit to image datasets, such as MNIST, for benchmarking. Four of the most prominent deep generative model classes for image datasets are variational autoencoders (VAEs) [22, 2013], generative adversarial networks (GANs) [19, 2014], normalising flows [36, 2015] and diffusion models [20, 2020]. Of particular interest to us, due to the inspirations drawn from them in developing CMPCs, are VAEs which can be seen as adaptations of autoencoders.

### 4.2.1 Autoencoders

An autoencoder $(\Omega_{\mathbf{X}}, d, \theta, \phi)$ consists of a sample space $\Omega_{\mathbf{X}} \subset \mathbb{R}^n$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, an encoder $\theta : \mathbb{R}^n \to \mathbb{R}^d$ and a decoder $\phi : \mathbb{R}^d \to \mathbb{R}^n$. Using a subset of $\Omega_{\mathbf{X}}$, one typically learns the encoder $\theta$ and the decoder $\phi$ with the goal of approximating the identity on $\Omega_{\mathbf{X}}$ via $\phi \circ \theta$. That is, roughly put, one seeks to obtain an encoder/decoder pair such that $\phi(\theta(\mathbf{x})) \approx \mathbf{x}$ for all $\mathbf{x} \in \Omega_{\mathbf{X}}$.

In intuitive terms, one can see the encoder $\theta$ as a compressor of samples in $\Omega_{\mathbf{X}}$ to their compressed (or latent) representation in $\mathbb{R}^d$ and so one might refer to the image of $\theta$ as the latent space $\Omega_{\mathbf{Z}}$. Similarly, the decoder $\phi$ can be seen as a decompressor of compressed representations $\mathbf{z} \in \Omega_{\mathbf{Z}}$ yielding the original sample $\mathbf{x}$. In line with this notion of an autoencoder as a compressor/decompressor, the latent dimension $d$ is typically taken to be far smaller than the dimension of the distribution of interest, i.e. $d \ll n$. Of course, when $d \ll n$, learning such mappings $\theta$ and $\phi$ typically involves some loss of information if $\Omega_{\mathbf{X}}$ is a manifold whose intrinsic dimension is greater than $d$. That is, autoencoders are typically lossy compressors.

**Example 4.3** *Suppose* $\Omega_{\mathbf{X}} = \{(a, a, b) \in \mathbb{R}^3 : ||(a, a, b)|| \leq 1\}$ *and* $d = 2$. *One immediately notices that* $\Omega_{\mathbf{X}}$ *is a two-dimensional surface embedded in* $\mathbb{R}^3$ *as it is the intersection of the unit ball and the plane* $\{(a, a, b) : a, b \in \mathbb{R}\}$. *To produce representations of* $\mathbf{x} = (a, a, b) \in \Omega_{\mathbf{X}}$ *in* $\mathbb{R}^2$ *(i.e. to compress a sample) one might employ the encoder* $\theta_2(x, y, z) = (x, z)$. *To reconstruct samples from their latent representation (i.e. to decompress) one might employ the decoder* $\phi_2(x, z) = (x, x, z)$. *The autoencoder* $(\Omega_{\mathbf{X}}, 2, \theta_2, \phi_2)$ *offers lossless compression on the sample space of interest as* $(\phi_2 \circ \theta_2)(\mathbf{x}) = \mathbf{x}$ *for all* $\mathbf{x} \in \Omega_{\mathbf{X}}$.

*If we instead desire latent representations of* $\mathbf{x} = (a, a, b) \in \Omega_{\mathbf{X}}$ *in* $\mathbb{R}$, *i.e. if* $d = 1$, *one might employ the encoder* $\theta_1(x, y, z) = x$ *and the decoder* $\phi_1(x) = (x, x, x)$. *The autoencoder*

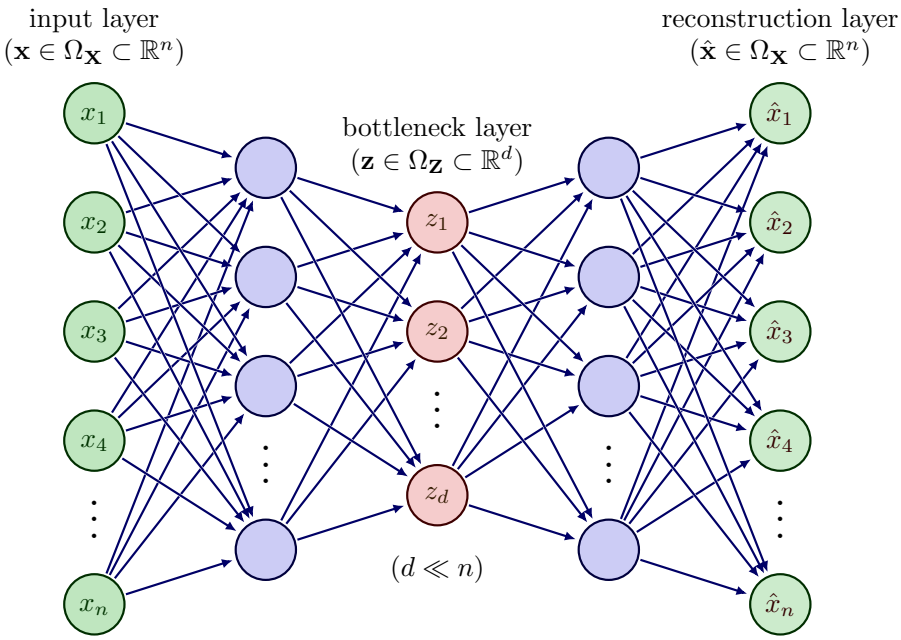


**Figure 17:** An autoencoder in which $\theta$ and $\phi$ are fit using multi-layer perceptrons (MLPs).

$(\Omega_{\mathbf{X}}, 1, \theta_1, \phi_1)$ *offers lossy compression, i.e. some information pertaining to a sample is lost in compressing and then decompressing it as* $(\phi_1 \circ \theta_1)(a, a, b) = (a, a, a)$ *for all* $a, b \in \mathbb{R}$.

One's tolerance for the loss incurred by a given encoder/decoder pair is task-dependent. Given a dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$ and a latent dimension $d$, one might compute the pair $(\theta^*, \phi^*) \in \Theta \times \Phi$ which minimises the empirical risk over $\mathcal{D}$ where $\Theta$ and $\Phi$ are chosen function classes. That is, computing

$$(\theta^*, \phi^*) = \underset{(\theta,\phi)\in\Theta\times\Phi}{\arg\min} \sum_{i=1}^{M} ||\mathbf{x}_i - \phi(\theta(\mathbf{x}_i))||^2.$$

If the function classes permit for computing the gradients of $\theta \in \Theta$ and $\phi \in \Phi$ then this computation may be done using gradient descent. For example, one could take $\Theta$ and $\Phi$ to be the classes of multi-layer perceptrons (MLPs) of appropriate input and output dimensions, as illustrated in Figure 17. MLPs are briefly summarised in Appendix C.

### 4.2.2 Variational autoencoders (VAEs)

Leaving aside, for now, how to learn one from data, variational autoencoders (VAEs) can informally be seen as an extension of autoencoders in which the encoder and decoder each output parameters of a distribution belonging to some pre-chosen distribution

family. Extending the notation used to define autoencoders, a variational autoencoder $(\Omega_{\mathbf{X}}, d, \mathcal{Q}_d, \mathcal{P}_n, \theta, \phi)$ consists of the sample space of the distribution of interest $\Omega_{\mathbf{X}} \subset \mathbb{R}^n$, a latent dimension $d \in \mathbb{Z}_{\geq 1}$, the parameter space $\mathcal{Q}_d$ of a family of $d-$dimensional conditional distributions denoted $q_\theta(\mathbf{z}|\mathbf{x})$, the parameter space $\mathcal{P}_n$ of a family of $n-$dimensional conditional distributions denoted $p_\phi(\mathbf{x}|\mathbf{z})$, an encoder $\theta : \mathbb{R}^n \to \mathcal{Q}_d$ and a decoder $\phi : \mathbb{R}^d \to \mathcal{P}_n$.

To illustrate the intended meaning of the newly-introduced parameter spaces $\mathcal{Q}_d$ and $\mathcal{P}_n$, one's encoder might yield the expectation vector and covariance matrix of a $d-$dimensional Gaussian $\mathcal{N}(\mu, \Sigma)$, i.e. $\mathcal{Q}_d$ could be the parameter space of the family of $d-$dimensional Gaussian distributions yielding

$$\mathcal{Q}_d = \{(\mu, \Sigma) : \mu \in \mathbb{R}^d, \Sigma \in \mathcal{S}^d_{++}\} = \mathbb{R}^d \times \mathcal{S}^d_{++}$$

where $\mathcal{S}^d_{++}$ denotes the set of all positive-definite matrices in $\mathbb{R}^{d \times d}$.

As the encoder of a VAE yields a $d-$dimensional distribution $q_\theta(\mathbf{z}|\mathbf{x})$ given the sample $\mathbf{x} \in \Omega_{\mathbf{X}}$, to obtain a latent $d-$dimensional representation $\mathbf{z}' \in \mathbb{R}^d$ of $\mathbf{x}$, one computes the parameters $\theta(\mathbf{x}) \in \mathcal{Q}_d$ of $q_\theta(\mathbf{z}|\mathbf{x})$, e.g. a $d-$dimensional Gaussian, via the encoder and samples $\mathbf{z}' \sim q_\theta(\mathbf{z}|\mathbf{x})$. To reconstruct $\mathbf{x}$ from its latent representation $\mathbf{z}'$, one computes the parameters $\phi(\mathbf{z}') \in \mathcal{P}_n$ of the $n-$dimensional reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}')$ via the decoder. Sampling $\mathbf{x}' \sim p_\phi(\mathbf{x}|\mathbf{z}')$ yields (ideally) a sufficiently-accurate reconstruction of the original sample $\mathbf{x}$. Achieving accurate reconstructions is done in a similar manner to autoencoders: by including a penalty term pertaining to reconstruction quality in the loss function used to learn the encoder and decoder.

Note that one's choice of $\mathcal{Q}_d$ and $\mathcal{P}_n$ should take into account the need to sample efficiently and so they should correspond to families of distributions which offer efficient means of sampling, e.g. Gaussians, as in Figure 18.

**Example 4.4** *Suppose $X_1 \sim \mathcal{N}(1, 2)$ and $X_3 \sim \mathcal{N}(-1, 1)$ are independent. Let $X_2 = X_1 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$ is independent of $X_1$ and $X_3$. If $X = (X_1, X_2, X_3)$ then $X \sim \mathcal{N}(\mu, \Sigma)$ with $\mu = (1, 1, -1)'$ and*

$$\Sigma = \begin{bmatrix} 2 & 2 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

*As such, $\Omega_{\mathbf{X}} = \mathbb{R}^3$. To obtain two-dimensional representations of samples $\mathbf{x} \in \Omega_{\mathbf{X}}$ (so $d = 2$), one might choose $\mathcal{Q}_2 = \mathbb{R}^2 \times \mathcal{S}^2_{++}$ and $\mathcal{P}_3 = \mathbb{R}^3 \times \mathcal{S}^3_{++}$. That is, we could fit a two-dimensional Gaussian to the latent space and a three-dimensional Gaussian to the reconstruction space. Knowing $X_2 = X_1 + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$, one might choose the encoder*

$$\begin{aligned} \theta : \Omega_{\mathbf{X}} &\to \mathcal{Q}_2 \\ (x_1, x_2, x_3) &\mapsto ((x_1, x_3), \sigma^2 I_2) \end{aligned}$$

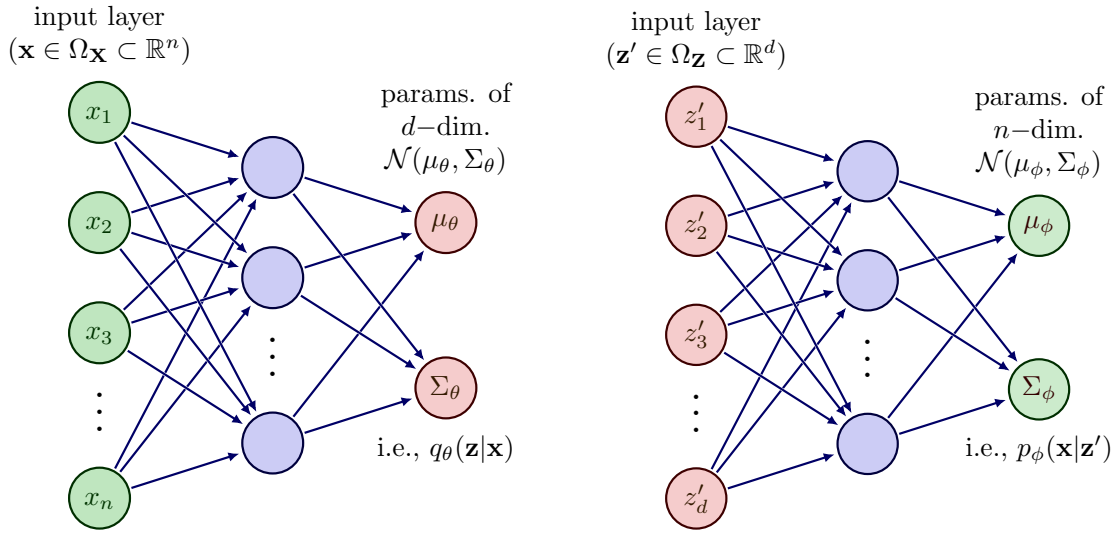


**Figure 18:** Left: an encoder which outputs parameters of the latent distribution $q_\theta(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\theta, \Sigma_\theta)$. Right: a decoder which outputs parameters of the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}') = \mathcal{N}(\mu_\phi, \Sigma_\phi)$ in which $\mathbf{z}' \sim q_\theta(\mathbf{z}|\mathbf{x})$.

