CARL VON OSSIETZKY universität OLDENBURG

Scalable Unsupervised Learning for Deep Discrete Generative Models

—

Novel Variational Algorithms and their Software Realizations

Supervisor:

Prof. Dr. Jörg Lücke

Second assessor:

Prof. Dr. Ralf Häfner

Author:

Enrico Guiraud

born in Saronno, Italy

on April 30, 1991

This thesis has been accepted as fulfilling the requirements
for the degree of Doctor of Natural Sciences in
Faculty V - Mathematics and Science

October 22, 2020

Abstract

Efficient, scalable training of probabilistic generative models is a highly sought after goal in the field of machine learning. One core challenge is that maximum likelihood optimization of generative parameters is computationally intractable for all but a few mostly elementary models. Variational approximations of the Expectation-Maximization (EM) algorithm offer a generic, powerful framework to derive training algorithms as a function of the chosen form of variational distributions. Also, usage of discrete latent variables in such generative models is considered important to capture the generative process of real-world data, which, for instance, has motivated research on Variational Autoencoders (VAEs) with discrete latents.

Here we make use of truncated posteriors as variational distributions and show how the resulting variational approximation of the EM algorithm can be used to establish a close link between evolutionary algorithms (EAs) and training of probabilistic generative models with binary latent variables. We obtain training algorithms that effectively improve the tractable likelihood lower bound of truncated posteriors. After verification of the applicability and scalability of this novel EA-based training on shallow models, we demonstrate how the technique can be mixed with standard optimization of a deep generative model's parameters using auto-differentiation tools and backpropagation, in order to train discrete-latent VAEs. Our approach significantly diverts from standard VAE training and sidesteps some of its standard features such as sampling approximation, reparameterization trick and amortization. For quantitative comparison with other approaches, we used a common image denoising benchmark. In contrast to supervised neural networks, VAEs can denoise a single image without previous training on clean data or on large image datasets. While using a relatively elementary network architecture, we find our model to be competitive with the state of the art in this "zero-shot" setting. A review of the open-source software framework developed for training of discrete-latent generative models with truncated posterior approximations is also provided. Our results suggest that EA-based training of discrete-latent VAEs can represent a well-performing, flexible, scalable and arguably more direct training scheme than alternatives proposed previously, opening the door to a large number of possible future research directions.

Declaration of Authorship

I, Enrico Guiraud, declare that this thesis and the work presented in it are my own.
I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- I am aware of the guidelines of good scientific practice of the Carl von Ossietzky University Oldenburg and observed them.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- I have not availed myself of any commercial placement or consulting services in connection with my promotion procedure.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signature

Date

Acknowledgements

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 05E15CHA).

I feel extremely grateful to all the colleagues and friends that helped me out in this journey, and to those that made it memorable.
I am under no illusion that I could have come this far on my own.
In particular, there are several people without whom this thesis would quite literally not exist, and to whom I would like to offer my heartfelt thanks. First of all, to my present and past supervisors Jörg Lücke, Danilo Piparo and Axel Naumann, for their unyielding support throughout my doctoral studies. To Jakob Drefs, for the numerous scientific discussions and his uplifting, positive attitude and Imke Brumund, who saved me from the clutches of bureaucracy in more than one occasion. I am grateful to my family, for their never-ending encouragement and for providing a home I can always go back to. And to Julie, who is with me in ups and downs, for her extraordinary patience and love; thank you so much.

Contents

Chapter 1

Introduction

On the sixth of August 1991, Tim Berners-Lee, employed as a fellow at CERN, Switzerland at the time, made the very first web page publicly available, effectively ushering the world into the Information Age. Since then, thanks to the internet, the World Wide Web and giant leaps in the capabilities of off-the-shelf computing hardware (both in terms of storage capacity and operations per second) data became, by and large, a commodity: it is not just scientists and researchers that are looking for correlations, anomalies and other interesting patterns hidden in datasets far too large to be combed through manually, but also marketers, entrepreneurs, journalists and policy-makers.

Correspondingly, many recent achievements of machine learning applications can be traced back to the increased availability of computational resources and large datasets, as well as to the development of algorithms that can efficiently bring both to bear. At Uber, deep Bayesian models process millions of observables to provide time series predictions with reliable uncertainty estimations (Zhu and Laptev, 2017): this helps allocating driver incentives and detect anomalous events in real-time. Deep learning methods have been proposed for quick identification of victims of domestic violence based on their public social media posts (Subramani et al., 2018). Fraud detection systems as well as trading algorithms often integrate machine learning techniques (see e.g. Faggella). Large-scale, automated content classification is a necessary component of web services such as YouTube (Karpathy et al., 2014), but similar image classification methods can assist medical professionals in diagnosing pathologies from X-ray scans, e.g. lung and breast cancer (Coudray et al., 2018; Ribli et al., 2018) or arthritis (Xue et al., 2017). Mental health issues can be diagnosed earlier (Shatte et al., 2019) and treated more effectively (Buchlak et al., 2019) with the help of machine learning techniques. Buchlak et al. (2019), in particular, note how more general methods that can aggregate heterogeneous sources of information (such as phenomenological patient features and neuroimaging data) are measurably more effective. Finally, as we have been reminded by the current COVID-19 pandemic, responsive and

data-driven policy-making is highly desirable, especially during emergencies. Anomaly detection algorithms are often proposed for early outbreak detection and a common challenge researchers face is the computational cost and the difficulty of scaling traditional methods to high-dimensional spaces (see e.g. Wong et al., 2003; Mohsin et al., 2012; Buckeridge et al., 2005).

Although many machine learning applications in industry rely on (possibly deep) supervised learning, unsupervised learning is just as an important learning paradigm: not only as a tool to reach for when large amounts of (often application-specific) labels are not available (e.g. to be integrated in semi-supervised learning methods, as in Forster et al., 2018; Kingma et al., 2014), but also because several common tasks are inherently unsupervised, such as clustering and anomaly detection. Meanwhile, Rich Sutton describes an interesting trend in his "bitter lesson" opinion piece (Sutton, 2019): general methods that leverage computation are ultimately the most effective when compared to highly application-specific methods engineered to incorporate a-priori domain knowledge.

Latent variable probabilistic generative models are certainly one of the most attractive unsupervised learning approaches in terms of generality. These are stochastic, parametric models of the data generation process. During training, the model parameters are tuned such that the generation process has high likelihood of producing datapoints similar to the observed data. Generality of application derives from the fact that many apparently different problems can be tackled as maximum likelihood optimization of an appropriate generative model (Hinton et al., 1999, offers an extensive discussion): for example, anomaly detection is achieved by flagging datapoints for which a trained model reports very low likelihood. Clustering is achieved by fitting a two-stage generative model (e.g., a Gaussian mixture model) that first selects a category and then samples from a distribution whose parameters depend on the category selected; in a trained model, the distribution corresponding to each category will represent a cluster.

At a fundamental level, solving the generative problem forces a model to pick up all sorts of complex correlations within data that, in comparison, a discriminative model can afford to ignore: while it is sufficient to pick up certain tell-tale patterns in order to classify images (e.g., fur or skin texture to classify different animals), a generative model must incorporate vastly more information in order to reproduce such images (e.g., eyes go above mouth, legs only come in certain numbers and with certain orientations, etc.). It is therefore to be expected that the training of probabilistic generative models is typically more computationally demanding and harder to scale than the training of discriminative models. Concretely, one important obstacle is represented by the computational intractability of the model likelihood when the dimensionality of the latent space is large. As a consequence, it is typical to reformulate the problem in terms of maximization of a computable lower-bound of the exact

likelihood: this is precisely what gave rise to the rich landscape of training algorithms based on variational approximations of the likelihood.

It is clear, at this point, that scalable methods for the training of probabilistic generative models are a desirable goal if not a necessary pursuit. It is equally desirable that such methods can be applied to a large variety of generative models with little to no modification; indeed, an expert machine learner will pick a generative model appropriate for the task at hand: Noisy-OR Bayesian networks are often used as easily interpretable models for disease/symptom correlation (e.g. Halpern and Sontag, 2013; Heckerman, 1990) or link analysis (Šingliar and Hauskrecht, 2006), linear sparse coding models have been used e.g. for image feature extraction (Henniges et al., 2010) or image compression (Haft et al., 2004) but more sophisticated models have been proposed e.g. to better deal with visual occlusion, which is an inherently non-linear problem (Lücke et al., 2009). Interpretability can be traded for expressiveness by choosing deeper, non-linear generative models such as Variational Autoencoders (Rezende et al., 2014; Kingma and Welling, 2014). The generality of the problems solved by generative approaches therefore demands a corresponding generality in the algorithms that can be employed to train them.

Variational Autoencoders in particular are hugely popular, and not without good reason: this deep, non-linear generative model with continuous latent variables proved to be able to learn complex correlations efficiently. It can be trained at scale with standard deep learning techniques (namely automatic differentiation and stochastic gradient descent) and can efficiently leverage modern computing hardware such as GPUs. Standard training of Variational Autoencoders involves an amortized, Gaussian approximation of the posterior distributions and a sampling approximation of the likelihood. However, many real-world use cases suggest the use of *discrete* latent variables for a more natural description of the generation process (e.g. Jojic and Frey, 2001; Sheikh et al., 2014; Rolfe, 2016). The success of continuous-latent VAEs has consequently spurred research on novel formulations that feature discrete latents (e.g. Rolfe, 2016; Khoshaman and Amin, 2018; Roy et al., 2018; Sadeghi et al., 2019; Vahdat et al., 2019). As the application of backpropagation is made difficult by discrete distributions, Variational Autoencoders with discrete latent variables are typically trained by relaxing discrete distributions into parameterized continuous distributions that contain the discrete ones as a limit case (Maddison et al., 2016; Jang et al., 2016) or by introducing auxiliary "smoothing" distributions (Rolfe, 2016); necessarily, these techniques add extra hyperparameters or approximations on top of standard VAE training algorithms.

Here we investigate a novel, scalable training algorithm for probabilistic generative models with discrete latent variables, including discrete Variational

Autoencoders. This novel training algorithm is based on a particular choice of variational approximation, i.e. truncated distributions: while truncated posteriors were already introduced by Lücke and Eggert (2010), a fully variational treatment of the approach, including a compact, efficiently computable form of the variational lower bound, was introduced by Lücke (2019).

Building on those theoretical results, we introduce an algorithm that mixes truncated variational approximations with evolutionary approaches to efficiently train discrete-latent probabilistic generative models. We also demonstrate, for the first time, how the truncated variational approximation can be coupled with standard deep learning techniques to directly optimize discrete-latent Variational Autoencoders. The resulting method is largely independent of the specific generative model trained, lending itself to "black-box" applications, can train large models with $\mathcal{O}(1000)$ latent variables and, coupled with discrete Variational Autoencoders, it establishes new state-of-the-art performance on standard denoising benchmarks in the "zero-shot" setting, in which the model is only given a single noisy image as input.

Special care was taken to produce a software implementation of the algorithm that could be immediately useful for the application of our novel methods on the part of other researchers and research groups. Our software provides a production-ready implementation of black-box unsupervised training of binary-latent probabilistic generative models with truncated variational distributions, marrying variational inference, genetic algorithms and gradient-based training techniques.

Part of the results described in this work were published as Guiraud et al. (2018) and I presented them on behalf of the authors in the occasion of that conference. More recent results, extending our research in the direction of more complex and deep generative models, are under review at the time of writing.

1.1 Outline

Chapter 2 introduces the Expectation-Maximization algorithm, the general framework for variational inference that we will employ in the rest of this work. Chapter 3 discusses the concrete variational approximation we use, Truncated Variational Expectation-Maximization (TVEM), which is based on truncated posteriors as variational distributions. Chapter 4 shows how TVEM can be mixed with evolutionary algorithms to provide a full training algorithm for probabilistic generative models with discrete latent variables. Experiments on shallow generative models are provided to demonstrate applicability and scalability of the approach. This chapter is based on the contents of our published paper Guiraud et al. (2018), "Evolutionary Expectation Maximization". Chapter 5 marries Evolutionary Expectation Maximization for variational optimization with gradient-based optimization of the model's parameters to train

deeper models, namely discrete-latent Variational Autoencoders. It contains novel state-of-the-art results on a denoising benchmark in the "zero-shot" setting. Finally, Chapter 7 presents the work I have done as part of my residency in the ROOT software development team at CERN in the context of high-level, declarative data analysis interfaces and bridging high-energy physics data formats with the Python data analysis ecosystem.

1.2 Mathematical notation

Throughout the document we will follow the convention that lower case subscripts take values from 1 to the corresponding capital letter, so that, for example, $\sum_n$ is equivalent to $\sum_{n=1}^{N}$.
Generally, variable names that refer to multi-dimensional quantities will be indicated with an arrow, such as in $\vec{y}$, otherwise variables are implied to be scalar, except when subscripts clearly indicate dimensionality such as in W_{dh}. For example, when discussing latent variable generative models:

- $\vec{s}$ refers to the array of values of all latent variables
- s_h represents the value of one of the H latent variables
- $\vec{y}$ stands for the array of observed variables
- y_d^n is one of the D observed values of one of the N datapoints

Curly braces around arrays indicate collections of such arrays: for example, $\{\vec{y}\}$ refers to the full observed dataset, i.e. $\{\vec{y}^1, \ldots, \vec{y}^N\}$, and $\sum_{\{\vec{s}\}}$ indicates a summation over all possible 2^H latent states.

Other symbols we will often make use of:

- Φ represents the set of variational parameters
- Θ is the set of a model's parameters
- $D_{\mathrm{KL}}(p;q)$ is the Kullback–Leibler divergence between distributions p and q
- $\mathcal{N}(x;\mu,\sigma^2)$ is the isotropic normal distribution with mean μ and variance σ^2
- $\mathrm{Bern}(p)$ is the Bernoulli distribution with parameter p

Chapter 2

Variational Inference with the Expectation-Maximization Algorithm

Producing algorithms that can learn to compactly describe interesting behaviour of real-world data, capturing complex correlations and robustly dealing with data noise, is a highly sought-after goal in the field of machine learning. In particular, for real-world applications, it is desirable that such algorithms require as little as possible expert knowledge and can provide out-of-the-box insights to domain experts, or can otherwise solve a given task with as little as possible human intervention.

As a starting point towards this goal, here we tackle this general problem using the theoretical framework of latent variable generative models trained with the Expectation-Maximization (EM) algorithm. Sections 2.1 and 2.2 introduce these concepts as well as the mathematical tools we will use throughout this document. Section 2.2.2, in particular, demonstrates the application of these tools to the real-world problem of inferring hearing-impairing health conditions from reported symptoms.

Exact EM quickly becomes computationally intractable as the complexity of the problem increases. The following chapter describes the generic variational approximation scheme we employ in this work to circumvent exact EM's intractability, *Truncated Variational Expectation-Maximization* (TVEM).

2.1 Latent variable generative models

Given a set of N data points $\{\vec{y}^1, \ldots, \vec{y}^N\}$, a latent variable, probabilistic, parametric generative model describes the data generation process as a joint probability distribution $p_\Theta(\vec{y}, \vec{s})$ where $\vec{y}$ are the observed variables (or observables), $\vec{s}$ are unobserved or latent variables, and Θ are the model parameters.

In other words, we think of observed data as sampled from the data distribution $p_\Theta(\vec{y}) = \sum_{\{\vec{s}\}} p_\Theta(\vec{y}, \vec{s})$, i.e. the joint distribution marginalized over unobserved state (the symbol $\sum_{\{\vec{s}\}}$ denotes the summation over all possible latent states). By including latent variables $\vec{s}$ in our model it is possible to take into account missing or unobserved information that is nevertheless part of the generative process: for example, if we were to model disease-symptom correlations, but our dataset only reported observed symptoms, we might describe presence/absence of a disease via a *latent cause*, i.e. an unobserved variable the state of which conditions the behaviour of the observables (presence/absence of symptoms). Concrete examples of latent variable generative models can be found below. For our purposes, we will only consider generative models with binary or discrete latent variables, which are the most natural application of the novel techniques we describe in Chapter 4.

In order to be able to answer meaningful questions about the data or the phenomenon they describe, we typically seek parameters Θ that make data generated by the model as similar as possible to the true N observed data points. More formally, we want our model distribution to be as close as possible an approximation of the true data distribution. To this end, one common approach is to seek maximum likelihood (ML) parameters, i.e. given the data log-likelihood

$$L(\Theta) = \sum_n \log \left(\sum_{\{\vec{s}\}} p_\Theta(\vec{y}^n, \vec{s}) \right), \tag{2.1}$$

the training algorithm should efficiently search for $\Theta^* = \text{argmax}_\Theta \, L(\Theta)$. This is precisely where the Expectation-Maximization algorithm described in Section 2.2 comes in.

A probabilistic generative model trained this way becomes then a powerful tool that can be used to solve a wide variety of common machine learning tasks: for example, a generative model with a single categorical latent variable $s = \{c^0, c^1, \ldots\}$ could perform *clustering* by assigning each data point to a cluster based on $\max_s p(s \,|\, \vec{y}, \Theta^*)$; whenever the latent space is smaller than the observed space, probabilistic association of latent states and observables could be used for *feature extraction* or *lossy compression*; the marginal probability $p_\Theta(\vec{y}^n)$ of data points is immediately useful for *anomaly* or *outlier detection*; finally, as the parametric distribution can directly model data noise, one can perform automatic *data noise estimation* by looking at the learned value of the noise parameters, and even *denoising* by using the model's learned distribution to estimate likely noise-free values from noisy observables (in Chapter 5 we discuss this use case in depth and provide state-of-the-art results for a specific "zero-shot" problem setting).

How well a given model will perform on a given task depends, of course, on how closely it can approximate the true data distribution (more expressive

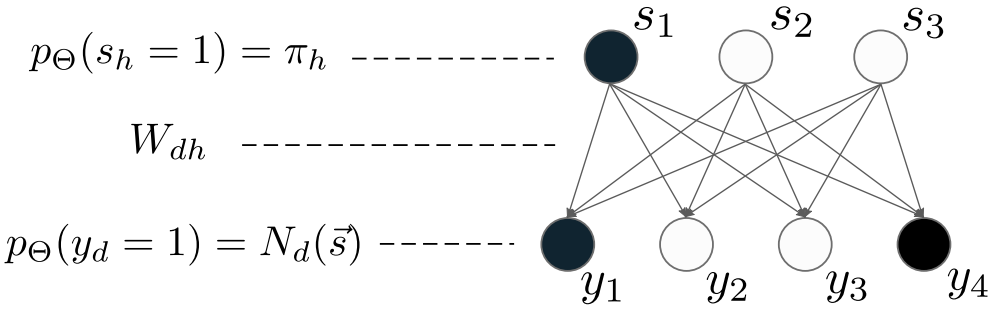

Figure 2.1: A small Noisy-OR model. The generative process first samples each s_h from a Bernoulli distribution; then each y_d is sampled from a Bernoulli distribution with parameter $N_d(\vec{s})$, generating a datapoint.

models can better approximate complex distributions), as well as how efficiently the training procedure can find model parameters that (approximately) maximize the data log-likelihood. In particular, for all but the simplest tasks a full search of the parameter space is unfeasible and the objective function is usually not convex; the performance of a model will therefore also depend on the propensity of a given training algorithm to get stuck in undesirable, shallow local optima.

2.1.1 Example: the Noisy-OR model

The Noisy-OR model (e.g. Singliar and Hauskrecht, 2006; Rotmensch et al., 2017) is a highly non-linear bipartite data model with all-to-all connectivity among hidden and observable variables. All variables take binary values. As in all generative models that we will consider in this work, the joint probability factorizes in a *prior* distribution and a *conditional data distribution*: $p_\Theta(\vec{y}, \vec{s}) = p_\Theta(\vec{s})\, p_\Theta(\vec{y} \,|\, \vec{s})$. Noisy-OR employs a Bernoulli prior, and latents activate observables based on the actual Noisy-OR rule:

$$\begin{aligned} p(\vec{s}) &= \prod_h \pi_h^{s_h} (1-\pi_h)^{1-s_h} \\ p(\vec{y} \,|\, \vec{s}) &= \prod_d N_d(\vec{s})^{y_d} (1 - N_d(\vec{s}))^{1-y_d} \\ \text{where} \quad N_d(\vec{s}) &= 1 - \prod_h (1 - W_{dh} s_h) \end{aligned} \tag{2.2}$$

In this context $\Theta = \{\vec{\pi}, W\}$, where $\vec{\pi}$ is the set of values $\pi_h \in [0, 1]$ representing the prior activation probabilities for the hidden variables s_h and W is a $D \times H$ matrix of values $W_{dh} \in [0, 1]$ representing the probability that a latent s_h, if active, activates in turn the observable y_d. The model owes its name to the fact that the probability that y_d is active corresponds to the probability that

at least one of the latents s_h successfully activates it. Figure 2.1 is a graphical representation of a small Noisy-OR network.

Noisy-OR is often employed in disease-symptom modeling: in this setting latent variables indicate presence/absence of a given disease, observables represent presence/absence of a given symptom, model priors π_h encode the incidence of a disease and model weights W_{dh} the probability that a given disease produces a given symptom. In section 2.2.2 we use precisely this problem setting and train a small Noisy-OR network on data related to hearing impairment.

2.1.2 Example: Binary Sparse Coding

As a second example of latent variable generative model we would like to introduce Binary Sparse Coding (BSC) (Haft et al., 2004; Henniges et al., 2010). Similarly to Noisy-OR, latents follow a univariate Bernoulli distribution (except that here we elect to use the same activation probability for each hidden unit, i.e. $\pi_h = \pi \; \forall \; h$). Differently from Noisy-OR, BSC employs continuous observables. The observables are drawn from a factorized multivariate Gaussian distribution whose mean is a linear superposition of the mean values W_{dh} associated to each of the latent variables:

$$\begin{aligned} p(\vec{s}) &= \prod_h \pi^{s_h} \left(1-\pi\right)^{1-s_h}, \\ p(\vec{y} \,|\, \vec{s}) &= \prod_d \mathcal{N}(y_d; \sum_h W_{dh} s_h, \sigma^2) \end{aligned} \tag{2.3}$$

The parameters of the model are $\Theta = \{\pi, \sigma, W\}$, where $\vec{\pi}$ has the same meaning as in Noisy-OR, W is a $D \times H$ matrix of real values and σ (also real-valued) determines the standard deviation of the isotropic Gaussian noise. BSC is a simple and yet versatile generative model that can be an appropriate choice whenever it is reasonable to assume that data noise is (approximately) Gaussian. As an example application, Guiraud et al. (2018) applied BSC to an image denoising task and showed that, when trained with the procedure outlined in Chapter 4, it can outperform other common sparse coding approaches in the setting examined.

2.2 Variational Expectation-Maximization

It can be tempting to try to directly optimize the model parameters Θ w.r.t. the log-likelihood of Eq. 2.1; however, analytical derivations for any but the simplest generative models will prove impractical if not unfeasible, and methods based on gradient ascent would require frequent evaluation of summation $\sum_{\{\vec{s}\}}$, which is intractable for models with more than a few latent variables. As we will discuss in this section, variational optimization algorithms based

on Expectation-Maximization (Dempster et al., 1977) help circumventing such intractability.

In particular, here we will follow Neal and Hinton's free energy reformulation of the log-likelihood maximization problem (Neal and Hinton, 1998): at the cost of introducing a set of appropriately chosen auxiliary probability distributions, this reformulation provides a tractable lower bound of the log-likelihood, the *variational free energy*, that can be optimized in its place. In this sense, variational Expectation-Maximization is a meta-algorithm, i.e. a framework to develop algorithms, because different choices of these auxiliary distributions, called *variational distributions*, yield potentially very different concrete training algorithms. The free energy is defined as follows:

$$F(\Phi, \Theta) = \sum_n \langle \log p_\Theta(\vec{y}^n, \vec{s}) \rangle_{q^n_\Phi} - \sum_n \langle \log q^n_\Phi \rangle_{q^n_\Phi}, \tag{2.4}$$

where $q^n_\Phi = q^n_\Phi(\vec{s})$ are the variational distributions, the exact functional form of which is now an extra degree of freedom. The expectation value still expands to the potentially computationally intractable summation over the full latent space:

$$\langle h(\vec{s}) \rangle_{q^n_\Phi} = \sum_{\{\vec{s}\}} q^n_\Phi(\vec{s}) h(\vec{s}) \tag{2.5}$$

It is easy to relate log-likelihood and free energy, assuming distributions q^n_Φ that are defined on and take strictly positive values for all latent states $\vec{s}$:

$$\begin{aligned} F(\Phi, \Theta) &= \sum_n \sum_{\{\vec{s}\}} q^n_\Phi(\vec{s}) \left(\log p_\Theta(\vec{y}^n, \vec{s}) - \log q^n_\Phi(\vec{s}) \right) && (2.6) \\ &= \sum_n \langle \log \frac{p_\Theta(\vec{y}^n, \vec{s})}{q^n_\Phi(\vec{s})} \rangle_{q^n_\Phi} && (2.7) \\ &= \sum_n \langle \log \frac{p(\vec{y}^n) p_\Theta(\vec{s} \,|\, \vec{y}^n)}{q^n_\Phi} \rangle_{q^n_\Phi} && (2.8) \\ &= \sum_n \langle \log p(\vec{y}^n) \rangle_{q^n_\Phi} + \langle \log \frac{p_\Theta(\vec{s} \,|\, \vec{y}^n)}{q^n_\Phi} \rangle_{q^n_\Phi} && (2.9) \\ &= L(\Theta) \; - \; \sum_n D_{\mathrm{KL}}(q^n_\Phi(\vec{s}); \, p_\Theta(\vec{s} \,|\, \vec{y}^n)) && (2.10) \end{aligned}$$

and from the non-negativity of the Kullback–Leibler (KL) divergence D_{KL} it derives that

$$F(\Phi, \Theta) \leq L(\Theta) \quad \forall \; \Phi \text{ and choice of (sufficiently well-behaved) } q_\Phi. \tag{2.11}$$

Another property of the free energy as defined in Eq. 2.4, that will be of interest in Chapter 5, is that a manipulation of its terms analogous to the one in Eq. 2.8

reveals it as an alternative writing of the *Evidence Lower BOund* (ELBO) commonly used as an optimization target in Variational Auto-Encoders (VAEs, Kingma and Welling, 2014; Rezende et al., 2014):

$$F(\Phi, \Theta) = \sum_n \langle \log \frac{p_\Theta(\vec{y}^n, \vec{s})}{q^n_\Phi(\vec{s})} \rangle_{q^n_\Phi} \tag{2.12}$$

$$= \sum_n \langle \log \frac{p_\Theta(\vec{y} \,|\, \vec{s}) p_\Theta(\vec{s})}{q^n_\Phi} \rangle_{q^n_\Phi} \tag{2.13}$$

$$= \sum_n \langle \log p_\Theta(\vec{y}^n \,|\, \vec{s}) \rangle_{q^n_\Phi} - \sum_n D_{\mathrm{KL}} \left(q^n_\Phi(\vec{s}); \, p_\Theta(\vec{s}) \right). \tag{2.14}$$

Note that, in this form, the first term contains the conditional data distribution rather than the joint distribution, and, differently from Eq. 2.10, the KL divergence contains the prior rather than the posterior distribution. In the following we will use the terms "free energy" and "ELBO" interchangeably, preferring the former in the context of variational Expectation-Maximization and the latter in the context of Variational Autoencoders.

2.2.1 Full E-steps and full M-steps

It is interesting to discuss, at this point, how this variational approximation represented by the free energy enables optimization of a given generative model. We will revisit the logical steps that we go through in this section when discussing truncated approximations in Chapter 3.

From Eq. 2.10 it can be observed that the variational free energy is maximized (i.e. becomes exactly equal to the log-likelihood) if and only if D_{KL} is zero for each datapoint, i.e. if and only if variational distributions are chosen to be equal to the model's posterior distributions, $q^n_\Phi(\vec{s}) = p_\Theta(\vec{s} \,|\, \vec{y}^n) \;\forall\; \vec{s}$. As a second key observation, please note that when optimizing the free energy of Eq. 2.4 w.r.t. model parameters Θ we can completely disregard the second term, which only depends on parameters Φ. These two observations are the main ingredients of the EM algorithm with full E-steps and full M-steps. Although it does not solve the computational intractability due to the exponentially large number of possible latent states $\{\vec{s}\}$ in summations $\sum_{\{\vec{s}\}}$, it does enable analytical derivations of optimization algorithms for a large number of latent variable generative models. For the rest of this section we will refer to the EM algorithm with full E-step and full M-step updates as "exact EM".

Exact EM alternatingly maximizes the free energy w.r.t. Θ and Φ, giving rise to an iterative optimization loop that never decreases (and, in practice, monotonously increases) $F(\Phi, \Theta)$:

Expectation step: the E-step requires that we evaluate $q^n_\Phi(\vec{s}) = p_\Theta(\vec{s} \,|\, \vec{y}^n)$ for all latent states $\vec{s}$ and all datapoints $\vec{y}^n$. Typically, for approximations of exact EM, the E-step also consists of a search for better Φ; here, however, we

make the optimal although computationally expensive choice of choosing posteriors as variational distributions and therefore setting Φ equal to Θ. After this step, the free energy is exactly equal to its upper bound, the log-likelihood, hence the E-step never decreases the free energy.
Maximization step: the M-step seeks better model parameter values Θ^{new} such that: $\Theta^{\text{new}} = \text{argmax}_{\Theta} F(\Phi, \Theta)$. Optimizing the free energy w.r.t. Θ is simpler than doing so for the log-likelihood directly: for many common generative models, including the Noisy-OR and BSC models introduced above, it is even possible to solve $\nabla_{\Theta} F(\Phi, \Theta) = 0$ w.r.t. Θ analytically to find its extrema (which inspection of the second derivatives would confirm to be maxima). After this step, however, due to the change in model parameters Θ, q^n_{Φ} have values different than the posterior distributions, so the free energy will generally have lower values than the log-likelihood. By definition, the M-step never decreases the free energy. Note that in general we are satisfied with finding new model parameters such that $F(\Phi, \Theta^{\text{new}}) \geq F(\Phi, \Theta)$ rather than the global optimum parameters: this is sufficient to eventually converge to a (local) optimum of the free energy. Equations for M-step parameter updates for the models presented in this work are available in Appendix A.

2.2.2 Application to medical data: training a Noisy-OR model on Common Audiological Functional Parameters

Exact EM is computationally expensive, but for small-enough scales it is still a viable training algorithm for standard probabilistic generative models. Medical data, production of which might require effort on the part of field experts, often is relatively small-scale, and calls for inference algorithms that can extract interesting correlations from relatively few datapoints. In the context of work by Mousavi et al. (2020) (currently in preparation), we applied Noisy-OR, trained with the exact EM algorithm, to a dataset containing information on the state of the audiological apparatus of anonymized patients. The Common Audiological Functional Parameters (CAFPAs) were introduced by Buhl et al. (2020) as an abstract representation of audiological tests performed on a patient. The dataset we use in these experiments is the output of an expert survey in which leading clinical audiologists and physicians labeled the database from Hörzentrum Oldenburg, Germany, by indicating audiological findings, treatment recommendations, and CAFPAs for single patient cases. CAFPAs take continuous values in the interval $[0, 1]$, where 0 represents a normal ("healthy") value and 1 a pathological value, and describe different functional aspects of the auditory system: four CAFPA values are related to hearing thresholds in different frequency ranges, two CAFPA values are related to suprathreshold deficits below and above 1.5 kHz, and finally binaural hearing, neural processing, cognitive abilities, and a socioeconomic component are associated to one CAFPA value each (see Buhl et al. (2020) for detailed

information).

In these experiments we ignore the socioeconomic CAFPA. Furthermore, we only take into account datapoints with audiological findings on high-frequency hearing loss and broadband hearing loss. Our aim will be to train a Noisy-OR model that can "diagnose" these conditions (i.e., infer the correct audiological findings, which will play the role of hidden state) based on the observed CAFPA values. As a performance metric, as it is common for medical applications, we will use the area under curve (AUC) for the receiver operating characteristic (ROC) curve (true positive over false positive rate). We will produce three such curves, one for high-frequency hearing loss, one for broadband hearing loss, and one for normal hearing. The dataset contains 124 data points with D=9 CAFPA values each, in which 52 patients are diagnosed with high-frequency hearing loss, 26 with broadband hearing loss, 9 with both and 37 have normal hearing.

