Data compression and computational efficiency

Thomas Bird

Department of Computer Science

University College London

This dissertation is submitted for the degree of

Doctor of Philosophy

Declaration

I, Thomas Bird, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

In this thesis we seek to make advances towards the goal of effective learned compression. This entails using machine learning models as the core constituent of compression algorithms, rather than hand-crafted components.

To that end, we first describe a new method for lossless compression. This method allows a class of existing machine learning models – latent variable models – to be turned into lossless compressors. Thus many future advancements in the field of latent variable modelling can be leveraged in the field of lossless compression. We demonstrate a proof-of-concept of this method on image compression. Further, we show that it can scale to very large models, and image compression problems which closely resemble the real-world use cases that we seek to tackle.

The use of the above compression method relies on executing a latent variable model. Since these models can be large in size and slow to run, we consider how to mitigate these computational costs. We show that by implementing much of the models using binary precision parameters, rather than floating-point precision, we can still achieve reasonable modelling performance but requiring a fraction of the storage space and execution time.

Lastly, we consider how learned compression can be applied to 3D scene data - a data medium increasing in prevalence, and which can require a significant amount of space. A recently developed class of machine learning models - scene representation functions - has demonstrated good results on modelling such 3D scene data. We show that by compressing these representation functions themselves we can achieve good scene reconstruction with a very small model size.

Impact statement

Given the ever-increasing amount of data generated and transmitted in the world, improvements to compression systems have the potential to be of high utility outside of academia. In this thesis we make steps towards improved learned compression. Since learned methods have come to outperform hand-crafted methods in many computational tasks, it seems likely that the task of compression will also follow this pattern.

Indeed the performance of such learned compression systems is, at the time of writing, already superior in many metrics to more traditional approaches. Some of the main barriers to adoption are now of a more practical nature - is the compressor fast to execute, and small enough in size to be distributed widely? This thesis makes preliminary approaches at answering these questions, which we hope will be useful on the progression of learned compression from a research project to something that is widely used.

The core of the research presented in this thesis has been peer-reviewed and published at machine learning conferences. The work on lossless compression and the use of binary precision parameters is published at three top-tier machine learning conferences [Townsend et al., 2019, 2020, Bird et al., 2021b]. The work on compressing 3D scene data was published at a more specialised compression conference [Bird et al., 2021a], and the corresponding presentation won the Best Presentation award, sponsored by Tencent Media Lab[1].

As such, there is a reasonable footprint of this thesis in the research world, including many methods which build directly on the work presented in this thesis [Kingma et al., 2019, Ho et al., 2019b, Hoogeboom et al., 2019, Kingma et al., 2021].

[1] See `https://pcs2021.org/awards/`

Collaborations and personal contributions

This thesis is, at its core, constructed from 4 separate publications [Townsend et al., 2019, 2020, Bird et al., 2021b,a]. All of these publications are collaborative in nature, to varying degrees, as are many modern machine learning papers. Indeed it wouldn't have been possible to write this thesis without the insights and input from all of my co-authors.

For the first paper presented [Townsend et al., 2019], Jamie Townsend was the primary author and the genesis of the main idea of the paper. My own personal contribution was largely in designing and running the experiments, as well as general contributions towards idea discussion and writing. The second paper Townsend et al. [2020] was a follow-up work, for which we shared primary authorship. This reflects the equal contributions we made towards the paper, although we did have some specialisation in what we worked on. In particular, Jamie did more of the work on vectorised coding, whereas I did more of the work on initialising the bits-back chain.

Both papers are presented in this thesis for completeness, although all content from the original papers which was not created by myself has been re-presented here in my own words and designs.

For the third [Bird et al., 2021b] and fourth [Bird et al., 2021a] papers, I was the primary author, and the main contributor to all aspects of the works.

Contents

List of Figures

List of Tables

Introduction

Sending and storing digital data has become ubiquitous in modern life, and by compressing this data, we can significantly reduce the storage and transmission costs incurred.

There is a wide range of literature and existing approaches to compression. In this thesis, we make progress in the relatively new methods of learned compression - that is, methods which use machine learning at their core rather than hand-crafted components.

The high-level structure of this thesis are the following 4 chapters:

1. *Background*

   We give a brief background on the relevant topics that we will build on in this thesis.

2. *Lossless compression with latent variable models*

   We demonstrate how to use the class of latent variable models as lossless compressors. The core of this method is to combine the relatively old bits-back coding argument [Hinton and van Camp, 1993] with a modern entropy coder, asymmetric numeral systems [Duda, 2009]. This combination overcomes difficulties which had previously prevented the method from being effective, and we demonstrate the resulting compressor practically in a variety of settings.

3. *Binary neural networks for probabilistic generative models*

   Neural networks form the computational backbone of the learned compressors that we are interested in. As such, we show how to effectively make these neural networks much less computationally expensive in terms of space and time requirements. This is done via implementing large parts of the neural networks

using binary weights, which has not been tried before to our knowledge, and requires a variety of innovations to work effectively.

4. *Scene compression*

   We explore how to apply learned compression to a relatively new data domain - that of 3-dimensional data from a scene. We compress a model which can render a scene from any perspective, and to evaluate its performance we construct a more traditional baseline methodology.

We now give a more detailed description of what motivates each chapter, and the contributions contained in each.

We begin in Chapter 2, with contributions towards learned lossless compression. Lossless compression is the often less-studied counterpart to lossy compression, where we impose the restriction that our compression system can incur no reconstruction error. Although lossy compression is generally more prevalent in the world, lossless compression is also widespread and many lossless compressors are ubiquitous (e.g. PNG). Indeed, there are many scenarios where lossless compression is essential. For example compression of executable files, where reconstruction errors from compression may result in program execution failures. Or in the world of scientific imaging, for example astronomical imaging is sensitive to small errors. Consider the case of the closest exoplanet to Earth - Proxima Centauri b. This would resolve to less than one pixel in images from the newly launched James Webb telescope, so clearly errors in the imaging process can affect the discovery or observation of similar celestial bodies.

Our contribution centres on developing methods to use latent variable models to create lossless compressors. Latent variable models are a class of model in which there exist hidden (i.e. latent) variables. Although latent variable models are popular in research and industry, they are often not straightforward to use as lossless compressors. This is due to the fact that in many latent variable models of interest, we cannot calculate a closed form of the marginal distribution over the data (i.e. observed variables). As such, to use entropy coding, we need to compress information about the state of the latent variables as well as the observed variables.

To overcome this difficulty, we consider a method known as bits-back coding which

is a theoretical argument regarding the compression rate of a latent variable model. The only previous attempts to turn bits-back coding into a lossless compressor [Frey, 1997] have suffered from high overheads and consequently sub-optimal compression rate, due to the choice of entropy coder used (arithmetic coding [Witten et al., 1987]). Instead, we show that using a relatively modern entropy coder, asymmetric numeral systems [Duda, 2009], does not suffer from such overheads. As such, it can be used in conjunction with a latent variable model and the bits-back coding method to achieve close to the theoretically optimal compression rate.

We initially demonstrate a proof-of-concept implementation of this, bits-back coding with ANS (BB-ANS), compressor. This uses a relatively small latent variable model and the simple MNIST dataset [LeCun et al., 1998] to demonstrate that we can achieve close to the desired compression rate. We then iterate upon the implementation to show that BB-ANS can have many of the required properties for a powerful, generic compressor.

To improve the BB-ANS implementation, we propose using a hierarchical latent variable model, which have been shown to have good modelling performance [Kingma et al., 2016, Maaløe et al., 2019]. We call the resulting system *hierarchical latent lossless compression*, or HiLLoC. We train the model on a more diverse dataset, ImageNet [Deng et al., 2009], to ensure the model performs well on a range of images. Although ImageNet is a labelled dataset (which we do not require), we use it since it is standard in machine learning research, and has a large number of images (1.3 million) from 1000 classes. We show that the resulting model achieves good compression performance even on unseen datasets[2] - a key requirement for an effective generic compressor. We also propose systems to effectively overcome some of the challenges posed by scaling up the model to a larger, hierarchical latent variable model. In particular, we demonstrate a simple method to mitigate the overheads incurred by BB-ANS at the start of compression. We also speed up our coding implementation such it can run in reasonable speeds on larger data-points that we wish to compress.

Although we investigate how to make the coding process efficient in Chapter 2, we

[2]Although the datasets we test are been collected in a similar manner to ImageNet, or are variants of ImageNet.

did not consider the computational efficiency of the model itself. As such, in Chapter 3 we consider how to improve the computational efficiency of the models, with both the space and speed requirements being factored in. For a learned compression algorithm to be viable to be adopted more widely, it is crucial that the models themselves do not have an excessive space requirement, and that they are fast to run.

There are various methods to improve the space efficiency of machine learning models, such as quantising the weights to a lower precision than floating-point, or compressing the weights using some other codec [Oktay et al., 2020]. In this dissertation we consider the method of quantising the weights down to binary precision - that is each weight being described by just one bit. This has been shown to be surprisingly effective at reducing the space requirement in neural networks [Courbariaux et al., 2015, Hubara et al., 2016, Rastegari et al., 2016, M., 2018, Gu et al., 2018]. Furthermore, the binary precision restriction means that the underlying operations used in neural networks can be implemented using fast versions that exploit the binary nature of the weights - this can increase the speed of the models in addition to the space improvements.

Despite having been shown to be effective for classes of models used for classification, to our knowledge there has been no research into whether using binary weights can be effective for probabilistic generative modelling. Since we use probabilistic generative models for lossless compression, we seek to understand whether they can be effectively implemented with binary weights.

We show that a popular weight normalisation scheme [Salimans and Kingma, 2016], widely used in generative models, has a simple analogue when the weights are restricted to be binary, which we term binary weight normalisation. This scheme is simpler than the binary version of batch normalisation [Ioffe and Szegedy, 2015], and also results in more stable learning. In addition, we motivate just implementing the residual layers [He et al., 2016] of the deep convolutional networks at the core of our generative models using binary weights, since they often account for most of the parameters, and importantly are robust to degradation in the quantisation process.

Taking these methods, we show that two modern and powerful classes of generative models can have large portions implemented using binary weights, reducing their

space requirement, and potentially their run-time, by a large amount. Thus we make progress towards having models that can be used as learned compressors which do not have prohibitive computational requirements.

Lastly, we consider how to implement learned compression in another data domain. In Chapters 2 and 3, we largely considered compressing 2D image data. In Chapter 4 we consider how to compress a collection of images that come from a single 3D scene. We consider the problem of how best to compress a system that allows the user to render the scene from any viewing position that they wish, which is a natural application for technologies such as augmented and virtual reality.

To compress such scene data losslessly is prohibitive, since the ability to render the scene from any angle means theoretically an extremely high number of images (from different angles) need to be recovered by the receiver with no error. So instead we seek to use lossy compression.

There has been a recent surge of developments in the field of learned methods for scene generation [Mildenhall et al., 2019, Sitzmann et al., 2019, Mildenhall et al., 2020], and so we leverage these methods for learned compression. We choose the neural radiance fields, or NeRF, method [Mildenhall et al., 2020], which represents a scene via a neural-network based function, which is trained to render a scene from any angle. The task of lossy compression is thus transformed into the task of compressing this function itself, effectively becoming a task for model compression.

As such, the method we used in Chapter 3 of using binary weights is technically applicable. However, since the models are smaller and without residual components, our methods used in Chapter 3 are less appropriate. As such we use a different model compression technique, which instead penalises the entropy of weights directly in the learning process [Oktay et al., 2020]. Combined with a simple codec to encode quantised versions of the low-entropy weights, the model can be effectively compressed.

Although this method requires decoding of the weights, which adds to the runtime, the decoding is a simple affine scaling per weight so is negligible when compared to the execution costs of the model. As with the previous compression methods presented in this thesis, the primary benefit of this approach is the reduced space required for

storage and reduced transmission costs of the compressed data (represented in this case by the model).