*where $\sigma > 0$ is small. That is, for a sample $(x_1, x_2, x_3) \in \Omega_{\mathbf{X}}$, the encoder's output would yield the latent distribution $q_\theta(\mathbf{z}|\mathbf{x}) = \mathcal{N}((x_1, x_3), \sigma^2 I_2)$. As decoder, if all we desire is accurate reconstructions, we might choose*

$$\begin{aligned} \phi : \mathbb{R}^2 &\to \mathcal{P}_3 \\ (z_1, z_2) &\mapsto ((z_1, z_1, z_2), \sigma^2 I_3). \end{aligned}$$

*That is, for the latent sample $\mathbf{z}' = (z'_1, z'_2)$, the decoder's output would yield the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}') = \mathcal{N}((z'_1, z'_1, z'_2), \sigma^2 I_3)$. Ideally, sampling $(x'_1, x'_2, x'_3) \sim p_\phi(\mathbf{x}|\mathbf{z}')$ would yield a sample sufficiently similar to the original sample $\mathbf{x} = (x_1, x_2, x_3)$.*

*Note that in practice, one does not know the true distribution $p(\mathbf{x})$ and so hand-picking the encoder and decoder as in this example is infeasible. Typically, the encoder and decoder are learned from a dataset $\mathcal{D} \subset \Omega_{\mathbf{X}}$.*

At this point, a natural question arises: for which tasks is learning a VAE more appropriate than learning an autoencoder? The answer lies in the purpose of VAEs which is two-fold: 1) to perform sufficiently-accurate compression/decompression and 2) to produce a sufficiently regularised approximation of the latent space $\Omega_{\mathbf{Z}}$. The latter is ensured by how one learns a VAE from data, which we soon consider. In brief, when learning an autoencoder one never imposes restrictions on the latent space beyond encouraging the model to yield sufficiently-accurate reconstructions. As a result, the latent space of an autoencoder is not

well-structured. For example, for latent samples $\mathbf{z}_1, \mathbf{z}_2 \in \Omega_{\mathbf{Z}}$ which are 'close' in the latent space, their reconstructions are not necessarily 'close' in $\mathbb{R}^n$. VAEs seek to remedy this.

To learn a VAE from data, we look to maximise the evidence lower bound (ELBO) over some dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$. Over a single sample $\mathbf{x} \in \Omega_{\mathbf{X}}$, the ELBO is a tight lower bound of $\log(p(\mathbf{x}))$. As such, maximising the ELBO over $\mathcal{D}$ can be seen as performing approximate maximum-likelihood estimation over $\mathcal{D}$ (sometimes referred to as evidence maximisation). To derive the ELBO, first note that given a decoder $\phi$ (which parameterises the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z})$), we may express the marginal distribution $p(\mathbf{x})$ using $p_\phi(\mathbf{x}|\mathbf{z})$ as

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z} = \int p_\phi(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}. \tag{2}$$

Using Equation 2, along with the encoder $\theta$ (which parameterises the latent distribution $q_\theta(\mathbf{z}|\mathbf{x})$) we derive the ELBO over a single sample $\mathbf{x} \in \Omega_{\mathbf{X}}$:

$$\begin{aligned}
\log(p(\mathbf{x})) &= \log\left(\int p_\phi(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}\right) \\
&= \log\left(\int q_\theta(\mathbf{z}|\mathbf{x}) \frac{p_\phi(\mathbf{x}|\mathbf{z}) p(\mathbf{z})}{q_\theta(\mathbf{z}|\mathbf{x})} d\mathbf{z}\right) \\
&= \log\left(\mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\frac{p_\phi(\mathbf{x}|\mathbf{Z}) p(\mathbf{Z})}{q_\theta(\mathbf{Z}|\mathbf{x})}\right]\right) \\
&\geq \mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_\phi(\mathbf{x}|\mathbf{Z}) p(\mathbf{Z})}{q_\theta(\mathbf{Z}|\mathbf{x})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\log(p_\phi(\mathbf{x}|\mathbf{Z}))\right] - \mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{q_\theta(\mathbf{Z}|\mathbf{x})}{p(\mathbf{Z})}\right)\right] \\
&= \mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\log(p_\phi(\mathbf{x}|\mathbf{Z}))\right] - D_{\mathrm{KL}}(q_\theta(\mathbf{Z}|\mathbf{x}) || p(\mathbf{Z})) \\
&=: \text{ELBO}
\end{aligned}$$

in which the inequality arises due to Jensen's inequality, as in $\mathbb{E}[\log(f(\mathbf{Z}))] \leq \log(\mathbb{E}[f(\mathbf{Z})])$, and $D_{\mathrm{KL}}(Q||P)$ denotes the KL-divergence between distributions $Q$ and $P$, which is detailed in Appendix B. The effect of maximising

$$\text{ELBO} = \mathbb{E}_{\mathbf{Z} \sim q_\theta(\mathbf{z}|\mathbf{x})}\left[\log(p_\phi(\mathbf{x}|\mathbf{Z}))\right] - D_{\mathrm{KL}}(q_\theta(\mathbf{Z}|\mathbf{x}) || p(\mathbf{Z}))$$

over a dataset is often explained term-by-term. The first term is a principled measure of the VAE's ability to reconstruct latent representations, as with autoencoders. The second term is a principled measure of the similarity of the latent distribution $q_\theta(\mathbf{z}|\mathbf{x})$ and the prior $p(\mathbf{z})$. The prior is chosen before training and the most common choice is a $d-$dimensional standard Gaussian, i.e. $p(\mathbf{z}) = \mathcal{N}(0, I_d)$. As such, minimising the negative KL-divergence between the latent distribution and said prior is often interpreted

as encouraging the learning of the encoder such that latent representations are distributed according to the prior. This is particularly useful in the case that one is learning a VAE to generate new samples from $\Omega_{\mathbf{X}}$: post-training, sample $\mathbf{z}' \sim p(\mathbf{z})$, compute the parameters $\phi(\mathbf{z}')$ of the reconstruction distribution $p_\phi(\mathbf{x}|\mathbf{z}')$ via the decoder and sample from it. Note that using a VAE for generative purposes does not invoke the use of the encoder, only the decoder is required post-training.

Given a dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$, we learn a VAE by choosing function classes $\Theta$ and $\Phi$ (e.g. MLPs as in Figure 18) and computing

$$\arg\max_{(\theta,\phi)\in\Theta\times\Phi} \left[ \sum_{i=1}^{M} \mathbb{E}_{\mathbf{Z}\sim q_\theta(\mathbf{z}|\mathbf{x}_i)} \left[\log(p_\phi(\mathbf{x}_i|\mathbf{Z}))\right] - D_{\mathrm{KL}}(q_\theta(\mathbf{Z}|\mathbf{x}_i)||p(\mathbf{Z})) \right].$$

In practice, after choosing a latent dimension $d$, we often take $p(\mathbf{z}) = \mathcal{N}(0, I_d)$, $\mathcal{Q}_d = \mathbb{R}^d \times \mathcal{S}^d_{++}$ and $\mathcal{P}_n = \mathbb{R}^n \times \mathcal{S}^n_{++}$, i.e. Gaussians for the latent and reconstruction distribution families and the standard $d-$dimensional Gaussian for the prior. A benefit of these choices is that it yields a differentiable and easy-to-implement closed form for the KL-divergence term in the ELBO [14, Subsection 2.2]. Additionally, the expectation pertaining to the reconstruction term is typically approximated via a single sample, boiling down to a mean square error term.

### 4.3 Completing the Overview of CMPCs

In Subsection 4.1, we motivated continuous mixtures using Monte Carlo (MC) integration, which are mixture models of the form

$$p(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} p_\phi(\mathbf{x}|\mathbf{z}_i)$$

for latent samples $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$ where $\mathbf{Z}$ is continuously distributed. We saw that if each component $p_\phi(\mathbf{x}|\mathbf{z}_i)$ is fit using a PC (e.g. an FFM or CLT) then $p(\mathbf{x})$ itself can be represented as a PC and is referred to as a continuous mixture of probabilistic circuits (PCs). From here, the final piece in overviewing CMPCs is how one might compute the parameters of the $N$ component distributions $p_\phi(\mathbf{x}|\mathbf{z}_1), \ldots, p_\phi(\mathbf{x}|\mathbf{z}_N)$ from their sampled latents $\mathbf{z}_1, \ldots, \mathbf{z}_N$. As detailed earlier, choosing simple subclasses of PCs, such as FFMs and CLTs, removes the need to use latents to learn the computational graph or the weights of the PCs fit to the component distributions. In the case of FFMs, the computational graph $\mathcal{G}$ is fixed independently of the data and there are no weights $\mathbf{w}$. In the case of CLTs, the computational graph $\mathcal{G}$ and weights $\mathbf{w}$ are learned from data by computing the maximum spanning tree in which weights correspond to mutual information between model variables. As such, fitting an FFM/CLT to one of the $N$ components amounts to computing the parameters $\theta$ (not to be confused with the notation used for encoders) of its leaf nodes given the component's latent sample.

To fit the $N$ components, Correia et al. took inspiration from VAEs and learn a decoder which outputs the parameters of FFM- or CLT-structured component distributions given $\mathbf{z}' \in \Omega_{\mathbf{Z}}$ as input. That is, a decoder $\phi : \mathbb{R}^d \to \mathcal{P}_n$ is learned where $d$ is the latent dimension and $\mathcal{P}_n$ is the parameter space corresponding to an FFM or CLT. To clarify when the parameters of a component distribution $p(\mathbf{x}|\mathbf{z}')$ are determined by the output $\phi(\mathbf{z}')$ of a decoder we write $p_\phi(\mathbf{x}|\mathbf{z}')$.

**Example 4.5** *Suppose we would like to model the joint distribution $p(\mathbf{x})$ pertaining to Binary MNIST using a CMPC with $N = 16$ components, so*

$$p(\mathbf{x}) = \frac{1}{16}\sum_{i=1}^{16} p_\phi(\mathbf{x}|\mathbf{z}_i).$$

*Further, suppose we choose the latent dimension $d = 4$, the prior $p(\mathbf{z}) = \mathcal{N}(0, I_4)$ and have learned a decoder $\phi$. Using $\phi$, we sample $\mathbf{z}_1, \ldots, \mathbf{z}_{16} \sim p(\mathbf{z})$ and fit FFMs to the 16 component distributions $p_\phi(\mathbf{x}|\mathbf{z}_1), \ldots, p_\phi(\mathbf{x}|\mathbf{z}_{16})$. From here, all one needs to complete their CMPC is a choice of parameter space $\mathcal{P}_{784}$. Samples of Binary MNIST consist of 784 features, each taking on a value of 0 or 1, i.e. $\Omega_{\mathbf{X}} = \{0, 1\}^{784}$. As such, fitting an FFM to each component distribution yields component distributions of the form*

$$p_\phi(\mathbf{x}|\mathbf{z}_i) = \prod_{j=1}^{784} p(x_j|\mathbf{z}_i)$$

*in which $X_j|\mathbf{z}_i \sim \text{Ber}(\hat{p}_j^i)$. That is, for the $i^{th}$ component $p_\phi(\mathbf{x}|\mathbf{z}_i)$, the $j^{th}$ pixel is attributed a Bernoulli distribution whose parameter $\hat{p}_j^i$ is determined by some latent sample $\mathbf{z}_i$ according to the output of the decoder, i.e. the parameter space is $\mathcal{P}_{784} = [0, 1]^{784}$ and so $\phi : \mathbb{R}^4 \to [0, 1]^{784}$. Our CMPC is thus of the form*

$$p(\mathbf{x}) = p(x_1, \ldots, x_{784}) = \frac{1}{16}\sum_{i=1}^{16} p_\phi(x_1, \ldots, x_{784}|\mathbf{z}_i) = \frac{1}{16}\sum_{i=1}^{16}\prod_{j=1}^{784} \text{Ber}(x_j|\hat{p}_j^i)$$

*where $\text{Ber}(x|p)$ denotes the probability mass function of $X \sim \text{Ber}(p)$ and $(\hat{p}_1^i, \ldots, \hat{p}_{784}^i) = \phi(\mathbf{z}_i)$ for $i = 1, \ldots, 16$.*

Note that once the decoder $\phi$ is learned, when constructing a CMPC at test time we may choose the number of components $N_{\text{test}}$ freely: simply draw $N_{\text{test}}$ latent samples and pass them through the decoder. For example, we could train $\phi$ using $N_{\text{train}} = 16$ components and at test time compile CMPCs consisting of $N_{\text{test}} = 32$ components for what would usually be a better estimator than one with $N_{\text{test}} = 16$ components.

### 4.3.1 Training the decoder

Given a dataset $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$, a latent dimension $d$ and a fixed parameter space for the structure used to fit the $N$ component distributions (the parameter spaces of FFMs or CLTs), the method used to learn the decoder $\phi : \mathbb{R}^d \to \mathcal{P}_n$ is maximum-likelihood estimation over $\mathcal{D}$. The reason that maximum-likelihood estimation is possible in learning CMPCs is that PCs are tractable with respect to **evi** queries and so evaluating $p(\mathbf{x})$ exactly for a particular realisation $\mathbf{x} \in \Omega_{\mathbf{X}}$ is tractable, which is not the case with VAEs. This is to say that there is no need to train according to approximate maximum likelihood using some sort of evidence lower bound as with VAEs. An interesting benefit of this is that we do not need to learn an encoder in conjunction with the decoder: we learn the decoder directly. Choosing a function class $\Phi$ to which the decoder $\phi$ will belong, we sample latents $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$ and compute

$$\begin{aligned}
\arg\max_{\phi \in \Phi} \sum_{j=1}^{M} \log(p(\mathbf{x}_j)) &= \arg\max_{\phi \in \Phi} \sum_{j=1}^{M} \log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right) \\
&= \arg\min_{\phi \in \Phi} \left[-\frac{1}{M}\sum_{j=1}^{M} \log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right)\right] \\
&= \arg\min_{\phi \in \Phi} \mathrm{NLL}(\phi, \mathcal{D})
\end{aligned}$$

where $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$ and NLL stands for negative log-likelihood. Correia et al. choose $\Phi$ to be the class of MLPs with $d$ input nodes and $n$ output nodes. With this choice, the parameters of the decoder are learned using gradient descent. Once the decoder is learned, constructing a CMPC amounts to choosing a number of components $N_{\text{test}}$, sampling $\mathbf{z}_1, \ldots, \mathbf{z}_{N_{\text{test}}} \sim p(\mathbf{z})$ and computing the parameters $\phi(\mathbf{z}_1), \ldots, \phi(\mathbf{z}_{N_{\text{test}}})$ to obtain the CMPC

$$p(\mathbf{x}) = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} p_\phi(\mathbf{x}|\mathbf{z}_i).$$

To avoid overfitting, Correia et al. incorporate early stopping with a patience of 15. That is, after each epoch of training, the validation error corresponding to the newly-proposed parameters is computed. Then, the old parameters are replaced only in the case that said validation error is lower than the validation error corresponding to the old parameters. If no change in parameters occurs after 15 consecutive epochs then the model loses patience and training ends. It is possible that the decoders learned by Correia et al. would benefit from further regularisation techniques, such as dropout, but they are not considered in their work.

### 4.3.2 Latent optimisation

As the compilation of CMPCs begins with randomly sampling $\mathbf{z}_1, \ldots, \mathbf{z}_N \sim p(\mathbf{z})$, a natural question is how much better CMPCs perform when the latent samples $\mathbf{z}_1, \ldots, \mathbf{z}_N \in \Omega_{\mathbf{Z}}$ are instead computed such that the corresponding CMPC maximises some principled measure. That is, does treating the latent samples used in constructing a CMPC as parameters to-be-optimised yield better CMPCs?