As Noisy-OR requires binary observables, we binarize CAFPA values based on a threshold and repeat the experiment for several binarization thresholds, selecting the one that yields best AUCs (0.25 for these experiments). We employ 3-fold cross-validation. Fig. 2.2 shows ROC curves for each fold (dashed lines) as well as the average across folds (solid lines) and the AUC value of the solid lines.

2.2.3 Factored variational distributions

Factored variational approximations, or *mean-field* approximations, are a common technique to obtain computationally tractable EM algorithms. In this setting, variational distributions are chosen so they have a factorized form:

$$q^n_\Phi(\vec{s}) = \prod_h \widetilde{q}^n_h(s_h; \phi^n_h) \tag{2.15}$$

The exact functional form of the factors $\widetilde{q}^n_h$ usually depends on the generative model, and it is chosen so that optimization is computationally tractable. A common choice is to pick $\widetilde{q}^n_h$ identical to the model's prior distributions (e.g. Jordan et al., 1999; Haft et al., 2004). Being a factored approach, mean-field has the disadvantage of being unable to capture correlations between hidden variables, potentially resulting in poor approximations of the exact log-likelihood. Another way to see this is that, as we know that the theoretical optimal choice would be to pick variational distributions equal to the model's posteriors, choosing q^n_Φ that can only poorly approximate the possibly complex modes of the posteriors can be sub-optimal (see e.g. Vértes and Sahani, 2018, for further discussion). Another downside of factored variational distributions is that M-step update rules for the variational parameters Φ require per-model mathematical derivations.

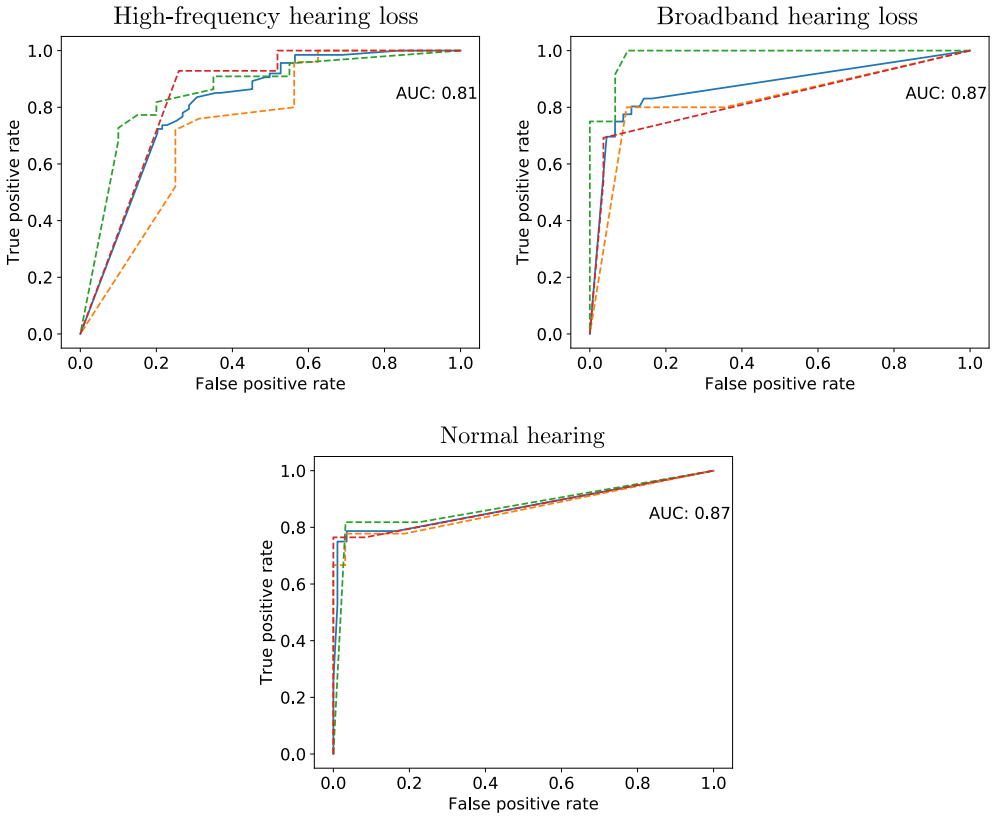

Figure 2.2: Noisy-OR trained with exact EM on a real-world CAFPA dataset. For the hearing conditions indicated in their titles, the sub-figures display ROC curves for each fold in the 3-fold cross-validation (dashed lines) as well as the average ROC curve across all folds (solid line) and its AUC.

2.2.4 Gaussian variational distributions

As a versatile distribution with useful analytical properties, another common choice of variational distributions is the Gaussian distribution:

$$q_{\Phi}^{n}(\vec{s}) = \mathcal{N}(\vec{s}, \mu^{n}, \Sigma^{n}) \tag{2.16}$$

where $\Phi = \{\mu^{1}, \Sigma^{1}, ..., \mu^{N}, \Sigma^{N}\}$. By definition, Gaussian variational distributions only capture one posterior mode. As such, they are a common choice for generative models known to have essentially mono-modal posteriors (see e.g. Opper and Winther, 2005; Seeger, 2008; Opper and Archambeau, 2009).

Chapter 3

Truncated Variational Expectation-Maximization

This chapter provides a review of truncated posteriors as a choice of variational distributions in the context of the Expectation-Maximization framework discussed in Chapter 2. Sec. 3.1 highlights several interesting properties of such a variational approximation; Sec. 3.2 presents the resulting training algorithm, which, notably, still features an important degree of freedom in the optimization of variational parameters; finally, Sec. 3.3 discusses several aspects related to implementing training algorithms based on truncated posteriors efficiently.

Truncated posteriors have previously been used in the training of probabilistic generative models (e.g. Lücke and Sahani, 2008; Sheikh et al., 2014) and fully variational such approaches are discussed in detail by Lücke (2019).

3.1 Truncated Posteriors as Variational Distributions

Rather than Gaussian or factored distributions, for the purpose of this work we will use truncated posteriors (e.g. Lücke and Sahani, 2008; Guiraud et al., 2018) as variational distributions of choice:

$$q^n_\Phi(\vec{s};\vec{y}^{\,n}) := \frac{p_\Theta(\vec{s}\,|\,\vec{y}^{\,n})}{\sum\limits_{\vec{s}'\in\Phi^n} p_\Theta(\vec{s}'\,|\,\vec{y}^{\,n})}\,\delta(\vec{s}\in\Phi^n) \tag{3.1}$$

$$= \frac{p_\Theta(\vec{y}^{\,n},\vec{s})}{\sum\limits_{\vec{s}'\in\Phi^n} p_\Theta(\vec{y}^{\,n},\vec{s}')}\,\delta(\vec{s}\in\Phi^n)\,. \tag{3.2}$$

where

$$\delta(\vec{s}\in\Phi^n) = \begin{cases} 1 & \text{if } \vec{s}\in\Phi^n \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

The truncated posterior $q^n_\Phi(\vec{s};\vec{y})$ is proportional to the true posterior $p_\Theta(\vec{s}\,|\,\vec{y}^{\,n})$ in a subset Φ^n of the latent state, and zero outside of the subset. The variational parameters Φ^n take then the form of sets of latent states, Φ being the collection of all Φ^n. Compared to factored distributions, truncated posteriors also yield a non-amortized training procedure (variational parameters Φ^n are per-datapoint), but are non-factoring (yet numerically tractable) and yield a generic E-step. Truncated posteriors are well-suited to generative models with discrete latent variables (e.g. Sheikh et al., 2014; Hughes and Sudderth, 2016; Lücke et al., 2018; Shelton et al., 2014; Forster and Lücke, 2018).

An interesting property of this choice of variational distributions is that it contains both exact Expectation-Maximization and its maximum a posteriori (MAP) approximation as limit cases: the former corresponds to the case in which Φ^n contains all possible variational states, so that summations $\sum_{\vec{s}\in\Phi^n}$ are equivalent to summations over the full latent space, and the latter corresponds to the case in which Φ^n only contains a single element – the (estimated) MAP state.

An immediate consequence of using truncated posteriors is that expectation values $\langle h(\vec{s})\rangle_{q^n_\Phi}$ can efficiently be computed given sufficiently small Φ^n:

$$\langle h(\vec{s})\rangle_{q^n_\Phi} = \sum_{\{\vec{s}\}} h(\vec{s})\, q^n_\Phi(\vec{s}) = \frac{\sum\limits_{\vec{s}\in\Phi^n} h(\vec{s})\, p_\Theta(\vec{y}^{\,n},\vec{s})}{\sum\limits_{\vec{s}'\in\Phi^n} p_\Theta(\vec{s}',\vec{y}^{\,n})}\,. \tag{3.4}$$

Eqn. 3.4 is required to evaluate many quantities of interest, including the free energy itself and, often, M-step parameter updates.

Note, however, that it is not a given that truncated posteriors can be plugged into the framework of Expectation Maximization without issues. In particular, we would like to highlight two technical hurdles: firstly, truncated posteriors, by definition, have hard zeros outside of the subset of latent space represented by each Φ^n. In first approximation, this would seem to invalidate the derivation of the variational lower bound in Eq. 2.11, in which q^n_Φ appears at denominator or as argument of a logarithm; however, it is possible to show that truncated posteriors are indeed a viable choice of variational distributions e.g. by defining them so they take a small positive value for $\vec{s}\notin\Phi^n$ and then taking the limit in which that positive value goes to zero. Secondly, given that our variational distributions depend on the model parameters Θ (since they are defined in terms of the model's posterior), it is not obvious that one can neglect the entropy term in the free energy of Eq 2.4 when taking derivatives w.r.t. the model parameters, which would greatly complicate derivations of M-step equations; however, it is possible to show that the entropy term can in fact be neglected, as one can consider, for the purpose of the optimization algorithm, the Θ on which q^n_Φ depends as a different set of parameters than those in p_Θ; the first set of parameters is then updated to be equal to the model's parameters at every iteration of the EM algorithm. Alternatively, one

could define the truncated posteriors as having the same functional form but different parameters $\hat{\Theta}$ than the model's posteriors, and then show that indeed setting $\hat{\Theta} = \Theta$ is the optimal choice. We refer the reader to Lücke (2019) for the relevant mathematical derivations and further discussion.

3.2 Training Generative Models using Truncated Posteriors

Centrally for this work, truncated posteriors enable a specific reformulation of the variational lower bound that is useful to efficiently optimize the variational parameters Φ (see Lücke, 2019, for a derivation):

$$F(\Phi, \Theta) = \sum_n \log \sum_{\vec{s} \in \Phi^n} p_\Theta(\vec{y}^n, \vec{s}) \,. \tag{3.5}$$

Note the similarity of Eq. 3.5 with the log-likelihood of Eq. 2.1. Thanks to this specific form of the free energy, a simple optimization algorithm for the variational parameters Φ becomes available: if we update the set Φ^n for a given $\vec{y}^n$ by replacing a state $\vec{s}^{\,\text{old}} \in \Phi^n$ with a state $\vec{s}^{\,\text{new}} \notin \Phi^n$, then $F(\Phi, \Theta)$ increases if and only if:

$$p_\Theta(\vec{y}^n, \vec{s}^{\,\text{new}}) > p_\Theta(\vec{y}^n, \vec{s}^{\,\text{old}}) \,. \tag{3.6}$$

or equivalently, via the monotonicity of the logarithm function,

$$\log p_\Theta(\vec{y}^n, \vec{s}^{\,\text{new}}) > \log p_\Theta(\vec{y}^n, \vec{s}^{\,\text{old}}) \,. \tag{3.7}$$

Consequently, thanks to Eq. 3.5, pairwise comparisons of joint probabilities are sufficient to optimize the free energy. This criterion is efficiently computable whenever the model's joint distribution is. The concrete form that the EM algorithm takes when truncated posteriors are chosen as variational distributions is denominated *Truncated Variational Expectation Maximization* (TVEM) and it is shown in Alg. 1.

This algorithm is guaranteed to converge to a (local) optimum of the free energy if M-step equations never decrease the free energy (which is commonly the case). Smart initialization of model parameters Θ can improve time to convergence and help avoid bad local optima. Also note that the same M-steps can be applied as for any EM algorithm. In our experiments, we favor sparse initialization of the variational parameters Φ, e.g. by sampling their initial contents from a Bernoulli distribution with a small parameter.

The crucial problem that is left to solve is the initialization of the proposed variational states Φ^{new} in TVEM's E-step: especially when the latent space is large, it is important that this search for proposal states is able to exploit available information to retrieve as many different high-posterior states as

Algorithm 1: Truncated Variational Expectation Maximization

Initialize model parameters Θ
Initialize each Φ^n with S distinct latent states
repeat
  for *each datapoint* $\vec{y}^n$ **do**
    Initialize Φ^{new} to S' states $\vec{s}'$ s.t. $\vec{s}' \notin \Phi^n$
    $\Phi^n = \Phi^n \cup \Phi^{\text{new}}$
    Truncate Φ^n to S elements with highest $\log p_\Theta(\vec{y}^n, \vec{s})$ values
  Update Θ using M-step equations and q^n_Φ as defined in Eq. 3.2
until *parameters* Θ *have sufficiently converged*

possible. A specific search procedure based on evolutionary algorithms is discussed in the next chapter, and it is a major original contribution of this work.

3.3 Efficient numerical implementation of TVEM

It is important, at this point, to ask whether an efficient numerical implementation of TVEM is possible. To answer the question, on one hand we have to study its theoretical scaling properties, on the other we have to ask whether implementations can easily take advantage of many-core architectures and what possible drawbacks we expect TVEM to have when compared to available alternatives.

3.3.1 Computational complexity and memory requirements

While TVEM M-steps employ the typical parameter update equation provided by the Expectation-Maximization framework, the E-step algorithm (i.e., the inner loop in Alg. 1) is specific to TVEM. Therefore, it is interesting to verify that TVEM's E-step does not involve undesirable scaling behavior with respect to the problem dimensionality.

The most computationally intensive operations involved in the E-step are the computation of joint probabilities and the selection of the best variational states to include in each Φ^n set. Therefore, we are interested in the computational complexity of TVEM's E-step with respect to the number of datapoints N and the size of sets Φ^n; the dependency on the number of hidden and observable variables only comes into play via joint probability computations and therefore it is really a function of the generative model and not a property of TVEM.

TVEM's E-step requires exactly $N \times S \times S'$ evaluations of the model's joint probability per training epoch, where N is the number of datapoints,

S is the size of Φ^n and S' is the size of Φ^{new} in Alg. 1; the number of joint probability evaluations therefore scales linearly with respect to N, S and also S'. Selection of the best S states in $\Phi^n \cup \Phi^{new}$ does not require a full sort of the union of the sets, but can be implemented in terms of efficient "k-th element" partitioning algorithms which result in $\mathcal{O}(S)$ time complexity (see e.g. Blum et al., 1973). The overall complexity of TVEM's E-step therefore scales linearly with respect to problem dimensionality.

In terms of memory requirements, however, TVEM does impose a non-negligible extra load: the Φ^n sets must be kept in memory across epochs. They occupy exactly $N \times S \times H$ bytes, assuming boolean values occupy one byte each, as it is typically the case. For large datasets and generative models, the memory required by the algorithm might therefore exceed the capacity of off-the-shelf single GPUs. Bit-packing (storing each boolean value as a single bit) would reduce the memory load eight-fold, at the cost of runtime performance. This kind of aggressive memory optimization is rarely necessary, however, because TVEM's data parallelism property (discussed above) implies that the N sets Φ^n could be distributed across computing nodes or, if required, even swapped to disk in chunks, with no changes required to the core algorithm logic.

3.3.2 Data parallelism

TVEM's E-step requires independent updates of each set Φ^n. This update, in turn, requires the evaluation of the model's joint (or log-joint) distributions $\log p_\Theta(\vec{y}^n, \vec{s})$ for each datapoint $\vec{y}^n$ and each state $\vec{s}$ already in Φ^n, as well as the new proposal states in Φ^{new}. Indeed, the evaluation of the joint distributions is typically the most computationally demanding part of TVEM's E-step. However, note that the update of each Φ^n can be performed independently and the evaluation of each $\log p_\Theta(\vec{y}^n, \vec{s})$ is also trivially data-parallel. The computational workload of TVEM's E-step can therefore be easily parallelized over datapoints across computing resources.

For what regards M-steps, it is very commonly the case that factors that are expensive to compute in M-step equations consist of sums of terms that depend on a single datapoint $\sum_n A_n$ (compare e.g. App. A where we report M-step equations for the Noisy-OR model and Binary Sparse Coding). Therefore, all of TVEM can be implemented in terms of a MapReduce-like strategy, where an arbitrary number of workers perform E-steps and evaluate M-step factors A_n for part of the dataset, and then the global parameter updates are evaluated by aggregating the terms computed by each worker. Data parallelism is a highly desirable property for any machine learning algorithm, and guarantees that proper TVEM software implementations can take full advantage of modern many-core architectures as well as state-of-the-art multi-GPU computing clusters.

3.3.3 Log-pseudo-joints, a useful numerical optimization

As mentioned above, evaluation of joint probabilities is often computationally expensive, and in practice, for the workloads that we will examine in subsequent chapters, it occupies a large fraction of the runtime (between 30% and 40% of a training epoch even after the optimizations described in this section are applied). Consequently, it can be interesting to make such computations more efficient. At the same time, especially when the problem dimensionality is high, it is likely that numerical values of joint probabilities become small enough to cause catastrophic arithmetic errors due to the finite precision of floating point representations. Log-pseudo-joints are a mathematical trick that largely mitigates both of these issues (see e.g. Bornschein et al., 2013).

First of all, note that in the pairwise comparison criterion of Eq. 3.7, which is at the core of the TVEM algorithm, joint or log-joint probabilities can be substituted by any function thereof that does not alter the result of these pairwise comparisons. Secondly, note that, when training a generative model with TVEM, joint probabilities are only used either in the pairwise criterion mentioned above, or as weights of expectation values in M-step equations (see e.g. App. A). These expectation values are evaluated as per Eq. 3.4, which first of all we re-express in terms of log-joint probabilities:

$$\langle h(\vec{s})\rangle_{q^n_\Phi} = \frac{\sum\limits_{\vec{s}\in\Phi^n} h(\vec{s})\, p_\Theta(\vec{y}^n,\vec{s})}{\sum\limits_{\vec{s}'\in\Phi^n} p_\Theta(\vec{s}',\vec{y}^n)} = \frac{\sum\limits_{\vec{s}\in\Phi^n} h(\vec{s}) \exp(\log p_\Theta\,(\vec{s},\vec{y}^n))}{\sum\limits_{\vec{s}'\in\Phi^n} \exp(\log p_\Theta\,(\vec{s}',\vec{y}^n))}\,.$$

At this point we define the log-pseudo-joints $\widetilde{\log p_\Theta}$ as the set of terms in the log-joint that depend on the latent variables $\vec{s}$, and aggregate other terms in a factor C^n that is constant with respect to $\vec{s}$:

$$\log p_\Theta(\vec{y}^n,\vec{s}) = \widetilde{\log p_\Theta}(\vec{y}^n,\vec{s}) + C^n(\vec{y}^n;\Theta)\,, \tag{3.8}$$

Finally, we note that expectations can be written as a function of the log-pseudo-joints only, as the factors C^n can be elided:

$$\begin{aligned}\langle h(\vec{s})\rangle_{q^n_\Phi} &= \frac{\sum\limits_{\vec{s}\in\Phi^n} h(\vec{s}) \exp(\widetilde{\log p_\Theta}(\vec{s},\vec{y}^n) + C^n(\vec{y}^n;\Theta))}{\sum\limits_{\vec{s}'\in\Phi^n} \exp(\widetilde{\log p_\Theta}(\vec{s}',\vec{y}^n) + C^n(\vec{y}^n;\Theta))} \\ &= \frac{\sum\limits_{\vec{s}\in\Phi^n} h(\vec{s}) \exp(\widetilde{\log p_\Theta}(\vec{s},\vec{y}^n))}{\sum\limits_{\vec{s}'\in\Phi^n} \exp(\widetilde{\log p_\Theta}(\vec{s}',\vec{y}^n))}\,.\end{aligned} \tag{3.9}$$

Not only expectation values $\langle h\rangle_{q^n_\Phi}$ can efficiently be computed in terms of the log-pseudo-joint functions as defined by Eq. 3.8, but also criterion Eq. 3.7:

$$\log p_\Theta(\vec{y}^n,\vec{s}^{\,\text{new}}) > \log p_\Theta(\vec{y}^n,\vec{s}^{\,\text{old}}) \quad\Leftrightarrow\quad \widetilde{\log p_\Theta}(\vec{y}^n,\vec{s}^{\,\text{new}}) > \widetilde{\log p_\Theta}(\vec{y}^n,\vec{s}) \tag{3.10}$$

We can therefore express all computations required for TVEM training of generative models in terms of log-pseudo-joints. Only when we wish to recover the full log-joint values (for instance if computation of the truncated free energy Eq. 3.5 is desired), the terms $C^n(\vec{y}^n;\Theta)$ need to be evaluated. Log-pseudo-joints are less computationally expensive than full joint probabilities and mitigate finite precision floating point arithmetic errors by moving computations to log-space. In order to avoid computation of exponentials with very large (positive or negative) arguments, we multiply both numerator and denominator of Eq. 3.9 by a term $\exp(B^n)$ with $B^n := -\min_{\vec{s}\in\Phi^n} \widetilde{\log p_\Theta}$, which guarantees that the largest exponent in each summation evaluates to exactly zero. Note that log-pseudo-joints are not probability distributions: they do not integrate to one. Explicit forms for the log-pseudo-joints of Noisy-OR and Binary Sparse Coding can be found in App. B.

3.3.4 No fast inference mode: a notable caveat

When applying a (supervised or unsupervised) machine learning algorithm to a given task, the expectation is often that a possibly long and computationally intensive training phase is followed by a relatively faster inference phase, in which the trained model is applied to the task at hand. Indeed this is the case for many commonly applied machine learning algorithms, including Boosted Decision Trees (Freund and Schapire, 1995), Deep Neural Networks used as supervised classifiers, Generative Adversarial Networks (Goodfellow et al., 2014) and Autoencoders (Kramer, 1991). Non-amortized learning algorithms such as TVEM, on the other hand, have to perform at least some form of parameter fitting at inference time as well as training time. In the specific case of TVEM, even though the learned model parameters Θ can be used as-is, this is not the case for the variational parameters Φ^n: valid sets of variational states $\vec{s}$ must be searched for each datapoint in order to properly perform inference tasks. Since this search amounts to a large fraction of the training time, this limitation is worth pointing out. Other standard, amortized EM approximation schemes such as the factored approximations discussed in Sec. 2.2.3 do not suffer from this limitation.

On the other hand, it has been pointed out (see e.g. Kim et al., 2018; Cremer et al., 2018) that amortization can result in worse inference performance. The TVEM algorithm is not subject to such an amortization gap.

Chapter 4

Evolutionary Expectation-Maximization

In Chapter 2 we have seen how the Expectation-Maximization framework can be applied to derive variational approximations of the maximum likelihood problem. In Chapter 3 we showed how, thanks to our choice of truncated posteriors Eq. 3.2 as variational distributions, it is possible to overcome the computational intractability of exact Expectation-Maximization without recurring to factored distributions that might not be able to capture complex modes of the model's posteriors.

In practice, the optimization of generative models with discrete latent variables has been re-framed as the optimization of sets of latent states Φ^n. The criterion of Eq. 3.7, that we report here as a reminder, guides the process: we seek latent states $\vec{s}^{\,\text{new}} \notin \Phi^n$ such that

$$\log p_\Theta(\vec{y}^{\,n}, \vec{s}^{\,\text{new}}) > \log p_\Theta(\vec{y}^{\,n}, \vec{s}^{\,\text{old}}) \qquad \text{(3.7 revisited)}$$

for any one $\vec{s}^{\,\text{old}} \in \Phi^n$. Assuming that computation of the model's log-joint distribution $\log p_\Theta(\vec{y}, \vec{s})$ is tractable, we are left with the challenge of efficiently searching the latent space for good proposal states $\vec{s}^{\,\text{new}}$. Solving this core problem will be the topic of this chapter: the behavior of the training algorithm itself is strongly dependent on the search procedure of choice.

The content of this chapter is based on Guiraud et al. (2018). The original idea of applying evolutionary algorithms to the search of proposal states $\vec{s}^{\,\text{new}}$ was developed jointly with Jörg Lücke and Jakob Drefs. Jakob Drefs and I settled the details of the algorithm, implemented, tested and fine-tuned it and performed the experiments necessary to its investigation and validation. In particular, I performed all experiments on the Noisy-OR generative model and developed and implemented the GPU-friendly genetic operators employed by the algorithm. The denoising results with the Binary Sparse Coding model presented in this chapter were produced by Jakob Drefs.

4.1 Related work: Truncated Variational Sampling

Previous work (e.g. Guiraud et al., 2016; Lücke et al., 2018) has tackled the challenge of finding good proposal states by sampling new variational states $\vec{s}^{\text{new}}$ from appropriate proposal distributions. Because the variational lower bound of Eq. 3.5 becomes a tighter and tighter approximation of the true log-likelihood as the truncated posteriors of Eq. 3.2 capture more and more posterior mass, the obvious choice would be to sample from the model's posterior distribution, $\vec{s}^{\text{new}} \sim p_\Theta(\vec{s} \,|\, \vec{y}^n)$, or, equivalently, from the joint distribution $\vec{s}^{\text{new}} \sim p_\Theta(\vec{y}^n, \vec{s})$ at fixed $\vec{y}^n$. Posterior sampling, however, typically requires per-model analytical derivations and can be computationally expensive. In practice, it can be found that more generic, faster alternatives perform just as well for our purposes.

First of all, note that, independently of the proposal distribution (and, more in general, of the algorithm used to propose variational states $\vec{s}^{\text{new}}$), Alg. 1 is guaranteed to never decrease the free energy. Therefore, at least in principle, even random uniform sampling would be a viable option. However, in practice, we typically seek a relatively small number of "good", high-posterior states $\vec{s}$, and random uniform sampling would explore the latent space (which for a model with H binary latent variables has size 2^H) terribly inefficiently. We would rather employ a proposal distribution that focuses on interesting portions of the latent space. One way to do so is to bias the sampling procedure towards latent states with sparsity compatible with the one learned by the model itself: this can be achieved by sampling from the model's prior distribution, $\vec{s}^{\text{new}} \sim p_\Theta(\vec{s})$. For a sparsity value of S (where we define sparsity as the average number of latents that are active during the generative process, i.e. $S = H\langle\pi_h\rangle_h$ for a model with priors π_h), we are now exploring a sub-set of the latent space with size proportional to $H^S \ll 2^H$ (assuming $S < H$ and sufficiently large H; the first assumption holds true whenever the task at hand is compatible with a sparse representation in latent space, which is often the case). Prior sampling, however, completely disregards information relative to the datapoint $\vec{y}^n$ for which latent states are being searched. To solve this issue, in Guiraud et al. (2016) truncated posteriors are leveraged to provide efficient sampling from an approximation of the *marginal* distribution of $\vec{s}$:

$$p_\Theta(s_h = 1 \,|\, \vec{y}^n) = \langle s_h \rangle_{p_\Theta(\vec{s} \,|\, \vec{y}^n)} \approx \langle s_h \rangle_{q^n_\Phi(\vec{s})} \,. \tag{4.1}$$

When approximate marginals are chosen as proposal distributions, each latent variable s_h^{new} is sampled independently from a Bernoulli distribution with parameter $\langle s_h \rangle_{q^n(\vec{s})}$: dependencies among latents are disregarded but information relating to the datapoint is taken into account. Work by Lücke et al. (2018) further explores this research direction: samples from the more exploratory prior distribution and the more exploitative marginal distribution are mixed into the sets Φ^{new}, and a neural network, trained alongside the generative

model, is used to evaluate the marginal probability $p_\Theta(s_h = 1 \,|\, \vec{y}^{\,n})$, which has the advantage of sharing information across similar datapoints via the network's weights.

4.2 Evolutionary Expectation-Maximization

As the model's posterior structure becomes more complex or as the latent space dimensionality becomes larger, prior or marginal sampling might not be able to find new and different high-posterior variational states $\vec{s}^{\,\text{new}}$ efficiently enough, because these distributions cannot accurately describe dependencies among latent variables. Therefore, it is interesting to investigate whether we can apply tools other than sampling to solve the optimization of the sets of discrete latent states Φ^n. In this vein, note how our problem setting involves a population of discrete variational states $\vec{s}$ that we wish to optimize based on a score function $f = \log p_\Theta(\vec{y}^{\,n}, \vec{s})$: from this perspective, our task is a natural fit for evolutionary algorithms (EAs). Note that the discrete nature of the latent variables, that we assumed when introducing the truncated posterior approximation in Ch. 3, becomes even more relevant in the context of the evolutionary algorithms we will consider.

Evolutionary algorithms are a well-known optimization technique inspired by biological evolutionary processes such as mutation, recombination and fitness-based selection (e.g. Fogel et al., 1966; Rechenberg, 1965), and they have been successfully applied to many typical machine learning tasks such as clustering (Pernkopf and Bouchaffra, 2005; Hruschka et al., 2009), reinforcement learning (Salimans et al., 2017), and hierarchical unsupervised (Myers et al., 1999) or deep supervised learning (see e.g. Stanley and Miikkulainen, 2002; Suganuma et al., 2017; Real et al., 2017, for recent examples). In some cases, EAs have been employed as alternatives to standard procedures (e.g. Hruschka et al., 2009), but most frequently EAs are used to solve specific subproblems: for example, for classification with Deep Neural Networks (DNNs, LeCun et al., 2015; Schmidhuber, 2015), EAs are frequently applied to select the best DNN architectures for a given task (e.g. Stanley and Miikkulainen, 2002; Suganuma et al., 2017) or more generally to find the best hyperparameters of a DNN (e.g. Loshchilov and Hutter, 2016; Real et al., 2017) given a specific dataset and application. Evolutionary algorithms have also been applied in conjunction with EM: Pernkopf and Bouchaffra (2005), for instance, have employed EAs in clustering applications with Gaussian mixture models (GMMs). In their approach, the GMM parameters are updated using EM, while EAs are used to select the best GMM models for the clustering problem (using a minimum description length criterion). Yet other approaches have used EAs to directly optimize, e.g., a clustering objective, but in these cases EAs *replace* EM approaches for optimization (compare e.g. Hruschka et al., 2009).

In Guiraud et al. (2018), we instead investigate the possibility to link EAs and variational optimization more tightly by leveraging evolutionary optimization as a core element of the Truncated Variational Expectation-Maximization algorithm described in Ch. 3. The variational nature of the approximation of the log-likelihood maximization problem is retained, and evolutionary algorithms are used by the inner loop to improve the truncated approximation of the model posteriors. There is a certain freedom in the choice of fitness function and genetic operators that can be employed, which we discuss below. Alg. 2 describes the resulting algorithm, and section 4.3 reports the numerical experiments we conducted in the context of this line of research and that have been published in Guiraud et al. (2018).