We show that by using this low entropy penalty directly on the NeRF model weights, which we term compressed NeRF (or cNeRF), we can effectively compress scenes down to fractions of their original size without noticeable degradation in their quality. We demonstrate this on a variety of synthetic and real scene data. To understand how cNeRF compares to other approaches, we construct a benchmark that does not use a scene representation function. Although this benchmark does use a modern, learned method (LLFF [Mildenhall et al., 2019]), cNeRF consistently outperforms the benchmark in terms of the rate-distortion trade-off.

We conclude with a review of the contributions of this thesis and potential avenues for future work.

Chapter 1

Background

In this chapter we cover the background material required to understand the contributions made in this thesis. Overall, this thesis is on the effectiveness of learned compression and the computational efficiency of such methods. As such we give background on a range of topics.

Firstly we describe the framework of probabilistic generative modelling. That is, the use of models which approximate the probability distributions of observed data. We then give more detail to the description of latent variable models in particular, a sub-class of model in which some variables are not observed (i.e. latent).

We also describe some of the model architectures used in approaches to probabilistic generative modelling, in particular focusing on those models that we use and iterate on in this thesis. These are models that are primarily designed for visual data, both two and three dimensional, which is the main data type that we test on.

Secondly, we describe the models for 3D scene data that we will utilise in this thesis. These models are defined by their ability to render a scene from an arbitrary viewpoint, thus in some sense representing the scene.

Thirdly, we give background for binary neural networks, a method of making neural networks more computationally efficient. In this thesis we will explore how to implement some of the models we use for compression with binary neural networks.

Lastly, we give background on the field of data compression, which is the topic which the majority of this thesis pertains to.

1.1 The machine learning paradigm and neural networks

This thesis will concern topics within the broader field of machine learning, for which we describe the key concepts that we will use. We also give a description of neural networks, which have become a common methodology in recent years, and which we will be using throughout this thesis. The following descriptions are necessarily brief and are written to give readers who are not familiar with the topics an understanding of some of the common concepts in this thesis. For a far more thorough and concrete introduction to these topics, we recommend reading Mackay [2003], Bishop [2006], Murphy [2022].

Machine learning can concisely be described as constructing a *model* which will *learn* from data how to perform some task of interest. By learning, we mean that the model will be shown data relevant to the task, referred to as the *dataset*, and the model will in some way use this data to improve its performance on the task.

In this thesis we will be concerned with methods which utilise neural networks as the backbone of the model. Neural networks can be thought of, at a high level, as function approximators, with their naming owing to the crude resemblance to neural structures observed within the brain.

A key aspect of neural networks is how they are trained[1]. Many popular training methods are based on *gradient descent*, which relies on the model being differentiable. By this we mean that gradients of the neural network with respect to some *loss* function parameters exist, and can be calculated efficiently by automatic differentiation. The loss function is some quantity that we wish to minimise as a proxy for improving the performance on the task of interest. Thus we can evaluate the gradients of the model parameters on our dataset, and update the parameters to move in the direction of steepest descent for the loss. Since datasets can be large, the gradients on the full dataset often cannot be evaluated, so instead they are evaluated on subsets of the dataset, referred to as *batches* or *mini-batches*[2].

[1] Note that "training" a model is analogous to a model "learning".

[2] Although these two terms are used somewhat interchangeably, we will use the term batch in this thesis.

1.2 Notation

Before the more technical sections of this thesis, we briefly lay out the notation that will be used.

- We will use bold for vectors, e.g. $\mathbf{x}$, as compared to scalars x. In general we will describe variables as vectors when their dimension is unspecified.

- We will take logarithms to be base 2, such that the corresponding unit of information is bits, so we abbreviate log to mean $\log_2$.

- In general we use the convention of using the characters $\mathbf{x}$ for data, $\mathbf{y}$ for labels and $\mathbf{z}$ for latents. The characters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ always refer to model parameters, although we also use other characters for model parameters where required.

- We will use p, q to denote density/mass functions. To avoid cluttered notation, the context of which density function we are specifying is made clear by the argument. For example, $p(\mathbf{z})$ refers to the density over the latents $\mathbf{z}$, whereas $p(\mathbf{x})$ refers to the density of the data. Despite the fact that we use the same function notation in both cases, namely $p(\cdot)$, the actual density function is different.

- We will often subscript functions and probability distributions, for example $p_{\boldsymbol{\theta}}$. This indicates that the $\boldsymbol{\theta}$ are the parameters which are used to specify p (or whichever function/distribution is subscripted).

- At some points we will wish to make the clear the difference between a sample from a distribution and the random variable name itself, in which case we will use 0 as a superscript to denote a sample. For example, $\mathbf{z} \sim p(\mathbf{z})$ defines a random variable $\mathbf{z}$ which is distributed according to $p(\mathbf{z})$, and $\mathbf{z}^0$ would be a sample of this random variable. We will use subscripts to identify data points within a dataset, and to identify layers within a hierarchy of layers, which should be identifiable from the context of their usage.

1.3 Probabilistic generative modelling

Probabilistic generative modelling is, broadly, the task of constructing a probability distribution $q(\mathbf{x})$ to fit to some distribution of interest $p(\mathbf{x})$. Usually, we do not have access to p itself, but rather just samples from it $\{\mathbf{x}_n\}_{n=1}^N$ where $\mathbf{x}_n \sim p(\mathbf{x})$. So in practice, since we normally do not have access to p, our task is to construct q such that it fits the dataset $\{\mathbf{x}_n\}_{n=1}^N$ well, in some sense to be defined.

We refer to the resulting models as probabilistic generative models (PGMs) since they are clearly probabilistic (we are constructing a probability distribution), and they are *generative* in the sense that they can be used to generate synthetic data by sampling from $q(\mathbf{x})$. The probabilistic nature differentiates PGMs from non-probabilistic approaches in generative modelling, such as GANs [Goodfellow et al., 2014] which can only generate samples from the distribution $q(\mathbf{x})$, but cannot evaluate the density itself. The generative nature differentiates them from *discriminative* approaches which seek to model the conditional probability $p(y|\mathbf{x})$ of some label y. Discriminative approaches have no method to generate synthetic data, and are instead only interested in the dependence of the labels on the data.

We also narrow the domain of interest for this thesis to PGMs for which we have a functional form for the density (or each conditional density in a structured model), and that the density is normalised. This precludes energy-based models in which the density is not normalised. Although it is possible in some circumstances to evaluate densities of such models, it is not necessarily possible in the general case, and as such our methods that we will use in this thesis may not be applicable.

1.3.1 The data domain

We now give some relevant properties of the types of data that we will be modelling in this thesis.

The data $\mathbf{x}$ can take values in some set $\mathcal{X}$, formally known as the sample space. For example, a 6-sided die has $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$, and a 64×64 image with 8-bit colours is in $\mathcal{X} = \{0, 1, ..., 255\}^{3\times64\times64}$. We will make the assumption that the sample space is discrete, since any continuous data can be taken to be discrete and

high-precision to approximate it well. This is common in computing, where only discrete values can be stored.

In this thesis we will be using PGMs to model image data, and the architectures we use are generally specific to the image data type. However, the general methods, for example the graphical models and application of lossless compression using PGMs, are applicable to other data types such as audio or text.

1.3.2 Classes of PGMs

We can divide probabilistic generative models (PGM) into two classes - *fully observed* models, and *latent variable* models. The difference between the two is defined by whether they contain latent variables, which, as the name implies, are variables that are not observed. We may simply be constructing such variables in the modelling process, or such latent variables may truly exist but not be observable to us as modellers.

We now describe both fully observed and latent variable models - giving their key properties, an example of such a model and the most common methodologies to train them.

1.3.2.1 Fully observed models

A fully observed model is one in which there are no latent/unobserved variables. In the PGM setting, we can denote the model therefore by $p_{\boldsymbol{\theta}}(x)$ where $\boldsymbol{\theta}$ are the model parameters.

As an example, consider modelling the number of cars observed passing a set of traffic lights in an hour. We may choose to model this variable with a Poisson distribution, which requires us to choose one free parameter - known as the rate. Since we have only one variable which we can observe (in this case to literally record how many cars pass the traffic lights in an hour), our model is indeed fully observed.

To train such fully observed PGMs there are a variety of techniques. A widely used, straightforward method, is that of *maximum likelihood*. The likelihood refers to the likelihood of a set of model parameters given the data, i.e. $p_{\boldsymbol{\theta}}(\{\mathbf{x}_n\}_{n=1}^N)$. Assuming the data is drawn independently the likelihood becomes $\prod_{n=1}^N p_{\boldsymbol{\theta}}(\mathbf{x}_n)$.

Thus by maximising this likelihood we are encouraging our model parameters to be appropriate for the data, in a sense defined by the likelihood function.

It is common to use the log-likelihood instead of the likelihood, which mitigates some numerical issues, and admits simpler computation of derivatives in many cases. Thus the objective becomes $\log p_{\boldsymbol{\theta}}(\{\mathbf{x}_n\}_{n=1}^N)) = \sum_{n=1}^N \log p_{\boldsymbol{\theta}}(\mathbf{x}_n)$. Since the log is a strictly monotonically increasing function, if we find a parameter setting which is a maximum of the log-likelihood function, it will be a corresponding maximum of the likelihood function.

1.3.3 Latent variable models

The counterpart to fully observed models, as described in Section 1.3.2.1, are latent variable models, in which there is at least one latent variable $\mathbf{z}$ - as well as the observed variable(s) $\mathbf{x}$ that we are truly seeking to model. In PGMs in particular, our model now consists of multiple components: the prior over our latent variables $p_{\boldsymbol{\theta}}(\mathbf{z})$, the likelihood conditioned on our data $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ and a posterior over our latents conditioned on our data $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. Given the prior and likelihood, the posterior can theoretically be computed using Bayes' theorem. However, in practice such a computation is intractable, and as such an approximate posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is used. Note that following convention we use $\boldsymbol{\phi}$ to denote the posterior parameters.

As an example of a latent variable model, suppose we wished to model the distribution of rainfall of a tropical country with a monsoon season. In this case the rainfall distribution is likely to be bimodal, since the rainfall is likely to be much higher during monsoon season than other times. As such we may introduce a binary latent variable, which we hope will capture whether it is the monsoon season or not. We can then model the rainfall conditional on this latent variable (for example as a Gaussian with mean and variance different for each setting of the latent).

As with fully observed PGMs, we can use the log-likelihood as the objective for latent variable PGMs also. However, the existence of the latent variables makes the optimisation slightly more complicated. In many cases of interest, we cannot perform the integration to marginalise the latent variables, and performing a numerical integration is not useful since we would like a closed form of the objective to optimise.

To overcome this intractability, we can form a lower bound on the log-likelihood by using Jensen's inequality:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) d\mathbf{z} \tag{1.1}$$

$$\geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] \tag{1.2}$$

Where we have assumed that our variables are continuous for simplicity (for discrete variables we replace the integration with a sum). This lower bound is known as the *evidence lower bound* or ELBO. By maximising this bound, we are maximising a lower bound on the log-likelihood, which serves as a proxy for direct maximum likelihood optimisation.

1.3.4 Model architectures

In this section we will describe the architectures of the probabilistic generative models which we use in this thesis. We include these descriptions here in the background, since they will be used or referred to in multiple chapters. Although the details are specific, the intention is for the reader to refer back to this section later on.

1.3.4.1 Variational autoencoders

The variational autoencoder (VAE) [Kingma and Welling, 2014, Rezende et al., 2014] is a latent variable PGM as per Section 1.3.3. The generative model can be decomposed into the prior on the latent variables $p_{\boldsymbol{\theta}}(\mathbf{z})$ and the likelihood of our data given the latent variables $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. The inference model $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is a variational approximation to the true posterior.

Training is generally performed by maximisation of the ELBO. Note that although the ELBO introduces an expectation over the latent variables $\mathbf{z}$, we can use Monte Carlo integration to form a differentiable estimate to Equation 1.1 with a number of samples from the posterior (even just a single sample). To reduce variance of the resulting gradients, it is also common to use the *reparameterisation trick* [Kingma and Welling, 2014].