This idea is employed by Correia et al. and is referred to as latent optimisation in line with similar approaches in literature [2, 31]. In treating the latent samples as parameters, one computes

$$\underset{\mathbf{z}_1,\ldots,\mathbf{z}_N \in \Omega_{\mathbf{Z}}}{\arg\max} \sum_{j=1}^{M} \log \left( \frac{1}{N} \sum_{i=1}^{N} p_\phi(\mathbf{x}_j | \mathbf{z}_i) \right)$$

using gradient descent. The effect of latent optimisation can be seen in Subsection 4.4. In cases where the number of components $N_{\text{test}}$ at test time is relatively small compared to the number of components employed during training, CMPCs learned using latent optimisation perform similarly to non-latent optimised CMPCs in which eight times as many components are used. For example, if the decoder was learned using $N_{\text{train}} = 2^{13}$ components then at test time, a latent optimised-CMPC with $N_{\text{test}} = 2^7$ components can, at times, perform as well as a CMPC consisting of $N_{\text{test}} = 2^{10}$ components to which latent optimisation was not applied. In cases where the number of components at test time is comparable or higher than the number of components used during training, the effect of latent optimisation is relatively small. This is due to the relatively low variance of such MC estimators at such a number of components.

Note that Correia et al. considered the effect on generalisation of CMPCs with latent optimisation applied during training [9, Latent Optimisation] but concluded that it would lead to undesirable behaviour, such as overfitting, and so did not employ it during training.

## 4.4 Results Obtained by CMPCs

Correia et al. benchmarked CMPCs on density estimation and sample fidelity. The effects of applying latent optimisation, changing the latent dimension $d$ and component number $N$ during training and testing were addressed. Additionally, comparisons between CMPCs to which FFMs/CLTs were fit to the $N$ components are presented. For density estimation, CMPCs were compared to the best performing PC at the time of their work (2022). For sample quality, CMPCs were compared to Small Einet and Big Einet, PCs consisting of five million and 84 million parameters respectively.

Whether a CMPC used FFMs or CLTs to fit the $N$ component distributions is denoted by $\text{cm}(S_{\text{F}})$ or $\text{cm}(S_{\text{CLT}})$ respectively. If latent optimisation was used in learning a CMPC then it is denoted by $\text{LO}(\text{cm}(S_{\text{F}}))$ or $\text{LO}(\text{cm}(S_{\text{CLT}}))$.

| Dataset | BestPC | cm($S_{\mathrm{F}}$) | cm($S_{\mathrm{CLT}}$) | LO(cm($S_{\mathrm{CLT}}$)) | Dataset | BestPC | cm($S_{\mathrm{F}}$) | cm($S_{\mathrm{CLT}}$) | LO(cm($S_{\mathrm{CLT}}$)) |
|---|---|---|---|---|---|---|---|---|---|
| accid. | **26.74** | 33.27 | 28.69 | 28.81 | jester | 52.46 | **51.93** | 51.94 | 51.94 |
| ad | 16.07 | 18.71 | 14.76 | **14.42** | kdd | **2.12** | 2.13 | **2.12** | **2.12** |
| baudio | 39.77 | **39.02** | **39.02** | 39.04 | kosarek | 10.60 | 10.71 | 10.56 | **10.55** |
| bbc | 248.33 | **240.19** | 242.83 | 242.79 | msnbc | **6.03** | 6.14 | 6.05 | 6.05 |
| bnetflix | 56.27 | 55.49 | **55.31** | 55.36 | msweb | 9.73 | 9.68 | 9.62 | **9.60** |
| book | 33.83 | 33.67 | 33.75 | **33.55** | nltcs | **5.99** | **5.99** | **5.99** | **5.99** |
| c20ng | 151.47 | 148.24 | **148.17** | 148.28 | plants | 12.54 | 12.45 | **12.26** | 12.27 |
| cr52 | 83.35 | 81.52 | **81.17** | 81.31 | pumbs | **22.40** | 27.67 | 23.71 | 23.70 |
| cwebkb | 151.84 | 150.21 | 147.77 | **147.75** | tmovie | 50.81 | **48.69** | 49.23 | 49.29 |
| dna | **79.05** | 95.64 | 84.91 | 84.58 | tretail | 10.84 | 10.85 | 10.82 | **10.81** |

**Table 3:** Mean negative log-likelihoods attained by CMPCs on the test sets of 20 density estimation datasets [24]. CMPCs, which are listed under cm($S_{\mathrm{F}}$), cm($S_{\mathrm{CLT}}$) and LO(cm($S_{\mathrm{CLT}}$)), are compared to the best performing PC (BestPC) at the time of their work which consisted of five PC learning methods: Einets [33], LearnSPN [16], ID-SPN [37], RAT-SPN [34] and HCLT [25]. All models pertaining to cm($S_{\mathrm{F}}$) and cm($S_{\mathrm{CLT}}$) were trained with $N_{\mathrm{train}} = 2^{10}$ components and at test time used $N_{\mathrm{test}} = 2^{13}$ components except for latent-optimised models which used $N_{\mathrm{test}} = 2^{10}$ components. Taken from [9, Table 1].

### 4.4.1 Density estimation

For benchmarking density estimation, 20 binary datasets were used. These 20 datasets, whose names are given in Table 3, are binary density estimation datasets often used in the PC literature [24]. All datasets pertain to binary model variables whose number ranges from 16 to 1556 between datasets. The number of samples within each dataset varies heavily too, ranging from fewer than 2,000 to hundreds of thousands. Additionally, how each dataset is split for training, validation and testing varies. Such details can be found on the GitHub repository in which the datasets are held [24]. It is important to note that the extent to which the features of each distribution correlate differs. For example, as noted by Correia et al., the `accidents`, `ad`, `dna` and `pumbs` datasets consist of features which depend highly on each other. This is reflected in Table 3 in which we see that cm($S_{\mathrm{CLT}}$) and LO(cm($S_{\mathrm{CLT}}$)) perform significantly better than cm($S_{\mathrm{F}}$) for these datasets.

Each CMPC fit to these datasets used a latent dimension of $d = 4$, the prior $p(\mathbf{z}) = \mathcal{N}(0, I_4)$, and $N_{\mathrm{train}} = 2^{10}$ components during training. Each decoder $\phi : \mathbb{R}^4 \to \mathcal{P}_n$ was fit using an MLP consisting of four hidden layers with LeakyReLU as activation functions with hyperparameter $a = 0.01$, i.e.

$$\sigma_{0.01}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if } x \leq 0. \end{cases}$$

The number of nodes in the four hidden layers are 4, $4 + \lfloor \frac{f-4}{4} \rfloor$, $4 + 2\lfloor \frac{f-4}{4} \rfloor$ and $4 + 3\lfloor \frac{f-4}{4} \rfloor$ where $f$ denotes the number of features in the given dataset. For example, if the number of features for a given dataset is $f = 20$ then the hidden layers of the corresponding MLP consist of 4, 8, 12 and 16 nodes respectively. As illustrated in Table 3, for 16 of the 20 density estimation datasets, CMPCs set state of the art mean negative log-likelihoods

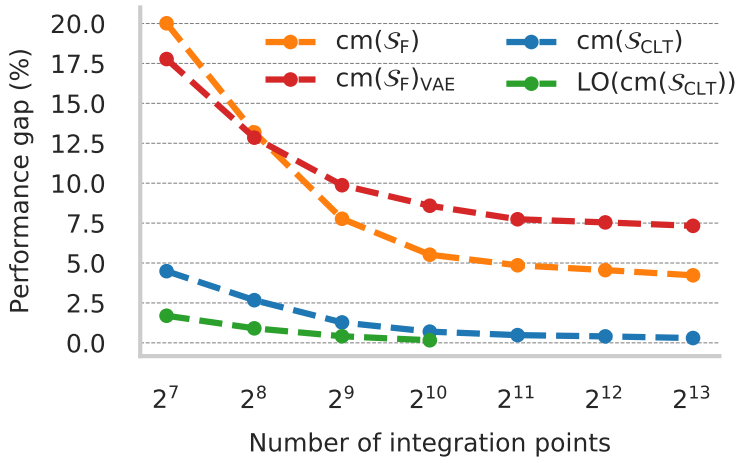


**Figure 19:** The effect of latent optimisation. Relative performance shown. Number of integration points refers to the number of components $N$. Taken from [9, Figure 1].

among PC-based approaches. The results of this benchmark did not state the parameter count of the competing models and so the per-parameter-performance is not considered.

As for latent optimisation, its use effectively reduces the number of components $N_{\text{test}}$ needed to attain a given mean NLL. Figure 19 illustrates the mean performance gap between the lowest mean NLL across all 20 datasets in Table 3 and what each model with $2^7, \ldots, 2^{13}$ components attained. We see immediately that CMPCs whose base components were fit using CLTs required eight times fewer components when applying latent optimisation to reach the performance gap attained by its non-latent optimised equivalent. This example illustrates precisely what the authors sought to achieve in applying latent optimisation: at test time, instead of randomly sampling latents, search the latent space for latent samples which yield CMPCs that generalise better. The precise log-likelihoods attained by $\text{cm}(S_{\text{F}})$, $\text{cm}(S_{\text{CLT}})$ and $\text{LO}(\text{cm}(S_{\text{CLT}}))$ with a varying number of components $N \in \{2^7, 2^8, \ldots, 2^{13}\}$ is provided in [9, Figure 6].

### 4.4.2 Negative log-likelihoods and sample quality

The image datasets used in benchmarking CMPCs were Binary MNIST, MNIST, Fashion-MNIST and Street View House Numbers (SVHN). MNIST is a dataset of 70,000 images of handwritten digits (0 to 9, i.e. 10 classes) in which each sample is 28 by 28 with pixel values in $\{0, \ldots, 255\}$. Fashion-MNIST is specified precisely as MNIST except that its samples depict items of clothing. It is generally considered to be a more complex distribution to encode than MNIST as items of clothing are less neatly-structured than handwritten digits. SVHN consists of around 100,000 images of single digit house number signs (10 classes,

| **Dataset** | **Small Einet** (5M) | **Big Einet** (84M) | **CMPC** |
|---|---|---|---|
| Binary MNIST | 0.206 | 0.184 | **0.179** (4.8M) |
| MNIST | 1.490 | 1.415 | **1.282** (100K) |
| Fashion-MNIST | 3.938 | 3.737 | **3.546** (100K) |
| SVHN | 6.442 | **5.961** | 6.307 (300K) |

**Table 4:** Mean negative log-likelihood (NLL) per dimension measured in bits ($\log_2$ used for their computation). 'Per dimension' refers to the fact that each NLL is normalised according to the dimension of its corresponding distribution, e.g. divided by 784 for Binary MNIST and MNIST. Number of parameters of decoders taken from [9, Table 3]. The Binary MNIST model had 1.2M parameters for the $\mathrm{cm}(S_{\mathrm{F}})$ models and 4.8M for the $\mathrm{cm}(S_{\mathrm{CLT}})$.

| | | Number of integration points at test time | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Model | N. Param | $2^7$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
| $\mathrm{CM}(S_{\mathrm{F}})$ (LO) | 1.2M | 167.29 (144.00) | 150.67 (135.89) | 138.55 (129.15) | 129.24 (123.44) | 121.96 | 116.42 | 112.03 | 108.69 |
| $\mathrm{CM}(S_{\mathrm{CLT}})$ (LO) | 4.8M | 127.59 (114.02) | 119.09 (110.02) | 113.15 (107.14) | 108.30 (104.37) | 104.50 | 101.55 | 99.23 | 97.48 |

**Table 5:** Binary MNIST mean negative log-likelihoods (NLLs) at test time for $\mathrm{CM}(S_{\mathrm{F}})$ and $\mathrm{CM}(S_{\mathrm{CLT}})$ trained with $N_{\mathrm{train}} = 2^{14}$ components. All NLLs measured in nats (natural logarithm used for their computation). In parentheses we report test mean negative log-likelihoods obtained via latent optimisation. Taken from [9, Table 2].

as with MNIST and Fashion-MNIST) in which each sample is 32 by 32 and pixel values are in accordance with an 8-bit RGB standard. As such, each pixel can take on one of $2^{24} = 16,777,216$ possible values.

For each of these datasets, the CMPCs learned used a latent dimension of $d = 16$, the prior $p(\mathbf{z}) = \mathcal{N}(0, I_{16})$ and the number of components used during training was $N_{\mathrm{train}} = 2^{14}$. Distinctions are made in the architectures of the decoders. For Binary MNIST, an MLP with six hidden layers was used to fit the decoder. For MNIST, Fashion-MNIST and SVHN, deconvolution neural networks [46, subsection 2.1] were employed to fit the decoders. Categorical distributions of 256 categories were attributed to the leaf nodes pertaining to the pixels. For MNIST and Fashion-MNIST, the 256 categories pertained to 256 gray scale values of the pixels. For SVHN, as the pixels of samples are coloured according to an 8-bit RGB standard, to fit each pixel value with 256 categories, the authors simplified samples in some way though it is unclear precisely how. This naturally results in some loss of information. The number of parameters pertaining to each model is illustrated in Table 4 in their corresponding entry of the column titled **CMPC**. We see that CMPCs remarkably offer lower mean NLLs than massively parameterised PCs. Recall that Small Einet and Big Einet consist of 5 and 84 million parameters respectively.

An assessment of how changing the number of components $N_{\mathrm{test}} \in \{2^7, \ldots, 2^{14}\}$ at test time for Binary MNIST is illustrated in Table 5. There we see that in some cases,

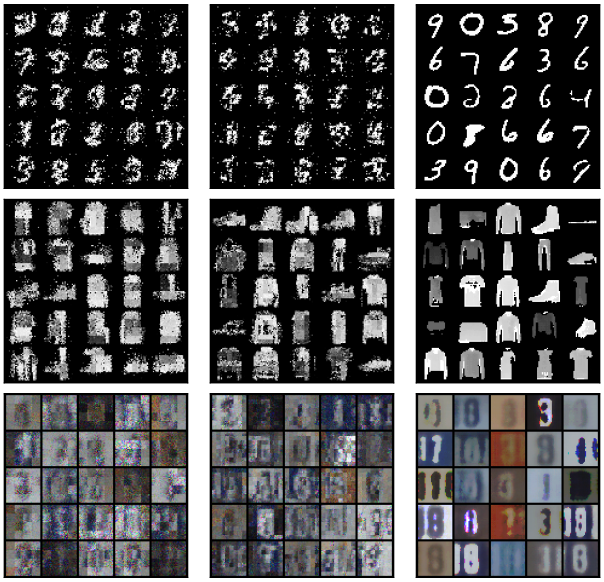

**Figure 20:** Samples of MNIST, Fashion-MNIST and SVHN drawn from Small Einet (left), Big Einet (centre) and a CMPC (right). Taken from [9, Figure 2].

particularly for the CLT-based CMPCs, latent optimisation can reduce the number of components needed to attain a given mean NLL by around a factor of four. This helps produce far more compact CMPCs. Note additionally that a latent-optimised CMPC using CLTs with $N = 2^{14}$ components attained a mean NLL of around 97.5 on Binary MNIST, improving on the performance offered by the most highly-parameterised PC at the time, consisting of 90 million parameters, which attained a mean NLL of 100 nats.