Algorithm 2: Evolutionary Expectation Maximization

```
Initialize model parameters Θ
Initialize each Φ^n with S distinct latent states
repeat
    for each datapoint y⃗^n do
        Φ^new = Φ^n
        for each generation do
            Φ^new = mutation (crossover (selection (Φ^new)))
            Φ^n = Φ^n ∪ Φ^new
        Truncate Φ^n to S elements with highest log p_Θ(y⃗^n, s⃗) values
    Update Θ using M-step equations and q^n_Φ as defined in Eq. 3.2
until parameters Θ have sufficiently converged
```

4.2.1 Choosing a fitness function

In order to improve the truncated free energy objective of Eq. 3.5, it is sufficient to select new individuals (i.e., new variational states) that respect criterion Eq. 3.7. Consequently, we can choose any fitness function $f_\Theta(\vec{s}; \vec{y}^n)$ for our evolutionary optimization of sets Φ^n as long as it fulfills the property:

$$f_\Theta(\vec{s}^{\,\text{new}}; \vec{y}^n) \;>\; f_\Theta(\vec{s}; \vec{y}^n) \quad \Leftrightarrow \quad \log p_\Theta(\vec{y}^n, \vec{s}^{\,\text{new}}) \;>\; \log p_\Theta(\vec{y}^n, \vec{s})\,. \tag{4.2}$$

In other words, if Eq. 4.2 is satisfied, any mutation of $\vec{s}$ that increases the fitness $f_\Theta(\vec{s}; \vec{y}^n)$ will result in provably increased free energies if the mutation of $\vec{s}$ substitutes the original state in the corresponding set Φ^n. Together with M-step optimizations of model parameters, the resulting variational EM algorithm will monotonically increase the free energy and therefore the log-likelihood. As long as the chosen fitness function satisfies Eq. 4.2 we are free to pick a form that enables an efficient evolutionary algorithm implementation.

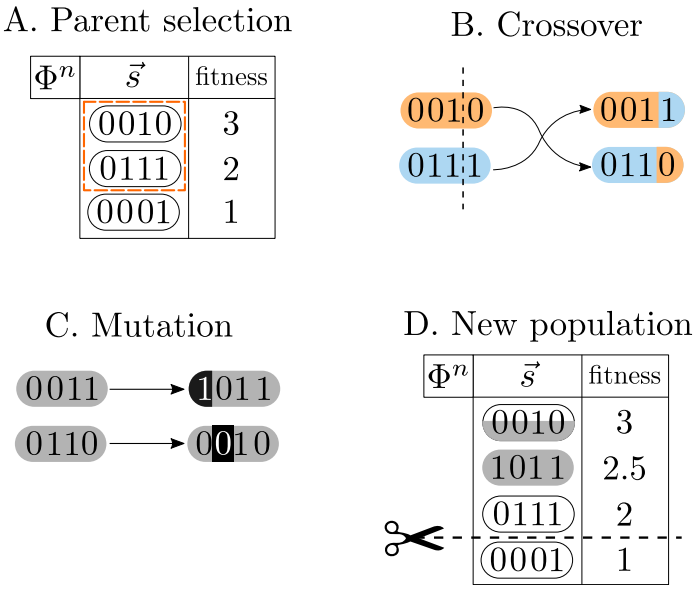

Figure 4.1: Schematic representation of the optimization process of the variational parameters $\Phi^{(n)}$ in EEM. **A.** Some states are selected as parents. **B.** Crossover generates new children. **C.** Each child undergoes random mutation (bitflips). **D.** Children are merged with the original population and the least fit are discarded. New states are marked in gray, one of the candidates was already part of the population.

Concretely, for our purposes we define the fitness as:

$$f_\Theta(\vec{s}; \vec{y}^n) := \widetilde{\log} p_\Theta(\vec{s}; \vec{y}^n) \tag{4.3}$$

where $\widetilde{\log} p_\Theta$, the "log-pseudo-joint" distribution, is a monotonically increasing function of the joint distribution defined to be more efficiently computable and to have better numerical stability. More details regarding log-pseudo-joints are available in Sec. 3.3.3.

4.2.2 Choosing the genetic operators

Our evolutionary algorithm includes three common genetic operators: parent selection, generation of children by single-point crossover and stochastic mutation of the children. We repeat this process over N_g generations in which subsequent iterations use the output of previous iterations as input population. These operations are performed on each datapoint-specific set Φ^n. The offspring generated this way, together with the initial contents of each Φ^n, are then sorted by fitness and the fittest S distinct states (where S is the size of Φ^n) are retained as the updated contents of Φ^n. The whole procedure can be seen as an evolutionary algorithm with perfect memory or very strong elitism

(individuals with higher fitness never drop out of the gene pool). Note that the improvement of the overall fitness of the population (and therefore of the variational lower bound) depends on generating as many as possible *different* children with high fitness over the course of training. A discussion of genetic operator follows, while Fig. 4.1 provides a visual representation of the process.

Parent Selection. This step selects N_p parents from the population Φ^n. Ideally, the selection procedure should balance exploitation of parents with high fitness (which will more likely produce children with high fitness) and exploration of mutations of low-fitness parents (which might eventually produce children with high fitness while increasing population diversity). Diversity is crucial, as Φ^n is a set of *unique* individuals and therefore the improvement of the free energy Eq. 2.4, which is our ultimate goal, depends on generating *different* children with high fitness. In our numerical experiments we explore fitness-proportional selection of parents and random uniform selection of parents. In order to sample states proportionally to their fitness, we must ensure that the fitness f always takes positive values, so that we can re-normalize it and use it as desired average sampling frequency; for this purpose, when fitness-proportional selection of parents is employed, we add an offset, constant w.r.t. $\vec{s}$, to the fitness defined in Eq. 4.3. For the experimental results presented below, this offset was set to $C = |2 \min_{\vec{s}} \left(\widetilde{\log p_\Theta}(\vec{s}; \vec{y}^n) \right)|$ (evaluated once per data point per EM iteration).

Crossover. During the crossover step, parents are combined in pairs; then each pair is assigned a number c from 1 to $H - 1$ with uniform probability: this is the crossover point; finally, the last $H - c$ bits of the parents in the pair are swapped to produce the offspring. We denote N_c the number of children generated in this way. The crossover step can be skipped, making the EA more lightweight but decreasing variety in the offspring. In our experiments, whenever crossover was employed, each possible parent pair performed one crossover, so this step produced $N_c = N_p(N_p - 1)$ children (two per parent pair).

Mutation. Finally, each of the N_c children undergoes a single random bit-flip to further increase offspring diversity. In our experiments we compare results of random uniform selection of the bits to flip with a more refined *sparsity-driven* bit-flip algorithm (detailed in App. C). This latter technique flips 0's and 1's with different probabilities so as to bias mutations towards children with a sparsity compatible with the one learned by the model. In case the crossover step is skipped, bit-flip mutations are performed on N_m identical copies of each parent, i.e. $N_c = N_p N_m$.

A full run of the evolutionary algorithm therefore produces $N_g N_c$ children (or new states $\vec{s}^{\text{new}}$).

4.3 Experimental validation

This section reports the numerical experiments that have been performed to investigate the behaviour as well as validate performance and scaling of the novel EEM algorithm in the context of Guiraud et al. (2018). Experiments with the BSC model have been performed by Jakob Drefs, co-author of the paper and doctoral student at the University of Oldenburg at the time.

Throughout the section, different choices of specific evolutionary algorithms (i.e., different combinations of the genetic operators described in Sec. 4.2.2) are named as follows:

- parent selection is labeled either "fitparents" for fitness-proportional selection or "randparents" for random uniform selection
- if crossover is employed, the label "cross" is present in the name of the algorithm
- the mutation operator is labeled either "sparseflips" or "randflips" for sparsity-driven bitflips and random uniform bitflips respectively

EA hyperparameters (N_g, N_p, N_m) were chosen by performing a grid-search on a sensible subspace, although dependency of final free energy values on these parameters was observed to be relatively weak as long as the product $N_g N_c$ remained constant and comparable to S (the size of each set Φ^n).

4.3.1 Validation on artificial data

To be able to properly characterize the behaviour of EEM and compare different combinations of genetic operators in controlled settings, we performed several numerical experiments on artificial data for which the ground-truth parameters are known. In particular, we employ the bars test as a standard setup for our purposes (Földiák, 1990; Hoyer, 2003; Lücke and Sahani, 2008). The Noisy-OR model and the BSC model, introduced in Sec. 2.1, are used as standard examples of probabilistic generative models with discrete latent variables. The first is a highly non-linear model that supports binary observables, while BSC is a standard linear superposition model that supports continuous observables and assumes Gaussian noise. These relatively simple experiments also serve to validate that EEM indeed has the highly desirable property of model-independence: the same algorithm, with no modification, will be applied to the training of two different models with no need for further model-specific derivations.

The bars test datasets

In this experimental setting, the generative fields W^{gen}_{dh} for $h = 1...H$ are interpreted as H images with $\sqrt{D} \times \sqrt{D}$ pixels each (where typically $D = H^2$).

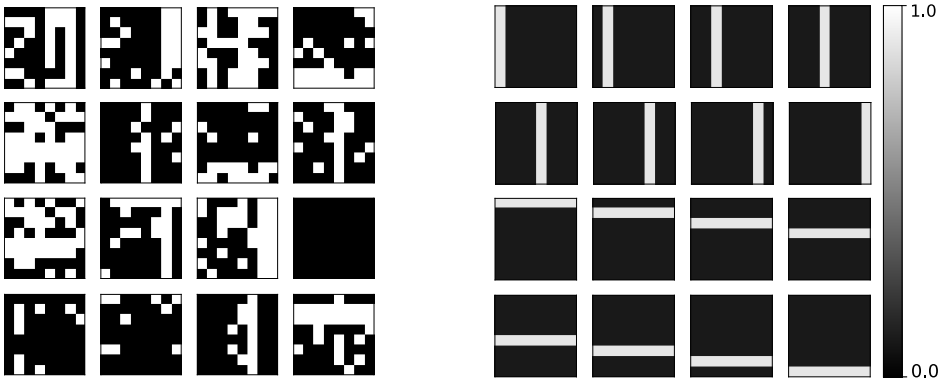

Figure 4.2: **Left:** 16 datapoints extracted from the dataset used for the Noisy-OR standard bars test. **Right:** ground-truth model weights W_{dh} displayed as H grayscale images of $\sqrt{D} \times \sqrt{D}$ pixels.

$H/2$ images (i.e., half of the generative fields) contain vertical bars on a neutral background, and the remaining half contains horizontal bars. The prior probability π_h^{gen} that a bar is present in a datapoint is identical for all components. The dataset is composed of N images generated by running the generative model with these parameters. Output images contain noisy superpositions of vertical and horizontal bars, in black and white for the Noisy-OR model and in gray-scale for BSC.

Training Noisy-OR on bars data

For this experiment, the dataset consists of bars data with $H = 16$ different bars (compare e.g. Spratling, 1999; Lücke and Sahani, 2008), with the standard average crowdedness of two bars per image ($\pi_h^{\text{gen}} = \frac{2}{H} \; \forall \; h$). We train Noisy-OR with EEM using different configurations of the evolutionary algorithm, and compare the *reliability* (compare e.g. Spratling, 1999; Lücke and Sahani, 2008) of the EAs. Reliability is defined here as the fraction of runs for which the learned free energy is above a certain minimum threshold and in which the full dictionary of bars as well as the correct values for the prior probabilities $\vec{\pi}$ are recovered.

Figure 4.3 shows reliabilities over 10 different runs for each of the EAs. On 8×8 images the more exploitative nature of "fitparents-sparseflips" is advantageous over the simpler and more exploratory "randparents-randflips". Note that this is not necessarily true for lower dimensionalities or otherwise easier-to-explore state spaces, in which a naive random search might still be able to quickly find high-fitness individuals. The addition of crossover increases variation in the EA's offspring and therefore in the variational states explored, with measurable benefit.

After the initial verification on a standard bars test, we now make the com-

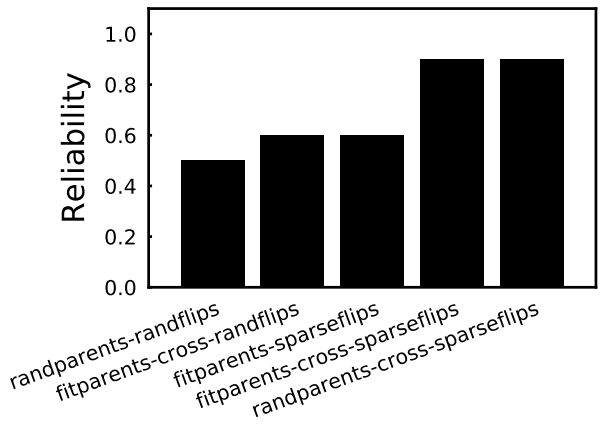

Figure 4.3: Reliability for the listed EAs over 10 runs of EEM for Noisy-OR on noisy 8×8 bars images. For all runs $H = 16$, $N = 5000$, $N_g = 2$, $N_p = 8$, $N_m = 7$, $S = 120$, iterations $= 100$.

ponent extraction problem more difficult by increasing overlap among the bars: a highly non-linear generative model such as Noisy-OR is a good candidate to model occlusion effects in images. Figure 4.4 shows the results of training Noisy-OR with EEM ("fitparents-cross-sparseflips") on a bars dataset in which the latent causes have sensible overlaps. The test parameters were chosen to be equal to those in Fig. 9 of Lücke and Sahani (2008); in particular, note the absence of data noise, also visible from Fig. 4.4 here. After applying EEM with Noisy-OR ($H = 32$) to $N = 400$ images with 16 strongly overlapping bars, we observed that all $H = 16$ bars were recovered in 20 of 25 runs. Quantitative comparison to NMF approaches, neural nets (DI, Spratling et al., 2009), and MCA (Lücke and Sahani, 2008) shows that although EEM for Noisy-OR performs well on average, other approaches can reach higher reliability. Of all the approaches which recover more than 15 bars on average, however, most require additional assumptions: NMF approaches, non-negative sparse coding (Hoyer, 2004) and R-MCA$_2$ require constraints on weights and/or latent activations. Only MCA$_3$ and presumably DI do not require such constraints. DI is a neural network approach, and MCA$_3$ is a generative model with a max-non-linearity as superposition model. When training, it explores all sparse combinations with up to 3 components. Applied with $H = 32$ latents, it hence evaluates more than 60000 states per data point per iteration. In comparison, EEM for Noisy-OR evaluates order of $S = 100$ states per data point per iteration.

Training BSC on bars data

As for Noisy-OR, we first evaluate EEM's ability to train the linear BSC model on a bars test. For BSC, the bars are superimposed linearly (Henniges et al.,

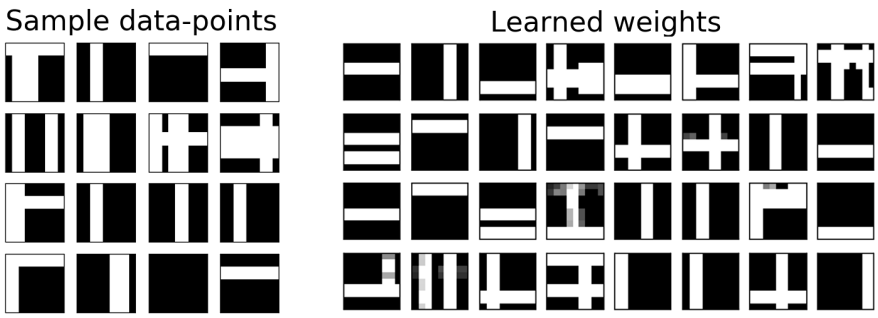

Figure 4.4: Sample input (left) and learned generative fields (right) for a run on overlapping bars. Out of 25 runs, 20 recovered all 16 ground-truth generative components, with the remaining runs missing one component or recovering a distorted version of it. As $H = 32$, the model makes use of the extra generative fields to explain common patterns with overlapping bars.

2010), which makes the problem easier. As a consequence, standard bars tests were solved with very high reliability when using settings similar to those of the Noisy-OR experiments. In order to make the task more challenging, we therefore (A) increased the dimensionality of the data to $D = 10 \times 10$ bars images, (B) increased the number of components to $H = 20$, and (C) increased the average number of bars per data point from two (the standard setting) to five. We employed $N = 5000$ training data points and tested the same five EA configurations as were evaluated for Noisy-OR. We set the number of hidden units to $H = 20$ and used $S = 120$ variational states. Per data point and per iteration, in total 112 new states ($N_p = 8$, $N_m = 7$, $N_g = 2$) were generated. Per configuration of the EA, we performed 20 independent runs, each with 1000 iterations. The results of the experiment are depicted in Fig. 4.5. Differently from what was observed for Noisy-OR, despite being the simplest of the examined approaches, the "randparents-randflips" EA achieved the highest reliability. Reliability decreases if parents are no longer selected randomly but proportionally to their fitness. This observation could also be made for lower dimensional data with lower crowdedness. A further impact on the reliability is the bit-flip procedure. EAs in which bits are flipped according to sparseness perform worse compared to the case of random bitflips. Taking these observations together, it could be concluded for BSC that EAs with a more exploratory and less exploitative nature are beneficial in a scenario of increased data dimensionality and increased crowdedness. The observation of sparseness-driven EAs performing poorly may also be explained by the initialization of Φ^n and with the initialization of the model parameter π. These values are initially drawn from a Bernoulli distribution with $p_\Theta(s_h = 1) = \frac{1}{H}$.

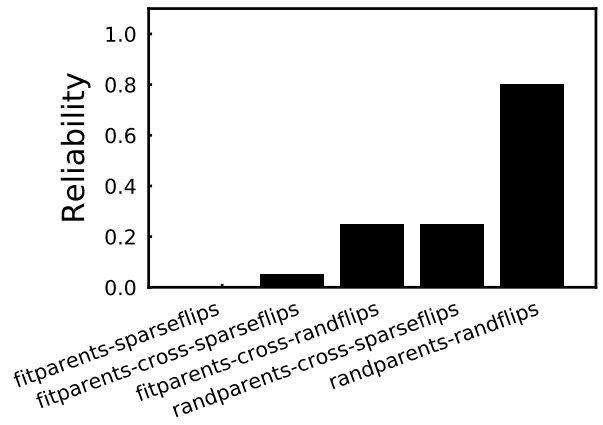

Figure 4.5: Reliability for the listed EAs over 20 runs of EEM for BSC on 10x10 bars images. On average five bars occurred per data point.

Such a choice makes it difficult for EAs with a search space restricted to sparse solutions to find new states with high crowdedness. The results show that EEM can achieve high reliabilities (more than 80 %) in challenging scenarios with high dimensionality and high crowdedness.

4.3.2 Scaling to larger datasets: training Noisy-OR on natural image patches

Next, we verify the approach on natural data. We use patches of natural images, which are known to have a multi-component structure, which are well investigated, and for which, typically, models with high-dimensional latent spaces are applied. The image patches are extracted from the van Hateren image database (van Hateren and van der Schaaf, 1998).

We consider raw images patches, i.e., images without substantial preprocessing which directly reflect light intensities. Such image patches were generated by extracting random square subsections of a single 255×255 image of overlapping grass wires (part of image 2338 of the database). We removed the brightest 1% pixels from the dataset, scaled each datapoint to have gray-scale values in the range $[0, 1]$ and then created data points with binary entries by repeatedly choosing a random gray-scale image and sampling binary pixels from a Bernoulli distribution with parameter equal to the gray-scale value of the original pixel. Note that components in such light-intensity images can be expected to superimpose non-linearly because of occlusion, which motivates the application of a non-linear generative model such as Noisy-OR. We employ the "fitparents-cross-sparseflips" evolutionary algorithm that was shown to perform best on artificial data (Fig. 4.3). Parameters were $H = 100$, $S = 120$, $N_g = 2$, $N_p = 8$, $N_m = 7$. To encourage the model to make use

of all generative fields, prior values π_h were clamped to a minimum of 0.01. Figure 4.6 shows 50 of the generative fields learned over 200 iterations. EEM allows learning of generative fields resembling curved edges, in line with expectations and with the results obtained in (Lücke and Sahani, 2008), showing that EEM scales to the training of non-linear models with complex posteriors and hundreds of latent variables.

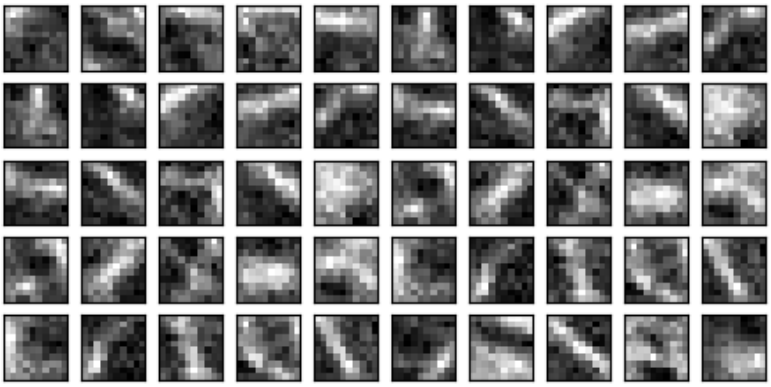

Figure 4.6: 50 gray-scale generative fields learned by applying EEM ("fitparents-sparseflips") for Noisy-OR to natural image patches.

4.3.3 Scaling to larger datasets: training BSC on natural image patches

Now we consider image patches pre-processed using common whitening approaches as they are customary for sparse coding approaches (Olshausen and Field, 1997). We use $N = 100,000$ patches of size $D = 16 \times 16$ pixels, randomly picked from the whole data set. The highest 2 % of the amplitudes were clamped to compensate for light reflections and patches without significant structure were excluded for learning. ZCA whitening (Bell and Sejnowski, 1997) was applied retaining 95 % of the variance (we used the procedure of a recent paper (Exarchakis and Lücke, 2017)). We trained the BSC model for 4,000 iterations using the "fitparents-cross-sparseflips" EA and employing $H = 300$ hidden units and $S = 200$ variational states. Per data point and per iteration, in total 360 new states ($N_p = 10$, $N_m = 9$, $N_g = 4$) were sampled to vary Φ^n. The results of the experiment are depicted in Fig. 4.7. The obtained generative fields primarily take the form of Gabor functions with different locations, orientations, phase, and spatial frequencies. This is a typical outcome of sparse coding being applied to images. On average more than five units were activated per data point showing that the learned code makes

use of the generative model's multiple causes structure. The generative fields converged faster than prior and noise parameters (similar effects are known from probabilistic PCA for the variance parameter). The finite slope of the free energy after 4000 iterations is presumably due to these parameters still changing slowly.

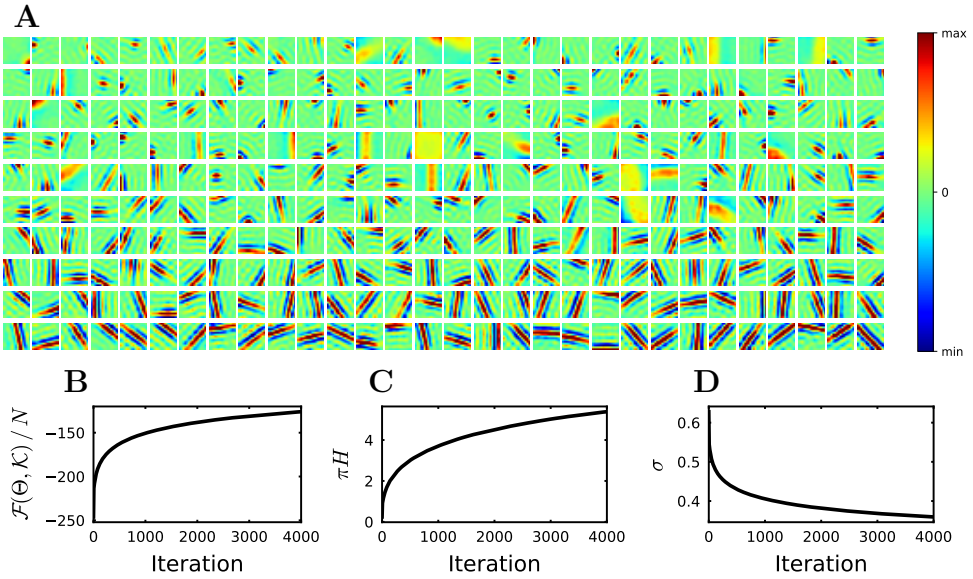

Figure 4.7: Results of training the BSC model on natural images. **A** Full dictionary learned from natural images by the BSC model trained with the "fitparents-cross-sparseflips" EA using 4,000 iterations. The generative fields are ordered according to their activation, starting with most active fields (highest π_h). The colormap of each field is normalized individually so that zero values sit in the middle of the scale (green color). **B** Evolution of the free energy per data point over iterations. **C** Evolution of the expected number of active hidden units per data point over iterations. **D** Evolution of the standard deviation over iterations.

4.3.4 Denoising the "house" image

Having validated the viability and scalability of the EEM training algorithm, we use it to apply BSC to the standard "house" image denoising benchmark. We do so in a "zero-shot" setting, i.e. the noisy "house" image is the only data that is used for the experiment: the algorithm has no access to a clean version of the image nor to other images (compare e.g., Shocher et al., 2018; Imamura et al., 2019). On one hand, this scenario often occurs in real-world applications, e.g., electromagnetic imaging of viruses, imaging of astronomical observations, very low resolution scans or pictures of hand-written documents or drawings,

etc.; on the other hand, the "zero-shot" setting fits EEM (and TVEM in general) particularly well, since it imposes no fast inference requirements (see Sec. 3.3.4).

Following the procedure described in (Sheikh et al., 2014) we first added Gaussian white noise with a standard deviation $\sigma = 15$ to the original 256×256 "house" image. Subsequently, we cut the noisy image into all possible 62,001 overlapping patches of size 8×8. For training we used the "fitparents-randflips" EA on a BSC model with $H = 256$ hidden units, $S = 200$ variational states, and hyperparameters $N_p = 10$, $N_m = 9$, $N_g = 4$. Training lasted 5,000 epochs.

Finally we used the trained model to estimate the most likely value of each of the image pixels in each of the patches. More concretely, given a trained model with parameters Θ, we estimated the value of a pixel in a single patch as $y_d^{\text{est}} = \langle y_d \rangle_{p_\Theta(y_d|\vec{y})}$. Using $p_\Theta(y_d \mid \vec{y}) = \sum_{\{\vec{s}\}} p_\Theta(y_d \mid \vec{s}) p_\Theta(\vec{s} \mid \vec{y})$ we obtain:

$$y_d^{\text{est}} = \left\langle \langle y_d \rangle_{p_\Theta(y_d|\vec{s})} \right\rangle_{p_\Theta(\vec{s}|\vec{y})} = \left\langle \sum_h W_{dh} s_h \right\rangle_{p_\Theta(\vec{s}|\vec{y})} = \sum_h W_{dh} \langle s_h \rangle_{p_\Theta(\vec{s}\,|\,\vec{y}^{\,n})} \,. \tag{4.4}$$

The expectation value in the right-hand-side of Eq. 4.4 is then approximated by means of the learned truncated distributions using Eq. 3.4. The final denoised estimate for each of the image's pixels is then a weighted average of the estimates of a pixel value in different patches (see e.g. Burger et al., 2012). Fig. 4.8 (right) shows the denoised picture resulting from this process and reports the Pixel Signal-to-Noise Ration (PSNR) values for the noisy (center) and denoised (right) versions of the image.

Figure 4.8: **Left:** Noiseless "house" image. **Center:** Image with additive Gaussian noise applied ($\sigma = 25$, PSNR $= 20.19\,\text{dB}$). **Right:** Image denoised with EEM for BSC (PSNR $= 32.32\,\text{dB}$).

These numerical experiments are a proof of concept which shows that EAs are indeed able to train linear as well as non-linear generative models with large latent spaces. The results published in Guiraud et al. (2018) that we reported in this chapter represent, to our knowledge, the first examples of Noisy-OR or probabilistic sparse coding models trained with EAs as an integral part of

variational EM, although both models had been studied extensively before. In the context of data reconstruction, EAs have been used e.g. for maximum a-posteriori optimization of sparse coding models in a non-probabilistic setting by Ahmadi and Salari (2016). Furthermore, note that EEM is formulated in terms of joint probabilities and therefore generalizes well and has straightforward applicability to other generative models with binary latents. In particular, it is interesting to ask whether these techniques can be successfully applied to more complex, deep hierarchical generative models and whether they can be combined with state-of-the-art gradient-based upgrades of the weights of such a model. The next chapter is dedicated to try answering this exact question.

Chapter 5

Truncated Variational Approximation and Variational Autoencoders

The previous chapters have introduced Truncated Variational Expectation Maximization and its specific incarnation that uses evolutionary algorithms to optimize the variational parameters Φ, namely Evolutionary Expectation Maximization (EEM). Given the novelty of the approach, the first question we asked was whether EEM would in fact work in practice, even for large latent spaces. Through the numerical experiments in Ch. 4, we were able to provide a positive answer; furthermore, even with the elementary generative models we employed, we could show that EEM is able to provide competitive performance e.g. in denoising applications. These results open the door to several further interesting research directions. One compelling question that can be asked at this point is whether EEM can be successfully applied to more sophisticated generative models, which will potentially feature a more complex posterior structure.

Research work by Drefs et al. (2020) (currently under review) therefore built on the results by Guiraud et al. (2018) and explored the application of EEM to Spike-and-Slab Sparse Coding (SSSC), which has been employed by several previous studies in a number of variants (e.g., Titsias and Lázaro-Gredilla, 2011; Lücke and Sheikh, 2012; Goodfellow et al., 2012; Sheikh et al., 2014). SSSC is a more expressive model than, for instance, BSC due to the presence of an extra set of latent variables: a first set of binary variables describes presence or absence of a given feature, just like in BSC, and one other set of continous values describes the intensity of the feature. Importantly for the application of EEM to this model, M-step equations for SSSC can be derived in closed form and can be formulated as expectations w.r.t. the posterior over just the binary latent space.

In parallel, I focussed on an equally interesting challenge: extending EEM to deep generative models with no closed-form M-steps; in this scenario, our algorithm for the optimization of truncated posteriors as variational distributions has to tie into model parameter optimization that is typically performed via backpropagation and gradient ascent. Among such deep generative models, Variational Autoencoders (VAEs; Kingma and Welling, 2014; Rezende et al., 2014) are a prominent and very actively researched topic in unsupervised learning. VAEs, in their many different variations, have successfully been applied to a large number of tasks including semi-supervised learning (e.g. Maaløe et al., 2016), anomaly detection (e.g. An and Cho, 2015; Kiran et al., 2018), sentence interpolation (Bowman et al., 2016), music interpolation (Roberts et al., 2018) and drug response prediction (Rampasek et al., 2017). A desired feature when applying VAEs to a given problem is that their latent variables (i.e., the encoder output variables) correspond to meaningful properties of the data, ideally to those latent causes that have originally generated the data. However, many real-world datasets suggest the use of *discrete* latents as they often describe the data generation process more naturally. For instance, the presence or absence of objects in images is best described by binary latents (e.g. Jojic and Frey, 2001). Discrete latents are also a popular choice in modeling sounds; for instance, describing piano sounds may naturally involve binary latents: keys are pressed or not (e.g., Sheikh et al., 2014; Titsias and Lázaro-Gredilla, 2011; Goodfellow et al., 2013). The success of standard forms of VAEs has consequently spurred research on novel formulations that feature discrete latents (e.g. Rolfe, 2016; Khoshaman and Amin, 2018; Roy et al., 2018; Sadeghi et al., 2019; Vahdat et al., 2019).

In this chapter we investigate the possibility to apply the EEM algorithm introduced in Ch. 4 to the training of Variational Autoencoders with binary latents: in principle, this combination would result in a mathematically-grounded, novel training algorithm for a widely popular deep generative model. First and foremost, our goal will be to prove that such training is indeed possible. It is interesting to ask, then, whether such a training algorithm offers any advantage with respect to other techniques applied to binary VAEs, and whether it results in competitive performance in real-world applications.

Sec. 5.1 introduces the mathematical framework behind Variational Autoencoders, and Sec. 5.2 complements it by discussing popular techniques to extend this framework to discrete latent variables. Sec. 5.3 discusses training of binary-latent VAEs with the EEM algorithm, and subsequent sections are dedicated to extensive testing and benchmarking of this model. At the time of writing, the results presented in this chapter are under review as Guiraud et al. (2020).