Generally the distributions $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ and $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ are implemented using neural networks. In order to parameterise distributions, it is common to model the parameters (or sufficient statistics) of a specific distribution. For example, we may model $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ as a diagonal Normal distribution via two functions $\mu(\mathbf{x})$ and $\sigma^{\mathbf{2}}(\mathbf{x})$ to give $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}), \sigma^{\mathbf{2}}(\mathbf{x})I)$.

1.3.4.2 Hierarchical VAEs

To give a more expressive model, the latent space can be structured into a hierarchy of latent variables $\mathbf{z}_{1:L}$. In the generative model each latent layer is conditioned on deeper latents $p_{\boldsymbol{\theta}}(\mathbf{z}_i|\mathbf{z}_{i+1:L})$. A common problem with hierarchical VAEs is that the deeper latents can struggle to learn, often *collapsing* such that the layer posterior matches the prior: $q_{\boldsymbol{\phi}}(\mathbf{z}_i|\mathbf{z}_{i+1:L}, \mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z}_i|\mathbf{z}_{i+1:L})$[3]. One method to help prevent posterior collapse is to use skip connections between latent layers [Kingma et al., 2016, Maaløe et al., 2019], turning the layers into residual layers [He et al., 2016]. Note that a skip connection refers to adding the input value to a layer to the output value of the layer (thus one component of the sum has "skipped' the layer). Blocks of residual layers are referred to as residual blocks.

An example of a hierarchical VAE that we use in this thesis is the ResNet VAE (RVAE) [Kingma et al., 2016]. In this model, both the generative and inference model structure their layers as residual layers. The ResNet VAE uses a bi-directional inference structure with both a bottom-up and top-down residual channel. This is a similar structure to the BIVA model [Maaløe et al., 2019], which has demonstrated state-of-the-art results for a latent variable model.

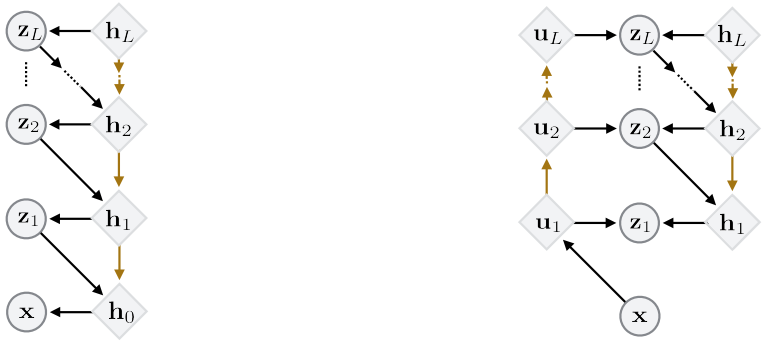
The generative model factors as:

$$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_{1:L}) = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}) p_{\boldsymbol{\theta}}(\mathbf{z}_L) \prod_{l=1}^{L-1} p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}) \tag{1.3}$$

The inference model is factored top-down:

$$q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z}_L|\mathbf{x}) \prod_{l=1}^{L-1} q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}, \mathbf{x}) \tag{1.4}$$

[3]We have assumed here that the inference model is factored *top-down*.



(a) Generative model (b) Bi-directional inference model

Figure 1.1: Graphical models of the generative and inference models in a hierarchical VAE with bi-directional inference. Stochastic nodes are circular, deterministic nodes are diamond. Brown lines indicate residual layers.

There is also a deterministic upwards pass (through the latent layers) performed in the inference model, which produces features used by the posterior, conditioned on just $\mathbf{x}$. We refer to the inference model as bidirectional, since there is both an upwards and downwards pass to be performed. The full graphical model is shown in Figure 1.1.

The objective is obtained by expanding the usual ELBO:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L})\right] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}_L|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}_L)) \tag{1.5}$$

$$- \sum_{l=1}^{L-1} D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}, \mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L})) \tag{1.6}$$

Where D_{KL} is the KL divergence. Both the prior and posterior for a latent layer are factorised when conditioned on deeper layers. Generally the distributions on the latent layers are Gaussian, although it is also possible to use logistic distributions.

In Figure 1.1 the residual connections are displayed in green, with the non-residual connections in black. The non-residual connections are convolutional layers [LeCun et al., 1989] with ELU activation functions [Clevert et al., 2016]. The residual connections are made from stacks of residual blocks. Each residual block is constructed as:

$$\text{Input} \rightarrow \text{Activation} \rightarrow \text{Conv2D}_{\text{3x3}} \rightarrow \text{Activation} \rightarrow \text{Conv2D}_{\text{3x3}} \tag{1.7}$$

With a skip connection adding the output to the input. 2D refers to the dimension of the convolution, and 3×3 refers to the shape of the convolution kernel.

1.3.4.3 Flow models

Flow models consist of a parameterised invertible transformation, $\mathbf{z} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})$, and a known density $p_{\mathbf{z}}(\mathbf{z})$ usually taken to be a unit normal distribution. Given observed data $\mathbf{x}$ we obtain the objective for $\boldsymbol{\theta}$ by applying a change-of-variables to the log-likelihood:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\mathbf{z}}(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})) + \log \left| \det \frac{d\mathbf{f}_{\boldsymbol{\theta}}}{d\mathbf{x}} \right| \tag{1.8}$$

For training to be possible, it is required that computation of the Jacobian determinant $\det(d\mathbf{f}_{\boldsymbol{\theta}}/d\mathbf{x})$ is tractable. We therefore aim to specify a flow model $\mathbf{f}_{\boldsymbol{\theta}}$ which is sufficiently flexible to model the data distribution well, whilst also being invertible and having a tractable Jacobian determinant. One common approach is to construct $\mathbf{f}_{\boldsymbol{\theta}}$ as a composition of many simpler functions: $\mathbf{f}_{\boldsymbol{\theta}} = \mathbf{f}_1 \circ \mathbf{f}_2 \circ ... \circ \mathbf{f}_L$, with each $\mathbf{f}_i$ invertible and with tractable Jacobian. So the objective becomes:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\mathbf{z}}(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})) + \sum_{i=1}^{L} \log \left| \det \frac{d\mathbf{f}_i}{d\mathbf{f}_{i-1}} \right| \tag{1.9}$$

There are many approaches to construct the $\mathbf{f}_i$ layers [Dinh et al., 2015, Rezende and Mohamed, 2015, Dinh et al., 2017, Kingma and Dhariwal, 2018, Ho et al., 2019a]. In this thesis we will use the Flow++ model [Ho et al., 2019a]. In the Flow++ model, the $\mathbf{f}_i$ are coupling layers which partition the input into $\mathbf{x}_1$ and $\mathbf{x}_2$, then transform only $\mathbf{x}_2$:

$$\mathbf{f}_i(\mathbf{x}_1) = \mathbf{x}_1, \quad \mathbf{f}_i(\mathbf{x}_2) = \sigma^{-1}\big(\text{MixLogCDF}(\mathbf{x}_2; \mathbf{t}(\mathbf{x}_1))\big) \cdot \exp(\mathbf{a}(\mathbf{x}_1)) + \mathbf{b}(\mathbf{x}_1) \tag{1.10}$$

Where MixLogCDF is the CDF for a mixture of logistic distributions. This is an iteration on the affine coupling layer [Dinh et al., 2015, 2017]. Note that keeping part of the input fixed ensures that the layer is invertible. To ensure that all dimensions are transformed in the composition, adjacent coupling layers will keep different parts of the input fixed, often using an alternating checkerboard or stripe pattern to choose

the fixed dimensions [Dinh et al., 2017]. The majority of parameters in this flow model come from the functions $\mathbf{t}, \mathbf{a}$ and $\mathbf{b}$ in the coupling layer, and in Flow++ these are parameterised as stacks of convolutional residual layers, with a convolution layer before and after the stack to project to and from the channel size of the residual stack. Each block is of the form:

$$\text{Input} \rightarrow \text{Activation} \rightarrow \text{Conv2D}_{3\text{x}3} \rightarrow \text{Activation} \rightarrow \text{Gate} \tag{1.11}$$

Where Gate is a 1×1 convolution followed by a gated linear unit [Dauphin et al., 2017]. There is a skip connection adding the input to the output, along with layer normalisation [Ba et al., 2016].

Our data is generally discrete, so we actually require a discrete distribution, not a continuous density. To allow this, the Flow++ model uses *variational dequantisation*. Suppose that the data is in $[0, 1, ..., 255]^D$. We can get a discrete distribution from a continuous density by integrating over the D-dimensional unit hypercube:

$$P_{\boldsymbol{\theta}}(\mathbf{x}) = \int_{[0,1)^D} p_{\boldsymbol{\theta}}(\mathbf{x} + \mathbf{u}) d\mathbf{u} \tag{1.12}$$

Variational dequantisation then proceeds by forming a lower-bound to this discrete distribution by applying Jensen's inequality:

$$\log P_{\boldsymbol{\theta}}(\mathbf{x}) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{u}|\mathbf{x})} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x} + \mathbf{u}) - \log q_{\boldsymbol{\phi}}(\mathbf{u}|\mathbf{x}) \right] \tag{1.13}$$

Where $q_{\boldsymbol{\phi}}(\mathbf{u}|\mathbf{x})$ is now a learned component, which *dequantises* the discrete data. This is itself parameterised as a flow, using a composition of coupling layers as above. So our model in total consists of a main flow $p_{\boldsymbol{\theta}}(\mathbf{x})$ and a dequantising flow $q_{\boldsymbol{\phi}}(\mathbf{u}|\mathbf{x})$.

1.4 Compression

In this section we describe the fundamentals of compression that we will utilise later when we describe learned compression methods.

1.4.1 Lossless compression

The task of lossless compression is to communicate data via a message, with as short a message length as possible, such that the decoded message is identical to the original data. We imagine that the communication occurs between the *sender*, who has the data, and the *receiver* who is in receipt of the message, and must reconstruct the original data. The pair of *coder* and *decoder*, which map from data to message and message to data respectively, are known as a *codec*. It is common to refer to the encoded version of each element of the data as a *codeword*.

We will restrict our discussion of lossless compression to the class of methods referred to as *entropy coders*. Entropy coders compress data $\mathbf{x}$ from a finite state space $\mathcal{X}$[4]. The compression is accomplished by using some mass function $q(\mathbf{x})$, which can possibly be conditioned on prior data points in a sequence if such a sequence exists. The mass function assigns probabilities (i.e. masses) to each element of the sample space, and the fundamental mechanism of entropy coders is to assign shorter codewords to more likely elements of data.

To be more precise, let us denote the mass function of the true distribution which generated the data as $p(\mathbf{x})$, which may be different from our mass function $q(\mathbf{x})$. Entropy coders assign a codeword of length $-\log q(\mathbf{x})$ for a given $\mathbf{x}$, thus the expected length of our codeword is $\mathbb{E}_{p(\mathbf{x})}[-\log q(\mathbf{x})]$. By Gibbs' inequality, the shortest expected codelength is achieved when $q(\mathbf{x}) = p(\mathbf{x})$, at which point the expected codelength is the entropy of the data $H(p) = \mathbb{E}_{p(\mathbf{x})}[-\log p(\mathbf{x})]$. When $q(\mathbf{x})$ is not equal to $p(\mathbf{x})$, the extra expected codelength over the entropy of the data is the KL-divergence between p and q: $\mathbb{E}_{p(\mathbf{x})}[-\log q(\mathbf{x})] = H(p) + D_{\mathrm{KL}}(p(\mathbf{x}) \,\|\, q(\mathbf{x}))$.

Note that although we cannot have shorter codelengths than the entropy of the data *on average*, which is stated formally by Shannon's source coding theorem [Shannon, 1948], we can have shorter codelengths for a given data point, i.e. it is possible that $-\log q(\mathbf{x}) < -\log p(\mathbf{x})$.

The main entropy coders we will discuss in this thesis are *arithmetic coding* [Witten et al., 1987] and *asymmetric numeral systems* [Duda, 2009], since they are

[4]In the compression literature the data is often referred to as symbols, and the sample space as an alphabet.

the most useful for the modelling situation which we are in. There are a wealth of other coders that may be of use in different circumstances, for example prefix codes such as Huffman coding [Huffman, 1952] and Golomb codes [Golomb, 1966], or universal codes such as Elias gamma coding [Elias, 1975]. Note that the requirement of a mass function distinguishes entropy coders from other lossless compressors such as dictionary coders which rely on replacing runs of symbols which have already occurred in the data with references to the previous occurrences[5]. Examples of dictionary coders include the widely used LZ77 [Ziv and Lempel, 1977], LZ78 [Ziv and Lempel, 1978] and LZW [Welch, 1984] algorithms which form the backbone of the popular gzip [Gailly and Adler, 1992] compressor.