Note that the discrepancy in the scale of negative log-likelihood values (NLLs) illustrated in Table 4 and Table 5 is due to the former having NLLs represented in bits ($\log_2$ used for computations) and having been normalised according to the dimension of the corresponding distribution. For example, the NLLs illustrated for Binary MNIST and MNIST (in bits) are divided by 784. All NLLs in Table 5 are represented in nats (natural logarithm used for NLL computations) and are not normalised.

Finally, in evaluating the sample quality of CMPCs, consider Figure 20 in which we illustrate samples drawn from Small Einet, Big Einet and CMPCs learned on MNIST, Fashion-MNIST and SVHN. We see that CMPCs produce samples, seen in the rightmost column of the figure, which do not pixelate nearly as harshly as those drawn from Small or Big Einet. For the samples of MNIST, the underlying structure of digits is far more clear and leaves very little ambiguity. Similar comparisons can be made for the samples drawn of Fashion-MNIST and SVHN.

| Dataset | $\mathrm{dm}(S_F)$ | $\mathrm{dm}(S_F^W)$ | $\mathrm{dm}(S_F^{EM})$ | $\mathrm{cm}(S_F)$ | Dataset | $\mathrm{dm}(S_F)$ | $\mathrm{dm}(S_F^W)$ | $\mathrm{dm}(S_F^{EM})$ | $\mathrm{cm}(S_F)$ |
|---|---|---|---|---|---|---|---|---|---|
| accidents | 42.58 | 40.61 | 35.38 | **33.94** | jester | 55.32 | 53.54 | 52.54 | **52.03** |
| ad | 104.57 | 97.79 | 24.91 | **20.42** | kdd | 6.81 | 2.15 | 2.14 | **2.13** |
| baudio | 42.24 | 40.41 | 39.76 | **39.14** | kosarek | 16.20 | 11.17 | 10.88 | **10.75** |
| bbc | 281.88 | 288.31 | 252.82 | **241.54** | msnbc | 6.36 | 6.12 | **6.03** | 6.15 |
| bnetflix | 58.19 | 57.00 | 56.34 | **55.71** | msweb | 18.29 | 11.36 | 10.00 | **9.72** |
| book | 41.72 | 35.61 | 34.66 | **33.79** | nltcs | 6.16 | 6.01 | 6.00 | **6.00** |
| c20ng | 163.04 | 157.80 | 151.79 | **149.10** | plants | 16.66 | 14.41 | 13.44 | **12.65** |
| cr52 | 104.91 | 98.79 | 87.07 | **82.33** | pumbs | 46.59 | 42.90 | 32.84 | **28.50** |
| cwebkb | 176.60 | 170.90 | 154.75 | **151.00** | tmovie | 66.94 | 61.64 | 52.80 | **49.12** |
| dna | 101.93 | 98.14 | **94.46** | 96.11 | tretail | 18.35 | 11.42 | 10.90 | **10.85** |

**Table 6:** Mean negative log-likelihoods on the test sets of 20 standard density estimation benchmarks for plain mixtures with non-learnable equally-probable weights trained using the Adam optimiser ($\mathrm{dm}(S_F)$), with learnable weights also using Adam ($\mathrm{dm}(S_F^W)$), with learnable weights trained using EM ($\mathrm{dm}(S_F^{EM})$), and a CMPC with FFMs fit to mixture components $\mathrm{cm}(S_\mathrm{F})$ evaluated with $N_\mathrm{test} = 2^{10}$ components at test time. All models had the same structure with $N_\mathrm{train} = 2^{10}$ components during training and were trained for up to 300 epochs with early stopping. Taken from [9, Table 6].

#### 4.4.3 Comparing performance of CMPCs to regular mixture models

A natural question when shown CMPCs as approximations of continuous latent variable models is whether or not they perform better than regular discrete mixture models. That is, does allowing for continuously distributed latent variables improve generalisation? Discrete mixture models are of the form

$$p_\mathrm{mix}(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} p(\mathbf{x}|\mathbf{Z} = i).$$

As with the CMPCs used for density estimation, the individual components of these plain mixture models were fit using FFMs. They are denoted by $\mathrm{dm}(S_F)$ in Table 6. Additionally, Correia et al. considered discrete mixture models in which the weights $w_1, \ldots, w_N \geq 0$ of the $N$ components are learned, as in

$$p_\mathrm{mix}(\mathbf{x}) = \sum_{i=1}^{N} w_i p(\mathbf{x}|\mathbf{Z} = i)$$

with $\sum_{i=1}^{N} w_i = 1$. These are learned separately using gradient descent (denoted $\mathrm{dm}(S_F^W)$) and the EM-algorithm (denoted $\mathrm{dm}(S_F^{EM})$). All three model types are compared to CMPCs with FFMs fit to the $N$ components (denoted $\mathrm{cm}(S_F)$, as before).

As illustrated in Table 6, CMPCs outperform the considered discrete mixture models on 18 of the 20 distributions considered. The authors note that latent optimisation is not

applied to the CMPCs whose results are shown in the table. These results are particularly interesting as, for a fixed number of components at test time, the class of CMPCs is necessarily less expressive than the class of discrete mixture models. Despite this, the use of a decoder in parameterising each component distribution, as in CMPCs, yields mixture models which generalise better. The authors explain this as the use of a decoder inherently regularising the parameterisation process of the mixture components through parameter sharing. This contrasts with the case of discrete mixture models in each component distribution is parametrised entirely independently. The authors additionally note that the use of a decoder may help to ensure that CMPCs do not consist of components which are effectively 'dead weight', as has been observed in discrete mixture models [11].

# 5 Hybrid CMPCs

As the intended applications of CMPCs, as proposed by Correia et al., were density estimation and image generation (sampling), their training objective was naturally in line with maximum likelihood estimation. That is, the loss function minimised during training each CMPC was of the form

$$\mathcal{L}(\phi, \mathcal{D}) = -\frac{1}{M}\sum_{j=1}^{M} \log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right) =: \mathrm{NLL}(\phi, \mathcal{D})$$

where $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\} \subset \Omega_{\mathbf{X}}$. This is a natural approach to learning a generative model, especially when one's objective is to apply the model to tasks which are generative in nature: sampling, density estimation, anomaly detection, etc.

In this section, we consider learning CMPCs for tasks that are discriminatory in nature such as classification. Further, we employ hybrid loss functions which encourage the learning of CMPCs applicable to both generative and discriminatory tasks. We refer to CMPCs learned in this hybrid fashion as hybrid CMPCs. We assess hybrid CMPCs by applying them to the classification of incomplete samples of Binary MNIST and to sample generation. Therein lies the purpose of benchmarking hybrid CMPCs: to evaluate the extent to which we can maintain the generative ability of CMPCs while encouraging some amount of discriminative learning.

## 5.1 Learning Discriminative CMPCs

Learning discriminative CMPCs amounts to employing a loss function which encourages discriminative learning during training. For a CMPC which performs classification, a natural choice for the loss function is one pertaining to the mean cross-entropy of class labels predicted by the model and those observed in a given labelled dataset

$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_M, y_M)\} \subset \Omega_{\mathbf{X}} \times \Omega_Y$. Such a loss function is given by

$$\begin{aligned}
\mathcal{L}(\phi, \mathcal{D}) &= -\frac{1}{M} \sum_{j=1}^{M} \log\left(p_\phi(y_j|\mathbf{x}_j)\right) \\
&= -\frac{1}{M} \sum_{j=1}^{M} \log\left(\frac{p_\phi(\mathbf{x}_j, y_j)}{p_\phi(\mathbf{x}_j)}\right) \\
&= -\frac{1}{M} \sum_{j=1}^{M} \log\left(\frac{\sum_{i=1}^{N} p_\phi(\mathbf{x}_j, y_j|\mathbf{z}_i)}{\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)}\right) \\
&=: \mathrm{CE}(\phi, \mathcal{D}).
\end{aligned}$$

Note that PCs are tractable with respect to **marg** queries and so computing $p_\phi(\mathbf{x}_j|\mathbf{z}_i)$, i.e. marginalising out the class label $y$, is straightforward in practice. Alternatively, one could marginalise the class label out manually, as in

$$p_\phi(\mathbf{x}_j|\mathbf{z}_i) = \sum_{y \in \Omega_Y} p_\phi(\mathbf{x}_j, y|\mathbf{z}_i),$$

but this is a cumbersome task if the number of class labels $|\Omega_Y|$ is sufficiently high.

As detailed in Subsection 3.4, the application of PCs to classification is straightforward. As a brief reminder, given a sample $\mathbf{x} \in \Omega_{\mathbf{X}}$, its corresponding label $y \in \Omega_Y$, according to our model, is given by

$$\arg\max_{y \in \Omega_Y} p(y|\mathbf{x}) = \arg\max_{y \in \Omega_Y} \frac{p_\phi(\mathbf{x}, y)}{p_\phi(\mathbf{x})} = \arg\max_{y \in \Omega_Y} p_\phi(\mathbf{x}, y)$$

which requires us to answer $|\Omega_Y|$ **evi** queries using our PC. In Subsection 5.3 we benchmark entirely (and partially) discriminative CMPCs on samples of Binary MNIST in which differing portions of pixel values are missing at random.

### 5.2 Learning Hybrid CMPCs

If a class of generative probabilistic models is tractable with respect to both **evi** and **marg** queries then such models can be learned in a generative fashion (using negative log-likelihood loss functions) or in a discriminative fashion (using cross-entropy loss functions). More interestingly, such models can be learned in a manner which encourages both generative and discriminative learning simultaneously. That is, they can be learned in a hybrid fashion. The earliest work known to us which utilises the learning of generative models in a hybrid fashion [3, Section 3] proposed a loss function akin to

$$\mathcal{L}_\lambda(\phi, \mathcal{D}) = \lambda \mathrm{CE}(\phi, \mathcal{D}) + (1 - \lambda)\mathrm{NLL}(\phi, \mathcal{D})$$

where $\lambda \in [0, 1]$ is a mixing hyperparameter which dictates the extent to which we encourage discriminative learning (due to the cross-entropy term) and generative learning (due to the negative log-likelihood term). For example, $\lambda = 1$ encourages entirely discriminative learning, as $\mathcal{L}_1(\phi, \mathcal{D}) = \text{CE}(\phi, \mathcal{D})$, and $\lambda = 0$ encourages entirely generative learning, as $\mathcal{L}_0(\phi, \mathcal{D}) = \text{NLL}(\phi, \mathcal{D})$. Taking $\lambda \in (0, 1)$ encourages both to varying extents. Using such a hybrid loss function, one would hope to produce a generative model that performs well for both generative and discriminative tasks.

The application of a hybrid loss function has already been applied to a different approach to learning PCs referred to as random and tensorised sum-product networks (RAT-SPNs) [34, Section 3]. When trained discriminatively on MNIST, RAT-SPNs consisted of around 1.2 million parameters and classified incomplete samples of MNIST with impressive accuracy. In line with this approach, we evaluate the generative and discriminative capabilities of CMPCs learned using such a hybrid loss function $\mathcal{L}_\lambda$, referred to further as hybrid CMPCs.

### 5.3 Benchmarking Hybrid CMPCs on Binary MNIST

In the literature, Binary MNIST typically refers to a binarisation of MNIST according to some assessment of pixel intensities across samples [40, Section 5]. It serves as an often-used baseline for evaluating approaches in generative modelling [21, 42, 5, 40, 43]. As its use is benchmarking models' generative abilities, the set of class labels corresponding to samples of Binary MNIST is not available. As such, to obtain a similarly-complex labelled dataset on which to learn hybrid CMPCs, we manually binarise MNIST ourselves. For each sample of MNIST, we set each pixel value to 0 if it is less than 128 and to 1 otherwise. Note that comparing benchmarks between models trained on the Binary MNIST used in literature and the manually binarised version proposed here is not recommended as the distinctions between the datasets are subtle but important. From here onward, Binary MNIST refers to our manual binarisation of MNIST. All Python programs used to train and test hybrid CMPCs on Binary MNIST can be found in our GitHub repository [1].

Binary MNIST consists of 70,000 samples. Each sample is a 28 by 28 image of a handwritten digit in $\{0, \ldots, 9\}$ in which each pixel value is either 0 or 1. As such, $\Omega_{\mathbf{X}} = \{0, 1\}^{784}$ and $\Omega_Y = \{0, \ldots, 9\}$. We partition the dataset into 50,000 samples for training, 10,000 for validation and 10,000 for testing. As all model variables pertaining to our hybrid CMPCs are binary, the class label of each sample during training is encoded into its final four pixels according to its binary representation. As illustration, consider the middle of Figure 21 which consists of a sample pertaining to the digit 7 (whose binarisation is $0111_2$). Its final four pixels, in the bottom right are manually lit or dimmed accordingly.

Note that, upon inspection, the final four pixels of all 70,000 samples of MNIST are zero. As such, this doctoring of samples during training does not alter the complexity of the underlying to-be-learned distribution. To classify a sample at test time, our model's prediction is whichever binarisation of the final four values yields the highest probability

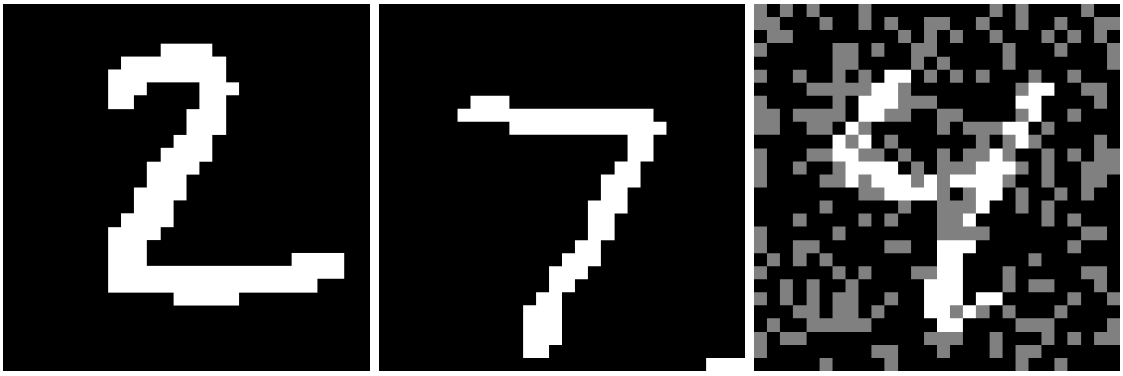

**Figure 21:** Samples of Binary MNIST. Left: A regular sample. Middle: a sample in which the final four pixels are set to $0, 1, 1$ and $1$ respectively, corresponding to the binarisation $0111_2$ of its class label which is 7. Right: A sample pertaining to the digit 4 in which 30% of pixel values are missing at random. The final four pixels are not considered when masking pixel values at random.

output by our hybrid CMPC. That is,

$$y^* = \underset{y\in\{0,\dots,9\}}{\arg\max}\, p(\mathbf{x}, y_{b_1}, y_{b_2}, y_{b_3}, y_{b_4})$$

where $y_{b_1}, \dots, y_{b_4}$ is the binarisation of $y \in \{0, \dots, 9\}$ and $\mathbf{x} \in \{0,1\}^{780}$ denotes the first 780 original pixel values of the sample. This approach is taken for its simplicity in maintaining the 28 by 28 image structure without deteriorating the complexity of the to-be-learned distribution. With this adjustment in mind, $\Omega_{\mathbf{X}} = \{0,1\}^{780}$ and $\Omega_Y = \{0,1\}^4$.