The idea was developed jointly with Jörg Lücke. I carried out first mathematical derivations to inspect the learning objective of a Variational Autoencoder

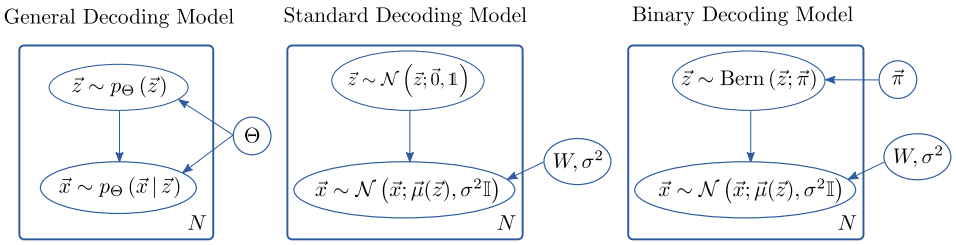

Figure 5.1: From left to right: generic VAE decoding model, continuous-latent VAE model with Gaussian noise and binary-latent VAE model, in plate notation.

model trained with the EEM algorithm, and verify theoretical viability in particular with respect to scaling properties. Subsequently, I proceeded with the numerical implementation of the training algorithms, designing the concrete model's architecture and incorporating cyclical learning rate schedules. I carried out all numerical experiments presented in this chapter, which were conceived and analyzed in consultation with Jörg Lücke and Jakob Drefs. The denoising experiments are based on the method used by Jakob Drefs for the BSC model in Guiraud et al. (2018), which I adapted to Variational Autoencoders. A literature review on previous discrete VAE implementations was carried out in collaboration Jörg Lücke. The overview of standard VAE training (Sec. 5.1) was written together with Jörg Lücke. As first author of Guiraud et al. (2020) I was responsible for most of its preparation, but an extensive review of previous work on image denoising was carried out by Jakob Drefs, who also conducted the denoising experiments with the noise2void model that we compare to in Section 5.4.6.

5.1 Training standard Variational Autoencoders

First of all, let us review standard training of Variational Autoencoders and the core set of equations supporting it. The starting point is precisely the log-likelihood maximization of probabilistic generative models discussed in Ch. 3; we will try to underline the similarities between standard VAE training and training of the simpler models that were introduced in previous chapters, as well as notable differences. To cover continuous as well as binary decoding models, in this section we will make use of the following generic VAE generative model:

$$p_\Theta(\vec{s}) = \mathcal{P}(\vec{s}; \Theta) \tag{5.1}$$

$$p_\Theta(\vec{y} \mid \vec{s}) = \mathcal{N}\left(\vec{y}; \vec{\mu}_\Theta(\vec{s}), \sigma^2_\Theta(\vec{s})\mathbb{I}\right), \tag{5.2}$$

where $\mathcal{P}$ could be a standard normal distribution (in the case of continuous latent variables) or the Bernoulli distribution (in the case of binary latent variables) and $\vec{\mu}_\Theta$ and σ^2_Θ are deep neural networks (DNNs) parameterized by Θ. Note that, if the task at hand were to call for it, the Gaussian noise model could be swapped for a different distribution with little change to the optimization procedure: for example, a common choice in case of binary data is Bernoulli noise. Fig. 5.1 reports the different kinds of architectures that we take into consideration here, in plate notation.

Given a dataset comprising N datapoints $\vec{y}^n$, variational autoencoders then seek model parameters Θ which maximize the usual data log-likelihood of Eq. 2.1, i.e.:

$$L(\Theta) = \sum_n \log p_\Theta(\vec{y}^n) = \sum_n \log \sum_{\{\vec{s}\}} p_\Theta(\vec{y}^n, \vec{s}) \,. \qquad \text{(2.1 revisited)}$$

As we will focus on binary latent spaces, we chose $\sum_{\vec{s}}$ in Eq. 2.1 as a placeholder for summation as well as integration: given a latent space of dimensionality H, in the case of binary latents, the summation $\sum_{\vec{s}}$ is intended over all possible bit vectors $\vec{s} \in \{0,1\}^H$; in the case of continuous latents, with slight abuse of notation we let the symbol denote an integral over the space in which the variables $\vec{s} \in \mathbb{R}^H$ are embedded.

As discussed in Ch. 3, the summation (or integration) over $\vec{s}$ makes the direct optimization of $L(\Theta)$ prohibitively computationally expensive for any practical purpose. For instance, in the binary case, computation of gradients for the optimization of the DNN weights of $\vec{\mu}_\Theta$ and $\vec{\sigma}_\Theta$ would require passing all possible 2^H states $\vec{s}$ through the DNNs. Instead of maximizing $L(\Theta)$ directly, VAEs therefore maximize the Evidence Lower BOund (ELBO), a particular rewriting of the variational free energy of Eq. 2.4:

$$F(\Phi, \Theta) = \sum_n \langle \log p_\Theta(\vec{y}^n \,|\, \vec{s}) \rangle_{q^n_\Phi} - D_{\mathrm{KL}}(q^n_\Phi(\vec{s}; \vec{y}^n); p_\Theta(\vec{s})) \qquad \text{(2.14 revisited)}$$

This rewriting of the free energy is usually regarded as the "core equation" of VAEs (e.g. Doersch, 2016).

5.1.1 Excursus: maximum likelihood and the autoencoding problem

Without going into excessive detail, it is useful at this point to mention how such a rewriting of the variational lower bound of the log-likelihood is connected to the autoencoding problem, that is, the problem of passing data through an encoding transformation (which typically compresses the input), a successive decoding transformation (which brings data back to the original variable space) and minimizing the *reconstruction error* of this process, i.e., the difference between original input and decoder output. This difference is

evaluated using a distance metric appropriate to the semantics of the kind of data involved, and the identification of such a metric is an interesting problem in itself: for example, much work has been devoted to researching robust image similarity metrics (see e.g. Wang et al., 2004; Zhao et al., 2016; Zhang et al., 2018b). Typically, encoder and decoder are implemented as deep neural networks joined at one extremity (the encoder's output is the decoder's input), whose weights are optimized via gradient descent using an aggregation of the reconstruction error of each datapoint as loss function.

The autoencoding problem is a popular research topic in unsupervised machine learning because a well-performing autoencoder can be easily applied to a variety of tasks, including dimensionality reduction or lossy compression (trivially by running the encoder to compress data and the decoder to decompress it), anomaly detection (anomalous data is decoded very poorly) or content generation (by "decoding" randomly sampled inputs).

The maximum-likelihood training of Variational Autoencoders as defined in Eqs. 5.1 and 5.2 can be interpreted as solving a *stochastic* version of the autoencoding problem. The variational distributions $q^n_\Phi(\vec{s}; \vec{y}^n)$ play the role of a stochastic encoder: they define a mapping from observed to latent space by introducing a correspondence between a datapoint $\vec{y}^n$ and latent states $\vec{s}$ that are likely to be sampled from q^n_Φ. Similarly, the generative distribution $p_\Theta(\vec{y}^n \mid \vec{s})$ can be seen as a stochastic decoder: it defines a mapping in the opposite direction, from latent to observed space, by associating a latent state $\vec{s}$ to datapoints that are likely to be generated by the model given that particular latent state. In this perspective, the first term on the right-hand side of Eq. 2.14 can then be interpreted as a stochastic reconstruction error: the summands $\langle \log\left(p_\Theta(\vec{y}^n \mid \vec{s})\right)\rangle_{q^n_\Phi}$ are greater when latent states $\vec{s}$ that are likely to "correspond" to a given datapoint $\vec{y}^n$ via the mapping defined by q^n_Φ in turn are likely to be mapped back to $\vec{y}^n$, i.e., produce high values of $p_\Theta(\vec{y}^n \mid \vec{s})$. From here on we will therefore refer to variational distributions q^n_Φ and generative distribution $p_\Theta(\vec{y}^n \mid \vec{s})$ as the encoding and decoding model of a VAE, respectively.

Bringing further the analogy between the approximate maximum likelihood objective of Eq. 2.14 and the autoencoding problem, the second term on the right-hand side of Eq. 2.14 can be seen as a regularization term that prevents overfitting of the variational distributions by pulling them towards a standard normal distribution.

5.1.2 Parameter optimization

The optimization of the VAE parameters Φ and Θ proceeds in a similar fashion as the step-wise free energy optimization of the Expectation-Maximization

algorithm:

$$\text{change } \Phi^{\text{old}} \text{ to } \Phi^{\text{new}} \text{ such that } F(\Phi^{\text{new}}, \Theta^{\text{old}}) > F(\Phi^{\text{old}}, \Theta^{\text{old}}) \tag{5.3}$$

$$\text{change } \Theta^{\text{old}} \text{ to } \Theta^{\text{new}} \text{ such that } F(\Phi^{\text{new}}, \Theta^{\text{new}}) > F(\Phi^{\text{new}}, \Theta^{\text{old}}) \tag{5.4}$$

Concretely, the optimization of the decoder parameters Θ in Eq. 5.4 typically only involves the first term on the right-hand side of Eq. 2.14 since for standard VAE decoding models (Eqs. 5.1 and 5.2) the prior is the standard normal distribution and does not depend on Θ. Consequently, the optimization of the first term reduces to computing the gradient of $\langle \log p_\Theta(\vec{y}^n \mid \vec{s}) \rangle_{q^n_\Phi}$ w.r.t. Θ. This is usually achieved via a sampling approximation to efficiently estimate the expectation value and subsequent gradient-based optimization of the DNN parameters. Standard DNN optimization tools can then be used for efficient training of the decoding parameters Θ, including automatic differentiation frameworks and sophisticated gradient-based algorithms. We report below the explicit form of the gradients that are required to be computed, as it will be useful in order to compare to the alternative training scheme that is the main focus of this chapter:

$$\begin{aligned}
\vec{\nabla}_W F(\Phi, \Theta) &= \sum_n \vec{\nabla}_W \langle \log \left(p_\Theta(\vec{y}^n \mid \vec{s}) \right) \rangle_{q^n_\Phi} \\
&= \sum_n \vec{\nabla}_W \langle \log \left(\mathcal{N}(\vec{y}^n; \vec{\mu}_\Theta(\vec{s}, W), \sigma^2 \mathbb{I}) \right) \rangle_{q^n_\Phi} \\
&\approx -\frac{1}{2\sigma^2} \sum_n \frac{1}{L} \sum_l \vec{\nabla}_W \| \vec{y}^n - \vec{\mu}_\Theta(\vec{s}^l, W) \|^2 \,, \\
&\quad \text{where } \; \vec{s}^l \sim q^n_\Phi(\vec{s})
\end{aligned}$$

We can slightly rewrite this expression to obtain:

$$\vec{\nabla}_W F(\Phi, \Theta) \;\approx\; -\frac{1}{2\sigma^2} \sum_n \sum_{\vec{s} \sim q^n_\Phi} \left(\frac{1}{L} \right) \vec{\nabla}_W \| \vec{y}^n - \vec{\mu}_\Theta(\vec{s}, W) \|^2 \,, \tag{5.5}$$

The optimization of the lower bound $F(\Phi, \Theta)$ w.r.t. the encoding parameters Φ (Eq. 5.3) poses a greater challenge. First and foremost, a specific functional form must be chosen for the variational distributions q^n_Φ such that optimization of the objective turns out to be efficiently implementable. Standard VAEs typically pick a Gaussian distribution which includes a second DNN (or a second set of DNNs):

$$q^n_\Phi(\vec{s}; \vec{y}^n) = \mathcal{N}(\vec{s}; \vec{\mu}_\Phi(\vec{y}^n), \vec{\Sigma}_\Phi(\vec{y}^n)) \tag{5.6}$$

where the mean $\vec{\mu}_\Phi(\vec{y}^n)$ and the variance $\vec{\Sigma}_\Phi(\vec{y}^n)$ are both non-linear functions of $\vec{y}^n$ parameterized by a DNN with weights Φ. Note that the covariance is often assumed diagonal as in Eq. 5.2, although non-diagonal covariances could

be important because the true posterior that $q^n_\Phi(\vec{s};\vec{y}^n)$ seeks to approximate can be expected to show strong explaining-away effects.

For standard VAEs (Eqs. 5.1 and 5.2), the choice of encoding model of Eq. 5.6 offers two main advantages: (A) the Gaussian form makes it possible to solve the KL-term in Eq. 2.14 analytically, and (B) the distribution $q^n_\Phi(\vec{s};\vec{y}^n)$ can be optimized by gradient ascent after computing gradients of $F(\Phi,\Theta)$ w.r.t. Φ. However, naive application of the gradient operator yields high-variance gradient estimations when coupled with the sampling approximation that is typically employed to make Eq. 2.14 efficiently computable. More concretely, using the usual log-derivative trick:

$$\begin{aligned}
\nabla_\Phi \langle \log p_\Theta(\vec{y}^n \mid \vec{s})\rangle_{q^n_\Phi} &= \sum_{\{\vec{s}\}} \nabla_\Phi q^n_\Phi(\vec{s};\vec{y}^n) \log p_\Theta(\vec{y}^n \mid \vec{s}) \\
&= \sum_{\{\vec{s}\}} q^n_\Phi(\vec{s};\vec{y}^n)\ \nabla_\Phi \log q^n_\Phi(\vec{s};\vec{y}^n)\ \log p_\Theta(\vec{y}^n \mid \vec{s}) \\
&= \langle \nabla_\Phi \log q^n_\Phi(\vec{s};\vec{y}^n) \log p_\Theta(\vec{y}^n \mid \vec{s})\rangle_{q^n_\Phi} \\
&\approx \frac{1}{L}\sum_l \nabla_\Phi \log q^n_\Phi(\vec{s}^{\,l};\vec{y}^n) \log p_\Theta(\vec{y}^n \mid \vec{s}^{\,l}) \\
&\text{where}\ \ \vec{s}^{\,l} \sim q^n_\Phi(\vec{s};\vec{y}^n)\,.
\end{aligned}$$

This gradient estimator exhibits very high variance (see e.g. Paisley et al., 2012) and is generally impractical for VAE training. In order to obtain low-variance estimates of the gradients, then VAEs take advantage of the *reparameterization trick* (Kingma and Welling, 2014; Rezende et al., 2014) to re-express sampling from the encoding distribution using a set of novel latent variables $\vec{\rho}$ such that:

$$\vec{s} \sim q^n_\Phi(\vec{s};\vec{y}^n) \quad \Leftrightarrow \quad \left\{ \begin{array}{rcl} \vec{\rho} & \sim & p(\vec{\rho}) \\ \vec{s} & = & \vec{g}(\vec{\rho},\vec{y}^n,\Phi) \end{array} \right. , \tag{5.7}$$

where $p(\rho)$ is a distribution that does not depend on Φ, i.e., all parameter dependencies are contained in the deterministic function $\vec{g}(\vec{\rho},\vec{y}^n,\Phi)$. Reparameterization serves the purpose of enabling efficient stochastic gradient ascent of $F(\Phi,\Theta)$ w.r.t. Φ: the gradient estimator after reparameterization is sufficiently low-variance and takes the following form:

$$\begin{aligned}
\nabla_\Phi \langle \log p_\Theta(\vec{y}^n \mid \vec{s})\rangle_{q^n_\Phi} &\approx \frac{1}{L}\sum_l \log p_\Theta(\vec{y}^n \mid \vec{s}^{\,n,l}) \\
\text{where}\ \ \vec{s}^{\,n,l} &= \vec{g}(\vec{\rho},\vec{y}^n,\Phi)\ \ \text{and}\ \ \vec{\rho} \sim p(\vec{\rho})\,.
\end{aligned} \tag{5.8}$$

The gradient $\nabla_\Phi F(\Phi,\Theta)$ can therefore be computed as follows:

$$\begin{aligned}
\vec{\nabla}_\Phi F(\Phi,\Theta) = &\frac{1}{L}\sum_n\sum_l \left(\vec{\nabla}_\Phi \vec{g}(\vec{\rho}^{\,l},\vec{y}^n,\Phi)\right)\vec{\nabla}_{\vec{s}} \log p_\Theta(\vec{y}^n \mid \vec{s})\Big|_{\vec{s}=\vec{g}(\vec{\rho}^{\,l},\vec{y}^n,\Phi)} \\
&- \sum_n \vec{\nabla}_\Phi D_{\mathrm{KL}}(q^n_\Phi(\vec{s};\vec{y}^n); p_\Theta(\vec{s}))\,.
\end{aligned} \tag{5.9}$$

Note that the first term of Eq. 5.9 now contains two gradients: the gradient $\left(\vec{\nabla}_\Phi \vec{g}\right)$ is a standard gradient w.r.t. the DNN(s) of the encoding model, while the second gradient involves the encoding as well as decoding model. Again, for standard VAEs, the D_{KL} term of Eq. 5.9 can typically be computed analytically, which makes the gradients directly computable.

Given the gradient of Eq. 5.9, the encoding parameters Φ are then routinely optimized using one or few samples of $\vec{\rho}$ for each data point $\vec{y}^n$. How many samples $\vec{\rho}$ and $\vec{y}^n$ are averaged over before Φ is updated is a design decision made on the grounds of empirical performance evaluations. Fig. 5.2 (left side) summarizes the sequence of methods used, which allows for the optimization of the encoding model for standard VAEs.

As a final observation, it must be mentioned that standard VAE training is amortized: a single set of parameters applies to the whole dataset. As discussed in Sec. 3.3.4, this enables fast inference and might yield quicker convergence thanks to the ability of the training procedure to pool information coming from similar datapoints. However, as discussed, the presence of an "amortization gap" between variational lower bound and true model log-likelihood can impose a measurable upper limit to the performance of such a model (see e.g. Kim et al., 2018; Cremer et al., 2018).

5.2 Discrete latent Variational Autoencoders

Existing optimization procedures for VAEs with discrete latents follow the same steps as those of standard VAEs described above. However, discrete or binary latents pose substantial further obstacles in learning, mainly due to the fact that backpropagation through discrete variables is generally not possible (Rolfe, 2016; Bengio et al., 2013). Specifically, the gradients w.r.t. latent variables $\vec{s}$ in Eq. 5.9, introduced via the reparameterization trick, are problematic.

In order to maintain the general VAE framework for the optimization of the encoding model, different groups have therefore suggested different possible solutions: work by Rolfe (2016), for instance, extends VAEs with discrete latents by auxiliary continuous latents such that gradients can still be computed. Work on the concrete distribution (Maddison et al., 2016) or Gumbel-softmax distribution (Jang et al., 2016) proposes newly defined continuous distributions that contain discrete distributions as limit cases. The continuous distributions can then be treated within the standard VAE framework. Finally, work by van den Oord et al. (2017) and Roy et al. (2018) combines VAEs with a vector quantization (VQ) stage in the latent layer. Latents become discrete through quantization but gradients for learning are adapted from latent values before they are processed by the VQ stage. All these approaches maintain the basic setup of VAEs but add specific additional mechanisms to treat discrete latents. These additional techniques are entangled, during training, with those

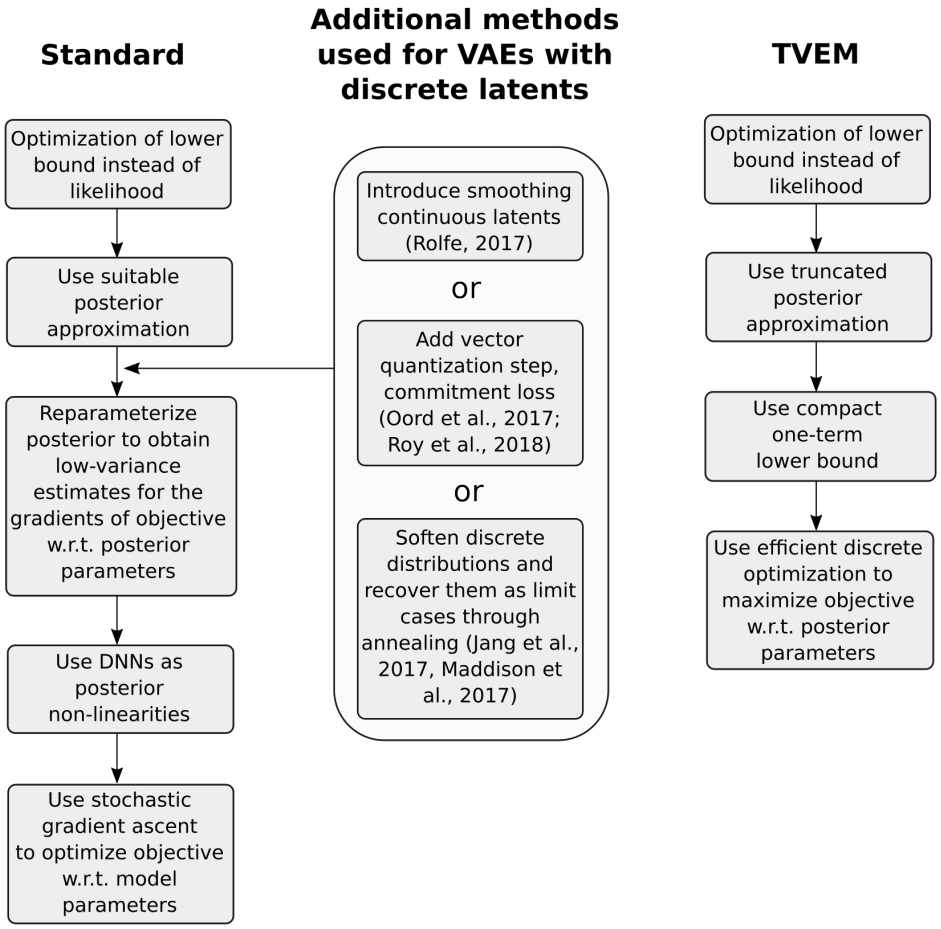

Figure 5.2: Standard series of methods applied to optimize the encoding model of VAEs. **Left:** methods applied for encoding models of standard VAEs (e.g., with encoding model 5.6). **Middle:** additional methods applied to maintain the standard procedure of encoding model optimization also for discrete latent variables. **Right:** alternative approach to optimize the VAE encoding model using direct discrete optimization in the form of EEM.

standard methods already in place for gradient computation. Furthermore, they add further types of design decisions and hyperparameters, for example parameters for annealing from softened discrete distributions to the (hard) discrete distributions of the encoder. Fig. 5.2 (left and center) summarizes this situation.

For VAEs with discrete latents, it may therefore be a desirable goal to investigate alternative, more direct optimization procedures that do not require a softening of discrete distributions or the use of other indirect solutions. Such a direct approach, however, is challenging because, once DNNs are used to de-

fine the encoding model, estimations of gradients of latent variables seem unavoidable. A more direct optimization procedure, as is investigated in Guiraud et al. (2020) (under review at the time of writing), consequently has to change VAE training more substantially.

TVEM, introduced in Ch. 3, and its particular implementation that is EEM (see Ch. 4) offer a possible way forward: we will be able to maintain the variational setting and a decoding model with DNNs as non-linearity; gradient-based optimization of the decoder's DNN will also be maintained. However, we will not use an encoder model parameterized by DNNs, but instead we will make use of the truncated variational distributions of Eq. 3.2. Notably, by employing truncated posteriors, the training algorithm will side-step all issues relative to backpropagation of gradients through discrete latent variables, completely removing the need for the techniques described above that all aim at reframing the discrete optimization problem so that, mathematically, the standard VAE training with reparameterization trick can still be employed. We will instead address the training of the encoder directly, as a discrete optimization problem, without requiring alteration to the generative model nor additional auxiliary probability distributions. The next section describes the resulting training algorithm.

5.3 Training discrete latent Variational Autoencoders with truncated posteriors

First of all, let us define a concrete generative model for VAEs with binary latent variables:

$$\begin{aligned} p_\Theta(\vec{s}) = \mathrm{Bern}(\vec{s};\vec{\pi}) = \prod_h \pi_h^{s_h}(1-\pi_h)^{(1-s_h)} \\ p_\Theta(\vec{y}^{\,n} \mid \vec{s}) = \mathcal{N}\left(\vec{y};\vec{\mu}_\Theta(\vec{s};W),\sigma^2\mathbb{I}\right) \end{aligned} \tag{5.10}$$

The set of model parameters is $\Theta = \{\vec{\pi}, W, \sigma^2\}$, where W incorporates DNN weights and biases that parameterize $\vec{\mu}_\Theta$. We assume homoscedasticity of the Gaussian distribution for simplicity, but note that there is no obstacle to generalizing the model by inserting a DNN non-linearity that outputs a correlation matrix; one advantage of employing a simpler (although less expressive) scalar variance rather than parameterizing it with a DNN is that M-step update rules can be derived analytically in closed form. Similarly, the training algorithm that we are about to introduce could easily be generalized to different noise distributions than Gaussian: indeed derivations for Poisson noise can be found in App. D and preliminary experiments with Bernoulli noise can be found in App. E.

For the purpose of this work, however, we will mainly focus on the most elementary VAE models, with the form shown in Eq. 5.10. This generative

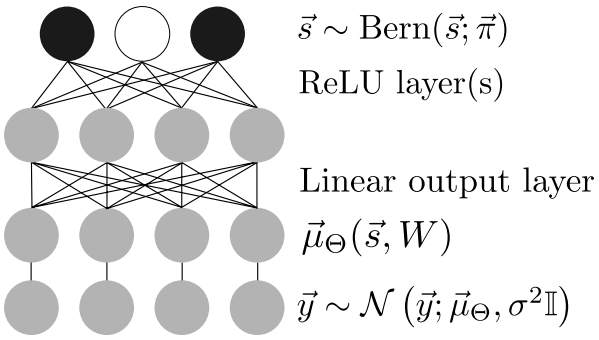

Figure 5.3: Graphical representation of the model architecture used in numerical experiments.

model has Bernoulli- rather than Gaussian-distributed latents but is in all other aspects identical to a vanilla VAE (see Fig. 5.1 for a schematic comparison). Fig. 5.3 is a graphical representation of this generative model, and includes the exact architecture of the DNN that we employed in our numerical experiments (see Sec. 5.4).

5.3.1 Decoder optimization

As discussed above, instead of using reparameterization or variance reduction, we will compute gradients based on the truncated posteriors introduced in Ch 3, that we report here for convenience:

$$q_\Phi^n(\vec{s}; \vec{y}^n) := \frac{p_\Theta(\vec{s} \,|\, \vec{y}^n)}{\sum\limits_{\vec{s}' \in \Phi^n} p_\Theta(\vec{s}' \,|\, \vec{y}^n)} \, \delta(\vec{s} \in \Phi^n) \qquad \text{(3.2 revisited)}$$

After substitution of the binary VAE model of Eqs. 5.1 and 5.2, we can compute the gradient of the ELBO of Eq. 2.14 w.r.t. the decoder weights W. Noting that the KL divergence term does not depend on W, this results in:

$$\begin{aligned} \vec{\nabla}_W F(\Phi, \Theta) &= \sum_n \langle \vec{\nabla}_W \log\left(\mathcal{N}(\vec{y}^n; \vec{\mu}_\Theta(\vec{s}, W), \sigma^2 \mathbb{I})\right) \rangle_{q_\Phi^n} \\ &= -\frac{1}{2\sigma^2} \sum_n \sum_{\vec{s} \in \Phi^n} q_\Phi^n(\vec{s}) \; \vec{\nabla}_W \|\vec{y}^n - \vec{\mu}_\Theta(\vec{s}, W)\|^2 \,. \end{aligned} \tag{5.11}$$

Notably, no sampling approximation and no reparameterization trick are necessary to obtain this result, and differently from standard VAE training this approach is not amortized.

The right-hand-side has salient similarities to the gradient of VAE decoders, Eq. 5.5: for example, the familiar gradient of the mean squared error (MSE), which shows that standard automatic differentiation tools can still be applied. However, the decisive difference resides in the weighting factors $q_\Phi^n(\vec{s})$:

instead of samples from a standard encoder, here the gradients are estimated using the members of sets Φ^n. The size of Φ^n can consequently be thought of as analogous to the number of samples used in a conventional estimation of the gradient. Standard VAE training estimates the gradient by weighting all samples equally and the gradient direction is approximated by drawing sufficiently many samples from the encoding model. In contrast, truncated gradient estimation uses the states in Φ^n, and the gradient is computed using a weighted summation with weights $q^n_\Phi(\vec{s})$. The gradient is then, notably, not a stochastic estimation but exact, and, for small enough gradient step sizes, the procedure is guaranteed to monotonically increase the variational lower bound.

To complete the decoder optimization, update equations for variance σ^2 and prior parameters $\vec{\pi}$ can be computed in closed-form (compare, e.g., Shelton et al., 2011) and are given by:

$$\begin{aligned} \vec{\pi}^{\text{new}} &= \tfrac{1}{N}\sum_n \sum_{\vec{s}\in\Phi^n} q^n_\Phi(\vec{s})\ \vec{s} \\ \sigma^{2,\text{new}} &= \tfrac{1}{DN}\sum_n \sum_{\vec{s}\in\Phi^n} q^n_\Phi(\vec{s})\ \|\vec{y}^n - \vec{\mu}_\Theta(\vec{s}, W)\|^2 \,. \end{aligned} \tag{5.12}$$

where N is the number of samples in the training dataset and D is the number of observables, that is, the dimensionality of $\vec{y}$.

5.3.2 Encoder optimization

The variational parameters of the encoding model, i.e., the sets of latent states Φ^n that define the truncated posteriors q^n_Φ, are optimized with Evolutionary Expectation-Maximization: at each epoch, for each datapoint $\vec{y}^n$, we seek latent states $\vec{s}$ with higher fitness, i.e., higher log-pseudo-joint distributions (see Sec. 4.2.1 for EEM's usage of the fitness function, and Sec. 3.3.3 for the definition of the log-pseudo-joints).

Let us take a closer look at the specific form of the log-pseudo-joints for the binary VAE model of Eq. 5.10:

$$\widetilde{\log p_\Theta}(\vec{y}, \vec{s}) = -\|\vec{y} - \vec{\mu}_\Theta(\vec{s}, W)\|^2 \ - \ 2\,\sigma^2 \sum_h \tilde{\pi}_h\, s_h \tag{5.13}$$

where $\tilde{\pi}_h = \log\big((1-\pi_h)/\pi_h\big)$. The expression assumes an even more familiar form, if we restrict ourselves for a moment to sparse priors $\pi_h = \pi < \frac{1}{2}$, i.e., $\tilde{\pi}_h = \tilde{\pi} > 0$. The pairwise comparison criterion for the update of the Φ^n sets in the TVEM algorithm, Eq. 3.7, then takes this form:

$$\|\vec{y}^n - \vec{\mu}_\Theta(\vec{s}^{\,\text{new}}, W)\|^2 + 2\,\sigma^2\tilde{\pi}\ |\vec{s}^{\,\text{new}}| \ \ < \ \ \|\vec{y}^n - \vec{\mu}_\Theta(\vec{s}^{\,\text{old}}, W)\|^2 + 2\,\sigma^2\tilde{\pi}\ |\vec{s}^{\,\text{old}}| \tag{5.14}$$

where $|\vec{s}| = \sum_{h=1}^H s_h$. Similar expressions are routinely encountered in sparse coding applications: for each set Φ^n, we seek those states $\vec{s}$ that are reconstructing $\vec{y}^n$ well while being sparse ($\vec{s}$ with few non-zero bits). For VAEs,

$\vec{\mu}_\Theta(\vec{s}^{\,\text{new}}, W)$ is a DNN and as such much more flexible in matching the distribution of observables $\vec{y}$ that it may be expected from linear mappings. Furthermore, criteria like Eq. 5.14 usually emerge for maximum a-posteriori (MAP) approximations of sparse coding. In contrast, here we seek a *population* of states $\vec{s}$ in Φ^n for each data point rather than a single MAP state. Similarly to what happens when increasing the number of samples in a Monte Carlo approximation of an expectation value, the sets Φ^n thus allow for capturing a richer posterior structure than a single MAP state.

We will refer to the generative model of Eq. 5.10, trained with TVEM, as *Truncated Variational Autoencoders* (TVAEs). The full training procedure for TVAEs is summarized in Alg. 3.