1.4.2 Lossy compression

When performing *lossy* compression, we do not operate under the restriction that the receiver has to have zero reconstruction error when decoding compressed data. As such, it is less obvious to define the utility of lossy codecs as compared to lossless codecs.

In lossless compression, we can simply seek methods which can compress data into as small a message as possible[6], since the reconstruction error is always zero. For lossy compression we instead have to consider how much reconstruction error there is for a given compressed size of the data (which we refer to as the *rate*). We expect that as we allow the rate to increase (possibly by altering the hyperparameters of the codec in some way) then we expect the reconstruction error (or *distortion*) to decrease. This is known as the rate-distortion trade-off.

The desired balance between the rate and distortion terms can be expressed by some weighting parameter λ. As such, it is common to define the learning objective for a lossy compressor as:

$$\mathcal{L}(\boldsymbol{\theta}) = D(\boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta}) \tag{1.14}$$

[5]There are also dictionary coders that rely on using a pre-computed dictionary, rather than calculating the dictionary on-the-fly. However, they are less common and we do not include them in this brief discussion for simplicity.

[6]Although there are of course other practical considerations, such as the computational demands of the codec.

Where $D(\boldsymbol{\theta})$ is the distortion for a given set of model parameters $\boldsymbol{\theta}$, for example the mean-squared error $D(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{x}_i - \text{dec}_{\boldsymbol{\theta}}(\text{enc}_{\boldsymbol{\theta}}(\mathbf{x}_i))\|_2^2$, where enc/dec are the encoder and decoder respectively. $R(\boldsymbol{\theta})$ is the rate, for example the mean compressed size of the encoded data.

Chapter 2

Lossless compression with latent variable models

The work presented in this chapter was published in two papers [Townsend et al., 2019, 2020]. These two papers are presented in this single chapter since the second, most recent paper [Townsend et al., 2020] follows on directly from the first [Townsend et al., 2019]. For the first paper [Townsend et al., 2019] my personal contribution to the paper was less than my co-author James Townsend, which is indicated by my position as second author. For the second paper [Townsend et al., 2020], we shared first authorship as our contributions were roughly equal. The concepts and results from both papers are presented here in their entirety, for a complete overview of the method we developed.

In this chapter, we describe a methodology to turn latent variable models into lossless compression algorithms. Previous attempts have incurred an overhead to compression which makes them impractical, and we demonstrate how to avoid this. We then explore the resulting algorithm and show its efficacy on a variety of datasets and a variety of latent variable models.

2.1 Introduction

As discussed in Section 1.4.1, a probabilistic model $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ can be used in conjunction with an entropy coder such as arithmetic coding (AC) or asymmetric numeral systems

(ANS) to perform lossless compression. In the case where can evaluate the normalised probabilities, and where the model is fully observed (i.e. has no latent, or unobserved, variables), the operation is straightforward. The model gives probabilities for discrete data $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ and thus with the use of an entropy coder the data can be compressed using approximately $-\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ bits.

For a latent variable model it is not so obvious how to perform lossless compression using the model and an entropy coder. This is because we do not have direct access to the probability $p_{\boldsymbol{\theta}}(\boldsymbol{x})$, since in most models of interest we cannot perform the integration[1] to calculate the functional form of the marginal probability over the data:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \,|\, \boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z}) dz \tag{2.1}$$

For example using a neural network to calculate the distribution parameters of $p_{\boldsymbol{\theta}}(\boldsymbol{x} \,|\, \boldsymbol{z})$ will usually result in an intractable integral, due to the non-linearities in the neural network.

Note that, as discussed in Section 1.3.3, using numerical integration is not useful since we would have to evaluate the probability of every possible value of $\mathbf{x}$ (to construct the full mass function), and for high-dimensional problems this would be computationally burdensome.

2.2 Bits back coding with ANS

2.2.1 Bits back coding

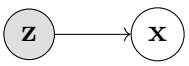


Figure 2.1: Graphical model with latent variable $\boldsymbol{z}$ and observed variable $\boldsymbol{x}$. The latent variable is greyed to indicate that it is not observed.

Bits back coding [Wallace, 1990, Hinton and van Camp, 1993] is an argument pertaining to latent variable models and their application to lossless compression. In

[1] Or if the latent $\boldsymbol{z}$ is discrete, replace integral with sum.

general, it claims that it is possible to use fewer bits than the naive $-\log p_{\boldsymbol{\theta}}(\mathbf{x}^0, \mathbf{z}^0)$ which result from simply entropy coding the data $\mathbf{x}^0$ plus a corresponding latent $\mathbf{z}^0$ according to the joint distribution[2]. We use the superscript 0 to denote that these are samples. We now present a detailed description of the argument.

As in the general case for lossless compression, assume a sender wishes to communicate data $\mathbf{x}^0$ to the receiver, with no reconstruction error. In addition, assume that the sender and receiver both have access to the same latent variable model, which consists of a generative model $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$ and an approximate posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$.

Naively, the sender may select $\mathbf{z}^0$ by some method, and encode both $\mathbf{z}^0$ and $\mathbf{x}^0$ according to the generative model. As discussed in Section 1.4.1, an ideal entropy coder would result in a message length of $-\big(\log p_{\boldsymbol{\theta}}(\mathbf{z}^0) + \log p_{\boldsymbol{\theta}}(\mathbf{x}^0 \mid \mathbf{z}^0)\big)$ bits, ignoring constant overheads. To minimise the message length, the sender should selected the maximum a posteriori (MAP) value $\mathbf{z}^0$ under the posterior $p(\mathbf{z}|\mathbf{x})$.

The receiver could then decode according to the generative model by first decoding $\mathbf{z}^0$ according to $p_{\boldsymbol{\theta}}(\mathbf{z})$ and then decoding $\mathbf{x}^0$ according to $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}^0)$. However, they can do better, and decrease the encoded message length significantly. To see that there is inefficiency here, we note that the receiver could, seeing $\mathbf{x}^0$, find the maximum a posteriori (MAP) value $\mathbf{z}^0$ themselves, so in a sense $\mathbf{z}^0$ has been "sent twice".

To resolve the inefficiency, we suppose that there is some other auxiliary information which the sender would like to communicate to the receiver[3]. We assume the other information takes the form of some random bits. As long as there are sufficiently many bits, the sender can use them to generate a sample $\mathbf{z}^0$ by *decoding* some of the bits to generate a sample from $p_{\boldsymbol{\theta}}(\mathbf{z})$. That is, the sender can treat the random bits as if they were an encoded message and "decode" them - the result is a sample from the distribution used to decode[4].

Generating this sample uses some amount of bits, dependent on the distribution used. The sender can then encode $\mathbf{z}^0$ and $\mathbf{x}^0$ with the generative model, and the

[2]Note we can select the latent to minimise the resulting codelength, which is equivalent to performing MAP estimation

[3]We will discuss the consequences of the absence of this auxiliary information later in the thesis.

[4]This demonstrates the equivalence of sampling and decoding.

message length will be $-\big(\log p_{\boldsymbol{\theta}}(\mathbf{z}^0) + \log p_{\boldsymbol{\theta}}(\mathbf{x}^0 \,|\, \mathbf{z}^0)\big)$ as before. But now the receiver is able to recover the other information, by first decoding $\mathbf{x}^0$ and $\mathbf{z}^0$, and then encoding $\mathbf{z}^0$, reversing the decoding procedure from which the sample $\mathbf{z}^0$ was generated, to get the "bits back".

Note that we can choose any distribution for the sender to sample $\mathbf{z}^0$ from - it does not have to be $p_{\boldsymbol{\theta}}(\mathbf{z})$, and it may vary as a function of $\mathbf{x}^0$. If we generalise and let $q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)$ denote the distribution that we use, possibly depending functionally on $\mathbf{x}^0$, we can write down the expected message length:

$$L(q) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)}\big[-\log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log p_{\boldsymbol{\theta}}(\mathbf{x}^0 \,|\, z) + \log q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)\big] \tag{2.2}$$

$$= -\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)} \log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}^0, \mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)} \tag{2.3}$$

This quantity is equal to the negative of the evidence lower bound (ELBO), sometimes referred to as the *free energy* of the model.

Having recognised this equivalence, it is straightforward to show using Gibbs' inequality that the optimal setting of $q_{\boldsymbol{\phi}}$ is the posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \,|\, \mathbf{x}^0)$, and that with this setting the message length is

$$L_{\text{opt}} = -\log p_{\boldsymbol{\theta}}(\mathbf{x}^0) \tag{2.4}$$

This is the information content of the sample $\mathbf{x}^0$, which by the source coding theorem is the best that we can achieve on average[5]. Thus bits back can achieve an optimal compression rate, if sender and receiver have access to the posterior. In the absence of such a posterior (as is usually the case), then an approximate posterior must be used.

2.2.2 Arithmetic coding versus asymmetric numeral systems

As discussed in Section 1.4.1, arithmetic coding (AC) and asymmetric numeral systems are entropy coders, which allow us to losslessly compress a symbol by utilising a probability distribution over the set of all possible symbols. The resulting

[5]We can assign shorter codelengths than this to a given sample, but it will be at the expense of other samples such that on average it is not beneficial.

message length for both schemes is the information content of the symbols, plus some small overhead. We will now discuss a key difference between AC and ANS which is instrumental in our method in applying latent variable methods for lossless compression.

AC encodes a sequence of symbols into a real number in the interval $[0, 1)$. The way it does this is to partition the interval into n sub-intervals, where n is the number of possible values the symbols can take, i.e. the size of the sample space. The length of each sub-interval is proportional to the probability of the corresponding symbol (possibly conditioned on previous symbols in the sequence). To encode the current symbol the corresponding sub-interval is selected, and then the process is repeated recursively, with the selected sub-interval now being divided up as $[0, 1)$ was divided to code the first symbol. As such, during the coding process, the running encoded sub-sequence is a shrinking interval that is some subset of $[0, 1)$. After all symbols have been encoded, a single number c that lies in the final interval can be chosen and transmitted.

To decode the resulting number from AC encoding, the receiver begins by mirroring the first step of encoding. That is, they divide the interval $[0, 1)$ into n sub-intervals of size proportional to the symbol probabilities. They then observe which sub-interval c lies in, which tells them the *first* symbol that was encoded by the sender. The process is then repeated recursively, much like the encoding process, by dividing the selected sub-interval according to the (conditional) symbol probabilities, observing which sub-interval c relies in and so on. As such, the receiver recovers the symbols in the same order that they were encoded. To borrow from the computer science terminology, we can say that AC behaves like a *queue*, that is first-in-first-out (with regards to symbols being coded and decoded).

We will now give a high-level description of ANS, which does not share this property of behaving like a queue. In particular we will describe the range variant of ANS (rANS). ANS encodes a sequence of symbols into a natural number $s \in \mathbb{N}$, referred to as the *state*. Given a current state (resulting from the encoding of some symbols, or given an initial state), we will describe the process to encode a symbol x and result in a new state s'. Assuming again that our sample space has size n,

we first partition $\mathbb{N}$ into n (infinite) disjoint subsets r_i, one for each symbol. For a random number sampled uniformly on the integers less than some upper bound K, i.e $u \sim U[1, 2, ..., K-1]$, we seek to make $p(u \in r_i) \approx p_i$ as $K \to \infty$.

To encode a symbol x, such that $\mathcal{X}_i = x$, into state s, we update the state s' to be the sth element of range r_i. To decode the symbol x from s' and recover s, we can note that the ranges $\{r_i\}$ partition $\mathbb{N}$, so by identifying which range s' is in, x can be identified. Once x has been obtained, we can simply look up which occurrence x was in r_i, which gives us s.