Further, we normalise the mean cross-entropy and mean negative log-likelihood terms in our hybrid loss function such that each is per-pixel. This is because the cross-entropy term corresponds to only the final four pixel values and the negative log-likelihood term corresponds to the first 780 pixel values. In line with this, the hybrid loss function used is given by

$$\begin{aligned}
\mathcal{L}_\lambda(\phi, \mathcal{D}) &= \frac{\lambda}{4}\mathrm{CE}(\phi, \mathcal{D}) + \frac{1-\lambda}{780}\mathrm{NLL}(\phi, \mathcal{D}) \\
&= -\frac{\lambda}{4}\frac{1}{M}\sum_{j=1}^{M}\log\left(\frac{\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j, y_j|\mathbf{z}_i)}{\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)}\right) - \frac{1-\lambda}{780}\frac{1}{M}\sum_{j=1}^{M}\log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right) \\
&= -\frac{1}{M}\sum_{j=1}^{M}\left[\frac{\lambda}{4}\log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j, y_j|\mathbf{z}_i)\right) + \left(\frac{1-\lambda}{780} - \frac{\lambda}{4}\right)\log\left(\frac{1}{N}\sum_{i=1}^{N} p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right)\right]
\end{aligned}$$

where $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_M, y_M)\} \subset \Omega_{\mathbf{X}} \times \Omega_Y$. Here, the logarithm in the first term was

split, which is why the additional

$$-\frac{\lambda}{4}\log\left(\frac{1}{N}\sum_{i=1}^{N}p_\phi(\mathbf{x}_j|\mathbf{z}_i)\right)$$

term appears in the second logarithm present in the sum.

### 5.3.1 Our experiments

As in the work of Correia. et al, we fix the prior $p(\mathbf{z}) = \mathcal{N}(0, I_d)$ and vary the latent dimension $d \in \{2, 4, 8, 16, 32\}$. During training, we fit FFMs to the $N_{\text{train}} = 2^{13}$ components $p_\phi(\mathbf{x}, y|\mathbf{z}_1), \ldots, p_\phi(\mathbf{x}, y|\mathbf{z}_{2^{13}})$. The decoder $\phi : \mathbb{R}^d \to [0, 1]^{784}$ itself is fit using an MLP consisting of six hidden layers of 16, 144, 272, 400, 528 and 656 nodes respectively. The number of nodes scale linearly from 16 to the number of outputs which is 784. Overall, each decoder consisted of 1.2 million tunable parameters. We employed LeakyReLU as activation functions, defined earlier, with hyperparameter $a = 0.01$. Training involved 100 max epochs and we employed early stopping with a patience of 15 epochs. A hybrid CMPC is learned using this criteria for each $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. As five latent dimensions $d \in \{2, 4, 8, 16, 32\}$ and six values of $\lambda$ are assessed, we trained $5 \cdot 6 = 30$ hybrid CMPCs in total. The benchmarks illustrated here forward are for the latent dimension 16. The results for other latent dimensions are given in Appendix D.

### 5.3.2 Classifying incomplete samples

Since PCs are tractable with respect to **marg** queries, we can apply hybrid CMPCs to classifying incomplete samples of Binary MNIST. As detailed earlier, the class label of an incomplete sample $\mathbf{x}' \in \Omega_{\mathbf{X}'}$, according to our model, is given by

$$\arg\max_{y\in\{0,\ldots,9\}} p(\mathbf{x}', y_{b_1}, y_{b_2}, y_{b_3}, y_{b_4}) = \arg\max_{y\in\{0,\ldots,9\}} \sum_{i=1}^{N_{\text{test}}} p_\phi(\mathbf{x}', y_{b_1}, y_{b_2}, y_{b_3}, y_{b_4}|\mathbf{z}_i)$$

where $(y_{b_1}, y_{b_2}, y_{b_3}, y_{b_4}) \in \{0, 1\}^4$ is the binarisation of $y \in \{0, \ldots, 9\}$. This involves answering nine **marg** queries. The portions of missing pixel values at random during our experiments are 0%, 10%, ..., 90% and 95%. Note that the first 780 pixel values can be sampled to be removed. The final four pixels are reserved for class labels. Additionally, our hybrid CMPCs consist of $N_{\text{test}} = 2^{14}$ components: twice the number of components used during training $N_{\text{train}} = 2^{13}$.

The results of this experiment are illustrated in Figure 22. Perhaps surprisingly, the best classification accuracies at different levels of missing pixel values is not given by the hybrid CMPC learned with $\lambda = 1$ (the entirely-discriminatively learned hybrid CMPC). In the right of Figure 22, we see that the highest classification accuracies are offered by $\lambda = 0.4$ and $\lambda = 0.6$ and that the lowest classification accuracies are offered by the hybrid CMPC learned

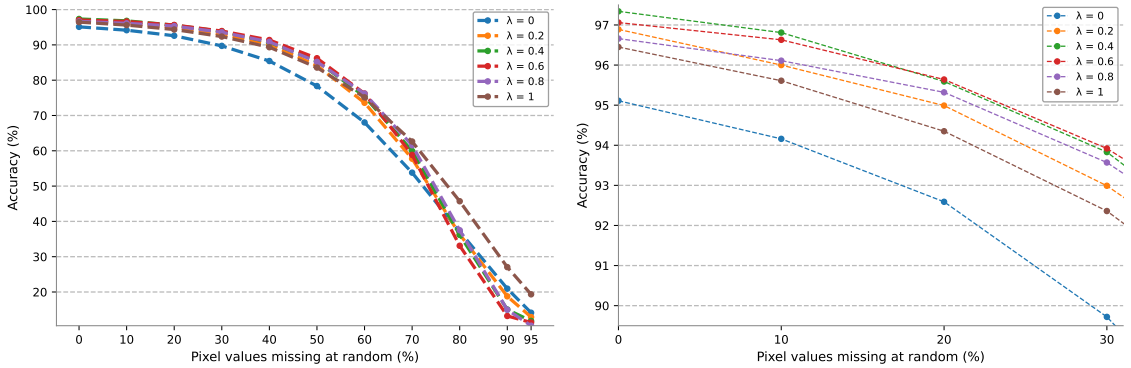


**Figure 22:** Classification accuracies of hybrid CMPCs with $d = 16$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ for incomplete samples of Binary MNIST in which differing portions of pixel values are masked at random. $N_{\text{test}} = 2^{14}$ components were used at test time. A zoomed version (for 0% to 30% missing pixels values) is given on the right for clarity.

with $\lambda = 0$. The latter is not at all surprising as it corresponds to the hybrid CMPC learned entirely generatively. We observe that the decrease in classification accuracy for $\lambda = 0$, as the portion of missing pixel values increases, is a lot more sudden than for other values of $\lambda$. This is especially observed when comparing the drop in accuracies corresponding to 20% and 30% missing pixel values. For the classification accuracies obtained by hybrid CMPCs for other latent dimensions $d \in \{2, 4, 8, 32\}$, consider Subsection D.2.

To demonstrate that the highest classification accuracies belonging to hybrid CMPCs trained with $\lambda \in \{0.4, 0.6\}$ is not due to a particular initialisation of the decoder, consider Appendix F.

### 5.3.3 Mean negative log-likelihoods and sampling

To demonstrate the extent to which hybrid CMPCs maintain generative ability while varying $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$, we illustrate the negative log-likelihoods obtained by each hybrid CMPC on our test set consisting of 10,000 samples in Table 7. We see that transitioning from entirely generative with $\lambda = 0$ to hybrid learning with $\lambda = 0.2$ invokes notable increases in the obtained mean negative log-likelihoods regardless of the number of components $N_{\text{test}}$ at test time. In particular, for each $N_{\text{test}}$, we see that the transition from $\lambda = 0.8$ to the fully discriminative $\lambda = 1$ invokes a significant increase in mean NLL. This is to be expected since hybrid CMPCs with $\lambda = 1$ are not at all encouraged to learn generatively. The mean negative log-likelihoods obtained by hybrid CMPCs with latent dimensions $d \in \{2, 4, 8, 32\}$ are given in Subsection D.1.

To complement this assessment of the mean negative log-likelihoods in assessing the

| $\lambda \backslash\ N_{\text{test}}$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|---|---|
| 0.0 | 141.58 | 129.61 | 119.39 | 111.93 | 105.76 | 101.03 | 97.33 |
| 0.2 | 147.62 | 137.97 | 130.82 | 125.18 | 120.20 | 116.53 | 113.63 |
| 0.4 | 152.20 | 145.21 | 139.08 | 134.01 | 130.56 | 127.66 | 125.40 |
| 0.6 | 159.49 | 152.20 | 146.38 | 142.04 | 138.83 | 136.24 | 134.39 |
| 0.8 | 176.34 | 172.47 | 168.94 | 166.41 | 164.35 | 162.47 | 161.16 |
| 1.0 | 239.40 | 235.96 | 233.38 | 231.50 | 229.97 | 228.79 | 227.73 |

**Table 7:** The mean negative log-likelihood obtained by hybrid CMPCs trained on Binary MNIST with $d = 16$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$.

generative ability of hybrid CMPCs for differing values of $\lambda$, we consider the quality of samples drawn from each model. For an illustration of nine samples drawn from each of the six hybrid CMPCs corresponding to $d = 16$, consider Figure 23. We see that as $\lambda$ increases, the underlying digits remain interpretable until $\lambda = 0.8$. That said, for $\lambda = 0.4$ and $\lambda = 0.6$, though the digits pertaining to samples remain mostly interpretable, heavy pixelisation occurs. For the entirely discriminative hybrid CMPC (with $\lambda = 1$, bottom right of Figure 23) we see samples in which the digit is seemingly painted in black while the background is painted in white. This may seem strange at first but it corresponds to the nature of the learning of the entirely discriminative model from which these samples were drawn. Obtaining entirely non-human interpretable samples is thus unsurprising. Samples drawn from hybrid CMPCs with latent dimensions $d \in \{2, 4, 8, 32\}$ are given in Subsection D.3.

### 5.4 Our GitHub Repository

Our repository, available at `github.com/dewi-batista/hybrid-cmpcs`, was initially forked from `github.com/AlCorreia/cm-tpm`, the repository published by Correia et al. as part of their work, and modified for the learning of decoders of hybrid CMPCs. Here, we offer a brief overview of our repository which has the following directory structure:

```
hybrid-cmpc/
├── data/
├── figures/
├── logs/
├── models/
├── utils/
├── hybrid_CMPC_test.ipynb
├── hybrid_CMPC_train.py
├── LICENSE.txt
├── README.md
└── requirements.txt
```

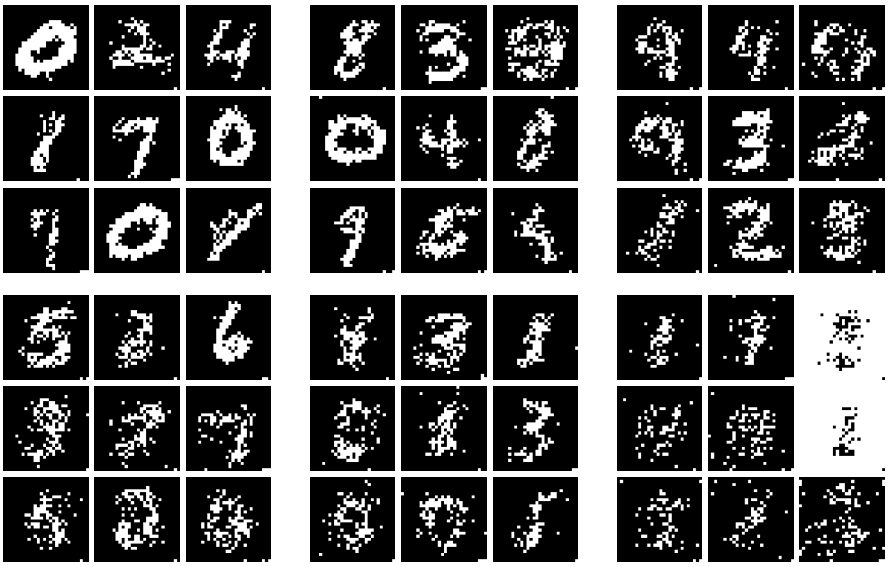

**Figure 23:** Samples of Binary MNIST drawn from hybrid CMPCs with $d = 16$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ ordered left-to-right top-to-bottom (so top right samples correspond to $\lambda = 0.4$ and bottom left to $\lambda = 0.6$).

Beginning with the files found in the root directory, `requirements.txt` consists of the Python dependencies needed to train and test hybrid CMPCs. How to install these dependencies is detailed in the root directory's `README.md` file. To train the decoder of a hybrid CMPC, the user must use `hybrid_CMPC_train.py` which is thoroughly-commented. The user can straightforwardly set the hyperparameters pertaining to the decoder that they would like to train, e.g. latent dimension $d$, the number of components used during training $N_{\text{train}}$ and $\lambda \in [0, 1]$.

Post-training, the parameters of the learned decoders can be found in `logs/`. Inside, a trained decoder is found within a series of nested directories according to its hyperparameters. In the scenario that more than one decoder has been trained with the same hyperparameters, each decoder will be listed according to its version number, indexing from 0, within its nested path. For example, if the user trains two decoders with latent dimension $d = 16$, $N_{\text{train}} = 2^{13}$ and $\lambda = 0.4$ then the directory `logs/latent_dim_16/lambda_0.40` will contain the directories `version_0/` and `version_1/`. The former corresponds to the first trained decoder and the latter corresponds to the second. Each directory will contain information pertaining to training, e.g. figures depicting convergence of training and validation errors. More importantly, each contains a `checkpoints/` directory within which is a single `*.ckpt` file. This is precisely where the parameters of the learned decoder are stored. The integer (between 1 and 100) which appears at the end of the name of said

`*.ckpt` file is the number of epochs reached before invoking the patience of 15 employed during training with early stopping. For example, if during training a decoder the validation error at epoch 66 was not improved upon within the proceeding 15 epochs then training halts and the parameters of the decoder are those pertaining to the $66^{\text{th}}$ epoch and are found in `checkpoints/best_model_valid-epoch=66.ckpt`.