Algorithm 3: Training Truncated Variational Autoencoders

```
Initialize model parameters Θ = {W, π⃗, σ²}
Initialize each Φ^n with S distinct latent states
repeat
    for each batch in dataset do
        for each sample n in batch do
            Φ^new = Φ^n
            for each generation do
                Φ^new = mutation (crossover (selection (Φ^new)))
                Φ^n = Φ^n ∪ Φ^new
            Truncate Φ^n to the S fittest elements based on Eq. 5.14
        Use Adam to update W using Eq. 5.11
    Use Eq. 5.12 to update π⃗, σ²
until parameters Θ have sufficiently converged
```

5.3.3 Computational requirements and theoretical scaling

In this section we would like to discuss the computational complexity of TVAE training; in particular, it is necessary to verify that TVAE does not introduce undesirable scaling behavior with respect to problem parameters when compared to standard VAE training. To this end, consider again the expressions of the gradients for standard VAEs and TVAEs, respectively corresponding to Eqs. 5.5 and 5.11. If, in standard VAEs, as many encoder samples L are used, per data point, as there are states in each Φ^n, then the two summations $\sum_{\vec{s}\sim q_\Phi^n}$ and $\sum_{\vec{s}\in\Phi^n}$ have the same number of summands. The evaluation of the gradients of the mean square error (MSE) consequently involves precisely the same computational load for both approaches.

The main computational difference then lies in the updates of Φ^n using evolutionary algorithms, to be compared to the update of encoder DNNs for

conventional VAEs. Once the parameters Θ are updated using Eq. 5.11, new states for Φ^n have to be sought based on the criterion Eq. 5.14. In practice, for each datapoint $\vec{y}^n$, we generate S' new states according to the applied evolutionary procedure. To select the best states we have to pass all the S' proposal states through the decoder DNN to evaluate Eq. 5.14. Furthermore, we also have to pass all S states already in Φ^n through the decoder to re-evaluate their fitness after parameters Θ have been updated. In summary, TVAE requires $\mathcal{O}(N \times (S + S'))$ passes through the decoder DNN per training epoch. As discussed in Sec. 3.3.1, selecting the S best states from the $(S + S')$ states does not add complexity as this can be done in $\mathcal{O}(S + S')$ for each n (Blum et al., 1973). In summary, we do expect the evolutionary procedure to have a visible impact in terms of computational load, but parent selection and mutation only add a constant offset for each of the considered states and do not increase algorithmic complexity.

On the other hand, in standard VAE training, if the encoder uses L samples, $\mathcal{O}(N \times L)$ passes through the decoder DNN are required to update the parameters Θ according to Eq. 5.5. For the encoder update, one requires $N \times L$ passes through encoder and decoder DNN to estimate the gradient w.r.t. the encoder weights. The additional overhead to actually draw the samples is usually negligible.

In summary, overall algorithmic complexity is comparable with the one of standard VAE training if $S \approx S' \approx L$. However, conventional VAE training is amortized, i.e., the update of encoder weights uses information from all data points n. As discussed in Sec. 3.3.4, amortization can be a double-edged sword, but it does mean that TVAE has higher memory requirements. Not only, but the number of samples L for conventional VAE training is usually smaller than best working sizes of Φ^n: for example, in the denoising experiments described in Sec. 5.4.5, we used sizes up to $|\Phi^n| = 200$; in contrast, even work that is dedicated to increasing VAE performance by making better use of multi-sample estimators such as Burda et al. (2015) does not make use of more than 50.

5.4 Numerical experiments

Truncated Variational Autoencoders significantly divert from any training procedure for binary VAEs that has been previously investigated. Indeed, to the best of our knowledge, it is the first example of a VAE that is trained without using a sampling approximation of the variational lower bound. A natural starting point is, consequently, the investigation of model architectures that are as elementary as possible. For this purpose, we used the binary VAE of Eq. 5.10 and a decoder DNN with the architecture shown in Fig. 5.3: a single middle layer with ReLU activations (Glorot et al., 2011) and an output layer with identity activation. We were then interested in addressing the following

two questions:

- How well does this direct discrete optimization of binary VAEs scale when dataset dimensionality and latent space dimensionality increase?
- Does the decoder's DNN non-linearity result in improved performance with respect to linear generative models?

Using relatively simple neural network architectures helps attributing benchmark performance to the capabilities of the training procedure itself rather than to the expressiveness of complex DNNs. Nevertheless, even the basic VAE decoder employed here can be benchmarked against recent state-of-the-art DNNs. For this purpose, as already discussed in Sec. 4.3.4, a benchmark which attracts recent attention is denoising in absence of clean data (the "zero-shot" setting). DNN-based approaches applicable to this task have received increasing attention: recent work (e.g. Lehtinen et al., 2018; Krull et al., 2019) is based on standard feed-forward DNNs whose training objectives have been altered to allow for training on noisy images. On the other hand, deep generative models are more naturally suited to training on noisy data as their learning objective models noise explicitly and can be directly be applied. Therefore, an interesting third research question that we would like to address is:

- Can TVAEs compete with state-of-the-art DNNs in the "zero-shot" denoising benchmark?

For the experiments that follow, training of the TVAE's DNN is performed with mini-batches, the Adam optimizer (Kingma and Ba, 2014) and decaying or cyclical learning rate scheduling (Smith, 2017). Xavier/Glorot initialization (Glorot and Bengio, 2010) is used for the DNN weights, while biases are always zero-initialized. Parameters $\vec{\pi}$ and σ^2 are updated via Eq. 5.12 and initialized to $\frac{1}{H}$ (H is the size of $\vec{\pi}$) and 0.01 respectively. Hyperparameter optimization was conducted manually and, for the more complex datasets, it also made use of black box Bayesian optimization based on Gaussian Processes (Nogueira, 2019). For what concerns the evolutionary optimization of EEM, for TVAE we made a relatively simple choice of genetic operators: fitness-proportional parent selection and random uniform one-bit mutations, with no crossover; we tried different combinations of evolutionary algorithms finding this setting to trade off computational cost and performance best for our purposes.

5.4.1 Bars test

Analogously to what was done for the simpler Noisy-OR and BSC models in Ch. 4, first of all we evaluated TVAE on bars datasets with known ground-truth parameters and log-likelihood, in order to verify the correct functioning of the algorithm and to investigate possible local optima effects. The dataset consisted of 500 4x4 images generated by linear superposition of vertical and

horizontal bars, with a small amount of Gaussian noise. The DNN's input and middle layers had 8 units each. The $\Phi^{(n)}$ variational sets consisted of 64 hidden states each. Fig. 5.4 shows the evolution of the run that achieved the highest ELBO value out of ten. All parameters were correctly recovered, and the ELBO value was consistent with actual ground-truth log-likelihood.

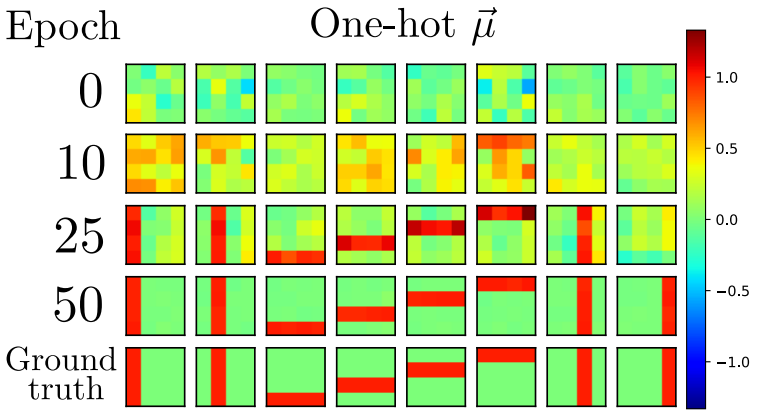

Figure 5.4: TVAE training on simple bars data: noiseless output of the TVAE's DNN for the 8 possible one-hot input vectors over several training epochs. Generating parameters are in the last row.

5.4.2 Correlated bars test

As we have seen in Ch. 4, however, such a simple test can also be solved by linear models such as Noisy-OR and BSC. In order to demonstrate that TVAEs can solve non-linear problems, taking advantage of the neural network non-linearity embedded in the generative model, we introduced correlations between pairs of bars: the bars combinations shown in the first two datapoints from the left in Fig. 5.5 were discouraged from appearing together. We employed the same evolutionary scheme and again we selected the run with highest peak ELBO value out of ten. The model correctly learns that certain combinations of bars are much more unlikely than others, and correctly estimates their likelihood.

Fig. 5.6 offers some more insight into this particular experiment. The left section of the figure shows the generative parameters for the dataset used: W_0 is the 8x8 weight matrix of the top-to-middle layer: this makes it so that the activation of the first latent variable inhibits activation of the second, and activation of the last latent variable inhibits activation of the first. Concretely, this results in a dataset where these specific bars combinations are discouraged from appearing. The weights W_1, visualized as 8 4x4 matrices, generate the actual bars. σ^2 was set to 0.01 and the dataset contained an average of two

superimposing bars per datapoint ($\pi_h = 2/8$ for each h). The middle section of the figure shows the ELBO values (averages over all batches for each epoch) as training progresses. The cyclical learning rate schedule is responsible for the oscillatory behavior. The right section shows some example datapoints together with samples from the trained TVAE model that reached the highest ELBO value out of the ten runs. Although maybe of little statistical significance, it can be observed that the combinations of bars with low log-likelihood (the two datapoints on the left of Fig. 5.5) indeed do not appear. Notably, a linear model such as BSC cannot, by construction, solve the correlated bars test, as it cannot model the negative correlations between generative components.

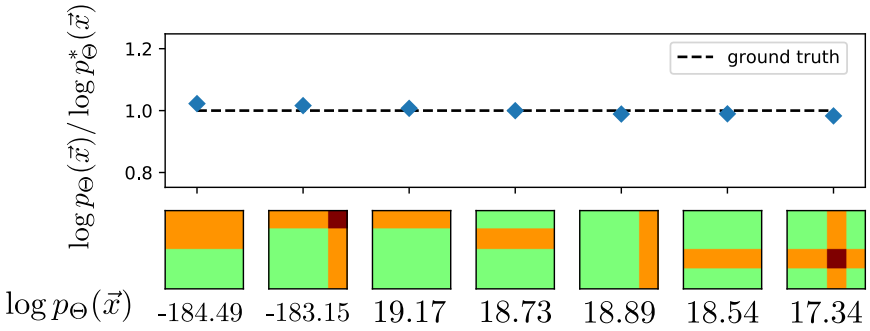

Figure 5.5: Correlated bars test. The plot shows the ratio between inferred free energy $\log p_\Theta(\vec{x})$ and ground-truth log-likelihoods $\log p^*_\Theta(\vec{x})$ of datapoints with interesting bar combinations. The inferred values are reported below the datapoints themselves.

5.4.3 A note on local optima

Deep generative models such as VAEs and Generative Adversarial Networks (Goodfellow et al., 2014) are known to be susceptible to local optima problems and posterior collapse. The bars test in particular features certain local optima that models often find difficult to overcome, in which superpositions of multiple bars rather than single bars are recovered as generative fields. As it often happens, a small amount of training noise can help the training procedure overcome such local optima. Alg. 3 inherently includes a few sources of noise, notably the variational approximation itself and the batch-wise model weights updates typical of stochastic gradient ascent. In our experiments, however, we have found that a cyclical learning rate schedule (Smith, 2017) also helps. Fig. 5.7 shows the free energy values reached by TVAE in several training configurations: on the left, we train a shallow TVAE model with no middle layer in the network, on the right we train the full architecture as depicted in Fig. 5.3. "Full EM" in the picture refers to the Expectation-Maximization

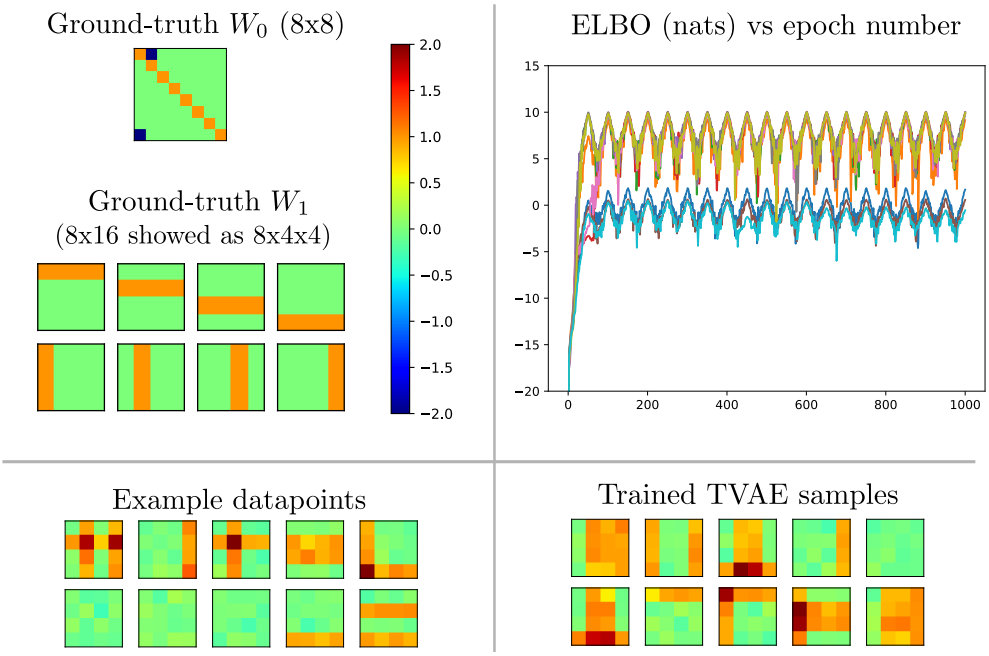

Figure 5.6: **Top left**: generative parameters for the correlated bars test; **Top right**: ELBO values over epochs for 10 runs; **Bottom left**: example datapoints; **Bottom right**: samples from the trained generative model.

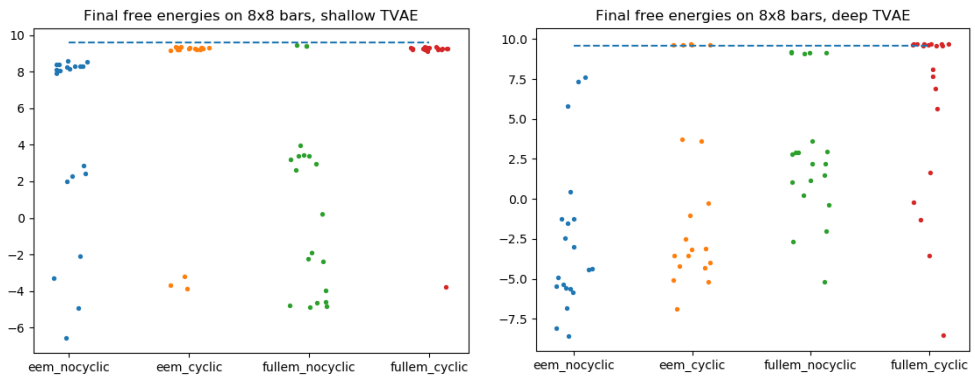

Figure 5.7: Highest free energy reached for several 8x8 bars test experiments, in different conditions. Each dot represents a different run of the experiment. **Left**: a shallow TVAE model with no middle layer is trained. **Right**: a TVAE model with a middle layer in the neural network is trained. The dotted lines indicate the ground-truth log-likelihood value. Cyclical learning rates help reaching higher free energy values more often.

algorithm with full E-steps and full M-steps. It can be observed that the addition of a cyclic learning rate schedule helps reaching the ground-truth

log-likelihood values more often in all training configurations.

5.4.4 Natural image patches

The test on correlated bars showed that TVAE can take advantage of the DNN's non-linearity, but it is still to be seen whether that corresponds to better approximate log-likelihoods on natural data. In order to answer this question we ran a direct comparison with BSC: in this regard, note that, interestingly, another way to interpret the binary VAE model of Eq. 5.10 is as a BSC model with non-linear superposition of generative components rather than the standard linear superposition; conversely, the TVAE model contains BSC as the edge case in which the hidden layers of the TVAE's neural network compute the identity function. This property makes BSC a well-suited candidate to test whether TVAEs can take advantage of the DNN non-linearity.

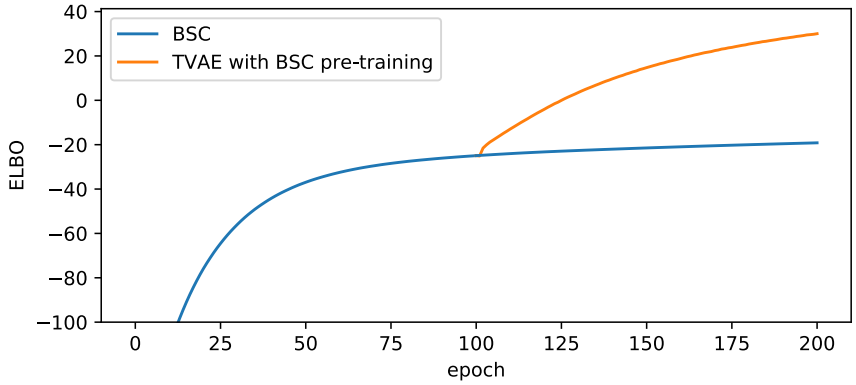

Figure 5.8: ELBO gain of TVAE compared to BSC on image patches.

We used the same dataset as for the BSC scaling tests of Sec.4.3.3. First we trained a BSC model with H=300 latent variables, with the same EEM algorithm as TVAE. After 100 epochs we copied the weights of the BSC model to the bottom layer of a TVAE model with three layers of 300, 300 and 16x16=256 units each. The upper layer weights were initialized to the identity matrix. We also retained the learned $\Phi^{(n)}$ sets as well as priors and variance. This BSC pre-training for TVAE guarantees a common starting point for the two models for the next 100 epochs of training. Fig. 5.8 shows the ELBO trajectories of BSC and TVAE for a typical such experiment. TVAE training quickly optimizes the ELBO to higher values, exploiting the extra flexibility of the neural network non-linearities.

While initializing TVAE weights with BSC can highlight the benefit of the DNN non-linearity, we observed this BSC pre-training to not be necessary in general. We also ran larger experiments in which we trained TVAE from scratch on the same dataset, scaling to H=1000 latent variables and 100 units

in the network middle layer, with further increases in ELBO values up to a value of 84.9 nats.

Fig. 5.9 shows some sample datapoints and samples from the trained generative model for the experiment on natural image patches.

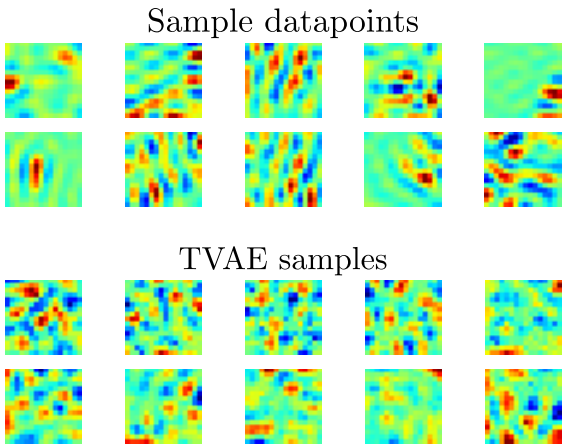

Figure 5.9: Example datapoints and generative model samples for the experiment on natural image patches. The colormap of each patch is normalized so that zero values sit in the middle of the scale (green color).

5.4.5 Denoising the "house" image

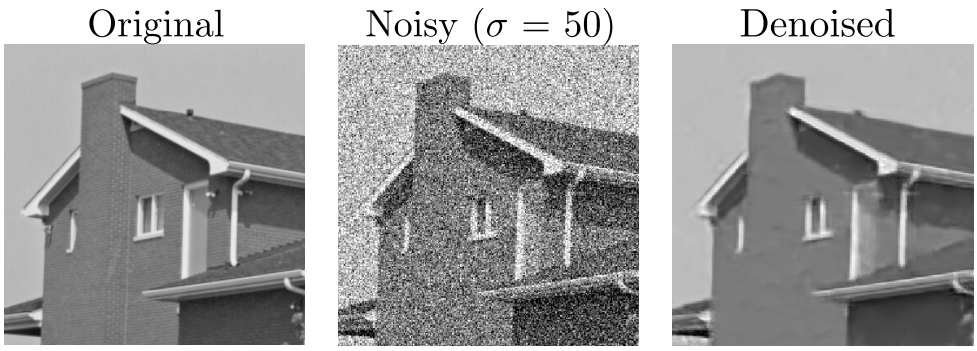

Figure 5.10: Denoising application of TVAE on house image with noise level $\sigma = 50$. The denoised image has PSNR=30.03, the best of the three runs of Tab. 5.2.

Having established scalability of the TVAE model to large latent spaces, and verified that it can in fact provide performance higher than shallow models such as BSC, we now focus on the third research question we posed at the

beginning of this section: can TVAEs compete with state-of-the-art DNNs in the "zero-shot" denoising benchmark?

In the context of denoising, the "house" image (Fig. 5.10 left), already introduced in Sec. 4.3.4, offers the broadest possible comparison to other methods. The standard benchmark setting for this image employs additive Gaussian white noise with standard deviations $\sigma \in \{15, 25, 50\}$. Fig. 5.10 (center) shows the corrupted image with noise level $\sigma = 50$.

Given a noise value σ, we trained a TVAE on square patches extracted from the noisy image. Subsequently, we used the trained model to estimate the most likely value of each of the image pixels. We apply the same denoising algorithm as described in Sec. 4.3.4 for BSC, with the following estimator:

$$y_d^{\text{est}} = \langle \mu_\Theta^d(\vec{s}) \rangle_{p_\Theta(\vec{s}|\vec{y})} \approx \langle \mu_\Theta^d(\vec{s}) \rangle_{q_\Phi(\vec{s};\vec{y})} \,. \tag{5.15}$$

Importantly, this benchmark enables direct comparison to other variationally optimized generative models including MTMKL (Titsias and Lázaro-Gredilla, 2011), GSC (Sheikh et al., 2014) and S5C (Sheikh and Lücke, 2016), which all showed state-of-the-art performance for probabilistic sparse coding when they were first published. As a first experiment, we compare TVAE to these other models in controlled conditions: all models benchmarked in Tab. 5.1 used the very same patch size of $D = 8 \times 8$ pixels and $H = 64$ latent variables. The table reports mean standard deviation of the PSNR achieved by TVAE across three runs with three different noise realizations. Values for MTMKL and GSC were taken from the respective original publications. As can be observed, TVAE performs significantly better than the other methods for high noise levels. TVAE is able to learn the best data representation for denoising and represents the state-of-the-art in this controlled setting. According to Tab. 5.1, two factors enable the good performance of TVAE compared to the other approaches: firstly, the evolutionary optimization training algorithm itself seems to be beneficial as a comparison of BSC to MTMKL and GSC suggests; secondly, denoising performance of TVAE is significantly better than BSC (1dB for this establish benchmark represents a major improvement), which implies that the decoder DNN of TVAE provides the decisive performance advantage.

For $\sigma = 25$ and $\sigma = 50$, TVAE also significantly improves on MTMKL and GSC results. All these three approaches are based on a spike-and-slab sparse coding model (also compare Goodfellow et al., 2012). The decoder DNN of TVAE more than compensates for its comparably less flexible Bernoulli prior, and results in the highest PSNR values for high noise.

In order to further extend our comparison, in the next experiment we considered this denoising task without controlled conditions. Concretely, we allow for any approach that performs denoising on the benchmark including approaches that are trained on large image datasets and/or use different patch sizes (including multi-scale and whole image processing). Note that dif-

Table 5.1: Denoising performance in PSNR (dB) for the house image under controlled conditions (D=8×8, H=64 for all algorithms).

	σ=15	σ=25	σ=50
MTMKL	**34.29**	31.88	28.08
GSC	32.68	31.10	28.02
BSC	32.25	31.15	28.62
TVAE	34.27 ± 0.02	**32.65 ± 0.06**	**29.61 ± 0.02**

ferent methods will involve very different sets of hyperparameters that can be optimized for denoising performance: for sparse coding approaches, hyperparameters include patch and dictionary sizes; for DNN approaches they include all network and training scheme hyperparameters. By allowing for comparison in this less controlled setting, we can include a number of recent approaches including DNNs trained on clean data as well as DNN training approaches dedicated to noisy training data. Tab. 5.2 shows the denoising performance for the three noise levels we investigated, with results for other algorithms taken from their corresponding original publications with the exceptions of WNNM and EPLL for which we cite values from Zhang et al. (2017). These other methods include a deterministic sparse coding baseline (KSVD, Aharon et al., 2006), a mixture model approach (EPLL, Zoran and Weiss, 2011), a non-local image processing method (WNNM, Gu et al., 2014) and state-of-the-art denoising methods based on deep neural networks (BDGAN, Zhu et al. (2019) and DPDNN, Dong et al. (2019)). These approaches can be distinguished, e.g., by the amount of employed training data and by the requirement for clean data. Note that the best performing approaches (lower half of the table) cannot be trained on just the noisy data, while the algorithms in the upper half of the table can.

Our novel TVAE model achieves new state-of-the art PSNR values in the "zero-shot" setting for the "house" image denoising benchmark and high noise levels. Performance is inferior, although competitive, with respect to algorithms that leverage large datasets of clean images and complex neural networks.

In Tab. 5.3 we report the exact hyperparameters used to obtain the PSNR values discussed above. In parentheses, the parameters for the run on data with noise level $\sigma = 50$ and unconstrained hyperparameters are given, when they differ from the other experiments.

Table 5.2: Denoising performance in PSNR (dB) for the house image for different algorithms with different optimized hyperparameters. The methods in the **top** section of the table only require the noisy data itself, the ones in the **middle** require the noise level, while methods in the **bottom** section cannot be trained on just noisy data and require (typically large) datasets of clean images.

	σ=15	σ=25	σ=50
MTMKL	**34.29**	31.88	28.08
GSC	33.78	32.01	28.35
S5C	33.50	32.08	28.35
VAR-BSC	33.50	32.32	28.91
TVAE	34.27 ± .02	**32.65 ± .06**	**29.98 ± .05**
KSVD	34.32	32.15	27.95
WNNM	35.13	33.22	30.33
BM3D	34.94	32.86	29.37
EPLL‡	34.17	32.17	29.12
BDGAN	34.57	33.28	30.61
DPDNN	**35.40**	**33.54**	**31.04**

Table 5.3: Hyperparameters for the denoising experiments on the house image.

Neural network units	
Input (H)	64 (512)
Middle	64 (512)
Output (D)	64 (144)
Cyclic Learning Rates	
Min l.r.	0.0001
Max l.r.	0.01 (0.05)
Epochs/cycle	20
Batch size	32
Evolutionary parameters	
Parents	10 (5)
Children	9 (4)
Generations	4 (1)
Size of $\Phi^{(n)}$	200 (64)

5.4.6 Performance comparison with noise2void

TVAE does not require clean images for training, and can be trained on just a single noisy image. Instead, EPLL, BDGAN and DPDNN require clean training data (typically tens or hundreds of thousands of samples are used in training). Approaches such as noise2noise (n2n, Lehtinen et al., 2018) and noise2void (n2v, Krull et al., 2019) occupy a middle ground: they can be trained on noisy data but they typically require much larger amounts of data than, e.g., TVAE or MTMKL. In the original n2v publication, for instance, 400 (noisy) 180×180 images from the BSD dataset (Martin et al., 2001) were used to create a training dataset (this procedure also involved data augmentation, see Krull et al. (2019)).

In Tab. 5.4 we report the denoising performance we obtained by applying n2v to the "house" benchmark. We made use of the official, publicly available code for n2v. After training on the default dataset ($\sigma = 25$) employed in the original n2v publication, we applied the n2v network to denoise the "house" image with $\sigma = 25$. The resulting PSNR value was 32.10 dB which is 0.76 dB lower than the PSNR value for BM3D (32.86 dB). The difference is consistent with an average 0.88 dB lower performance of n2v compared to BM3D on the BSD68 test set (see Krull et al., 2019). The same network can also be used to denoise an image with lower or higher noise level. The entry marked $n2v^{\dagger}$ in Tab. 5.4 reports PSNR values for the n2v network trained on $\sigma = 25$ and then applied to a noisy image with $\sigma = 15$ or $\sigma = 50$. Especially for high noise levels, performance can be much improved, however, if the n2v network is trained using images with the same noise level as the test image. In order to do so, we followed the procedure described in the n2v publication, adapting the training noise level to the one of the test image. The resulting PSNRs are listed as $\text{n2v}^{\ddagger}$ in Tab. 5.4. The PSNR values obtained in this matched-noise-level scenario are much higher compared to the scenario with unmatched noise level (e.g., for $\sigma = 50$ the PSNR improvement is approximately 8 dB). The much lower performance for mismatched noise for n2v is in this respect consistent with observations for standard DNN denoising for which training with the ground-truth noise level has been pointed out as important for performance (Chaudhury and Roy, 2017; Zhang et al., 2018a).

The n2v model can also be trained on the single noisy image, without a-priori knowledge of the noise level. We investigated this "zero-shot" denoising feature of n2v and applied the algorithm to denoise the "house" image while using the same noisy image for training that we seek to denoise. We took the publicly available code of n2v as an example and manually adjusted hyperparameters as follows: we set the "Percentage of pixel to manipulate per patch" to a value of 0.4, "Number of training epochs" to 400 and "Number of parameter update steps per epoch" to 33. The obtained PSNR values are listed as n2v^{*} in Tab. 5.4.

Table 5.4: Denoising performance of n2v in PSNR (dB) for the "house" image with additive white Gaussian (AWG) noise. For comparison, we also list the performance of TVAE (numbers copied from Tab. 5.2). PSNR values for n2v* are obtained by training only on the noisy image. n2v† reports performance when additional training data in the form of noisy images with AWG noise $\sigma = 25$ is used. n2v‡ corresponds to the performance obtained for the n2v trained on data with noise level that matches the noise of the test image. See the main text for further details.

	σ=15	σ=25	σ=50
n2v*	32.05	29.20	25.42
n2v†	32.93	32.10	20.96
n2v‡	33.91	32.10	28.94
TVAE	**34.27 ± 0.02**	**32.65 ± 0.06**	**29.98 ± 0.05**

Note that for all considered training settings of n2v and all noise levels, PSNR values of TVAE are consistently higher. This is true even if n2v is trained on external data with matched-noise level. Additional parameter tuning may improve performance of n2v* to a certain extent but PSNRs are in general much lower than n2v‡. Although for n2v‡ we followed the standard hyperparameter setting of the original publication (Krull et al., 2019), we cannot exclude that further improvements could be obtained by fine-tuning training parameters. However, we remark that the difference of n2v‡ and BM3D for the "house" benchmark is comparable to the differences between n2v and BM3D reported (for the BSD dataset) in the original n2v publication. However, it must be noted that, once trained, n2v is typically faster to apply to new noisy images than BM3D as well as TVAE.

For what concerns noise2noise (n2n), its PSNR values are usually very closely aligned with those achievable by feed-forward DNNs: for example, n2n uses a RED30 network (Mao et al., 2016) which achieves 31.07 dB PSNR on the BSD300 data set if trained on clean data. If directly trained on noisy data, RED30 achieves 31.06 dB (Lehtinen et al., 2018). n2n is therefore strongly performing in terms of PSNR. The caveat of n2n compared to n2v is, however, that the noisy data n2n uses is rather artificial. The pairs of images n2n is trained on consist of two different noise realization of the very same underlying clean image. For real data, such a setting is only approximately occurring at most, which has motivated the n2v approach.

For the "zero-shot" setting, TVAE is consequently the best performing system on the "house" benchmark. Notably, such a high performance is achieved using a basic DNN and relatively small patch sizes of $D = 8 \times 8$ (for $\sigma = 15$ and $\sigma = 25$) or $D = 12 \times 12$ (for $\sigma = 50$). All feed-forward DNNs for denoising

use much larger patches (e.g., n2v use 64×64). That a competitive denoising performance can be achieved with small patch sizes argues in favor of VAE approaches to denoising. Indeed, TVAE even comes close to state-of-the-art approaches (BDGAN and DPDNN) that use very intricate DNN architectures and large amounts of clean training data. We believe that such results underline the potential of the approach investigated here, especially considering its novelty: further improvements to the decoder DNN as well as to the evolutionary optimization of the variational distributions can potentially yield further gains in performance.