To give a concrete example, suppose we have a sample space of two symbols $\{a_1, a_2\}$ with respective probabilities 2/3 and 1/3, and we wished to encode the sequence of symbols $a_2a_1a_1$. We would first divide the natural numbers into two disjoint subsets $r_1 = \{1, 2, 4, 5, 7, 8, ...\}$ and $r_2 = \{3, 6, 9, ...\}$, and start with an initial state $s_1 = 1$.

To then encode the first symbol a_2, we would choose the first (since our state is equal to 1) element of r_2 (since our symbol we wish to encode is a_2), which is 3, so we set $s_2 = 3$. Our next symbol is a_1, so we pick the third element of r_1, to get $s_3 = 4$. To encode our final symbol we pick the fourth element of r_1, to get $s_4 = 5$.

The receiver would receive this state $s_4 = 5$, and then lookup that this is in r_1, which tells them the *last* element encoded was a_1. They then calculate that this is the fourth element of r_1, which tells them that the previous state was $s_3 = 4$. They then lookup that 4 is the third element in r_1, so the penultimate element encoded was a_1 and the previous state was $s_2 = 3$. They finally lookup that 3 was the first element of r_2, which tells them that the first element encoded was a_2. They know this is the end of the decoding since they reached a state of 1.

So in summary, we have two operations, one to encode a symbol to the state and return a new state enc : $s, x \to s'$ and one to decode the symbol and recover the old state dec : $s' \to s, x$. Again to borrow terminology from wider computer science, we would say that ANS behaves like a *stack*, and that symbols are encoded last-in-first-out, or LIFO.

Thus we have seen that AC behaves like a queue and ANS like a stack, which we will utilise in the next section. For a far more detailed introduction to arithmetic

coding, see Witten et al. [1987], for asymmetric numeral systems, see Duda [2009].

It is also important to note that we gave a simple example of ANS in action above, but in practice there are many implementation details, which we have omitted for the sake of clarity. For example, to avoid integer overflow it is common to constrain the state to be bounded, and when the state would increase beyond the upper bound to remove and store the least significant bits such the state is within the bounds again. These details do have a small impact on compression performance, but the extend of the overhead is implementation dependent.

Algorithm 1: BB-ANS encode

Data : $\mathcal{D}$
Model : $p_{\boldsymbol{\theta}}(\mathbf{z}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}), q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$
Require: ANS stack with sufficient data encoded
while $\mathcal{D} \neq \emptyset$ **do**
  pick $\mathbf{x}^0 \in \mathcal{D}$;
  decode $\mathbf{z}^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$;
  encode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}^0)$;
  encode $\mathbf{z}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z})$;
  $\mathcal{D} \leftarrow \mathcal{D} \setminus \mathbf{x}^0$;
Send : ANS stack (serialised into bitstream), $N := |\mathcal{D}|$

Algorithm 2: BB-ANS decode

Model : $p_{\boldsymbol{\theta}}(\mathbf{z}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}), q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$
Require: ANS stack with data encoded (deserialised), N
$\mathcal{D} \leftarrow \emptyset$;
for $n = 1$ *to* N **do**
  decode $\mathbf{z}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z})$;
  decode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}^0)$;
  encode $\mathbf{z}^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathbf{x}^0$;
Output : $\mathcal{D}$

2.2.3 Combining bits back coding and ANS

Suppose now that we wish to implement bits-back coding as described in Section 2.2.1 using an entropy coder such as AC or ANS - which one should we use? We will now show why the stack-like nature of ANS is naturally compatible with the bits-back scheme, and admits coding with a minimal overhead.

The first step in bits-back coding of some data $\mathbf{x}^0$ is to decode a latent $\mathbf{z}^0$ according to the posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$, since before the latent is sampled the sender does not have access to the required distribution over the data $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. After the latent is sampled, both $\mathbf{x}^0, \mathbf{z}^0$ must be encoded according to the generative model $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$.

The receiver does not have access to $\mathbf{x}^0, \mathbf{z}^0$, so the only thing they can do is to decode them according to the generative model (which is shared by sender and receiver). Thus we see the requirement that bits-back coding imposes: elements must be decoded in the opposite order to which they were encoded. Frey [1997] noted this and thus that it is not possible to implement bits-back with AC directly, since it is FIFO. His solution was to implement a stack-like wrapper using AC which is LIFO, however the issue is that an overhead of 2 bits must be paid for every iteration. In contrast, ANS can be used to directly implement bits-back coding, since the stack-like nature of ANS means that elements are decoded in the opposite order to which they are encoded naturally.

We give a precise algorithmic definition in Algorithm 1. Since the first step of bits-back encoding is to decode from the stack, we require that the stack already contains some encoded data. Note that this may be random data, if we have not actually previously encoded anything. We then show the decoding process in Algorithm 2. Note that we recover the same encoded data that we begun with, after decoding $\mathbf{x}^0$. We refer to the codec resulting from these encoding and decoding processes as *bits back with ANS*, or BB-ANS.

2.2.4 Chaining BB-ANS

An important observation, first noted in Frey [1997], about bits-back coding is that the encoding of a sequence of symbols can also be done in sequence with the same stack. In Frey [1997], the stack was artificially constructed from the queue-like AC, but in the case of BB-ANS we use the ANS coder, which is naturally stack-like. In other words, once a symbol has been encoded, the resulting ANS stack can then be added to encode the next symbol in the sequence, with the encoding of each symbol following the steps in Algorithm 1 (and correspondingly Algorithm 2 for the decodes). We refer to this process of sequential BB-ANS encodes (and

correspondingly, decodes), as *chaining*.

As noted in Section 2.2.3, BB-ANS has the peculiar property that we require data to already be in the ANS stack in order to encode a symbol, since the first step of encoding is to decode a latent according to the approximate posterior $\mathbf{z}^0 \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$. A major benefit of chaining is that we (generally) only need enough encoded data in the ANS stack such that we can decode a latent *for the first symbol in the sequence*. Once we have encoded the first symbol, the ANS stack will have grown, and we should have enough data in the stack to then encode the next symbol and so on.

One consideration the above argument ignored is that the entropy of the posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$ may change with $\mathbf{x}^0$. It is therefore possible that we may have enough data in the stack to encode the first symbol, but not enough for the second symbol if the posterior is higher entropy under the second symbol than the first. The problem could also occur at some later symbol. In practise we find that this occurs very rarely, since the size of the stack always increases after coding a data point. If it were to occur, one simple method to mitigate it would be to restart compression with a larger initial stack size.

2.3 Practical considerations for BB-ANS

In this section we explore the issues that may prevent BB-ANS from achieving the theoretical compression rate of the negative ELBO.

2.3.1 Initial data in the ANS stack

As discussed in Section 2.2.4, although we can chain BB-ANS to use previously encoded data as the source of auxiliary data for future BB-ANS encoding, we do still require the ANS stack to have some encoded data in it to code the initial symbol. We cannot assume the existence of some auxiliary information that we would also wish to send (such as metadata for images), and so we must treat the data that initially fills the ANS stack as overhead to our compression scheme. It is overhead since the initial data in the stack must also be communicated to the receiver, so adds to the message length. We refer to the size of this initial cost as the *initial bits*, to

convey that they are effectively random (uniform) bits in the ANS stack, required to begin BB-ANS coding.

When using BB-ANS to compress a large dataset, the initial bits can effectively be amortised, and as such will generally be insignificant. However, for a small number of data points, or in the extreme, for a single data point, this cost may be prohibitive. For example, if the latent space is large enough then it could well be that the BB-ANS encoded version of a single data point would be *larger* than the uncompressed data point. This is since, for a single data point, the message length will be equal to the negative log joint $-\log p_{\boldsymbol{\theta}}(\mathbf{x}^0, \mathbf{z}^0)$, which will generally increase as the latent space grows.

There are various ways to mitigate this starting cost of data in the ANS stack. We can optimise for the initial data point(s) by selecting the latents which minimises the message length (rather than decoding from a random ANS stack, which amounts to sampling), and this is equivalent to maximum a posteriori (MAP) estimation.

Another way to reduce the initial bits is to use a different codec to encode a number of data points at the beginning of compressing a sequence. Thus we fill the ANS stack until there is sufficient data encoded to permit BB-ANS encoding. This will result in a net compression rate somewhere between that of the auxiliary codec and BB-ANS. A simple alternative to an actual codec is also to simply fill the ANS stack with some of the raw data, i.e. the first few data points. This way we can ensure BB-ANS is never increasing the data size - in this case it will be at the uncompressed size until we have enough data in the stack to permit BB-ANS encoding.

In practice we are usually concerned with compressing vectors of symbols, rather than individual symbols, for example images. In such cases, another way to mitigate the initial BB-ANS cost is to begin by compressing sub-vectors independently to build up the ANS stack, since smaller vectors generally require a smaller amount of initial bits to begin BB-ANS encoding. Again using the image example, we can compress patches from the images, rather than full images, until we have sufficient data in the ANS stack such that we can code full images.

2.3.2 Non-uniform random bits in the stack

To achieve the negative ELBO as a message length (Equation 2.3) we require that the latent we decode from the ANS stack is a sample from the posterior $\mathbf{z}^0 \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^0)$. The decoded latent is a deterministic function of the bits inside the stack, so the randomness for the sample comes from the randomness of the bits. A sufficient condition to ensure $\mathbf{z}^0$ is a true sample is that the bits inside the ANS stack are independent and uniform distributed (on $\{0, 1\}$). We refer to such bits as *clean*, and if they are not clean, then we refer to them as *dirty*.

At initialisation of BB-ANS, we are free to put clean bits in the stack. During chaining, we effectively use each compressed data point as the seed for the next. Specifically, we use the bits at the top of the ANS stack, which are the result of coding the previous latent $\mathbf{z}^0$ according to the prior $p_{\boldsymbol{\theta}}(\mathbf{z})$. Will these bits be clean? The latent $\mathbf{z}^0$ is originally generated as a sample from $q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)$. This distribution is clearly not equal to the prior, except in degenerate cases, so naively we wouldn't expect encoding $\mathbf{z}^0$ according to the prior to produce clean bits. However, the true sampling distribution of $\mathbf{z}^0$ is in fact the *average* of $q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}^0)$ over the data distribution. That is, $q_{\boldsymbol{\phi}}(\mathbf{z}) \triangleq \int q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}) p(\mathbf{x}) \mathrm{d}\mathbf{x}$. This is referred to by Hoffman and Johnson [2016] as the *average encoding distribution*.

If q is equal to the true posterior, then evidently $q_{\boldsymbol{\phi}}(\mathbf{z}) \equiv p_{\boldsymbol{\theta}}(\mathbf{z})$, however in general this is not the case. Hoffman and Johnson [2016] measure the discrepancy empirically using the *marginal KL divergence* $\mathrm{KL}[q_{\boldsymbol{\phi}}(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z})]$, showing that this quantity contributes significantly to the ELBO for three different VAE like models learned on MNIST. This difference implies that the bits at the top the ANS stack after encoding a sample with BB-ANS will not be perfectly clean, which could adversely impact the coding rate.

When initialising the ANS stack, if we choose to mitigate the initial bits cost using one of the methods discussed in Section 2.3.1 then we may also be starting with dirty bits. For example if we initially fill the stack with uncompressed data (e.g. an image) then these are clearly dirty. One method suggested in Frey [1997], to make the bits cleaner, is to XOR the dirty bits with uniform random bits generated by some PRNG and known seed (such that the receiver can invert the process).

2.3.3 Discretising a continuous latent space

Up til now we have assumed that all variables have been discrete, as they must be to be entropy coded via AC/ANS. This is generally true for our data $\mathbf{x}$, since, as discussed in Section 1.3, we can generally safely make the assumption that the sample space is discrete. However, many latent variable models have latents in a continuous space. For example, the canonical VAE example [Kingma and Welling, 2014] has continuous and Gaussian-distributed latents.

If our model has continuous latents, then we must convert the latents (and corresponding distributions) into discrete analogues in order to be compatible with BB-ANS (or any compression scheme based on entropy coding). We refer to this process as *discretisation*.