As for the remaining unexplained directories, `figures/` contains figures pertaining to performance during testing, including classification accuracies, samples drawn and Monte Carlo analysis. The `utils/` directory consists of utility-like programs, e.g. the program used to seed all relevant random components during training and testing to ensure the reproducibility of our results. Finally, the `models/` directory consists of the implementation of the multi-layer perceptron architectures used in fitting our decoders (`nets.py`) and the program used to compile hybrid CMPCs from latent samples $\mathbf{z}_1, \ldots, \mathbf{z}_{N_{\text{test}}}$ and a trained decoder (`cm_hybrid.py`).

# 6 Discussion

The classification accuracies obtained by hybrid CMPCs for incomplete samples of Binary MNIST should be interpreted with a few important considerations. The first consideration to be made is that without a point of comparison in literature, it is difficult to address the raw discriminative performance of hybrid CMPCs. That said, we are still able to assess the effect that increasing $\lambda \in [0, 1]$ has on the extent to which hybrid CMPCs gain discriminative power while losing generative power. We saw earlier, in the form of mean negative log-likelihoods and sample quality, that reasonable generative power is maintained using $\lambda = 0.4$ and $\lambda = 0.6$ while classification accuracies on incomplete samples improved noticeably compared to the entirely-generative hybrid CMPC with $\lambda = 0$.

The second consideration pertains to the fact that evaluating classification accuracies on incomplete samples begs a natural question: are the missing pixel values within a sample equally informative of the sample's class label? In our case of Binary MNIST, the answer is that pixel values are certainly not equally informative of the class label and that a large portion of pixels are almost entirely uninformative of the class label. This is unsurprising as the samples pertain to neatly pre-processed images of handwritten digits. As such, pixels close to the boundary of each sample are almost exclusively 0 across the dataset and so their lack of presence when removed at random is far less detrimental to classification accuracy than pixels found in the centre. Perhaps an assessment of which pixel values are most informative, using mutual information, could be conducted. Then, an evaluation of the classification accuracy of incomplete samples in which some portion of the most informative pixel values are removed could be done. We expect that such an evaluation would require careful consideration: what is written here is only an informal description of the idea.

Regarding sample quality as $\lambda \in [0, 1]$ increases: samples drawn do not degrade im-

mediately as $\lambda$ is increased incrementally. The underlying structure of digits remains interpretable to a reasonable extent until $\lambda = 0.8$. From this point, the model's learning is too heavily discriminatory to produce convincing samples and heavy pixelation is observed. This is reflected further in the significant increase in the mean negative log-likelihoods obtained by the model from $\lambda = 0.6$ to $\lambda = 0.8$ and from $\lambda = 0.8$ to $\lambda = 1$. To add to this, it is entirely possible that a more appropriate weighting of the mean cross-entropy and mean negative log-likelihood terms would make for a smoother transition in negative log-likelihoods obtained (and samples drawn) at test time. In our work, each term was divided by the number of model variables to which they correspond: four for cross-entropy (in line with the class label represented in binary) and 780 for the digit's pixel values.

**Computational limitations**
It is worth stating that, given more time and computational resources, fitting hybrid CMPCs to datasets which are more complicated than Binary MNIST, such as MNIST, FASHION-MNIST, SVHN, etc., is a natural avenue of further assessment. It can be argued that benchmarking on Binary MNIST is not entirely representative of the power of hybrid CMPCs as its distribution is low in complexity. In line with this, our contribution in introducing hybrid CMPCs and benchmarking them on Binary MNIST can be seen as a tutorial-like example of benchmarking hybrid CMPCs in general.

# 7 Conclusion

We gave overviews of probabilistic circuits (PCs) and continuous mixtures of probabilistic circuits (CMPCs). Their application to tractable probabilistic inference was addressed. In particular, overviews of the tractability of PCs with respect to certain probabilistic queries, such as **evi** and **marg** queries, were offered. In extending CMPCs, a hybrid loss function which encourages simultaneous discriminatory and generative learning was employed in introducing hybrid CMPCs, an original contribution of our work.

Our motive in assessing hybrid CMPCs was to see to what extent the generative ability of CMPCs, developed in the original work of Correia et al. [9], could be maintained while increasing the extent to which discriminative learning is encouraged. Typically, the learning of both paradigms are at odds with one another but the tractability of PCs with respect to **evi** and **marg** queries facilitates the learning of both paradigms at once. The performance of hybrid CMPCs was assessed according to their classification accuracy on incomplete samples of Binary MNIST as well as the quality of samples drawn from them. Our results demonstrate that increasing the extent to which discriminative learning is encouraged reduces the generative ability of hybrid CMPCs notably. Perhaps surprisingly, hybrid CMPCs learned entirely discriminatively (with $\lambda = 1$) did not attain the highest classification accuracies. Instead, the hybrid CMPC learned with $\lambda = 0.4$ performed best during classification.

As for future work, for concrete comparisons to literature, hybrid CMPCs learned on MNIST, FASHION-MNIST and SVHN should be investigated. Alongside, transposed convolutional decoders should be employed. While the computational demands of transposed convolutional decoders is certainly higher, noticeable improvements in performance are expected. Further, using a transposed convolutional neural network to fit the decoder drastically reduces the decoder's parameter count. To complement this, an assessment of the performance of hybrid CMPCs in which the $N$ mixture components are fit using PCs more sophisticated than FFMs, such as CLTs, should be considered. Perhaps the most natural next step in this regard would be Bayesian networks in which nodes have at most two parents. We believe that the performance of such hybrid CMPCs for both classification and sampling would be significantly better than the hybrid CMPCs considered in our work.

# References


[1] Dewi Batista. Hybrid continuous mixtures of probabilistic circuits. `https://github.com/dewi-batista/hybrid-cmpcs`.

[2] Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks. *arXiv preprint arXiv:1707.05776*, 2017.

[3] Guillaume Bouchard and Bill Triggs. The tradeoff between generative and discriminative classifiers. In *16th IASC International Symposium on Computational Statistics*, pages 721–728, 2004.

[4] Cory J Butz, Jhonatan S Oliveira, André E dos Santos, and André L Teixeira. Deep convolutional sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3248–3255, 2019.

[5] Song Cheng, Lei Wang, Tao Xiang, and Pan Zhang. Tree tensor networks for generative modeling. *Physical Review B*, 99(15):155131, 2019.

[6] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf`, 2020.

[7] CKCN Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[8] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649. IEEE, 2012.

[9] Alvaro HC Correia, Gennaro Gala, Erik Quaeghebeur, Cassio de Campos, and Robert Peharz. Continuous mixtures of tractable probabilistic models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7244–7252, 2023.

[10] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[11] Meihua Dang, Anji Liu, and Guy Van den Broeck. Sparse probabilistic circuits via pruning and growing. *Advances in Neural Information Processing Systems*, 35:28374–28385, 2022.

[12] Meihua Dang, Antonio Vergari, and Guy Broeck. Strudel: Learning structured-decomposable probabilistic circuits. In *International Conference on Probabilistic Graphical Models*, pages 137–148. PMLR, 2020.

[13] Nicola Di Mauro, Gennaro Gala, Marco Iannotta, and Teresa MA Basile. Random probabilistic circuits. In *Uncertainty in Artificial Intelligence*, pages 1682–1691. PMLR, 2021.

[14] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[15] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[16] Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In *International Conference on Machine Learning*, pages 873–880. PMLR, 2013.

[17] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[19] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems*, 27, 2014.

[20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[21] Ajay Jain, Pieter Abbeel, and Deepak Pathak. Locally masked convolution for autoregressive models. In *Conference on Uncertainty in Artificial Intelligence*, pages 1358–1367. PMLR, 2020.

[22] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014.

[23] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA, 2009.

[24] UCLA StarAI Lab. Density estimation benchmark datasets, 2025. Available at `https://github.com/UCLA-StarAI/Density-Estimation-Datasets`.

[25] Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. *Advances in Neural Information Processing Systems*, 34:3558–3570, 2021.

[26] Anji Liu, Honghua Zhang, and Guy Van den Broeck. Scaling up probabilistic circuits by latent variable distillation. *arXiv preprint arXiv:2210.04398*, 2022.

[27] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *Advances in Neural Information Processing Systems*, 30, 2017.

[28] Stefan Lüdtke, Christian Bartelt, and Heiner Stuckenschmidt. Outlier explanation via sum-product networks. *arXiv preprint arXiv:2207.08414*, 2022.

[29] Shakir Mohamed and Balaji Lakshminarayanan. Learning in implicit generative models. *arXiv preprint arXiv:1610.03483*, 2016.

[30] Andrew Ng and Michael Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. *Advances in Neural Information Processing Systems*, 14, 2001.

[31] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 165–174, 2019.

[32] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(10):2030–2044, 2016.

[33] Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020.

[34] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Random sum-product

networks: A simple and effective approach to probabilistic deep learning. In *Uncertainty in Artificial Intelligence*, pages 334–344. PMLR, 2020.

[35] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the 2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690, Barcelona, Spain, 2011. IEEE.

[36] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538. PMLR, 2015.

[37] Amirmohammad Rooshenas and Daniel Lowd. Learning sum-product networks with direct and indirect variable interactions. In *International Conference on Machine Learning*, pages 710–718. PMLR, 2014.

[38] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[39] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[40] Ruslan Salakhutdinov and Iain Murray. On the quantitative analysis of deep belief networks. In *Proceedings of the 25th International Conference on Machine Learning*, pages 872–879, 2008.

[41] Edward H Shortliffe. Mycin: A knowledge-based computer program applied to infectious diseases. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*, page 66, 1977.

[42] Jakub Tomczak and Max Welling. VAE with a VampPrior. In *International Conference on Artificial Intelligence and Statistics*, pages 1214–1223. PMLR, 2018.

[43] Arash Vahdat and Jan Kautz. NVAE: A deep hierarchical variational autoencoder. *Advances in Neural Information Processing Systems*, 33:19667–19679, 2020.

[44] Yang Yang, Gennaro Gala, and Robert Peharz. Bayesian structure scores for probabilistic circuits. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 206, pages 563–575. PMLR, 2023.

[45] Lotfi A. Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy Sets and Systems*, 11(1–3):199–227, 1983.

[46] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.

[47] Han Zhao, Mazen Melibari, and Pascal Poupart. On the relationship between sum-product networks and Bayesian networks. In *International Conference on Machine Learning*, pages 116–124. PMLR, 2015.

# Appendices

## A Sample Spaces

Here, we define the sample space $\Omega_S$ over a subset $S \subset \{X_1, \ldots, X_n\} = \mathbf{X}$. For this, consider the lexicographic ordering function

$$\begin{aligned} \mathcal{O} : \mathcal{P}(\{X_1, \ldots, X_n\}) &\to \tau_n \\ S &\mapsto (X_{i_1}, \ldots, X_{i_k}) \end{aligned}$$

where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and

$$\tau_n = \bigcup_{k=0}^{n} \{(X_{i_1}, \ldots, X_{i_k}) | 1 \leq i_1 < \cdots < i_k \leq n\}.$$

$\mathcal{O}$ outputs a tuple in which the elements of the input subset $S \subset \mathbf{X}$ are ordered according to their subscripts, e.g. $\mathcal{O}(\{X_5, X_1, X_8\}) = (X_1, X_5, X_8)$. For our purposes, simply define

$$\Omega_S := \Omega_{X_{i_1}} \times \cdots \times \Omega_{X_{i_k}}$$

where $S \subset \mathbf{X}$ and $(X_{i_1}, \ldots, X_{i_k}) = \mathcal{O}(S)$.

## B Kullback–Leibler Divergence (KL-divergence)

Given probability density/mass functions $p$ and $q$ on the same space $\Omega$, their Kullback–Leibler divergence (KL-divergence) is given by

$$D_{\mathrm{KL}}(p||q) = \mathbb{E}_{X \sim p}\left[\log\left(\frac{p(X)}{q(X)}\right)\right].$$

It is often used as a measure of similarity between two distributions: it follows from Gibbs' inequality that it is non-zero and it is often proposed as a loss function when assessing how well a model $q$ encodes an underlying distribution $p$.

The KL-divergence of two distributions can be expressed in terms of self-entropy and cross-entropy terms as

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \mathbb{E}_{X \sim p}\left[\log\left(\frac{p(X)}{q(X)}\right)\right] \\ &= \mathbb{E}_{X \sim p}\left[-\log(q(X))\right] - \mathbb{E}_{X \sim p}\left[-\log(p(X))\right] \\ &= H(p, q) - H(p) \end{aligned}$$

where $H(p, q)$ denotes the cross-entropy between $p$ and $q$ while $H(p)$ denotes the self-entropy of $p$. It follows that minimising the KL-divergence $D_{\mathrm{KL}}(p, q)$ in $q$ corresponds to minimising the cross-entropy $H(p, q)$.

# C Multi-Layer Perceptrons (MLPs)

Multi-layer perceptrons (MLPs) are fully-connected feed-forward networks consisting of an input layer (where data is input), hidden layers and an output layer. An MLP with three hidden layers is illustrated in Figure 24. Each layer is made up of a number of neurons, each of which has a real-valued activation value derived from the activation values of the nodes in its preceding layer. The only exception to this is the input layer whose nodes' activation values are determined by the raw input data.

The suitability of MLPs for learning functions from data is due to their expressivity and their expressive-efficiency. Their expressivity is known due to a proof of the universal function approximation theorem for MLPs [10]. To add to this, their high expressive-efficiency has been demonstrated empirically [27] and ensures that the number of model components (hidden layers and neurons) needed to represent arbitrary functions is often far lower than competing methods.