5.5 Computational performance

An important limitation of TVAE with respect to standard DNN-based methods is its computational demand. For our experiments on the "house" image with noise level $\sigma = 50$ in Tab. 2 we used $N = 60025$ patches of $D = 12 \times 12$ pixels, which amounts to all possible non-overlapping square patches of that size that can be extracted from the image. For training and denoising we used a TVAE with $H = 512$ latent variables, sizes of $|\Phi^n| = 64$, and 512 units in the DNN middle layer of the decoder. TVAE training required 49 seconds per training epoch when executing on a single NVIDIA Titan Xp GPU and 2.5 GB of GPU memory. We ran for 500 epochs which required between seven and eight hours on the single GPU. We did not observe significant changes in variational bound values or in denoising performance after 500 epochs in any of the experiments we conducted for Tabs 1 and 2. Runtime complexity increased linearly with the number of data points N, with the dimensionality of the data D, with the number of the latents H, and with the size of the DNN used. Empirically, we observed a sub-linear scaling with $|\Phi^n|$: for example, increasing from $|\Phi^n| = 64$ to $|\Phi^n| = 128$ (while keeping all other parameters as above) computational time increases from 49 seconds per training epoch to 75 seconds; presumably, this is due to a significant constant overhead in computing time w.r.t. the size of Φ^n.

For noise levels $\sigma = 15$ and $\sigma = 25$ in Tab. 2 we used smaller patch sizes ($D = 8 \times 8$) and fewer stochastic latents ($H = 64$) but larger Φ^n (i.e., $|\Phi^n| = 200$). In general, if the patch size D is increased, more structure has to be captured. This can be done either by increasing the size of the stochastic latents H or by using larger DNNs. Both, in turn, requires more training data in order to estimate the increased number of parameters. In the current setup, the sizes of D which are currently feasible are comparably small. The denoising performance based on small patches is, however, notably very high.

For comparison, n2v uses up to $D = 64 \times 64$ and also all other feed-forward DNN approaches use significantly larger patch sizes than TVAE. Still, n2v can be trained efficiently on large patches requiring approximately 19 hours on a NVIDIA Tesla K80 GPU for training on approximately 3k noisy images of

shape 180x180 and mere seconds for the denoising of one 256x256 image. The higher computational demand of TVAE is also the reason why averaging across databases with many images (such as BSD68) or applications to large single images quickly becomes infeasible. As a novel approach, TVAE is, however, far from being fully optimized algorithmically compared to large feed-forward approaches, and there is certainly further potential to improve training efficiency.

Chapter 6

A Software Framework for Truncated Variational EM

In a research field such as machine learning, in which theory is deeply intertwined with computing, powerful and user-friendly software libraries can greatly encourage the adoption of new algorithms or paradigms. For deep learning applications, this is nicely demonstrated by Theano (Theano Development Team, 2016): at the time it was published, its versatile engine for symbolic differentiation of mathematical expressions involving multi-dimensional arrays lowered the bar for the development and investigation of new models immensely. Ergonomic software libaries are just as important as effective algorithms, both for rapid prototyping and research work as well as to foster widespread adoption of new techniques.

On top of the promising scientific results described in previous chapters, another notable product of my doctoral work has been a ready-to-use, well-tested and extensible Python framework for Truncated Variational Expectation Maximization. The software targets all Linux platforms and it has been used for the majority of the experiments described in Chapters 4 and 5. In particular, it has enabled the state-of-the-art denoising results described in the previous chapter. It will soon be made publicly available at `https://gitlab.com/mloldenburg/tvem`.

I designed and implemented the framework in collaboration with Jakob Drefs. My most notable personal contributions to the project were the protocol-based design of model interfaces, efficient GPU-friendly implementations of EEM's genetic operators, black-box training functionality based on auto-differentiation, and the implementations of the Noisy-OR model and of Truncated Variational Autoencoders (TVAEs). The implementation of BSC's data reconstruction algorithms was contributed by Jakob Drefs. I later added support for TVAE's data reconstruction. I also set up most of the framework's "DevOps" infrastructure, including continuous integration and deployment, testing, packaging

and automatic documentation generation.

6.1 Why a new framework?

The motivation for developing a full-fledged TVEM software framework is twofold: firstly, we want to facilitate adoption of and experimentation with the TVEM algorithm on the part of other researchers, within our same research group as well as other groups that are interested in our novel training methods; secondly, we wish to ensure the reproducibility of our experimental results and provide a robust starting point for further improvements. These reasons alone might not be seen as sufficient motivation to create a new software package rather than integrating our methods with existing frameworks. Indeed, the availability of production-grade machine learning software has dramatically grown in recent years: mostly thanks to the fruitful marriage of deep learning techniques, auto-differentiation libraries and more powerful commodity hardware (namely GPUs), software frameworks which provide industry as well as academia with out-of-the-box solutions for the training and inference of discriminative models have flourished in number and amount of contributors (see e.g. Nguyen et al., 2019, for a recent review). Frameworks that natively support probabilistic generative models or, more in general, probabilistic programming, also exist: recently, Edward (Tran et al., 2016) and Pyro (Bingham et al., 2019) have been gaining popularity. However, due to the fundamentally different nature of TVEM's variational parameters (sets of discrete latent states rather than continuous parameters of a given variational distribution) integration of TVEM training with such frameworks is non-trivial. Furthermore, both Edward and Pyro require users to learn their ad-hoc probabilistic programming language, which might represent an undesirable entry barrier.

In comparison to a tool such as Pyro, our TVEM software framework is narrower in scope: it is limited to training of discrete latent probabilistic generative models with truncated posterior approximations and consequent inference applications. However, we believe it provides a simple, ready-to-use programming model that can feel natural to researchers and most (also non-expert) Python users, as well as a robust starting point for algorithmic extensions and modifications. The TVEM framework was designed with the following goals in mind:

We strived for an intuitive programming model that minimizes the lines of code required to reach the desired result. Great care has been put in making standard use cases simple and expert use cases possible by using only few, shallow abstraction layers on top of the Python programming language and PyTorch (more on this below).

The framework is extensible via flexible protocol-based interfaces: the most important extension points provided are the definition of arbitrary generative models as well as TVEM optimization schemes (e.g. EEM or

marginal sampling). Rather than basing customization on inheritance and polymorphism, the TVEM framework embraces Python's "duck typing" capabilities and allows users to implement their own custom logic as long as it fits a well-defined protocol definition (i.e., user-defined objects must expose a certain interface with pre-defined semantics). Sec. 6.3 elaborates on this point.
GPU acceleration and transparent multi-core parallelization on CPU clusters: user code requires little to no change in order to be executed as a single-threaded program or as a multi-process highly data-parallel application. Similarly, when appropriate, computation can be dispatched to GPUs simply by setting an ad-hoc environment variable.

Naturally, we avoided re-implementing standard tools, and opted instead to offload automatic differentiation as well as heterogeneous hardware support to the PyTorch library (Paszke et al., 2017), on which our framework relies heavily. Compared to other production-grade frameworks such as Tensorflow (Abadi et al., 2016) and MXNet (Chen et al., 2015), PyTorch provides the right balance between computational performance, non-opinionated and flexible APIs (other frameworks, by design, heavily target deep learning use cases at the expense of other areas) and ease of debugging. Finally, it is worth pointing out that the development of our framework follows modern software development best practices: a pipeline of continuous integration, testing and deployment is in place to validate all changes before they are deployed. Similarly, each release is automatically packaged and made available as part of the Anaconda software ecosystem to facilitate installation. The code has over 95% test coverage thanks to a mix of unit tests as well as integration tests, for a total of 450 individual test cases. Documentation is automatically generated from code annotations and it is published as browsable web pages. Extensive usage of Python 3 type annotations provides further documentation of the interfaces and is used for static validation of the code-base as well as protocol-based extensions.

The framework is fully open source and released under the Academic Free License v3.0.

6.2 Framework components and programming model

Users of the TVEM framework manipulate three types of objects: *models*, *variational state optimizers* and *experiments*. Fig. 6.1 reports a schematic representation of each component's role and interface. Listing 1 reports a full usage example of the framework that trains a BSC model on the specified input dataset.

A model object, at the very least, is expected to store the parameters of a given generative model and implement the calculation of the corresponding log-joint probabilities. The free energy of the model on a given dataset can then be evaluated using the compact truncated free energy of Eq. 3.5,

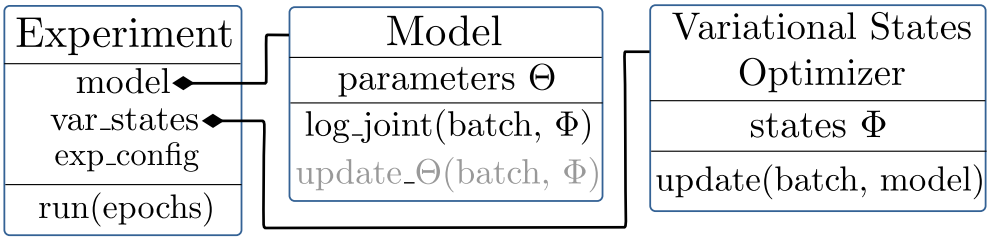

Figure 6.1: Schematic (simplified) representation of the main functional components of the TVEM framework, as UML class diagrams: type name is shown at the top of each box, data attributes in the middle section, methods at the bottom. The `update_`Θ method of `Model` is in gray because its implementation is optional: if not present, the framework falls back to gradient-based parameter updates.

which in turn can be processed by an automatic differentiation tool and yield gradient-based model parameter updates. Therefore, training of a model in the TVEM framework does not require much more than the definition of a log-joint probability. If M-step parameter update rules are available in closed form, models can expose them as an optional method (method `update_`Θ in gray in Fig. 6.1). The TVEM framework expects parameters to be stored as PyTorch tensors; input data and experiment outputs are stored in HDF5 format. The Noisy-OR, BSC and TVAE generative models are made available by the library, but users can easily integrate their own.

A variational state optimizer contains the variational parameters Φ and updates them according to a specific algorithm. The TVEM framework implements Evolutionary Expectation Maximization (as presented in Ch. 4) as well as full E-steps (i.e., "exact EM" as presented in Ch. 2) and random uniform sampling of variational states (useful as a performance baseline for other algorithms or for testing purposes).

Finally, an experiment object aggregates a model and a variational state optimizer and runs the Expectation-Maximization loop. Experiments are passed a configuration object in order to set training parameters such as batch size, optional random shuffling of the input data, training log output file or which parameters should be stored in the training log. As it is visible from listing 1, the optimization loop is run as a standard Python `for` loop so that users can insert custom logic to be executed at the end of every epoch: this is useful, for example, to implement custom logging, to validate model parameters or to implement early quitting of the optimization loop.

The framework also provides logging facilities, and a companion code repository that provides a few useful pre- and post-processing routines is available.

```
from tvem.exp import ExpConfig, EEMConfig, Training
from tvem.models import BSC

# Create a BSC model with random weight initialization
bsc = BSC(H=8, D=16)

# Configure and create a Training experiment
estep_conf = EEMConfig(n_states=16,
                       n_parents=3,
                       n_children=2,
                       n_generations=2,
                       parent_selection="fitness",
                       mutation="uniform")
exp_conf = ExpConfig(shuffle=True, output="exp_log.h5")
exp = Training(exp_conf, estep_conf, bsc,
               train_data_file="some_data.h5")

# Run 500 epochs of training
for epoch_log in exp.run(500):
    epoch_log.print()
```

Listing 1: Simple example usage of the TVEM framework. A BSC model is trained with EEM for 500 epochs. Data is read from a file in HDF5 format.

6.3 Extensibility via protocol-based interfaces

Two major factors behind Python's widespread adoption as the language of choice of researchers (and specifically machine learners) are its rich library ecosystem and its ability to enable quick iteration, refactoring and prototyping. Not only, but it could be argued that the growth of the Python software ecosystem has been, in fact, greatly facilitated precisely by the short development cycles the language enables. In this light, it might be easier to appreciate the important role played by the language's flexible type system: even though the Python programming language is statically typed, its syntax does not require that function declarations explicitly specify the type of their arguments. Instead, functions can accept arguments of any type, and no errors will be produced as long as the argument type supports the usage the function makes of it, or in other words adheres to the *protocol* the function requires. This property of the Python language is often referred to as "duck typing", a reference to the "duck test" of logical inference: "if it walks like a duck and quacks like a duck, then it must be a duck". Although protocols have always been part of idiomatic Python usage (e.g., for the implementation of custom iterators or context managers), with the advent of type annotations in Python 3, programming interfaces can explicitly document protocol requirements; not only, but static analysis tools such as `mypy` can verify they are satisfied in client code, making protocol-based programming interfaces the obvious choice to provide user-facing customization points.

Thanks to duck typing, each function argument has the potential to serve as a customization point for a given library or framework. In practice, library designers can promote certain arguments to customization points by defining, in code, a custom protocol they are required to satisfy. Compared to polymorphism through type inheritance (possibly the most common software pattern for behavior customization, at least in object oriented programming languages), protocols are more flexible and more easily composed: firstly, while multiple inheritance is typically cumbersome in languages such as C++ (as well as Python itself, due to non-obvious method resolution order), a given type can easily adhere to multiple protocols; secondly, similarly to what happens in policy-based design (Alexandrescu, 2001) library code can easily provide fall-back implementations as well as take alternative code-paths depending on what *parts* of a protocol a type implements. Finally, and maybe trivially, protocol-based customization points in Python are also preferred because they fit nicely with the language's duck typing mechanism.

Tab. 6.1 lists the protocols available when implementing a new model to be integrated in the TVEM framework. `Trainable` is the simplest protocol for a trainable TVEM model. It represents the bare minimum requirements for a Python class to act as a TVEM model. Optionally, a `Trainable` model can also implement `update_`Θ to take advantage of analytical M-step update rules

Table 6.1: TVEM framework protocols for generative models. A TVEM model must implement at least "Trainable", and can optionally also implement "Optimized", "Sampler" and "Reconstructor" to add support for the corresponding features. See text for a description of each protocol.

Protocol	Requirements
Trainable	`log_joint(batch, Φ)`, parameter set `theta`
Optimized	same as "Trainable", plus `log_pseudo_joint(batch, Φ)`
Sampler	`generate_data(n_samples)`
Reconstructor	`data_estimator(batch, states)`

instead of gradient-based optimization of the parameters. `Optimized` builds on `Trainable` by adding support for the log-pseudo-joint numerical optimization (see Sec. 3.3.3). `Sampler` is orthogonal to the other protocols and adds support for sampling of datapoints from the generative model. `Reconstructor` is also orthogonal to other protocols and adds support for data reconstruction for this model. This is how denoising as well as in-painting are supported by the TVEM framework.

6.4 Black-box variational inference through automatic differentiation

An advantage of TVEM when compared to other common variational approximation schemes such as factored distributions (see Sec. 2.2.3) is that the TVEM E-step does not require model-specific derivations. In fact, TVEM's E-step only depends on the specific generative model via the log-joint distributions: log-joints are used to select the best variational states to employ as variational parameters of truncated posteriors. This is in contrast with other frameworks for variational inference such as Pyro, which in general require that users define "guides", i.e. that they provide a specific definition for the variational distributions. For what regards the M-step, on the other hand, closed-form parameter update equations such as those listed in App. A naturally require model-specific mathematical derivations; however, given an efficiently computable objective function such as the compact free energy of Eq. 3.5, automatic differentiation techniques can be applied to evaluate gradients of the optimization objective with respect to model parameters. Therefore we can optimize model parameters without requiring model-specific analytical derivations at the cost of exchanging exact M-step updates with gradient-based updates. On one hand, exact M-steps can reach optima faster and are less computationally demanding; on the other, stochastic gradient ascent

introduces noise in the optimization process that is often useful to avoid undesirable, shallow local optima.

TVEM's generic E-step, combined with automatic differentiation for M-step parameter updates, yields a truly black-box optimization system for probabilistic generative models: the only ingredient required is the definition of the probabilistic generative model in terms of log-joint probabilities.

Listing 2 shows how the TVEM framework supports training of a model for which only the log-joint probabilities have been specified.

6.5 Automatic inference of computing targets

The TVEM framework adds facilities to automatically distribute computation on heterogeneous computing targets (single-node, MPI CPU clusters, GPU nodes) on top of the capabilities of PyTorch. In particular, the `Experiment` objects are able to automatically detect whether the program is being executed as an MPI application and properly initialize process-local variables as well as partition datasets between all available workers. For what regards GPU processing, E-steps and M-steps can transparently be deployed to GPU targets by setting the `TVEM_GPU` environment variable to the index of the device where computation should be executed. Mixing MPI multi-processing computation with deployment to GPU is currently not supported although it is on our road-map. Listing 3 shows how this looks in practice.

6.6 Performance profiling

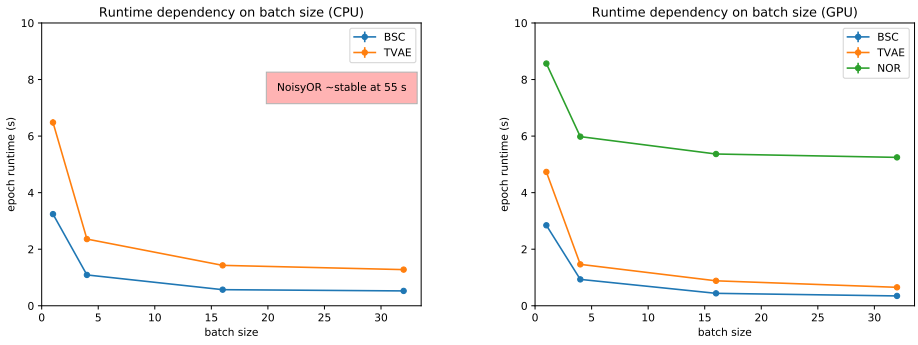

Figure 6.2: Runtime per epoch (in seconds) as a function of batch size on CPU (left) and GPU (right)

Much care has been taken in implementing efficient generic E-steps and model-specific M-steps. One example, namely usage of numerically stable

```
from tvem.exp import ExpConfig, EEMConfig, Training
from tvem.utils.model_protocols import Trainable
import torch as to

class BlackBoxModel(Trainable):
    def __init__(self, H: int, D: int):
        """
        Initialize model parameters as a dictionary
        of PyTorch tensors _theta.
        """
        self._theta = dict(pi=to.rand(H),
                           logsigma=to.tensor(1e-3),
                           W=to.rand(H, D))

    def log_joint(self, batch: to.Tensor,
                  var_states: to.Tensor) -> to.Tensor:
        """
        Evaluate log_joint probabilities for batch given
        var_states.
        """
        ...

model = BlackBoxModel(H=16, D=256)
estep_conf = EEMConfig(n_states=8, n_parents=3, n_generations=2)
exp = Training(ExpConfig(), estep_conf, model,
               train_data_file="blackbox_test.h5")
for log in exp.run(200):
   log.print()
# Model parameters `model._theta` have been updated.
# Variational states can be accessed as `exp.train_states`.
```

Listing 2: This Python snippet shows how to train a user-defined model for which only the log-joint probabilities have been defined with the TVEM framework. Model parameters must be PyTorch tensors and are registered in the `self._theta` attribute.

```
$ # run the program on a single node
$ python run_exp.py
$ # run the program as an MPI application
$ mpirun run_exp.py
$ # run the program on GPU number 1
$ TVEM_GPU=1 python run_exp.py
```

Listing 3: Running a TVEM program on different computing targets.

and more computationally efficient log-pseudo-joints in place of full log-joint computations, has already been discussed in Sec. 3.3.3. During performance profiling, EEM's genetic operators appeared as another potential hot spot; this is not so surprising as evolutionary algorithms are applied to several variational states per datapoint, making them part of the few computations that are executed in "hot" loops with size $\mathcal{O}(N \times \|\Phi^n\|)$. Therefore we implemented batch-wise genetic operators that do not require Python for loops, but instead, by substituting an imperative programming style with array programming, move such expensive loops to efficient linear algebra libraries that execute them at the speed of optimized machine code. As a consequence, the performance of the TVEM software framework depends on the chosen batch size, i.e. the amount of datapoints that E-steps and M-steps process in one go; similarly to what typically happens in deep learning applications, a small batch size results in worse runtimes due to an increase in overhead computations and a lack of low-level parallelization or CPU vectorization of the computations, while larger and larger batch sizes require more and more memory.

In order to investigate the dependency of runtime on hyperparameters such as batch size or number of processing cores, we run a simple toy experiment: different generative models are trained on a dataset consisting of N=1000 randomly generated arrays of size D=784. Models have H=128 latent variables and TVAE, in particular, also has 128 units in the middle layer of its neural network. We train with EEM using fitness-proportional parent selection and random uniform mutations, and no crossover. Unless specified otherwise, we will use $\|\Phi^n\| = 64$ variational states and evolutionary algorithms will select 8 parent states and produce 7 children per parent in 1 generation. These tests were run on a machine with a single NVIDIA Titan Xp GPU and 8 physical Intel i7-7820X cores, with two threads per core.

Fig. 6.2 shows how the runtime of a training epoch changes w.r.t. the batch size employed, for executions on CPU (left) and GPU (right). The plots contain error bars, but they are actually smaller than the markers: epoch runtimes are extremely stable for this use case. It can be noticed that, for both CPU and GPU executions, the framework does take advantage of larger batch sizes but there is no further benefit beyond batches of size 32, so in further

experiments we will keep it fixed at this value. Except for Noisy-OR, GPU runtimes are only marginally smaller than corresponding CPU runtimes for this experiment. The reason is twofold: firstly, the lack of large speed-ups when switching to GPU is due to the relatively small problem size, which highlights the overhead of memory transfers between CPU and GPU when computations are run on GPUs; in fact, for one of the TVAE denoising experiments of Sec. 5.4.5, executing one epoch on CPU takes between 90 and 110 seconds while, on GPU, epoch runtime is around 28 seconds. Secondly, even when GPUs are targeted, there is one potentially expensive part of the algorithm that is unconditionally run on CPUs: detection of duplicated variational states when updating Φ^n by selecting the best $\|\Phi^n\|$ states in $\Phi^n \cup \Phi^{\text{new}}$ (compare Alg. 1). Such checks are necessary because Φ^n is required to be a set of unique latent states. This search of duplicate latent states is not trivial to write as a batch operation in array programming style, so for the TVEM framework it has been implemented as a Python extension module using Cython (Behnel et al., 2011), and it executes at C speeds. However, it is always executed on CPU, even when a GPU is employed for all other computations, which does require expensive data copies between devices. As model size increases w.r.t. $\|\Phi^n\|$, though, this becomes less and less of an issue.

Noisy-OR runtimes are much larger and display a slightly more surprising behavior than BSC and TVAE: when running on CPU, runtime does not depend on batch size, the speed-up when switching to GPU is much larger than for the other models, but scaling with batch-size already plateaus at size 16. We believe the reason for this behavior is that, compared to the other two models, Noisy-OR's log-pseudo-joint and M-step computations make use of larger arrays of size $\|\Phi^n\| \times D \times H \times \text{batch\_size}$ (computations on which dominate runtimes). As a consequence, with a batch size of 1 Noisy-OR already fills the vectorization capacity of PyTorch's CPU implementation of the necessary linear algebra operators, and smaller batch sizes also fill the GPU's vectorization capacity. In the current implementation, such large arrays also require expensive allocations of temporaries (which might hinder scaling unless specially crafted memory allocators are used), although our attempts to reduce the amount of allocations did not improve the scaling with batch size.

Fig. 6.3 reports result for a similar experiment but with batch size fixed to 32 and a varying number of MPI worker processes, on a single worker node. Because Noisy-OR computations require the allocation of large arrays (see above) and MPI usage makes it so that the arrays need to be allocated once per MPI worker, we had to reduce the problem's dimensionality for Noisy-OR and set $N = 200$. As mentioned before, EEM's E-step is trivially data parallel and the most computationally expensive of models' M-steps also often are, so distributing parts of the dataset to each worker process and aggregating the partial results in a final reduction steps scales fairly well up to the available

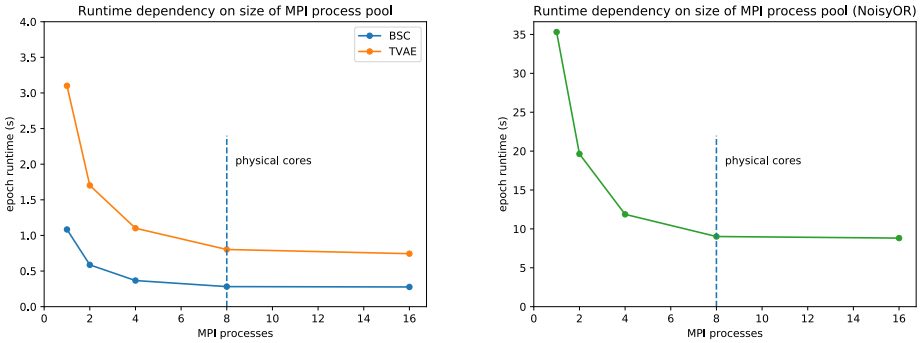

Figure 6.3: Runtime per epoch (in seconds) as a function of the number of MPI worker processes for BSC and TVAE (left) and Noisy-OR (right).

number of physical cores. Note that runtimes with one worker process are larger than the equivalent runtimes with no usage of MPI (i.e. those reported in Fig. 6.2): multi-processing does involve a non-negligible runtime overhead and also increases the amount of memory required to run the training, so it is advisable to only adopt it when the problem is very large and the workload can be distributed on different machines.

Finally, Fig. 6.4 shows how runtime per epoch changes as a function of the size of the variational sets $\|\Phi^n\|$. The plot shows runtimes for the experiment described above and the TVAE model, with batch size set to 32 (the full epoch runtime does not match what was reported in Fig. 6.2 for batches of size 32 because this last experiment was run on a different machine with four Intel i7-4790 physical cores). The lower line shows runtimes for the E-step only, and the higher line the full epoch runtime. As expected, the dependency of runtime on $\|\Phi^n\|$ is clearly linear, but note the small coefficient: runtimes go from 0.64 seconds to 7.28 seconds when $\|\Phi^n\|$ goes from 1 to 256. Another interesting observation is that, at least for the TVAE model, the impact on runtime of larger $\|\Phi^n\|$ is dominated by the more and more expensive M-steps rather than the increase runtime cost of EEM's E-steps.

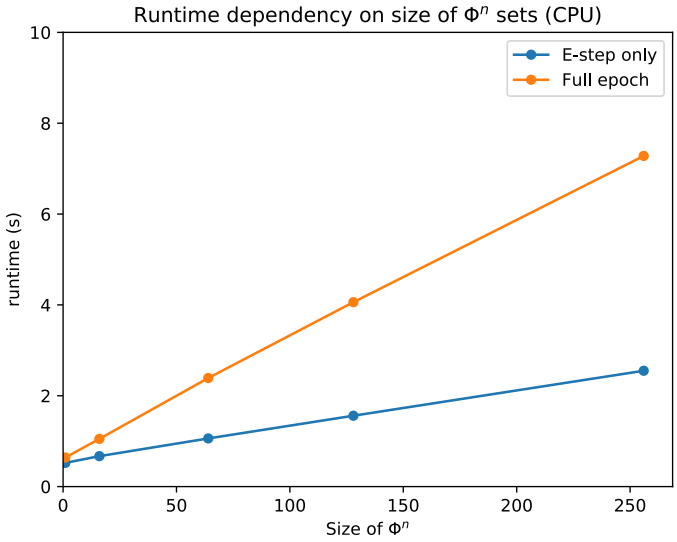

Figure 6.4: Runtime per epoch as a function of the size of the sets Φ^n of variational states. Time spent in the E-step is shown in blue, total epoch runtime in orange.

Chapter 7

Handling Large Datasets: Improving the Machine Learning Pipeline at CERN

Machine learning tools, including prominent deep learning frameworks such as Tensorflow (Abadi et al., 2016) and Pytorch (Paszke et al., 2017), are typically designed with the expectation that training, validation and testing will be performed on datasets that fit into the machine's fast memory access capacity (i.e., that fit in a machine's or a cluster's RAM). At the same time, however, typical dataset sizes have increased greatly with time, and faster than the capacity of memory units. "Big data" is commonly used to refer to datasets that are not only larger than memory, but that also exceed the capacity of a single off-the-shelf storage drive; state-of-the-art deep learning architectures are notoriously data-hungry; in general, as we have discussed in previous chapters, both for supervised and unsupervised learning it is interesting to develop algorithms and tools that can deal with large datasets as, at a fundamental level, larger sample sizes provide a greater amount of statistics and therefore a more precise insight on the problem at hand.

Workarounds for handling larger-than-memory datasets exist: for instance, for many applications, including Truncated Variational Expectation Maximization, datasets can be stored on hard drives and batches of datapoints can be swapped in and out of working memory at will during training. However, such solutions typically have a high cost in terms of computing runtime.

As part of my thesis, I have been involved in the field of data analysis software for the European Organization for Nuclear Research (CERN). High Energy Physics (HEP) data analyses often involve machine learning techniques as well as datasets with sizes in the order of terabytes or tens of terabytes, but the very nature of HEP data analysis offers an alternative solution to the problem of handling large datasets. Machine learning algorithms, in fact,

are typically applied at the final stages of the pipeline, at which point the researcher has selected a small percentage of interesting datapoints (or "events") which are interesting for this last step. This event selection and pre-processing step is often a bottleneck for researchers.

ROOT (Brun and Rademakers, 1997) is a large, mature software framework that provides the necessary foundation for the processing and analysis of data produced by CERN's Large Hadron Collider as well as many other HEP experiments. Working with the ROOT development team, I have helped delivering new high-level, modern data processing interfaces in Python and C++, named `ROOT::RDataFrame` [1]. As an outcome, not only HEP scientists have at their disposal a new declarative, high-performance interface for data manipulation, but they can also leverage `RDataFrame` to efficiently perform complex event selections on larger-than-memory or larger-than-disk datasets and then export interesting data to formats useful for further processing: interestingly for machine learning applications, `RDataFrame` allows exporting ROOT data as Numpy arrays, which makes it easy to select relevant parts of a dataset, pre-process them and finally feed the transformed data to standard machine learning libraries and tools.

This chapter first provides a brief introduction to ROOT and data analysis at CERN (Sec. 7.1) as well as an overview of `RDataFrame`'s design and features (Sec. 7.2). Then Sec. 7.4 presents a real-world use case in which `RDataFrame` was used to build a data transformation tool for the ATLAS experiment. Finally, Sec. 7.6 shows how this tool makes it possible to efficiently skim larger-than-memory datasets and feed selected events to Python's data science ecosystem.

I have personally contributed most of the design and implementation of `RDataFrame`'s interfaces in collaboration with Danilo Piparo, who was supervising my work in the ROOT team. Multi-thread reading capabilities are based on previous work by Enric Tejedor, and parallel writing leverages previous work by Guilherme Amadio (Amadio et al., 2019). I presented `RDataFrame` at the CHEP 2018 computing conference; the proceedings of the conference contribution, on which the content of this chapter is based, are available as (Piparo et al., 2019). The scaling plot of Fig. 7.7 was produced by Xavier Valls, another doctoral student that was collaborating with the ROOT team at the time. The exporting of ROOT data as Numpy arrays has been developed in collaboration with Stefan Wunsch, also a doctoral student at the time.