In the following we will treat the latent as a scalar z - for a vector $\mathbf{z}$ we can apply the same discretisation scheme but applied independently in each dimension, which is appropriate if the relevant distributions over $\mathbf{z}$ factorise into a product over each dimension. We will also use capital letters for the discretised mass functions we obtain, to make clear the difference to the density over the continuous latent space[6]. We will not discuss the more complicated case, where the distributions do not factorise, since in our models the distributions over the latent space will factorise by design.

The discretisation process can be thought of as partitioning the real line into *buckets* of width δz_i indexed by $i \in I$. We can then convert a density $p_{\boldsymbol{\theta}}(z)$ over the continuous latent space to a mass function by taking the probability of a bucket i to be $P(i) \approx p_{\boldsymbol{\theta}}(z_i)\delta z_i$ where z_i is some point in the bucket, for example its centre-point[7]. This discrete mass function is clearly an approximation of the continuous density, so will introduce some overhead compared to the theoretical message length (using continuous latents). However, for sufficiently small bucket widths, this error can be negligible.

A separate concern is whether the precision (i.e. bucket widths) affects the coding rates independent of the difference in continuous density and discrete mass function.

[6]Note that elsewhere in this thesis we use lower case for all densities/mass functions.
[7]Note that we may have to normalise this to make it a proper distribution.

Naively, we may expect that using a higher precision would increase message length. A straightforward example to motivate this is that the entropy of a discrete uniform distribution is $h = \log N$ where N is the number of elements in the distribution. However, there is a simple argument that shows that in bits-back coding this effect will not occur. In bits-back coding we must discretise a continuous latent space and then apply the prior and posterior over the same set of buckets, since if the buckets differed then a sample from the posterior could not be coded according to the prior (since they would operate over different spaces). If we denote the discretised version of the posterior as Q (which is a distribution over the buckets I), then we can write the expected message length for bits-back as:

$$L \approx -\mathbb{E}_{Q(i \mid \mathbf{x}^0)}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}^0 \mid z_i) p_{\boldsymbol{\theta}}(z_i) \delta z_i}{q_{\boldsymbol{\phi}}(z_i \mid \mathbf{x}^0) \delta z_i}\right]. \tag{2.5}$$

We can see that the δz_i terms cancel, thus using a higher precision (equivalently, smaller buckets) does not affect the message length. It would seem, therefore, that we should wish to use a high precision such that the continuous density and discrete analogue are very similar, but there is one important caveat. Using a high precision requires more bits to draw a sample (again this can be understood by our trivial example about the uniform distribution), so using a high precision means we need more initial bits to start BB-ANS. These initial bits are an overhead, so in reality we would prefer to keep the precision low in order to reduce this overhead. In practice, we seek to strike a balance between having the precision high enough to negate the impact from approximating our continuous densities, but not so high as to impose a prohibitive cost to starting the BB-ANS chain.

2.3.3.1 Choosing the discretisation scheme

We have established that we must discretise any continuous latent spaces in order to proceed with BB-ANS, but what is the best way to assign the buckets? Some important points to note about the discretisation are that:

- The discretisation must be appropriate for the densities that will use it for

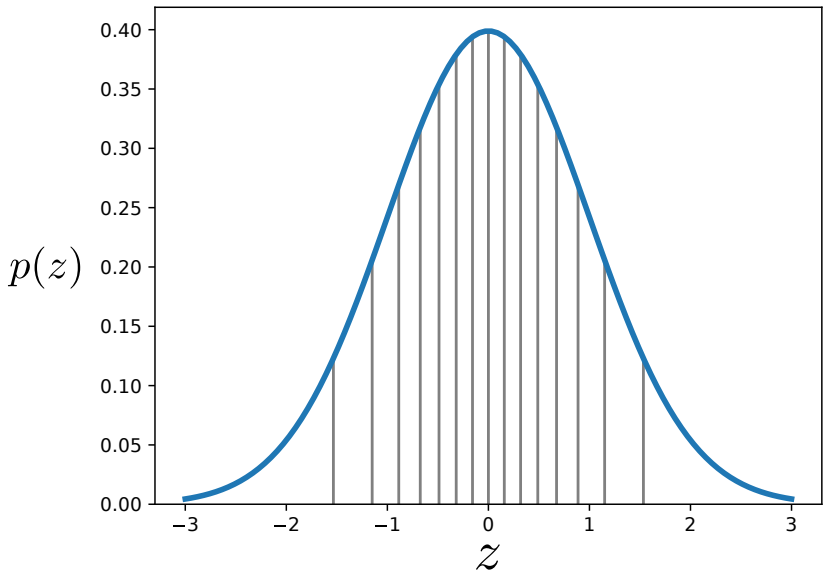


Figure 2.2: An example of the discretisation of the latent space with a standard Gaussian prior, using 16 buckets with equal prior probability mass.

coding. For example, imagine we were to discretise such that all but one of our buckets were in areas of very low density, with just one bucket covering the area with high density. This would result in almost all of the latent variables being coded as the same symbol (corresponding to the one high density bucket). Clearly this cannot be an efficient discretisation.

- The prior and the approximate posterior must share the same discretisation. This is due to the fact that they both describe the same set of latent variables, and if we do not share a discretisation then the argument used in Section 2.3.3 does not apply, since there is no cancellation of δz_i in 2.5.

- The discretisation must be known by the receiver before seeing data, since the first step of decoding is to decode $\mathbf{z}^0$ according the prior.

We propose to satisfy these considerations, by using the *maximum entropy discretisation* of the prior, $p_{\boldsymbol{\theta}}(\mathbf{z})$. This amounts to allocating buckets of equal mass under the prior and so results in a uniform distribution over the buckets, which is the discrete distribution with the maximal entropy for a given number of buckets. We visualise this for a standard Gaussian prior in Figure 2.2.

Having the discretisation be a function of the prior (which is fixed) allows the receiver to know the discretisation up front, which we have noted is necessary for the receiver to begin decoding. This would not be true for a discretisation that depended on the posterior.

This discretisation is appropriate for coding according to the prior, since we are maximising the entropy for this density. However, it is not obvious that it will be appropriate for coding according to the posterior, which it must also be used for.

Note that we can write our the expected message length (negative ELBO) for a single data point as:

$$L(q_{\boldsymbol{\phi}}) = -\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}\,|\,\mathbf{x}^0)}\big[\log p_{\boldsymbol{\theta}}(\mathbf{x}^0\,|\,\mathbf{z})\big] + D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}\,|\,\mathbf{x}^0)\,\|\,p_{\boldsymbol{\theta}}(\mathbf{z})) \tag{2.6}$$

We can see that minimising this objective encourages the minimisation of the KL divergence between the posterior and the prior. Therefore a trained model will generally have a posterior *close* (in a sense defined by the KL divergence) to the prior. This indicates that the maximum entropy discretisation of the prior may also be appropriate for coding according to the posterior.

The above discussion has been regarding the model with a single layer of latent variables, i.e. those models following the straightforward graphical model seen in Figure 2.1. We will return to the discussion of discretisation when we consider models with more complicated latent structures, in particular those with a hierarchy of latent layers.

2.4 Proof-of-concept experiments

In this section we provide the results of experiments designed to verify that the BB-ANS codec performs as expected on relatively small models and datasets, before we proceed onto more challenging datasets and complex models.

2.4.1 Compressing MNIST with a VAE

For our proof-of-concept implementation, we use a Variational Auto-Encoder (VAE) [Kingma and Welling, 2014]. The VAE is a popular probabilistic generative model with latent variables, and we have given a thorough background in Section 1.3.4.1.

We use a relatively simple VAE with a single (but multi-dimensional) layer of latent variables. As is common, we place a unit Gaussian prior over the latents, and model the approximate posterior as a diagonal Gaussian.

$$p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, I) \tag{2.7}$$

$$q_{\boldsymbol{\phi}}(\mathbf{z} \,|\, \mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}), \mathrm{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x}))) \tag{2.8}$$

$$\tag{2.9}$$

We will be modelling the MNIST dataset [LeCun et al., 1998], which consists of 28×28 monochrome images, with each pixel described in 8 bits. As such, we need an output distribution $p_{\boldsymbol{\theta}}(\mathbf{x} \,|\, \mathbf{z})$ that is discrete and produces a vector of 256 probabilities. We use the Beta-binomial distribution with 255 trials, which is equivalent to a binomial distribution with 255 trials, with parameter p unknown, and modelled as a random Beta variable with parameters α, β. We factor $p_{\boldsymbol{\theta}}(\mathbf{x} \,|\, \mathbf{z})$ over each image dimension. We also consider a simpler case of a stochastically binarised MNIST [Salakhutdinov and Murray, 2008]. For this case, we use a Bernoulli output distribution on each image dimension. So our two output distributions are:

$$p_{\text{full}}(\mathbf{x} \,|\, \mathbf{z}) = \mathrm{BetaBin}(\mathbf{x}; 255, \boldsymbol{\alpha}_{\boldsymbol{\theta}}(\mathbf{z}), \boldsymbol{\beta}_{\boldsymbol{\theta}}(\mathbf{z})) \tag{2.10}$$

$$p_{\text{binary}}(\mathbf{x} \,|\, \mathbf{z}) = \mathrm{Bernoulli}(\mathbf{x}; \boldsymbol{\gamma}_{\boldsymbol{\theta}}(\mathbf{z})) \tag{2.11}$$

Where we have used γ rather than the usual p to denote the Bernoulli parameter, to avoid confusion with the prior. So the functions that we learn in training are $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}), \boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x})$ for the posterior and either $\boldsymbol{\alpha}_{\boldsymbol{\theta}}(\mathbf{z}), \boldsymbol{\beta}_{\boldsymbol{\theta}}(\mathbf{z})$ or $\boldsymbol{\gamma}_{\boldsymbol{\theta}}(\mathbf{z})$ depending on whether we model full MNIST or the binarised version.

		Binarised MNIST	Full MNIST
	Raw data	1	8
	VAE test ELBO	0.19	1.39
	BB-ANS	0.19	1.41
	bz2	0.25	1.42
Codecs	gzip	0.33	1.64
	PNG	0.78	2.79
	WebP	0.44	2.10

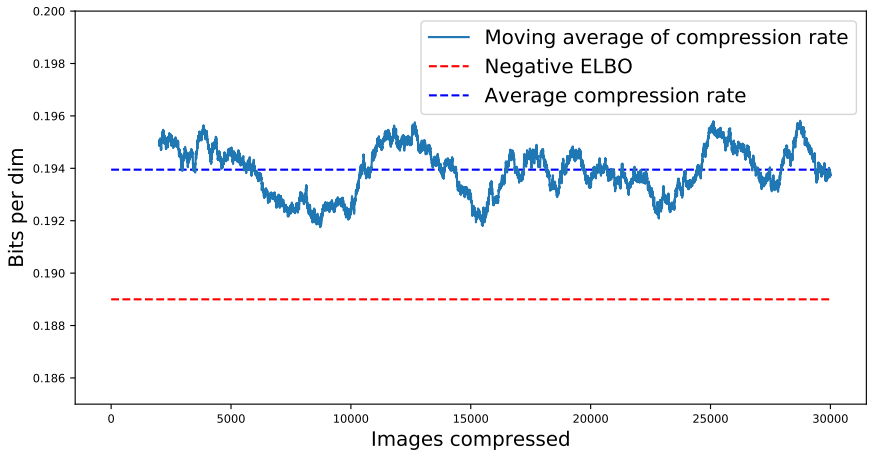
Table 2.1: Compression rates on the binarised MNIST and full MNIST test sets, using BB-ANS and other benchmark compression schemes, measured in bits per dimension. We also give the negative ELBO value for each trained VAE on the test set.

All functions are approximated using MLPs, with ReLU activations [Fukushima, 1975]. For the binarised data we use networks with a single hidden layer of dimension 100, and the latent of dimension 40. For the full MNIST data we use dimensions of 200 and 50 for the hidden layer and latent respectively. Note that it is generally acknowledged that convolutional neural networks are advantageous to use for image modelling tasks, but for this proof-of-concept experiment we found it simpler to use MLPs as a minimal working example.