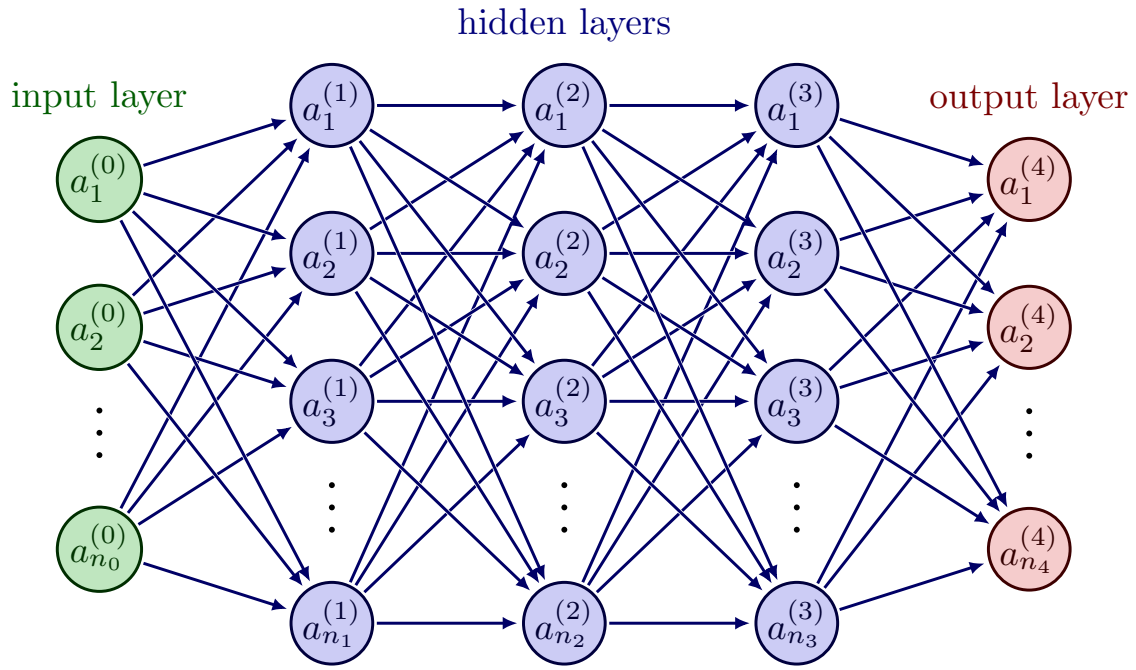


**Figure 24:** A multi-layer perceptron (MLP) with $k = 3$ hidden layers.

For ease of notation, given an MLP with $k$ hidden layers, denote the index of the input layer by 0, the output layer by $k+1$ and by extension the $j^{\text{th}}$ layer by $j \in \{0, \ldots, k+1\}$. Additionally, note the following:

- Let $n_j \in \mathbb{Z}_{\geq 1}$ denote the number of neurons in layer $j$.

- Let $a_i^{(j)} \in \mathbb{R}$ denote the activation value of neuron $i$ in layer $j$.
- Let $w_{i,l}^{(j)} \in \mathbb{R}$ denote the weight associated with the edge pointing to neuron $i$ in layer $j \in \{1, \ldots, k+1\}$ from neuron $l$ in layer $j-1$.
- Let $b_i^{(j)} \in \mathbb{R}$ denote the bias of neuron $i$ in layer $j \in \{1, \ldots, k+1\}$.
- Let $\sigma_j : \mathbb{R} \to \mathbb{R}$ denote the non-linear activation function of layer $j \in \{1, \ldots, k+1\}$.

From here we can express the activation value of any neuron in a non-input layer in terms of the activation values of the neurons in the layer which precedes it as

$$\begin{aligned} a_i^{(j+1)} &= \sigma_{j+1}\left(\sum_{l=1}^{n_j} w_{i,l}^{(j+1)} a_l^{(j)} + b_i^{(j+1)}\right) \\ &= \sigma_{j+1}\left(\begin{bmatrix} w_{i,1}^{(j+1)} \ldots w_{i,n_j}^{(j+1)} \end{bmatrix} \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{n_j}^{(j)} \end{bmatrix} + b_i^{(j+1)}\right) \end{aligned}$$

for $j \in \{0, \ldots, k\}$. The reason for writing the second equality above, involving the dot product of two vectors, is that it helps us to see how using matrix-vector notation allows us to write an elegant and compact expression for the activation values of all nodes in a non-input layer in terms of the activation values of the neurons belonging to its preceding layer as

$$\begin{bmatrix} a_1^{(j+1)} \\ \vdots \\ a_{n_{j+1}}^{(j+1)} \end{bmatrix} = \sigma_{j+1}\left(\begin{bmatrix} w_{1,1}^{(j+1)} & \ldots & w_{1,n_j}^{(j+1)} \\ \vdots & \ddots & \vdots \\ w_{n_{j+1},1}^{(j+1)} & \ldots & w_{n_{j+1},n_j}^{(j+1)} \end{bmatrix} \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{n_j}^{(j)} \end{bmatrix} + \begin{bmatrix} b_1^{(j+1)} \\ \vdots \\ b_{n_{j+1}}^{(j+1)} \end{bmatrix}\right)$$

which we abbreviate to

$$\mathbf{a}^{(j+1)} = \sigma_{j+1}\left(\mathbf{W}^{(j+1)}\mathbf{a}^{(j)} + \mathbf{b}^{(j+1)}\right)$$

where the activation function $\sigma_{j+1}$ is applied element-wise.

If we fix the structure and choice of activation functions of an MLP then all that is left to learn are its weights and biases. These are often learned from data using gradient descent to minimise some loss function in which gradients are computed via back-propagation [39]. Such a loss function can be a principled measure, e.g. corresponding to maximum likelihood, but this is not always necessary: ad-hoc loss functions are sometimes employed.

# D Additional Results

## D.1 Mean Negative Log-likelihoods for $d \in \{2, 4, 8, 32\}$

Table 8 contains the mean negative log-likelihoods of hybrid CMPCs for $N_{\text{test}} \in \{2^8, \ldots, 2^{14}\}$ and differing latent dimensions $d \in \{2, 4, 8, 32\}$. Each was trained with $N_{\text{train}} = 2^{13}$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ on Binary MNIST.

| $\lambda \backslash\ N_{\text{test}}$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|---|---|
| 0.0 | 139.86 | 127.70 | 120.40 | 111.80 | 106.84 | 104.12 | 102.47 |
| 0.2 | 144.26 | 137.31 | 133.77 | 127.37 | 124.71 | 123.37 | 122.42 |
| 0.4 | 152.82 | 146.82 | 143.40 | 138.89 | 136.87 | 135.67 | 134.76 |
| 0.6 | 162.29 | 157.23 | 154.57 | 150.98 | 149.60 | 148.40 | 147.58 |
| 0.8 | 181.48 | 178.20 | 175.42 | 173.54 | 172.51 | 171.62 | 170.87 |
| 1.0 | 282.71 | 281.04 | 279.93 | 278.99 | 278.25 | 277.55 | 276.85 |

| $\lambda \backslash\ N_{\text{test}}$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|---|---|
| 0.0 | 145.55 | 129.86 | 120.48 | 111.27 | 105.65 | 101.04 | 98.27 |
| 0.2 | 147.01 | 136.50 | 129.15 | 122.32 | 118.19 | 114.63 | 112.56 |
| 0.4 | 152.88 | 144.45 | 138.02 | 131.45 | 127.98 | 125.02 | 123.15 |
| 0.6 | 162.69 | 156.26 | 152.57 | 148.49 | 146.34 | 144.31 | 142.92 |
| 0.8 | 173.88 | 167.48 | 164.41 | 161.36 | 159.51 | 157.60 | 156.42 |
| 1.0 | 256.01 | 253.45 | 250.98 | 248.96 | 247.77 | 246.67 | 245.85 |

| $\lambda \backslash\ N_{\text{test}}$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|---|---|
| 0.0 | 143.37 | 130.19 | 120.21 | 111.69 | 105.54 | 100.67 | 97.04 |
| 0.2 | 146.66 | 137.54 | 131.08 | 123.77 | 119.22 | 115.80 | 113.10 |
| 0.4 | 152.14 | 144.13 | 138.73 | 134.43 | 131.14 | 128.59 | 126.55 |
| 0.6 | 160.37 | 154.96 | 149.70 | 145.84 | 142.93 | 140.66 | 138.87 |
| 0.8 | 175.87 | 170.93 | 167.16 | 164.75 | 162.05 | 160.17 | 158.83 |
| 1.0 | 240.90 | 238.53 | 236.05 | 234.53 | 233.13 | 232.02 | 230.98 |

| $\lambda \backslash\ N_{\text{test}}$ | $2^8$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|---|---|
| 0.0 | 141.28 | 129.49 | 119.36 | 111.96 | 105.94 | 101.26 | 97.46 |
| 0.2 | 145.67 | 137.35 | 130.65 | 124.92 | 120.61 | 117.17 | 114.43 |
| 0.4 | 152.36 | 144.58 | 139.28 | 134.56 | 131.12 | 128.30 | 126.13 |
| 0.6 | 163.24 | 155.98 | 151.24 | 147.48 | 144.52 | 142.14 | 140.34 |
| 0.8 | 179.00 | 175.04 | 170.42 | 167.73 | 165.84 | 163.82 | 162.32 |
| 1.0 | 228.26 | 226.23 | 224.18 | 222.06 | 220.48 | 219.28 | 218.16 |

**Table 8:** Latent dimensions $2, 4, 8$ and $32$ respectively top-to-bottom.

## D.2 Classification accuracies for $d \in \{2, 4, 8, 32\}$

Figure 25 contains the classification accuracies of hybrid CMPCs for $N_{\text{test}} = 2^{14}$ and differing latent dimensions $d \in \{2, 4, 8, 32\}$. Said classification accuracies are for varying portions of pixel values missing at random. Each was trained with $N_{\text{train}} = 2^{13}$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ on Binary MNIST. Zoomed copies are given to the right of each.

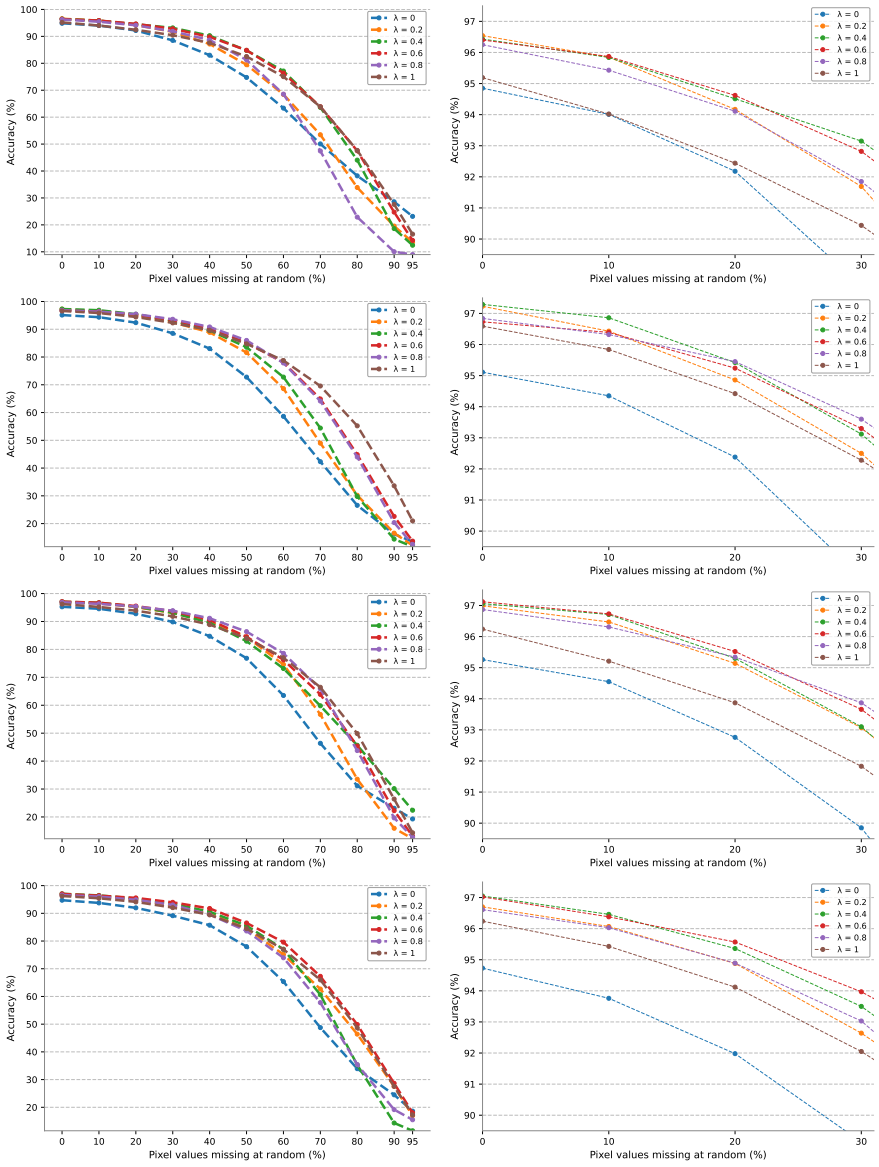


**Figure 25:** Latent dimensions $2, 4, 8$ and $32$ respectively top-to-bottom.

### D.3 Sample quality for $d \in \{2, 4, 8, 32\}$

Figure 26, Figure 27, Figure 28 and Figure 29 contain samples of Binary MNIST drawn from hybrid CMPCs for $N_{\text{test}} = 2^{14}$ and differing latent dimensions $d \in \{2, 4, 8, 32\}$. Each was trained with $N_{\text{train}} = 2^{13}$ and $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ on Binary MNIST. Ordered left-to-right top-to-bottom increasing in $\lambda$.