[1] in the following just `RDataFrame`, first presented publicly, when still at the experimental stage, at the ACAT 2017 conference by (Amadio et al., 2018) as `ROOT::Experimental::TDataFrame`

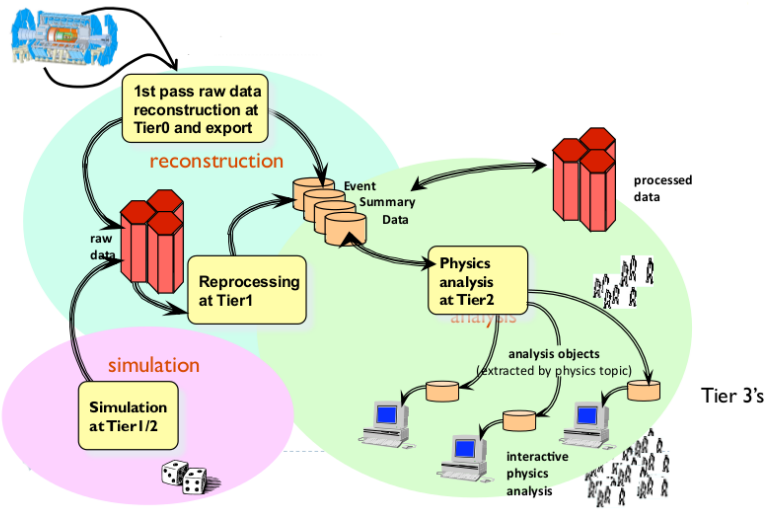

Figure 7.1: A schematic representation of the tiered data processing architecture of the Worldwide LHC Computing Grid (WLCG). Raw signals coming from the detector are "reconstructed" into physical entities (e.g., particles, tracks, jets); a large amount of simulated data is also produced, and will serve as validation data for all analyses. Given reconstructed events, datasets with contents specific to a given physics topic are produced; these datasets are typically in the order of hundreds of GBs or a few TBs. Further analysis-specific event selection brings datasets down to the order of GBs, at which point in-memory processing becomes feasible; this final selections and analysis-specific manipulations are the use case that `RDataFrame` mainly aims to satisfy. After the reconstruction step, data is typically stored in ROOT format.

7.1 ROOT: a cornerstone of CERN's software ecosystem

ROOT (Brun and Rademakers, 1997) is an open-source software framework for data processing, analysis and visualization. It was born at CERN in 1995 and it is mainly targeted at High Energy Physics (HEP) use cases.

First and foremost, ROOT provides a common data format for efficient I/O of the kind of data that is typically handled by HEP analyses: columnar data with nested collections, in which rows represent independent physical events (e.g., collisions of proton bunches within CERN's Large Hadron Collider) and columns represent different kinds of physical entities (e.g., the momentum of each electron produced by a collision, the missing transverse energy and so forth). Although the ROOT format supports reading and writing of complex C++ data structures (thanks to the Cling C++ interpreter, Vasilev et al., 2012), at the level of interactive physics analysis (which is `RDataFrame`'s main

target, see Fig. 7.1[2]) column values mostly consist of fundamental data types (integers and floating point numbers) and arrays thereof; nevertheless, the presence of these nested arrays gives HEP datasets a "jagged" shape (as opposed to the rectangular shape of HDF5 tables, for instance, which is common in deep learning applications) that often requires ad-hoc tools and interfaces to be manipulated efficiently.

On top of the ROOT data format and the corresponding I/O layer, ROOT offers a vast range of features, including a library for statistical fitting, efficient (single- and multi-dimensional) histograms as well as a GUI toolkit and facilities for data visualization and multi-thread and multi-process computation.

The ROOT project is committed to take physicists from data acquisition to publication as effectively as possible. The need to offer analysts simpler and yet powerful interfaces that could easily let them exploit the full potential of their hardware became all the more apparent with the increased luminosity and the upgrades of the LHC experiments foreseen for Run III (Albrecht et al., 2019), HL-LHC (Apollinari et al., 2017) and FCC (Fernandez et al., 2012) – with the consequent increase in the amount and complexity of available data. `RDataFrame` has been developed in order to address these requirements. In a similar vein to other modern data analysis frameworks such as Apache Spark's DataFrames (Zaharia et al., 2010) and Python's data analysis library pandas (McKinney et al., 2010), `RDataFrame` exposes a declarative API designed to be easy to use correctly and hard to use incorrectly. Novel elements introduced by `RDataFrame` with respect to pre-existing libraries are the choice of programming language (C++), which allows usage of template metaprogramming to avoid runtime overhead while maintaining generality of interfaces, the integration of just-in-time compilation of user-defined expressions to make analysis definition concise when top performance is not required, and of course a tight integration with the rest of the ROOT data analysis toolkit. User-transparent task-based parallelism has been a goal since inception, and recently published results (Avati et al., 2019) demonstrates that the programming model lends itself to multi-node distributed execution with little to no changes.

7.2 RDataFrame's software design

7.2.1 Design principles

At a high level, `RDataFrame` strives to expose modern, elegant and safe interfaces. The introduction of elements of **declarative programming** in the design (users say *what* they need to compute, `RDataFrame` chooses *how* to compute it) provides user-visible advantages such as less typing, increased readability and abstraction of complex operations. At the same time, by decoupling

[2]Image source: "Building the world's largest Scientific Grid", a presentation by Jamie Shiers at the Oracle Tech Day 2004.

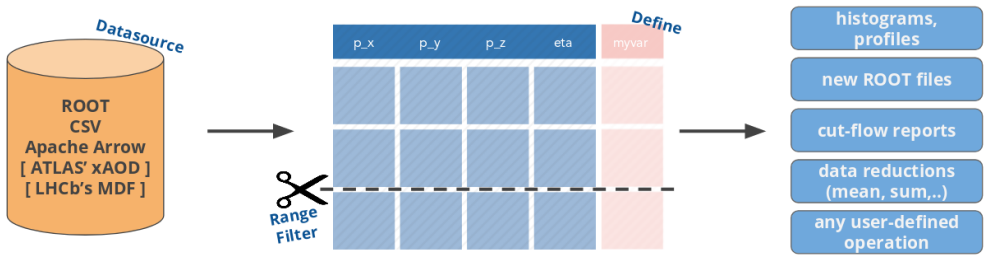

Figure 7.2: The `RDataFrame` framework reads from a columnar data format via a data source, applies transformations to the data (i.e. selects rows and/or defines new columns) and produces results (i.e. data reductions like histograms, new ROOT files, or any other user-defined object or side effect). Data sources for ATLAS' xAOD data format and LHCb's MDF binary data format exist but are not distributed with ROOT.

```
ROOT::EnableImplicitMT();                          Run a parallel analysis
ROOT::RDataFrame df(dataset);                      on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("x > 0")                      only accept events for which x > 0
             .Define("r2", "x*x + y*y");           define r2 = x² + y²
auto rHist = df2.Histo1D("r2");                    plot r2 for events that pass the cut
df2.Snapshot("newtree", "out.root");               write the skimmed data and r2
                                                   to a new ROOT file
```

Figure 7.3: A simple C++ `RDataFrame` analysis that performs event selection, defines a new quantity, produces a histogram and writes processed data to disk. All registered operations will be executed in a single event loop.

the API from the underlying implementation, the declarative paradigm allows transparent optimizations (e.g. user-transparent parallel processing of ranges of events, lazy evaluation, caching) that it would not have been possible to introduce in previous ROOT data analysis facilities such as `TTree` with a more imperative API.

As shown in figure 7.3, the design also features elements of **functional programming** such as pure and higher order functions which encourage users to program in terms of small and reusable components with less side effects and less shared state; this in turn increases thread safety and code correctness: pure functions are thread safe by construction and are easier to test as they do not carry dependencies on global state. Furthermore, thanks to PyROOT's automatic python binding generation (Generowicz et al., 2004), most of the framework's functionality is seamlessly available in Python, guaranteeing a

consistent user experience across programming languages.

7.2.2 Functional parts

Concretely, the framework is composed of three kinds of objects:

- *data sources* read columnar data and expose a common format-independent interface. This is a customization point: expert users can implement a data source for their columnar data format of choice. Data sources are discussed in more detail in Sec. 7.3.

- *nodes* are objects that represent one step of the data analysis workflow specified by the user. They form a computation graph which represents the full analysis workflow. With reference to figure 7.3, each `Filter`, `Define` or `Histo1D` invocation creates a node object which is appended to the node on which the method was called, hence forming a graph.

- *results*: most `RDataFrame` methods return a smart pointer to an object produced through data processing. For example, `Histo1D` returns a smart pointer to a histogram (`TH1D`) object. These smart pointers are the mechanism through which lazy execution is implemented: the actual data processing is only triggered upon access to one of the results produced by an `RDataFrame` (i.e. dereferencing of the smart pointer). During the data processing all previously booked results are produced simultaneously.

We distinguish between `RDataFrame` methods which return new `RDataFrame`-like objects (such as `Filter` or `Define`) and methods which return results. We refer to the former as *transformations* and to the latter as *actions*, following Spark's nomenclature.

7.2.3 Parallelization scheme

The actual event loop is parallelized by processing different chunks of data in different tasks. Tasks are scheduled for execution on a thread pool, and the execution of each task updates a thread-local copy of each desired result with the output of the processing of the relevant data chunk. As an optional final step, thread-local partial results are merged into a single result object that will be handed to users. ROOT's task scheduler is currently Intel's Threading Building Blocks (TBB) library (see Piparo et al., 2017).
Assuming the task scheduler employs a pool of worker threads of appropriate size (as it happens with TBB), task-based parallelism offers several advantages: there is no risk of over-committing computing resources, as the task scheduler

ensures that each thread runs one task at a time, also taking into account dependencies between tasks; this scheme also integrates well with other entities (e.g. experiments' frameworks) which schedule their own tasks, as long as all tasks are submitted to a common scheduler. Finally, redundant decompression of ROOT data is avoided by chunking inputs with the same granularity at which they were compressed.

7.3 High-level customization points

7.3.1 Data sources

Figure 7.4: RDataFrame can read any columnar data format through a dedicated data source implementation. Expert users can implement and seamlessly integrate data sources for their format of choice.

Although the ROOT data format certainly plays a large role in HEP analyses, it is however not the only format of interest in science: a common user requirement is to read text files, in CSV format or similar, into ROOT; the ALICE experiment also expressed interest in the possibility to use `RDataFrame`'s declarative analysis paradigm to process Apache Arrow in-memory tables, as part of their data analysis framework renovation effort (Eulisse et al., 2019). In order to address these use cases, the framework provides the interface type `RDataSource` which defines a minimal API that `RDataFrame` can use to read arbitrary tabular data formats.

In practice, `RDataSource` is a C++ abstract base class which imposes certain functional requirements onto its implementations; concrete derived types will function as adaptors that `RDataFrame` can leverage to read any kind of tabular data formats. `RDataFrame` calls into `RDataSources` to retrieve information about the data, to obtain (thread-local) readers or "cursors" for selected columns and to advance such readers to the desired data entry.

`RDataSource` not only extends `RDataFrame` to support other formats than ROOT, but it decouples the analysis code from the format analysed so that users can use the same exact code to process potentially very different datasets.

CSV and Apache Arrow inputs are currently supported through this mechanism and prototypes for LHCb's MDF binary data format and ATLAS' xAOD event data model (see Dharmaji, 2018) have been developed, which goes to show the flexibility of the approach.

```
auto h = tdf.Histo1D("x");                          book the creation of a histogram

TCanvas c("c", "x hist");
auto drawHisto = [&c](TH1D &h_) {                   define the callback function:
  c.cd();                                             it updates a TCanvas
  h_.Draw();                                          with the drawing of a
  c.Update();                                         partially filled histogram
};

h.OnPartialResult(100, drawHisto);                  book invocation of callback on a
                                                    partially filled `h` every 100 entries
h->Draw();                                          trigger event loop
```

Figure 7.5: An example callback usage, commented line by line. The `drawHisto` function is called by one of the worker threads every one hundred entries processed, and it refreshes a canvas on which the state of the histogram is displayed. The callback need not be thread-safe, as it is never executed concurrently. In multi-thread executions, the partial result that the callback will receive as argument will be the thread-local copy that the relevant worker thread is employing.

7.3.2 User-defined callbacks

It is possible to schedule execution of arbitrary functions (callbacks) during the event loop. Callbacks can be used, for example, to inspect the ongoing filling of a histogram as the event loop is running, to save results to a file every time a certain number of new entries are processed, or to display a progress bar that indicates event loop progress.
As an example, figure 7.5 shows how one can draw an up-to-date version of a result histogram every 100 entries.

At user's discretion, callbacks can be called once at the beginning of the event loop or every time a certain amount of new entries have been processed. Users can also decide whether the callback should be called only by one thread at a time (*which* thread calls the callback might vary during execution, but the framework guarantees that the given amount of entries will have been processed between calls) or by all threads, potentially concurrently, in which case the user is responsible for providing a thread safe callback function.
This feature is currently only available in C++, not Python.

7.4 A real-world RDataFrame application

As a case study, we would like to discuss a real-world `RDataFrame` C++ application developed by ROOT users from the ATLAS collaboration. Refer

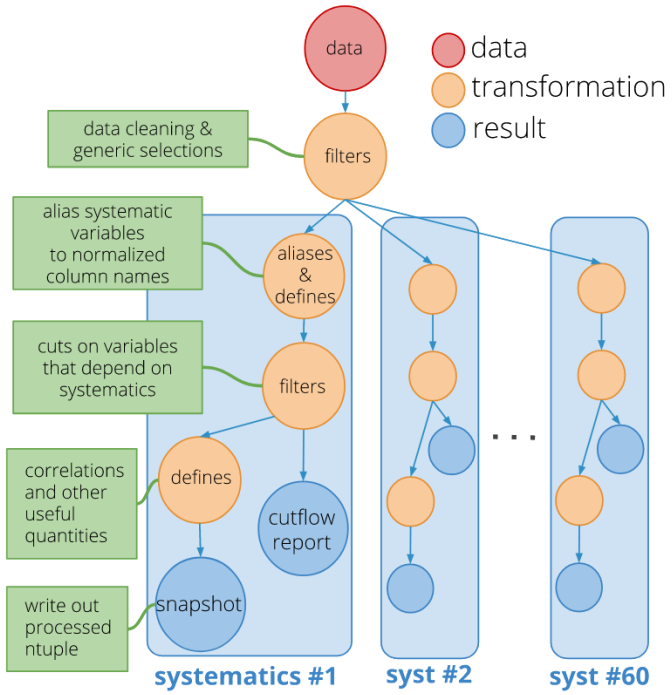

Figure 7.6: A representation of the computation graph of an `RDataFrame` C++ application that performs ntuple to ntuple processing of simulated data. For each of sixty different systematics, the input ntuple is skimmed and several columns containing quantities relevant for further processing and dependent on systematics are added. Sixty new output ROOT files are produced within the same event loop.

to figure 7.6 for the application's computation graph and a brief explanation of its purpose. It is worth highlighting a few striking features of the application's code-base: first of all, thanks to the declarative programming model, the program's `main` function is a simple sequence of `Filter` and `Define` calls, followed by a call to schedule a write-out of the processed ntuple (`RDataFrame`'s `Snapshot` method); the definition of the analysis workflow is clearly separated from the definition of the smaller helper functions. Given a little familiarity with `RDataFrame`'s API, the workflow is grasped quickly, with no need to dive into finer details if not required: such details are encapsulated in several small, pure helper functions. Of course good software design practice is to always make applications as readable and modular as possible; however, `RDataFrame`'s programming model makes it natural and encourages users to write code with such qualities.

Finally, it is worth noting that event selection, calculation of new quantities and writing of the sixty output ROOT datasets all happen within a single parallelized event loop, an achievement that would have required signif-

icant effort and attention to low-level details with ROOT interfaces preceding `RDataFrame`.

7.5 Scaling and performance benchmarks

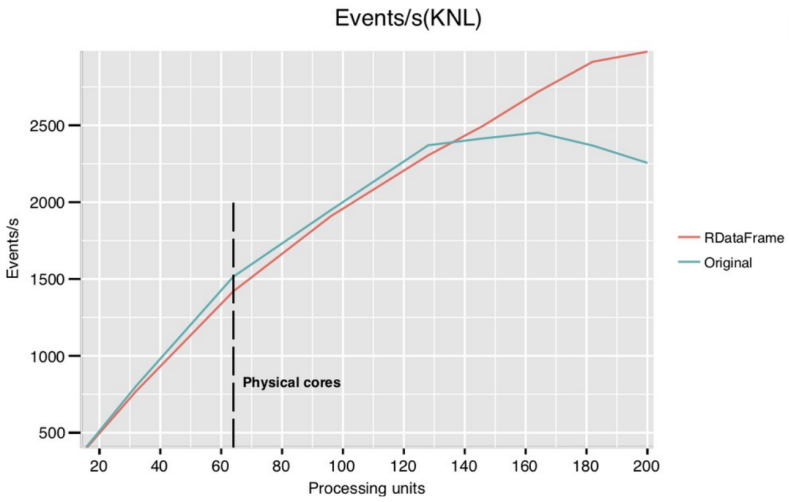

Figure 7.7: Scaling of a Monte Carlo QCD Low-PT event generation and analysis on the fly for an ad-hoc implementation using a patched ROOT 5 and POSIX threads (labeled "original" in the plot) and an `RDataFrame` rewrite of that same application (yielding identical results). No disk reads or writes are performed by either application. KNL architecture, 64 physical cores.

In order to measure the scaling of a realistic `RDataFrame` application, we took a pre-existing parallel code that generates low-pt QCD events, processes them and plots some quantities of interest as ROOT histograms. This application has been developed for research purposes by an expert ROOT user, who based it on a fork of ROOT version 5 patched to allow multi-thread data analysis. Data is produced and analyses on the fly: absence of direct disk I/O makes it possible to scale to a large number of cores without being limited by the hardware's reading speed. We compare the original code with an `RDataFrame`-based rewriting which produces identical results and reuses most of the numerical computation logic. As figure 7.7 shows, `RDataFrame` introduces a small overhead with respect to the original ad-hoc code, which made direct use of lower-level ROOT interfaces. On the other hand, `RDataFrame` scales to a larger amount of cores thanks to task-based parallelism and less lock contention during the event loop.

In order to assess `RDataFrame`'s I/O performance, the measurements of (Blomer, 2018) were repeated with the latest release of ROOT; the results are displayed in figure 7.8. No significant changes with respect to the original

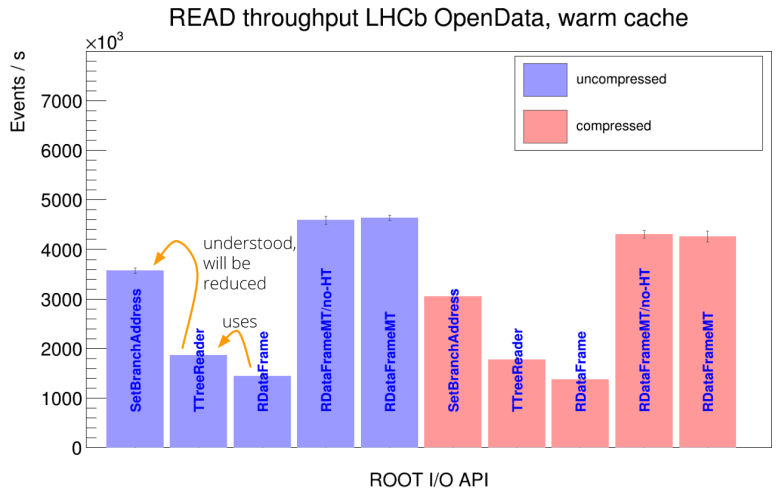

Figure 7.8: Read speed (events/s) on an LHCb OpenData dataset for three different reading APIs available in ROOT: `TTree`+`SetBranchAddress`, `TTreeReader`, `RDataFrame` (with and without implicit multi-threading enabled). The benchmark was run on a machine with four physical cores, 3.6 GHz each, and an off-the-shelf SSD. `TTreeReader` adds a non-negligible overhead on top of direct `TTree` usage, whose origin is understood. This overhead will be reduced in future ROOT releases. `RDataFrame` employs `TTreeReader` internally, inheriting the overhead as a consequence.

measurements of (Blomer, 2018) were detected: `RDataFrame` is the fastest interface ROOT offers to analyse ROOT data if one takes into account the simplicity of expressing parallelism, although single-thread execution suffers from an important overhead with respect to direct usage of `TTree`. The cause of such overhead has been identified and mitigations will be introduced in future releases.

7.6 Exporting HEP data to the Python data science ecosystem with RDataFrame

ROOT provides efficient C++ implementations of machine learning algorithms that are typically employed in High Energy Physics (e.g., Boosted Decision Trees) as well as facilities for fast inference with standard deep learning models. However, ROOT cannot (nor tries to) compete with the rich and ever-evolving landscape of machine learning tools and algorithms that the Python data science ecosystem provides. When it comes to training models with the latest variation of stochastic gradient descent, or applying recently published learning algorithms to HEP data, ROOT's goal is to seamlessly bridge the HEP ecosystem (namely, the ROOT data format) and the Python data science

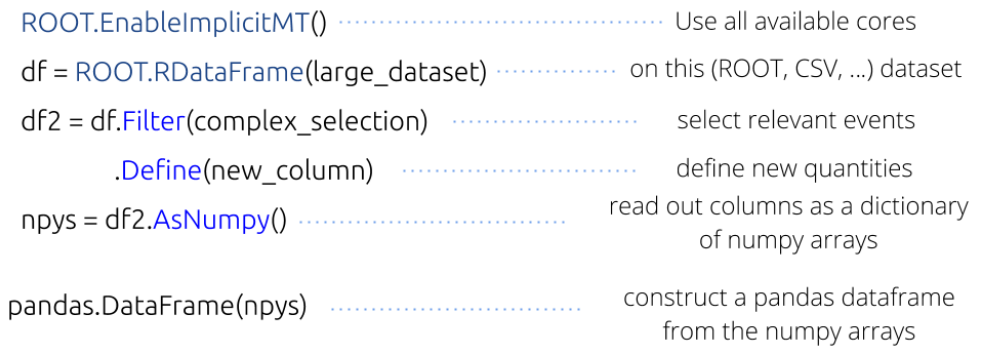

Figure 7.9: An example usage of the `AsNumpy` method, commented line by line.

ecosystem.

Numpy arrays are de-facto the "lingua franca" of data analysis and processing tools in Python; therefore, an efficient export mechanism from the ROOT data format to Numpy arrays is a highly desirable feature. Indeed, until recent years ROOT lacked such an export mechanism and multiple third-party libraries have been developed in order to fill this gap.

Thanks to `RDataFrame`, however, ROOT now provides a high-level, declarative interface that not only allows fast multi-thread reading of ROOT data, but, notably, can also apply complex event selection rules and can evaluate new derived quantities on the fly while reading larger-than-memory data from disk or from remote storage over the network. `RDataFrame` is therefore the ideal tool to apply a pre-selection step and a pre-processing step that transforms a dataset such that it is made digestible by external machine learning libraries. Fig. 7.9 shows how the `AsNumpy` method can be used for this purpose. Data is processed as it is read and large datasets can be reduced to a form that fits in memory *while* they are being exported to Numpy arrays. As usual, transparent multi-thread task parallelism guarantees that this potentially time-consuming pre-processing step is executed quickly.

Chapter 8

Summary

The intractability of exact maximum likelihood inference in probabilistic generative models has spurred the development of a large number of approximate inference techniques. In particular, the variational Expectation-Maximization framework offers a generic, mathematically grounded procedure to derive efficient, scalable training algorithms, but the performance of such algorithms strongly depends on the choice of variational distributions.

Although evolutionary algorithms (EAs) have been applied in conjunction with the Expectation-Maximization algorithm before (see e.g. Pernkopf and Bouchaffra, 2005, in which EAs are used to search the space of model hyperparameters), we here, for the first time, link EAs and optimization of the variational lower bound much more intimately: genetic operators are employed to optimize the variational parameters themselves, not to select the model's architecture or other hyperparameters. Indeed, one of our most notable contributions consists of the first working application of EA-based variational inference to approximate maximum-likelihood training of probabilistic generative models, opening the door to a large variety of applications and possible future improvements. In Ch. 4, we provided numerical experiments that demonstrate applicability of the approach to discrete-latent generative models, scalability to large latent spaces, and a promising application to denoising.

In Ch. 5, we investigated whether this novel training algorithm can successfully be applied to the training of Variational Autoencoders (VAEs) with binary latent variables. Thanks to the direct optimization of the VAE encoding model, estimation of gradients of the variational lower bound with respect to the encoder parameters is side-stepped completely, making it possible to train discrete-latent VAEs without exceptionally common approximations such as sampling approximations of the gradients and factored Gaussian approximations of the model posteriors. Such approximations can introduce learning biases (see e.g. the discussion by Vértes and Sahani, 2018; **?**), so the investigation of alternative training methods could help shed light on the consequences of specific approximation choices that are de-facto standard in the training of

Variational Autoencoders. Not only we show that EA-based optimization of the encoder can be married with gradient-based optimization of the model parameters in a simple way, but we also prove that discrete-latent VAEs trained this way can produce state-of-the-art results in one of the most frequently used denoising benchmarks (the "house" image) in the "zero-shot" setting. Even though we show that our approach scales up to $\mathcal{O}(1000)$, we acknowledge that the current computational requirements of this non-amortized training algorithm represent its main limitation.

As discussed in Ch. 6, a notable output of this thesis (as well as an important tool for its realization) is a production-grade open source software library that makes our novel training algorithm as well as all the models discussed readily available to researchers that are interested in applying our techniques or would like to reproduce our findings. That same software library is also capable of performing "black-box" variational inference of arbitrary generative models with discrete latent variables: only the definition of the model's log-joint probability is required in order to start training. Furthermore, as part of my residency at CERN I designed and developed a high-level, declarative data analysis interface that CERN scientists can use to export high-energy physics datasets to the Python data science ecosystem.

Finally, in Ch. 7 we presented a modern, declarative, parallel framework for data analysis and manipulation. `RDataFrame` is officially part of ROOT as of version 6.14, offers a C++ interface as well as Python bindings and it has already been employed successfully in real-world HEP analyses. `RDataFrame`'s task-based parallelism scales successfully to many-core architectures.

In summary, the following may be considered the main contributions of this thesis:

- the very first investigation of the viability and scalability of the novel Evolutionary Expectation Maximization algorithm (published as Guiraud et al., 2018, which I presented at the GECCO 2018 conference); further work on more complex sparse coding models is currently under review as Drefs et al. (2020)

- the implementation, validation and performance optimization of Truncated Variational Autoencoders, a novel training algorithm for Variational Autoencoders with discrete latents, including state-of-the-art results for the "house" denoising benchmark in the "zero-shot" setting (under review as Guiraud et al., 2020, at the time of writing)

- the development of a full-blown, ready-to-use software library for black-box variational inference which mixes Truncated Variational Expectation Maximization and automatic differentiation for the training of arbitrary discrete-latent generative models

- the development of a modern, high-level data analysis tool for High Energy Physics (HEP) datasets with C++ and Python interfaces, which provides transparent multi-threading parallelism and a simple way to export HEP data to the Python data science ecosystem (published as Piparo et al., 2019, which I presented at the CHEP 2018 conference); this work was immediately useful as the foundation for R&D such as by Avati et al. (2019)

The introductions of Chs. 4, 5, 6 and 7 highlight my personal contributions as well as collaborations with or contributions of other researchers in full detail.

8.1 Outlook

The work presented here can be extended in several different directions. The development of evolutionary algorithms and genetic operators specialized for the specific optimization problems arising in the training of generative models have great potential for future improvements of accuracy and scalability of learning. In particular, operators that reflect hierarchical relationships between latent variables might prove beneficial for the application of truncated approximations to models with structured latent spaces. Another natural research direction is the one that tackles the interesting mathematical problem of generalizing truncated posteriors to continuous latent spaces: this is a non-trivial extension, as a naive replacement of sums with integrals does not automatically result in continuous analogous of the compact free energy formulation Eq. 3.5 and the pair-wise criterion Eq. 3.7 that is at the core of Truncated Variational Expectation Maximization.

For what concerns the application to more complex, deeper generative models, a possible way to relax the algorithm's computational requirements is to optionally introduce amortization or another form of variational parameter sharing. Such an extension would pave the way towards experimentation with larger DNNs, less elementary decoders and, in general, more complex architectures that have been shown to improve inference on complex data such as images (e.g. Gulrajani et al., 2016; Sadeghi et al., 2019).

There are also several important upgrades planned for the TVEM software framework, geared towards consolidation of the existing features as well as increased ease of use. Currently, the framework supports multi-node CPU computation via MPI or single-node GPU computation: adding support for transparent computation on multi-GPU setups is a natural extension. We plan to add more probabilistic generative models as well as TVEM optimization schemes. In order to facilitate adoption and integration of new models, it is desirable to develop a thin logical layer on top of current TVEM black box inference capabilities that would be able to parse the mathematical expression that represents a probabilistic generative model definition and automatically

generate the corresponding Python code necessary to train such a model. Such a feature, together with the automatic computing target inference described in Sec. 6.5, would simplify usage of the TVEM framework dramatically, lowering the entry barrier and increasing adoption of the software and therefore the TVEM optimization methods.

Finally, future work on `RDataFrame` will revolve around offering more "pythonic" Python bindings and low-level performance optimization, especially aimed at reducing single-thread overhead with respect to direct data access. R&D on distributed execution of `RDataFrame`-based analyses is also being carried on and its first published application was Avati et al. (2019).

Appendix A

M-step equations

Here we report the M-step equations for the models introduced in Sec. 2.1.

A.1 Noisy-OR

The M-step equations for Noisy-OR are obtained by taking derivatives of the free energy Eq. 2.4 after plugging in the model Eq. 2.2, equating them to zero and solving the resulting set of equations. We report the results here for completeness:

$$\pi_h^{\text{new}} = \frac{1}{N} \sum_n \langle s_h \rangle_{q_\Phi^n} \tag{A.1}$$

$$W_{dh}^{\text{new}} = 1 + \frac{\sum_n (y_d^{(n)} - 1) \langle D_{dh}(\vec{s}) \rangle_{q_\Phi^n}}{\sum_n \langle C_{dh}(\vec{s}) \rangle_{q_\Phi^n}} \tag{A.2}$$

where

$$\begin{aligned} D_{dh}(\vec{s}) &:= \frac{\widetilde{W}_{dh}(\vec{s}) s_h}{N_d(\vec{s})(1 - N_d(\vec{s}))} \\ C_{dh}(\vec{s}) &:= \widetilde{W}_{dh}(\vec{s}) D_{dh}(\vec{s}) \\ \widetilde{W}_{dh}(\vec{s}) &:= \prod_{h' \neq h} (1 - W_{dh'} s_{h'}) \end{aligned} \tag{A.3}$$

The update equations for the weights W_{dh} do not have a closed form solution. We instead employ a fixed-point equation whose fixed point is the exact solution of the maximization step. We exploit the fact that, in practice, one single evaluation of A.3 is enough to (noisily, not optimally) move towards convergence to iteratively improve on the parameters W_{dh}.

A.2 Binary Sparse Coding

Similarly to what was described in the case of the Noisy-OR model, taking derivatives of the BSC free energy w.r.t. the model parameters π, σ and W leads to the M-step equations for BSC (see e.g. Henniges et al., 2010):

$$\pi^{\text{new}} = \frac{1}{NH} \sum_n \sum_h \langle s_h \rangle_{q_\Phi^n} \tag{A.4}$$

$$\sigma^{\text{new}} = \sqrt{\frac{1}{ND} \sum_n \langle || \vec{y}^n - W\vec{s} ||^2 \rangle_{q_\Phi^n}} \tag{A.5}$$

$$W^{\text{new}} = \left(\sum_n \vec{y}^n \langle \vec{s} \rangle_{q_\Phi^n}^T \right) \left(\sum_n \left\langle \vec{s}\vec{s}^T \right\rangle_{q_\Phi^n} \right)^{-1} \tag{A.6}$$

Appendix B

Log-pseudo-joints

As described in Sec. 3.3.3, it is often convenient to express expectations over posteriors, as well as other quantities of interest, in terms of the "log-pseudo-joints": this quantity is equal to the model's log-joint distributions with terms that do not depend on latent variables elided. Assuming Noisy-OR and Binary Sparse Coding are defined as in Sec. 2.1, we report below the explicit expressions of their log-pseudo-joints.