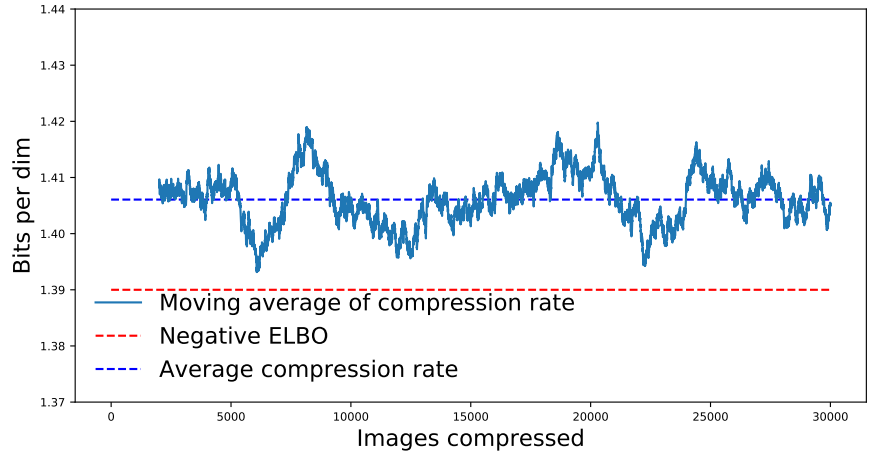
The usual VAE training objective is the ELBO, which, as we noted in Section 2.2.1, is the negative of the expected message length with bits back coding. Thus if we train a VAE as usual, by maximising the ELBO we are also minimising the message length of the resulting codec when the VAE is used with BB-ANS.

For these proof-of-concept experiments, we do the simplest thing to initialise the BB-ANS stack. That is, instead of directly sampling the first latents at random, to simplify our implementation we instead initialise the BB-ANS stack with a supply of clean bits. Thus to "sample" the first latents we simply decode according to the posterior from our stack. We find that around 400 bits are required for this in our experiments. The precise number of bits required to start the chain depends on the entropy of the discretised approximate posterior (from which we are initially sampling).

We show the results of the BB-ANS compression scheme against a suite of standard image compression benchmarks in Table 2.1. We can see that despite using small, fully-connected, neural networks we can outperform many of the benchmark methods.



(a) Binarised MNIST



(b) Full MNIST

Figure 2.3: A 2000 point moving average of the compression rate, in bits per dimension, during the compression process using BB-ANS with a VAE. We compress a concatenation of three shuffled copies of the MNIST test set.

This is encouraging, particularly as a potential use case of learned compression is to make specialised codecs for a particular dataset. For example we could simply be interested in transmitting MNIST in as small a size as possible. In such cases though we must take the model size into account, since we do not get to amortise the model size over many compression use cases. We will return to this thought in Chapter 3.

However it is important to note that BB-ANS is compressing data which is very close to that on which it was trained. That is, we trained the model on the train set of MNIST and tested on the test set - but the method is not robust to a distributional

shift in the data we seek to compress. For example, compressing a colour image dataset would fail.

In contrast, the benchmark methods are more generic codecs, which do not require training and have been shown to compress a wide set of image data effectively. In fact, gzip and bz2 can compress arbitrary files, not even restricted to image data. It might seem surprising that the more generic compressors actually outperform specialised image codecs (PNG, WebP), but this effect is due to the simple nature of the MNIST dataset. Namely, that there are large portions of the image which are blank, which run-length encoding methods can compress very effectively.

In light of the above, perhaps the more important observation from Table 2.1 is that the achieved compression rate is very close to the value of the negative test ELBO seen at the end of VAE training.

So we can conclude, that at least in this example, the detrimental effects identified in Section 2.3 of finite precision, discretising the latent and of dirty bits do not have a material impact on the compression rate. We visualise their effects in Figure 2.3, which plots a moving average of the compression rate against the lower bound (negative ELBO). We can see that there is a gap of around 1% between the negative ELBO and the achieved compression rate, which is reasonable (or at least not enough to conclude that there is a serious issue).

2.5 Scaling up BB-ANS

We verified in Section 2.4, that in some proof-of-concept experiments that BB-ANS does come close to the theoretical message length, and that the various practical considerations do not appear to manifest in a significant way in such experiments.

We now discuss the techniques we introduce to scale up BB-ANS to larger models and more complex datasets, and the considerations required to ensure that BB-ANS is a candidate to go beyond research and be a viable alternative to existing, generic codecs.

To outline the rest of this section, we will identify shortcomings of the method described in the previous section that manifest when scaling up the codec. For each shortcoming we will then describe methods to mitigate it.

- As acknowledged in Section 2.4, the BB-ANS codec as described previously was trained and tested on a narrow set of data, and would not generalise to other image data. However, a strength of BB-ANS is that it simply a method to translate a latent variable PGM into a codec. As such, we instead move to use a more powerful model, trained on a more broad set of image data, namely ImageNet [Deng et al., 2009]. We also consider how to compress images of different sizes, which is a crucial element of making a more generic image codec. We discuss our choice of model in Section 2.5.1, and examine the performance on a diverse test set in Section 2.6.

- Another issue that manifests as we increase the scale of our model is that the cost to initialise the BB-ANS stack, discussed in Section 2.3.1, increases as our latent space grows. For a very large latent space, this cost can become prohibitive. As such, we propose methods to mitigate this in Section 2.5.2.

- It is not obvious how to extend the discretisation scheme presented in Section 2.3.3 to the scenario in which latent variables have a prior which does not factorise over the latent dimensions. We will discuss our method to resolve this for a particular class of latent priors in Section 2.5.3. We propose a simple method we refer to as *dynamic discretisation*, which is applicable for

an important subset of latent variable models which have a hierarchy of latent layers.

- A point we glossed over in the previous sections is the overhead of actually performing the coding/decoding operations. In fact, in our previous experiments we coded the dimensions of the data and latent in serial. This scales poorly to high dimensions, and so it is crucial that we vectorise this coding procedure. We discuss how to do this in Section 2.5.4.

  We identify the resulting codec that uses the methods referred to in the above list as *Hierarchical Latent Lossless Compression*, or HiLLoC.

2.5.1 Model selection

The main choice to be made when considering how to scale up a codec resulting from BB-ANS is the choice of model that will be used in conjunction with BB-ANS.

We identify necessary conditions that the model must satisfy in order to be considered as a viable replacement for generic image codecs.

- Firstly, the model must be trained on (and perform well on) a sufficiently diverse set of training data. This ensures that performance will be acceptable when attempting to code from image sources different from the source used for training.

- Secondly, the model must be able to process different image sizes.

The model we used in Section 2.4 was a small VAE, trained on the simple MNIST dataset [LeCun et al., 1998]. To improve the model performance we used a more powerful, modern model as the VAE. Namely we use the ResNet VAE introduced by Kingma et al. [2016]. This is a hierarchical latent variable model, as described in Section 1.3.4.2, which has shown to be effective at probabilistic modelling of a variety of image datasets.

We use the ImageNet dataset [Deng et al., 2009] as the training dataset, since this is a relatively large dataset, with approximately 1.3M images in the training set, and which covers a diverse range of subjects spanning over 1000 classes. These

images are also in a variety of sizes, so by training and testing on this dataset we ensure that we have performance robust to changes in image size. Note that although it is unnecessary to use a labelled dataset for our use case of compression, we also are motivated to use ImageNet simply because it, at least at the time of publication, is a widely used benchmark dataset. So many models that we wish to compare to report performance on it.

It is possible to take a model with a fixed input size and obtain outputs on a different model size with a combination of running the model on patches (if the image is larger than the requested input size) or padding the input image (if it is smaller). Neither of these are optimal solutions, since image context that could be used by the model is being discarded in the process.

We seek instead to use a class of models which can be run on multiple different image sizes, namely *fully convolutional* models. This refers to a model in which there are no densely connected layers - every layer is either convolutional or operates elementwise. Both convolutions and elementwise operations produce an output that scales in size with the input, unlike a densely connected layer which requires a fixed size input and output.

2.5.2 Starting the bits back chain when using hierarchical latent variable models

As discussed in Section 2.3.1, to code our first data point (or batch of data points), we require some data to be in the ANS stack in order to allow us to decode a latent sample according to the posterior. With a hierarchical latent variable model, our latent space may be very large, since many layers can be added to improve the modeling performance (i.e. increase the ELBO). For example very deep hierarchies are used in Kingma et al. [2016], Sønderby et al. [2016], Maaløe et al. [2019]. This is problematic, since the entropy of the posterior distribution over the latent space, and thus the amount of data initially required to be in the ANS stack, generally scales with the dimensionality of the latent space. For a sufficiently large space, this could result in requiring such a large amount of initial data to be in the ANS stack

such that effective compression is impossible.

Note that we can in theory amortise the cost of these *initial bits* over all of the data points being compressed. So for compressing large datasets it may not be problematic. However, for a compression scheme to be truly practical, it's imperative that it also be useful for compressing smaller datasets, and even when compressing a single data point.

We now discuss methods that can be used to reduce the initial bits required when using BB-ANS and a hierarchical latent variable model.

2.5.2.1 Bit-Swap

One method proposed to reduce the initial bits is Bit-Swap [Kingma et al., 2019]. This core of this method is, at the beginning of compression, to not decode all latents according to the posterior at once. Instead layers of latents are alternately decoded according to the posterior, then encoded according to the prior. We show the encode process precisely in Algorithm 3, and the decode in Algorithm 4.

The key benefit of Bit-Swap is that the latents are not all decoded with the posterior at the start of BB-ANS coding. This means that the initial bits required is smaller than the naive BB-ANS implementation, in which all latents $\mathbf{z}_{1:L}$ are decoded with the posterior at the first step of compression. The precise number of initial bits for Bit-Swap will depend on the entropy of the latents and the data, but will always be less than or equal to the number of initial bits required by the naive BB-ANS algorithm, and in many cases be significantly less.

However, Bit-Swap has two main disadvantages. Firstly, the hierarchical posterior must be *bottom-up*, namely deeper latents must be conditioned on earlier latents in order to permit the alternative encode/decode steps. This precludes the use of certain architectures that use top-down posteriors. Secondly, and most importantly, the hierarchical prior, likelihood and posterior must be *Markov*. That is, $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}) = p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1})$, $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}) = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_1)$ and $q_{\boldsymbol{\phi}}(\mathbf{z}_{l+1}|\mathbf{z}_{1:l}, \mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z}_{l+1}|\mathbf{z}_l)$ for all l[8]. Again, this is required to permit the alternating encode/decode steps. To see why, if we suppose that our likelihood $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L})$ was not Markov, then we would be unable to

[8]Apart from the very first layer, where the posterior is just conditioned on $\mathbf{x}$.

perform the "encode $\mathbf{x}$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_1)$" step in the encode algorithm (and in fact any of the other encodes according to the prior) since we do not have access to $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_1)$, as it would require integrating out the latents $\mathbf{z}_l$ for $l > 1$.

The Markov restriction one is significant, since it forbids the use of skip connections, described in Section 1.3.4.2 as the fundamental component of residual layers. Skip connections have proven to be a crucial architectural component of deep hierarchical latent variable models, increasing the stability of very deep models and preventing posterior collapse [Kingma et al., 2016, Sønderby et al., 2016, Maaløe et al., 2019].