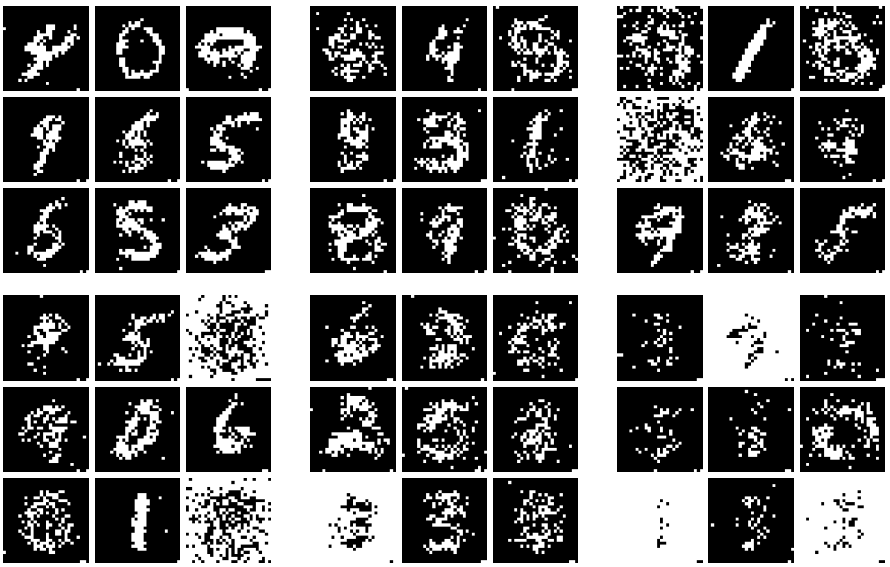

**Figure 26:** Latent dimension $d = 2$.

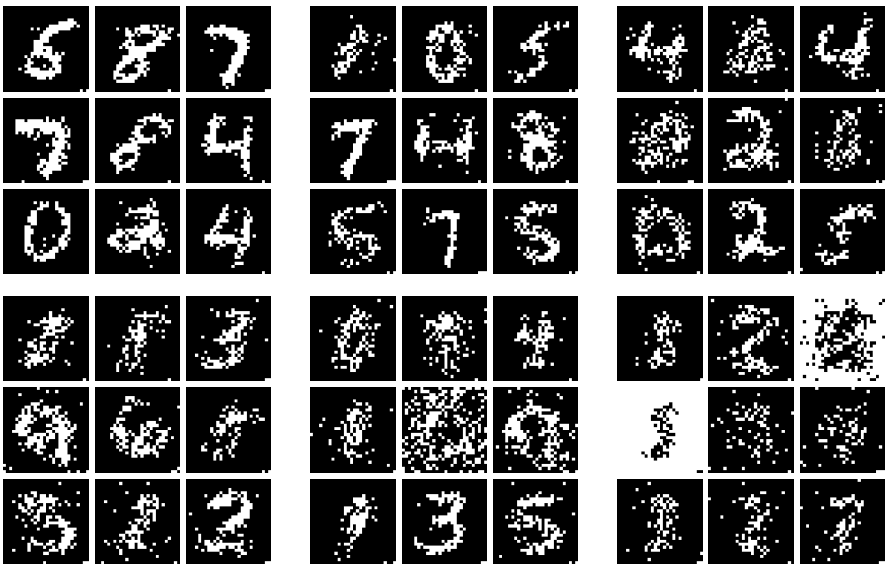

**Figure 27:** Latent dimension $d = 4$.

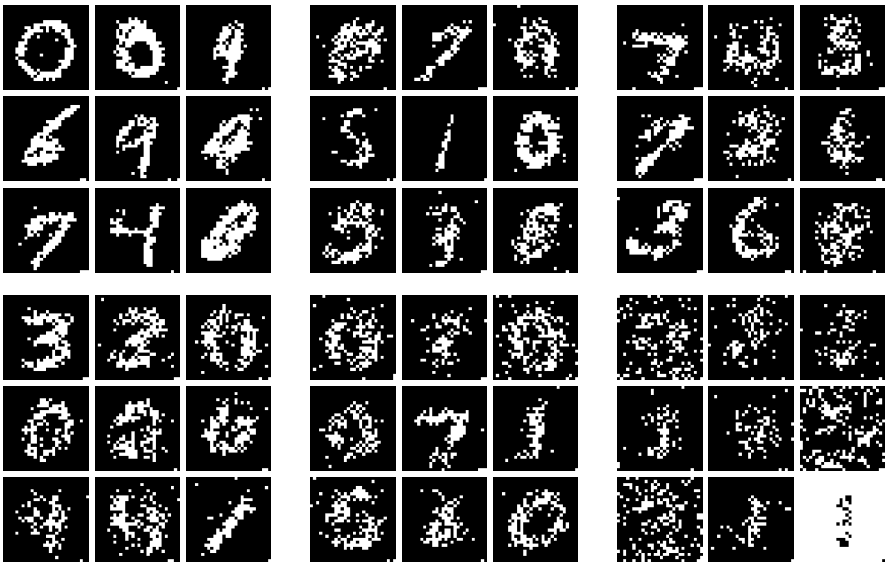

**Figure 28:** Latent dimension $d = 8$.

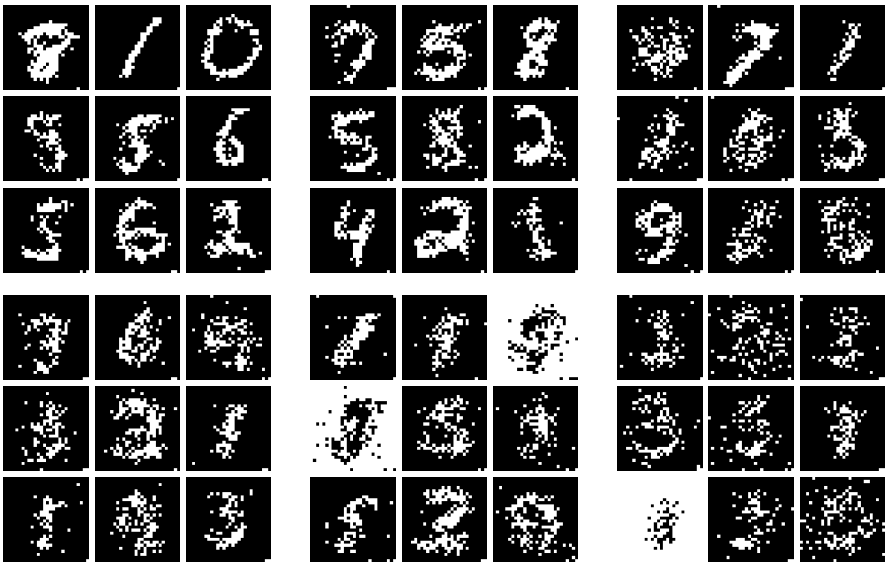

**Figure 29:** Latent dimension $d = 32$.

# E Monte Carlo Analysis

In assessing the variance in classification accuracies of our estimators with respect to sampled latents $\mathbf{z}_1, \ldots, \mathbf{z}_{N_{\text{test}}} \in \Omega_{\mathbf{Z}}$, we sampled 50 separate times and computed the mean and standard deviations of the obtained classification accuracies for said latents on the Binary MNIST test set. We illustrate the means and single standard deviations in Figure 30, Figure 31, Figure 32, Figure 33, Figure 34 and Figure 35 for $\lambda \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. Zoomed copies are given to the right of each figure.

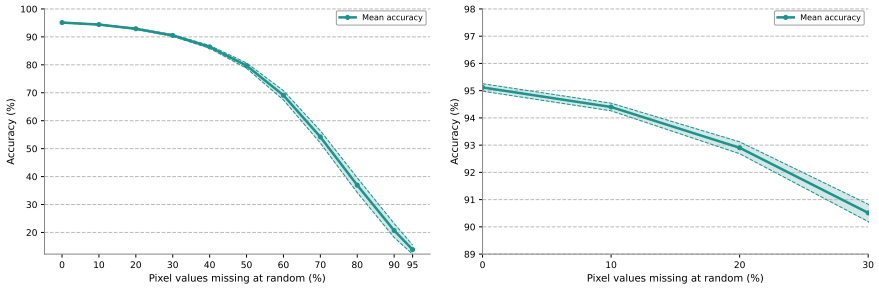


**Figure 30:** Mixing hyperparmeter $\lambda = 0$

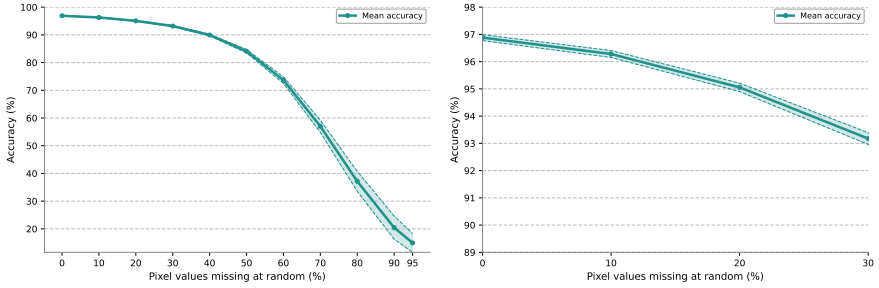


**Figure 31:** Mixing hyperparmeter $\lambda = 0.2$

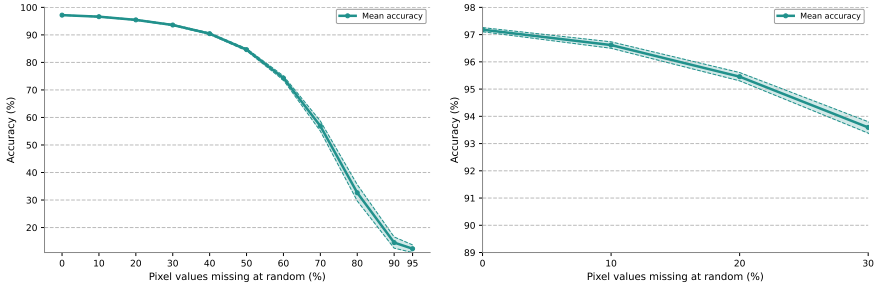


**Figure 32:** Mixing hyperparmeter $\lambda = 0.4$

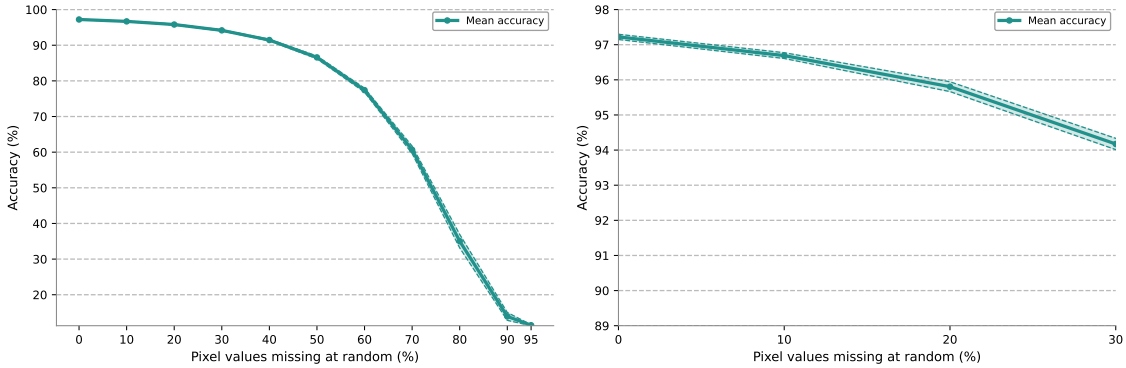


**Figure 33:** Mixing hyperparmeter $\lambda = 0.6$

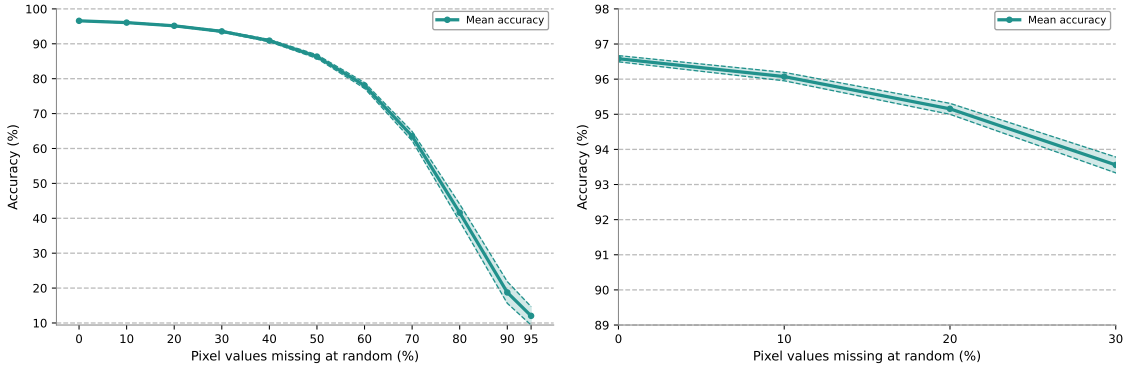


**Figure 34:** Mixing hyperparmeter $\lambda = 0.8$

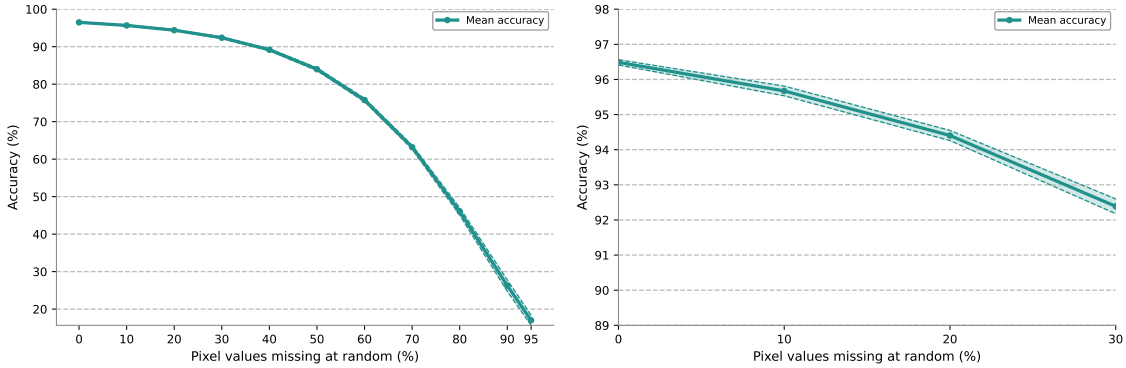


**Figure 35:** Mixing hyperparmeter $\lambda = 1$

# F Invariance of Classification Accuracies on Binary MNIST to Initial Conditions of Decoder

We seek to demonstrate that hybrid CMPCs with $\lambda = 0.4$ obtaining the highest classification accuracies on Binary MNIST in our experiments is not due to initial conditions. To do so, we learn the decoders of ten hybrid CMPCs with $\lambda = 0.4$ and ten with $\lambda = 1$ whose parameters are initialised according to Xavier initialisation [17]. The Xavier initialisation of an MLP (the decoders used in constructing our hybrid CMPs are MLPs, detailed in Appendix C) amounts to initialising all biases to 0 and initialising all weights in the $j^{\text{th}}$ layer by sampling each from

$$\mathcal{U}\left(-\sqrt{\frac{6}{n_{j-1}+n_j}}, \sqrt{\frac{6}{n_{j-1}+n_j}}\right)$$

where $n_j$ denotes the number of neurons in layer $j$. In initialising an MLP in this way, we ensure that the variance of activations (and gradients) remains roughly constant across layers, preventing undesirable behaviour during training, e.g. vanishing or exploding gradients during training.

The results are illustrated in Figure 36 in which we see that hybrid CMPCs with $\lambda = 0.4$ consistently yield higher classification accuracies on Binary MNIST than hybrid CMPCs with $\lambda = 1$.

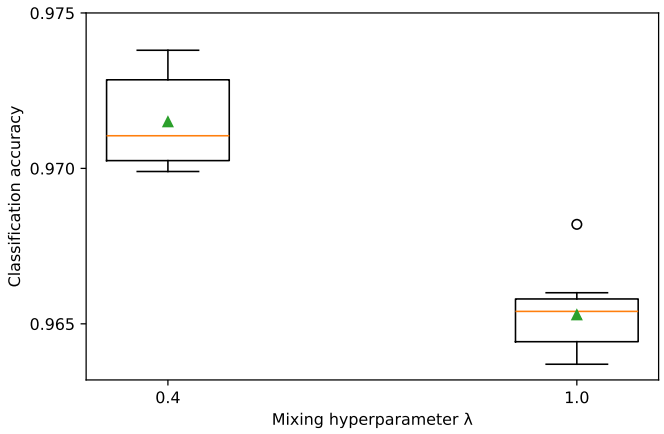


**Figure 36:** Boxplots pertaining to classification accuracies over ten independent runs for the mixing hyperparameters $\lambda = 0.4$ and $\lambda = 1.0$. Each box spans the interquartile range (IQR), representing the central 50% of the data pertaining to the runs, with the median shown by the horizontal line. Whiskers extend to the most extreme values within $1.5 \cdot \text{IQR}$ of the quartiles, and red triangles denote the mean.