B.1 Noisy-OR

$$\log p_\Theta(\vec{y}, \vec{s}) = \widetilde{\log p}_\Theta(\vec{y}, \vec{s}) + \sum_{h=1}^{H} \log(1 - \pi_h) \tag{B.1}$$

$$\widetilde{\log p}_\Theta(\vec{y}, \vec{s}) = \sum_d \left(y_d \log\left(N_d(\vec{s})\right) + (1 - y_d) \log\left(1 - N_d(\vec{s})\right)\right) + \sum_{h=1}^{H} s_h \log\left(\frac{\pi_h}{1 - \pi_h}\right) \tag{B.2}$$

$$N_d(\vec{s}) = 1 - \prod_{h=1}^{H} (1 - W_{dh} s_h) \tag{B.3}$$

with the special exception of $\vec{s} = \vec{0}$ for which

$$\widetilde{\log p}_\Theta(\vec{s} = \vec{0}, \vec{y}) = \begin{cases} 0 & \text{if } \vec{y} = \vec{0} \\ -\inf & \text{otherwise} \end{cases} \tag{B.4}$$

For practical computations, we set $\widetilde{\log p}_\Theta$ to an arbitrarily low value rather than the floating point infinite representation.

B.2 Binary Sparse Coding

$$\log p_\Theta(\vec{y}, \vec{s}) = \widetilde{\log p}_\Theta(\vec{y}, \vec{s}) + H \log(1-\pi) - \frac{D}{2} \log(2\pi\sigma^2) \tag{B.5}$$

$$\widetilde{\log p}_\Theta(\vec{y}, \vec{s}) = \log\left(\frac{\pi}{1-\pi}\right) \sum_{h=1}^{H} s_h - \frac{1}{2\sigma^2} (\vec{y} - W\vec{s})^{\mathrm{T}} (\vec{y} - W\vec{s}) \tag{B.6}$$

Appendix C

Sparsity-driven mutation operator

When performing sparsity-driven bitflips, we assign each bit of a particular individual $\vec{s}$ a probability to be flipped that depends on its current state: bits that are "off" are flipped with probability p_0, the others with probability p_1. We denote p_{bf} the average probability of flipping any bit in $\vec{s}$. Our goal is then to set p_0 and p_1 to values such that the mutation operator tends to produce offspring with sparsity $\xi = \sum_h s_h$ compatible to the one learned by the model, $\xi^* = H\langle \pi_h \rangle$.

To this end, we impose the constraint that $p_1 = \alpha p_0$ for some constant α and that the average number of "on" bits after mutation be set to $\widetilde{s}$. This yields the following expressions for p_0 and p_1:

$$p_0 = \frac{H p_{\text{bf}}}{H + (\alpha - 1)|\vec{s}|}, \qquad \alpha = \frac{(H - |\vec{s}|) \cdot (H p_{\text{bf}} - \widetilde{s} + |\vec{s}|)}{(\widetilde{s} - |\vec{s}| + H p_{\text{bf}})|\vec{s}|} \tag{C.1}$$

Random uniform bitflips correspond to the case $p_0 = p_1 = p_{\text{bf}}$. For our numerical experiments we chose $\widetilde{s}$ based on the sparsity learned by the model: we set $\widetilde{s} = \sum_h \pi_h$ for Noisy-OR and $\widetilde{s} = H\pi$ for BSC. The overall probability of a bitflip was set to $p_{\text{bf}} = \frac{1}{H}$ as to perform one bitflip per variational state on average.

Appendix D

Truncated Variational Autoencoders with Poisson noise

Truncated Variational Autoencoders as discussed in Ch. 5 leave a high degree of flexibility in terms of the choice of data noise model. Although for the purposes of the experiments in the main text we employed a Gaussian noise model, if the application called for it we could instead, for instance, define TVAEs with Poisson noise as follows:

$$p(\vec{s}) = \prod_h \pi_h^{s_h} \left(1 - \pi_h\right)^{(1-s_h)} \tag{D.1}$$

$$p(\vec{y} \mid \vec{s}) = \prod_d e^{-\lambda_d(\vec{s})} \frac{\lambda_d(\vec{s})^{y_d}}{y_d!} \tag{D.2}$$

where $\vec{\lambda}(\vec{s})$, the mean of the Poisson distribution, is the output of a deep neural network.

The training algorithm of Alg. 3 does not require any modification. In this scenario, the objective of gradient-based optimization of the decoding model takes the following form:

$$E = \frac{1}{N} \sum_n \sum_d \langle -y_d^n \log \lambda_d(\vec{s}) + \lambda_d(\vec{s}) \rangle_{q_\Phi^n(\vec{s})} \tag{D.3}$$

This is similar to a cross-entropy loss but an extra regularization term is present, namely $\frac{1}{N} \sum_{n,d} \langle \lambda_d(\vec{s}) \rangle_{q_\Phi^n(\vec{s})}$, that comes from the Poisson's distribution normalization factor $e^{-\lambda_d(\vec{s})}$. The common cross-entropy loss used in deep learning ignores the regularization factor because it assumes the normalization $\sum_d \lambda_d = 1$. If we introduce a normalization of the neural network output, $\sum_d \lambda_d = A$ and, correspondingly, we were to assume data was

contrast-normalized, then the neural network's loss function would indeed take the form of a cross-entropy weighted by the truncated variational distributions q^n_Φ.

Appendix E

Truncated Variational Autoencoders with Bernoulli noise

As an empirical verification that our training scheme is easily adapted to noise models other than Gaussian, we trained a TVAE model with Bernoulli noise on dynamically binarized MNIST (see e.g. Rolfe, 2016; Sadeghi et al., 2019). For each image in a batch, we sampled binary pixel values from Bernoulli distributions with parameters equal to the original pixel values. The 60000 original images in the training set were used for training, while the test set, as usual, consists of another 10000 images. The exact generative model is:

$$\begin{aligned} p_\Theta(\vec{s}) &= \mathrm{Bern}(\vec{s}; \vec{\pi}) = \prod_h \pi_h^{s_h} (1-\pi_h)^{(1-s_h)} \\ p_\Theta(\vec{y}^n \mid \vec{s}) &= \mathrm{Bern}\left(\vec{y}^n; \vec{\mu}_\Theta(\vec{s})\right) \end{aligned} \tag{E.1}$$

where $\vec{\mu}_\Theta$ is a neural network with the same architecture as in Fig. 5.3 except that the output non-linearity is a sigmoid function to guarantee an output between zero and one.

Although we have not performed extensive benchmarking or hyperparameter search on this dataset, Bernoulli TVAE trained with the settings of Tab. E.1 yields a best ELBO value of -99.63 nats on the test set after 500 epochs of training and running the evolutionary search for variational parameters on the test set until ELBO converged sufficiently. This result is in line with what e.g. Maddison et al. (2016) reports (for a statically binarized version of this data) for simpler model architectures, although the ELBO value is lower than the best reported in that work. It is to be noted however that this value is a strict lower bound of the true log-likelihood, not a sampling approximation like what it is typically reported.

We also trained on the standard static binarization of the MNIST dataset of Salakhutdinov and Murray (2008) like it is done e.g. in Maddison et al.

(2016). Fig. E.1 shows example datapoints as well as samples from the trained Bernoulli-noise TVAE.

These results are to be considered preliminary and are not indicative of the best possible TVAE performance on this task.

Table E.1: Hyperparameters for training Bernoulli TVAE on dynamically binarized MNIST.

Neural network units	
Input	200
Middle	200
Output	784
Learning Rate Schedule	
Min l.r.	0.0001
Max l.r.	0.001
Batch size	128
EEM parameters	
Parents	8
Children	7
Generations	2
Size of $\Phi^{(n)}$	200

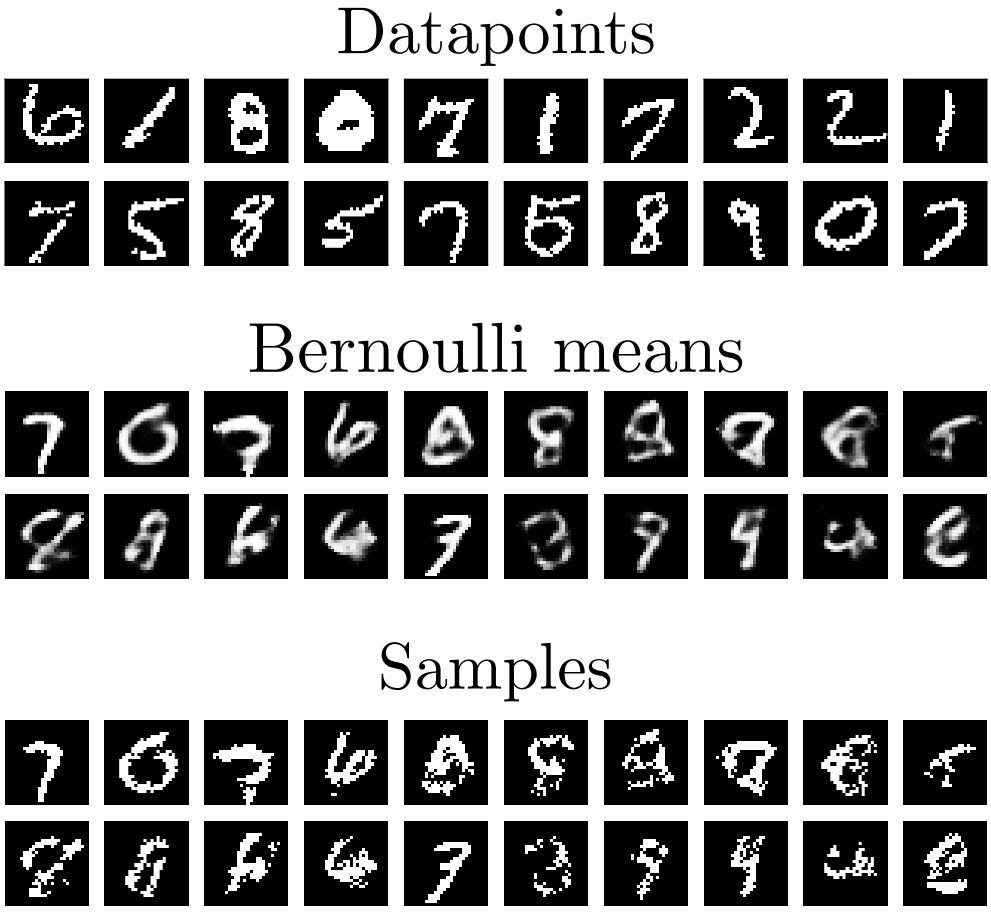

Figure E.1: **Top**: example datapoints. **Middle**: Bernoulli means for 20 random latent states. **Bottom**: 20 generated samples corresponding to the Bernoulli means reported above.

Appendix F

Performance profiling of Truncated Variational Autoencoders Denoising Application

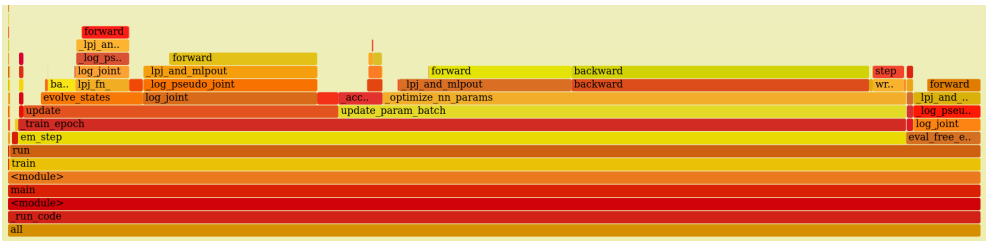

Figure F.1: Flamegraph of the denoising experiment with TVAE described in Sec. 5.4.5.

Figure F.1 reports a "Flamegraph" visualization for the denoising experiments with the TVAE model described in Sec. 5.4.5. A flamegraph shows call stacks on the y axis (e.g., `em_step` calls `_train_epoch` which calls `update` and `update_param_batch`, etc.) and the length of each block on the x axis is proportional to the time the program spent in that function.

For this model and application, it can be observed that around 30% of the runtime is spent in the `update` method, which implements the E-step (it updates the variational parameters Φ^n and around 60% is spent in `update_param_batch`, which is responsible for the M-step. Both of these methods actually end up spending most of their time in the `forward` and `backward` methods, which respectively implement a neural network forward pass and backpropagation pass. This is the best case scenario, as it appears that logic specific to our

EEM algorithm is not a performance bottleneck except for the extra `log_joint` calls that it naturally involves.

Bibliography

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *Signal Processing, IEEE Transactions on*, 54(11):4311–22, 2006.

K. Ahmadi and E. Salari. Single-image super resolution using evolutionary sparse coding technique. *IET Image Processing*, 11(1):13–21, 2016.

J. Albrecht, A. A. Alves, G. Amadio, G. Andronico, N. Anh-Ky, L. Aphecetche, J. Apostolakis, M. Asai, L. Atzori, M. Babik, et al. A Roadmap for HEP Software and Computing R&D for the 2020s. *Computing and software for big science*, 3(1):7, 2019.

A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P. M. Vila, L. Moneta, D. Piparo, E. Tejedor, and X. V. Pla. Novel functional and distributed approaches to data analysis available in ROOT. *Journal of Physics: Conference Series*, 1085(4):042008, 2018.

G. Amadio, P. Canal, E. Guiraud, and D. Piparo. Writing ROOT Data in Parallel with TBufferMerger. In *EPJ Web of Conferences*, volume 214, page 05037. EDP Sciences, 2019.

J. An and S. Cho. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2(1), 2015.

G. Apollinari, I. Béjar Alonso, O. Brüning, P. Fessia, M. Lamont, L. Rossi, and L. Tavian. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1.* CERN Yellow Reports: Monographs. CERN, Geneva, 2017.

V. Avati, M. Blaszkiewicz, E. Bocchi, L. Canali, D. Castro, J. Cervantes, L. Grzanka, E. Guiraud, J. Kaspar, P. Kothuri, et al. Declarative Big Data Analysis for High-Energy Physics: TOTEM Use Case. In *European Conference on Parallel Processing*, pages 241–255. Springer, 2019.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13 (2):31–39, 2011.

A. J. Bell and T. J. Sejnowski. The "independent components" of natural scenes are edge filters. *Vision Research*, 37(23):3327–38, 1997.

Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

J. Blomer. A quantitative review of data formats for hep analyses. *Journal of Physics: Conference Series*, 1085(3):032020, 2018.

M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

J. Bornschein, M. Henniges, and J. Lücke. Are v1 simple cells optimized for visual occlusions? a comparative study. *PLoS Comput Biol*, 9(6):e1003062, 2013.

S. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating sentences from a continuous space. In *Proceedings of the Twentieth Conference on Computational Natural Language Learning (CoNLL).*, 2016.

R. Brun and F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *Proceedings AIHENP '96 Workshop*. Nuclear Instruments and Methods in Physics, 1997.

Q. D. Buchlak, N. Esmaili, J.-C. Leveque, F. Farrokhi, C. Bennett, M. Piccardi, and R. K. Sethi. Machine learning applications to clinical decision support in neurosurgery: an artificial intelligence augmented systematic review. *Neurosurgical review*, pages 1–19, 2019.

D. L. Buckeridge, H. Burkom, M. Campbell, W. R. Hogan, A. W. Moore, et al. Algorithms for rapid outbreak detection: a research synthesis. *Journal of biomedical informatics*, 38(2):99–113, 2005.

M. Buhl, A. Warzybok, M. R. Schädler, O. Majdani, and B. Kollmeier. Common audiological functional parameters (cafpas) for single patient cases: deriving statistical models from an expert-labelled data set. *International Journal of Audiology*, pages 1–14, 2020.

Y. Burda, R. Grosse, and R. Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

H. C. Burger, C. J. Schuler, and S. Harmeling. Image denoising: Can plain neural networks compete with bm3d? In *2012 IEEE conference on computer vision and pattern recognition*, pages 2392–2399. IEEE, 2012.

S. Chaudhury and H. Roy. Can fully convolutional networks perform well for general image restoration problems? In *International Conference on Machine Vision Applications*, pages 254–257, 2017.

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

N. Coudray, P. S. Ocampo, T. Sakellaropoulos, N. Narula, M. Snuderl, D. Fenyö, A. L. Moreira, N. Razavian, and A. Tsirigos. Classification and mutation prediction from non–small cell lung cancer histopathology images using deep learning. *Nature medicine*, 24(10):1559–1567, 2018.

C. Cremer, X. Li, and D. Duvenaud. Inference suboptimality in variational autoencoders. In *International Conference on Machine Learning*, pages 1078–1086, 2018.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society B*, 39:1–38, 1977.

W. U. Dharmaji. xAOD-DataSource: An implementation of ROOT's RDataSource interface for reading xAOD files through RDataFrame interface., July 2018.

C. Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

W. Dong, P. Wang, W. Yin, G. Shi, F. Wu, and X. Lu. Denoising Prior Driven Deep Neural Network for Image Restoration. *TPAMI*, 2019.

J. Drefs, E. Guiraud, and J. Lücke. Evolutionary variational optimization of generative models. *Under review*, 2020.

G. Eulisse, P. Konopka, M. Krzewicki, M. Richter, D. Rohr, and S. Wenzel. Evolution of the ALICE Software Framework for Run 3. *EPJ Web Conf.*, 214:05010, 2019. doi: 10.1051/epjconf/201921405010.

G. Exarchakis and J. Lücke. Discrete Sparse Coding. *Neural Computation*, 29:2979–3013, 2017.

D. Faggella. Machine learning in finance – present and future applications. *Mode of access: https://www.techemergence.com/machine-learning-in-finance, last visited 30-08-2020.*

J. A. Fernandez, C. Adolphsen, A. Akay, H. Aksakal, J. Albacete, S. Alekhin, P. Allport, V. Andreev, R. Appleby, E. Arikan, et al. A large hadron electron collider at cernreport on the physics and design concepts for machine and detector. *Journal of Physics G: Nuclear and Particle Physics*, 39(7):075001, 2012.

L. J. Fogel, A. J. Owens, and M. J. Walsh. Artificial intelligence through simulated evolution. 1966.

P. Földiák. Forming sparse representations by local anti-Hebbian learning. *Biological Cybernetics*, 64:165–170, 1990.

D. Forster and J. Lücke. Can clustering scale sublinearly with its clusters? A variational EM acceleration of GMMs and k-means. In *AISTATS*, pages 124–132, 2018.

D. Forster, A.-S. Sheikh, and J. Lücke. Neural Simpletrons: Learning in the Limit of Few Labels with Directed Generative Networks. *Neural Computation*, 30(8):2113–2174, 2018.

Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37, 1995.

J. Generowicz, W. T. Lavrijsen, M. Marino, and P. Mato. Reflection-based python-c++ bindings. 2004.

X. Glorot and Y. 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, 2010.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

I. Goodfellow, A. C. Courville, and Y. Bengio. Large-Scale Feature Learning With Spike-and-Slab Sparse Coding. In *ICML*, 2012.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In *NIPS*. 2014.

I. J. Goodfellow, A. Courville, and Y. Bengio. Scaling up spike-and-slab models for unsupervised feature learning. *TPAMI*, 35(8):1902–1914, 2013.

S. Gu, L. Zhang, W. Zuo, and X. Feng. Weighted Nuclear Norm Minimization with Application to Image Denoising. In *CVPR*, pages 2862–2869. IEEE, 2014.

E. Guiraud, J. Bornschein, and J. Lücke. Large-Scale Noisy-OR Networks as Models for Inference and Learning in Neurosensory Systems. In *NIPS-Workshop, Brains and Bits*, 2016.

E. Guiraud, J. Drefs, and J. Lücke. Evolutionary Expectation Maximization. In *GECCO*, 2018.

E. Guiraud, J. Drefs, and J. Lücke. Direct discrete optimization of variational autoencoders with binary latents. *Under review*, 2020.

I. Gulrajani, K. Kumar, F. Ahmed, A. A. Taiga, F. Visin, D. Vazquez, and A. Courville. Pixelvae: A latent variable model for natural images. *arXiv preprint arXiv:1611.05013*, 2016.

M. Haft, R. Hofman, and V. Tresp. Generative binary codes. *Formal Pattern Analysis & Applications*, 6:269–84, 2004.

Y. Halpern and D. Sontag. Unsupervised learning of noisy-or bayesian networks. *arXiv preprint arXiv:1309.6834*, 2013.

D. Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Machine Intelligence and Pattern Recognition*, volume 10, pages 163–171. Elsevier, 1990.

M. Henniges, G. Puertas, J. Bornschein, J. Eggert, and J. Lücke. Binary Sparse Coding. In *LVA/ICA*, pages 450–57. Springer, 2010.

G. E. Hinton, T. J. Sejnowski, T. A. Poggio, et al. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

P. O. Hoyer. Modeling receptive fields with non-negative sparse coding. *Neurocomputing*, 52-54:547–52, 2003.

P. O. Hoyer. Non-negative matrix factorization with sparseness constraints. *Journal of Machine Learning Research*, 5:1457–69, 2004.

E. R. Hruschka, R. J. Campello, A. A. Freitas, et al. A survey of evolutionary algorithms for clustering. *IEEE Transactions on Systems, Man, and Cybernetics*, 39(2):133–155, 2009.

M. C. Hughes and E. B. Sudderth. Fast Learning of Clusters and Topics via Sparse Posteriors. *arXiv preprint arXiv:1609.07521*, 2016.

R. Imamura, T. Itasaka, and M. Okuda. Zero-Shot Hyperspectral Image Denoising With Separable Image Prior. In *IEEE International Conference on Computer Vision Workshop*, 2019.

E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax, 2016.

N. Jojic and B. Frey. Learning flexible sprites in video layers. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2001.

M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37:183–233, 1999.

A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

A. H. Khoshaman and M. Amin. Gumbolt: extending gumbel trick to boltzmann priors. In *Advances in Neural Information Processing Systems*, pages 4061–4070, 2018.

Y. Kim, S. Wiseman, A. Miller, D. Sontag, and A. Rush. Semi-amortized variational autoencoders. In *International Conference on Machine Learning*, pages 2678–2687, 2018.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *ICLR*, 2014.

D. P. Kingma, S. Mohamed, D. Jimenez Rezende, and M. Welling. Semi-supervised learning with deep generative models. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3581–3589. Curran Associates, Inc., 2014.

B. Kiran, D. Thomas, and R. Parakkal. An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos. *Journal of Imaging*, 4(2):36, 2018.

M. A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

A. Krull, T.-O. Buchholz, and F. Jug. Noise2void-learning denoising from single noisy images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2129–2137, 2019.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila. Noise2Noise: Learning image restoration without clean data. In *ICML*, volume 80, pages 2965–2974, 2018.

I. Loshchilov and F. Hutter. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. In *ICLR Workshop*, pages 513–520, 2016.

J. Lücke. Truncated Variational Expectation Maximization. *arXiv preprint, arXiv:1610.03113*, 2019.

J. Lücke and J. Eggert. Expectation Truncation And the Benefits of Preselection in Training Generative Models. *Journal of Machine Learning Research*, 11:2855–900, 2010.

J. Lücke and M. Sahani. Maximal causes for non-linear component extraction. *Journal of Machine Learning Research*, 9:1227–67, 2008.

J. Lücke and A.-S. Sheikh. Closed-Form EM for Sparse Coding and Its Application to Source Separation. In *LVA/ICA*, pages 213–221, 2012.

J. Lücke, R. Turner, M. Sahani, and M. Henniges. Occlusive Components Analysis. *NIPS*, 22:1069–77, 2009.

J. Lücke, Z. Dai, and G. Exarchakis. Truncated Variational Sampling for "Black Box" Optimization of Generative Models. In *LVA/ICA*, pages 467–478, 2018.

L. Maaløe, C. K. Sønderby, S. K. Sønderby, and O. Winther. Auxiliary deep generative models. In *33rd International Conference on Machine Learning (ICML 2016)*, 2016.

C. J. Maddison, A. Mnih, and Y. W. Teh. The concrete distribution: A continuous relaxation of discrete random variables, 2016.

X. Mao, C. Shen, and Y.-B. Yang. Image Restoration Using Very Deep Convolutional Encoder-Decoder Networks with Symmetric Skip Connections. In *NIPS*, pages 2802–2810, 2016.

D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.

W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

M. F. M. Mohsin, A. R. Hamdan, and A. A. Bakar. Review on anomaly detection for outbreak detection. In *International Conference on Information Science and Managemen (ICoCSIM)*, volume 1, pages 22–28, 2012.

S. H. Mousavi, M. Buhl, E. Guiraud, J. Drefs, and J. Lücke. Learning the maximal causes for beta distributed interval data. *In preparation*, 2020.

J. W. Myers, K. B. Laskey, and K. A. DeJong. Learning bayesian networks from incomplete data using evolutionary algorithms. In *GECCO*, 1999.

R. Neal and G. Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In M. I. Jordan, editor, *Learning in Graphical Models*. Kluwer, 1998.

G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.

F. M. F. Nogueira. Bayesian optimization package. `github.com/fmfn/BayesianOptimization`, 2019.

B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, 1997.

M. Opper and C. Archambeau. The variational Gaussian approximation revisited. *Neural computation*, 21(3):786–792, 2009.

M. Opper and O. Winther. Expectation consistent approximate inference. *Journal of Machine Learning Research*, 6(Dec):2177–2204, 2005.

J. Paisley, D. Blei, and M. Jordan. Variational bayesian inference with stochastic search. *arXiv preprint arXiv:1206.6430*, 2012.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

F. Pernkopf and D. Bouchaffra. Genetic-based EM algorithm for learning Gaussian mixture models. *TPAMI*, 27(8):1344–1348, 2005.

D. Piparo, E. Tejedor, E. Guiraud, G. Ganis, P. Mato, L. Moneta, X. V. Pla, and P. Canal. Expressing Parallelism with ROOT. *Journal of Physics: Conference Series*, 898(7):072022, 2017.

D. Piparo, P. Canal, E. Guiraud, X. V. Pla, G. Ganis, G. Amadio, A. Naumann, and E. Tejedor. RDataFrame: Easy Parallel ROOT Analysis at 100 Threads. In *EPJ Web of Conferences*, volume 214, page 06029. EDP Sciences, 2019.

L. Rampasek, D. Hidru, P. Smirnov, B. Haibe-Kains, and A. Goldenberg. Dr.vae: Drug response variational autoencoder, 2017.

E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *ICML*, pages 2902–2911, 2017.

I. Rechenberg. Cybernetic solution path of an experimental problem. 1965.

D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *ICML*, 2014.

D. Ribli, A. Horváth, Z. Unger, P. Pollner, and I. Csabai. Detecting and classifying lesions in mammograms with deep learning. *Scientific reports*, 8 (1):1–7, 2018.

A. Roberts, J. Engel, C. Raffel, C. Hawthorne, and D. Eck. A hierarchical latent vector model for learning long-term structure in music, 2018.

J. T. Rolfe. Discrete variational autoencoders. *arXiv eprint 1609.02200*, 2016.

M. Rotmensch, Y. Halpern, A. Tlimat, S. Horng, and D. Sontag. Learning a Health Knowledge Graph from Electronic Medical Records. *Scientific Reports*, 7(1):5994, 2017.

A. Roy, A. Vaswani, A. Neelakantan, and N. Parmar. Theory and experiments on vector quantized autoencoders. *arXiv preprint arXiv:1805.11063*, 2018.

H. Sadeghi, E. Andriyash, W. Vinci, L. Buffoni, and M. H. Amin. Pixelvae++: Improved pixelvae with discrete prior. *arXiv preprint arXiv:1908.09948*, 2019.

R. Salakhutdinov and I. Murray. On the quantitative analysis of deep belief networks. In *ICML*, pages 872–879, 2008.

T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *arXiv preprint arXiv:1703.03864*, 2017.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

M. Seeger. Bayesian inference and optimal design for the sparse linear model. *Journal of Machine Learning Research*, 9:759–813, 2008.

A. B. R. Shatte, D. M. Hutchinson, and S. J. Teague. Machine learning in mental health: a scoping review of methods and applications. *Psychological Medicine*, 49(9):1426–1448, 2019. doi: 10.1017/S0033291719000151.

A.-S. Sheikh and J. Lücke. Select-and-Sample for Spike-and-Slab Sparse Coding. In *NIPS*, 2016.

A.-S. Sheikh, J. A. Shelton, and J. Lücke. A Truncated EM Approach for Spike-and-Slab Sparse Coding. *Journal of Machine Learning Research*, 15: 2653–2687, 2014.

J. A. Shelton, J. Bornschein, A.-S. Sheikh, P. Berkes, and J. Lücke. Select and Sample – A Model of Efficient Neural Inference and Learning. *NIPS*, 24:2618–2626, 2011.

J. A. Shelton, J. Gasthaus, Z. Dai, J. Lücke, and A. Gretton. GP-select: Accelerating EM using adaptive subspace preselection. *arXiv:1412.3411, now published by Neural Computation 29(8): 2177–2202, 2017*, 2014.

A. Shocher, N. Cohen, and M. Irani. "Zero-Shot" Super-Resolution using Deep Internal Learning. In *CVPR*, pages 3118–3126. IEEE, 2018.

T. Singliar and M. Hauskrecht. Noisy-OR Component Analysis and its Application to Link Analysis. *Journal of Machine Learning Research*, 7:2189–2213, 2006.

L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

M. W. Spratling. Pre-synaptic lateral inhibition provides a better architecture for self-organising neural networks. *Network: Computation in Neural Systems*, 10:285 – 301, 1999.

M. W. Spratling, K. De Meyer, and R. Kompass. Unsupervised learning of overlapping image components using divisive input modulation. *Computational Intelligence and Neuroscience*, pages 1–19, 2009.

K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computing*, 10(2):99–127, 2002.

S. Subramani, H. Wang, H. Q. Vu, and G. Li. Domestic violence crisis identification from facebook posts based on deep learning. *IEEE Access*, 6: 54075–54085, 2018.

M. Suganuma, S. Shirakawa, and T. Nagao. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. In *GECCO*, 2017.

R. Sutton. The bitter lesson. *Incomplete Ideas (blog), March*, 13:12, 2019.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

M. K. Titsias and M. Lázaro-Gredilla. Spike and Slab Variational Inference for Multi-Task and Multiple Kernel Learning. In *NIPS*, 2011.

D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

A. Vahdat, E. Andriyash, and W. G. Macready. Learning undirected posteriors by backpropagation through mcmc updates. *arXiv preprint arXiv:1901.03440*, 2019.

A. van den Oord, O. Vinyals, et al. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pages 6306–6315, 2017.

J. H. van Hateren and A. van der Schaaf. Independent component filters of natural images compared with simple cells in primary visual cortex. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 265: 359–66, 1998.

V. Vasilev, P. Canal, A. Naumann, and P. Russo. Cling – the new interactive interpreter for ROOT 6. In *Journal of Physics: Conference Series*, volume 396, page 052071, 2012.

E. Vértes and M. Sahani. Flexible and accurate inference and learning for deep generative models. In *Advances in Neural Information Processing Systems*, pages 4166–4175, 2018.

T. Šingliar and M. Hauskrecht. Noisy-or component analysis and its application to link analysis. *Journal of Machine Learning Research*, 7:2189–2213, 2006.

Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

W.-K. Wong, A. W. Moore, G. F. Cooper, and M. M. Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 808–815, 2003.

Y. Xue, R. Zhang, Y. Deng, K. Chen, and T. Jiang. A preliminary examination of the diagnostic value of deep learning in hip osteoarthritis. *PLoS One*, 12 (6):e0178992, 2017.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang. Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising. *IEEE Transactions on Image Processing*, 26(7):3142–3155, 2017.

K. Zhang, W. Zuo, and L. Zhang. FFDNet: Toward a fast and flexible solution for CNN based image denoising. *IEEE Transactions on Image Processing*, 2018a.

R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–595, 2018b.

H. Zhao, O. Gallo, I. Frosio, and J. Kautz. Loss functions for image restoration with neural networks. *IEEE Transactions on computational imaging*, 3(1): 47–57, 2016.

L. Zhu and N. Laptev. Deep and confident prediction for time series at uber. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 103–110, 2017.

S. Zhu, G. Xu, Y. Cheng, X. Han, and Z. Wang. BDGAN: Image Blind Denoising Using Generative Adversarial Networks. In *Chinese Conference on Pattern Recognition and Computer Vision*, pages 241–252, 2019.

D. Zoran and Y. Weiss. From Learning Models of Natural Image Patches to Whole Image Restoration. In *IEEE International Conference on Computer Vision*, pages 479–486, 2011.