Algorithm 3: Bit-Swap encode

Data : $\mathcal{D}$
Model : $p_{\boldsymbol{\theta}}(\mathbf{z}_{1:L}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}), q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x})$
Require : ANS stack with sufficient data encoded
while $\mathcal{D} \neq \emptyset$ **do**
  pick $\mathbf{x}^0 \in \mathcal{D}$;
  decode $\mathbf{z}_1^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_1|\mathbf{x}^0)$;
  encode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_1^0)$;
  for $l = 1$ *to* $L - 1$ **do**
    decode $\mathbf{z}_{l+1}^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_{l+1}|\mathbf{z}_l^0)$;
    encode $\mathbf{z}_l^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1}^0)$;
  encode $\mathbf{z}_L^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_L)$;
  $\mathcal{D} \leftarrow \mathcal{D} \setminus \mathbf{x}^0$;
Send : ANS stack (serialised into bitstream), $N := |\mathcal{D}|$

Algorithm 4: Bit-Swap decode

Model : $p_{\boldsymbol{\theta}}(\mathbf{z}_{1:L}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}), q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x})$
Require : ANS stack with data encoded (deserialised), N
$\mathcal{D} \leftarrow \emptyset$;
for $n = 1$ *to* N **do**
  decode $\mathbf{z}_L^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_L)$;
  for $l = L - 1$ *to* 1 **do**
    decode $\mathbf{z}_l^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1}^0)$;
    encode $\mathbf{z}_{l+1}^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_{l+1}|\mathbf{z}_l^0)$;
  decode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_1^0)$;
  encode $\mathbf{z}_1^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_1|\mathbf{x}^0)$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathbf{x}^0$;
Output : $\mathcal{D}$

2.5.2.2 Using auxiliary information sources as initial bits

We seek to decrease the number of initial bits, but also avoid the main disadvantage of Bit-Swap, discussed in Section 2.5.2.1, that the model must be Markov. To do so, we utilise one of the core properties of the bits-back argument - that auxiliary data must be present in order to permit compression in the first place. As such, we can simply use *useful* information to fill the ANS stack until we have sufficient initial bits to permit decoding the latents. This way, the initial bits do not have to be viewed as pure overhead to the codec, instead they may simply cause the compression rate to be sub-optimal at the beginning of compression, and as more data is compressed the compression rate will approach the optimal rate with BB-ANS (the negative ELBO).

There are two key questions that we are required to answer to make this method effective:

- What useful auxiliary information can we insert into the ANS stack?
- How can we minimise the initial bits required to begin BB-ANS coding, allowing us to approach the optimal rate faster?

A straightforward method to address the first question is proposed by Frey [1997] - we can insert the data itself into the ANS stack, coded according to a uniform distribution. This does not achieve compression, since our coding distribution is uniform, but crucially is not worse than the raw data itself. For example for raw data that is described by 8 bits per dimension, we can ensure that (on average) we do not use more than 8 bits per dimension at the beginning of coding. This may sound underwhelming, but it is noted in Bit-Swap [Kingma et al., 2019] that using a deep hierarchical model and naive BB-ANS can result in significant size inflation at the beginning of coding. For example, their 8 layer model requires roughly 44 bits per dimension to compress a single data point, which is almost six times more than the raw data itself.

One complication is that we require that our data we code via the uniform distribution to be roughly random, and uniform. Otherwise BB-ANS coding may not be effective due to the dirty bits issue described in Section 2.3. Clearly, if we use our raw data, this constraint will not usually be satisfied, since the data will not be

random and uniform. Frey [1997] deals with this by applying a XOR operation to the data with a random bit-mask. As long as a seed for a pseudo-random number generator is shared between sender and receiver to generate this bit-mask this step is reversible on the receiver's end.

Although the above method guarantees that BB-ANS will never be worse than sending the raw data (as long as the dirty bits issue is not manifest), it is more effective instead to use a different codec to fill the ANS stack with initial bits of compressed data. We refer to this codec as the *auxiliary codec*. The resulting compression rate would begin at the compression rate of the auxiliary codec, and would then step to the compression rate of BB-ANS once the auxiliary codec had compressed enough data points to permit BB-ANS coding to proceed. The auxiliary codec would have to be known to both sender and receiver, but they can use a simple, generic codec with a worse compression rate than BB-ANS.

In our experiments we use the Free Lossless Image Format (FLIF) [Sneyers and Wuille, 2016] to build up the buffer. We chose this codec because it performed better than other widely used codecs.

The above methods allow us to construct a codec which is practical at compressing a small number of data points, but clearly there is still overhead. We will not achieve our desired compression rate of the negative ELBO until we have sufficient initial bits in the ANS stack. This brings us to the second question posed above - if we can reduce the amount of bits required to begin BB-ANS coding then we will be able to achieve the BB-ANS compression rate sooner.

Since the first step of BB-ANS is to decode the latents $\mathbf{z}_{1:L}$ with the posterior $q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x})$, we wish to reduce the dimensionality of $\mathbf{z}_{1:L}$ - this in turn will reduce the initial bits required to decode it.

Each latent $\mathbf{z}_l$ will in general have shape $B \times H_l \times W_l \times C_l$ where B is the batch size, H_l, W_l, C_l are the respectively the height, width, and number of channels in the latent layer l. Our latent space has two spatial dimensions (height and width) since we assume that we are using a fully convolutional model as discussed in Section 2.5.1. Although we cannot alter the channels of the latents, each of the other 3 dimensions

generally scales in accordance with the input data[9]. Therefore, to minimise the initial bits required for BB-ANS we simply seek to minimise the batch and spatial dimensions of the input image that we seek to code. As such, at the beginning of coding we use a batch size of 1, and code small patches of images rather than full images themselves.

These size reductions are effective at reducing the initial bits overhead, but by using a small batch size we increase the run-time of our codec, since generally neural network based models are most time efficient when running on larger batch sizes since batch computations can be performed in a parallel fashion. By coding small patches, rather than full images, we also expect a worsening in compression rate. This is since the model has less context for each pixel it seeks to compress. As an example, consider using the smallest possible patch size - a single pixel. The best possible compression would be the marginal distribution over the pixel values, which would not result in effective compression.

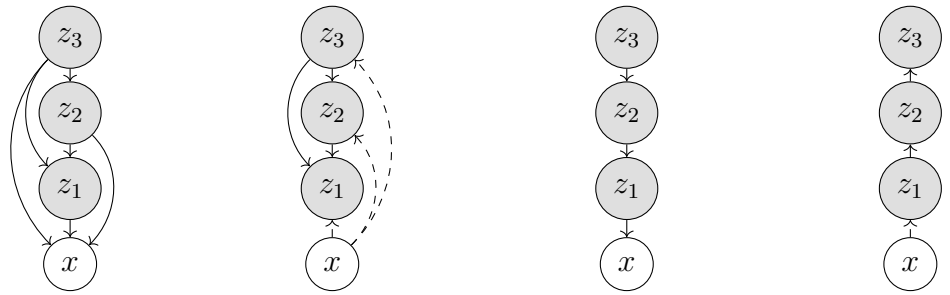
For our experiments on compressing full ImageNet images, we compress 32×32 patches, then 64×64, then 128×128 before switching to coding the full size images directly. Note that since our model can compress any shape image, we can compress the edge patches which will have different shape if the patch size does not divide the image dimensions exactly. Using this technique means that our coding rate improves gradually from the FLIF coding rate towards the coding rate achieved by HiLLoC on full images. We compress only 5 ImageNet images using FLIF before we start compressing 32×32 patches using HiLLoC.

Having to compress 5 images before we begin to utilise BB-ANS is an overhead of our codec, but it is worth noting some important justifications. Firstly, it achieves better performance than the primary alternative, Bit-Swap. With an 8 layer model, Bit-Swap report an initial compression rate on ImageNet 32[10] of 6.97 bits per dimension. FLIF achieves a much better compression rate than this. Secondly, we utilise a very deep latent hierarchy in our experiments of 24 latent layers. This is to

[9]There are some model architectures that may not have this property, but it is often true, and true for the models that we utilise.

[10]This refers to a version of the ImageNet dataset with images cropped and down-sampled to shape 32×32.

illustrate that our codec performs well even with such large models, but in practice models would often use many less layers than this, which means that we would require less images to be coded with the auxiliary codec.



(a) HiLLoC generative (b) HiLLoC inference (c) Bit-Swap generative (d) Bit-Swap inference

Figure 2.4: Graphical models representing the generative and inference models with HiLLoC and Bit-Swap, both using a 3 layer latent hierarchy. The dashed lines indicate dependence on the fixed observation.

2.5.3 Dynamic discretisation

2.5.3.1 Structured versus unstructured priors

In Section 2.3.3 we described a straightforward method to discretise a continuous latent space to permit entropy coding via ANS. We referred to the method as the maximum entropy discretisation, since it amounted to dividing a 1D latent space into buckets with equal mass under the prior. With higher dimensional latent spaces, the assumption made was that the prior factorises over latent dimensions, i.e. $p_{\boldsymbol{\theta}}(\mathbf{z}) = \prod_{d=1}^{D} p_{\boldsymbol{\theta}}(\mathbf{z}_d)$ where D is the number of dimensions. Note that this assumption can of course be violated, for example we could use a Gaussian prior with a non-diagonal covariance matrix. This would be problematic, as now our discretisation would have to index higher dimensional buckets. We do not consider this case, since it is, for one, more complicated, but also for the simple fact that it is more common in PGMs to just use a factorised prior.

However, for more complex latent variable models, it is usual to design the latent space to be *structured*, in some sense. For example, we discussed models with a hierarchy of layers of latent variables in Section 1.3.4.2. The hierarchy amounts to

structuring the latent $\mathbf{z}$ into L layers $\mathbf{z}_{1:L}$ and the prior is thus ordered:

$$p_{\boldsymbol{\theta}}(\mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z}_L) \prod_{l=1}^{L-1} p_{\boldsymbol{\theta}}(\mathbf{z}_l | \mathbf{z}_{l+1:L}) \tag{2.12}$$

where the prior for each latent layer is conditioned on the previous layers. Compare this to the simpler method that we used previously, i.e. in Section 2.4, where the prior had no conditional dependencies, which we can think of as *unstructured*.

From a mathematical perspective, the hierarchical prior is a richer prior than the unstructured prior, since it is a more expressive factorisation of the joint distribution over all of the latent dimensions. Indeed, having an unstructured prior is actually a very restrictive condition on the space of priors that can be modelled, since all latent dimensions are independent. In contrast, the hierarchical prior places no restrictions on the joint distribution - every multivariate distribution can be written as per Equation 2.12 by Bayes' theorem. But if we insist that each conditional prior $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L})$ be factorised over the dimensions of $\mathbf{z}_l$ then this does again place a restriction on the set of priors that can be modelled. Note also that many other choices of a structured prior could be specified, for example the prior could be Markov: $p_{\boldsymbol{\theta}}(\mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z}_L) \prod_{l=1}^{L-1} p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1})$.

2.5.3.2 Static versus dynamic discretisation

Returning now to the task of discretising the latent space, we note that it is not as straightforward to discretise a structured latent space as compared to an unstructured. This is due to the fact that we do not know the marginal prior distribution $p(\mathbf{z}_l)$ on a latent layer l, since the prior itself is conditional on the values of other latent layers (aside from the deepest layer L, which is not conditional).

Using the latent hierarchy described previously as an example, we see that to access the marginal prior on a latent layer l we would have to integrate out the deeper layers:

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}_l) = \int d\mathbf{z}_{l+1:L}\ p_{\boldsymbol{\theta}}(\mathbf{z}_L) \prod_{k=l}^{L-1} p_{\boldsymbol{\theta}}(\mathbf{z}_k | \mathbf{z}_{k+1:L}) \tag{2.13}$$

This integral is usually intractable, and even if it were tractable it would probably

induce dependencies between dimensions of the latent space, which would make entropy coding more difficult. Therefore we cannot simply allocate buckets to be of equal mass under the prior as we did previously to obtain the maximum entropy discretisation.

One method to resolve this issue is to simply sample from the prior, and then divide the latent dimensions into buckets which assign a roughly equal number of samples to each bucket. This is equivalent to performing a Monte Carlo integration of Equation 2.13. This is the method utilised by Bit-Swap [Kingma et al., 2019]. We refer to this method as *static discretisation*, since the buckets are estimated once and then constant for the coding of any data points.

We propose using a different method, namely to simply make the discretisation scheme itself conditional in the same manner as the prior. Instead of discretising according the marginal priors $p_{\boldsymbol{\theta}}(\mathbf{z}_l)$, we discretise according to the conditional priors $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L})$. Specifically, for each latent layer l, we partition each dimension into intervals which have equal probability mass under the conditional $p_{\boldsymbol{\theta}}(\mathbf{z}_l \,|\, z_{l+1:L})$. This directly generalises the maximum entropy discretisation scheme we described previously, and we refer to this method as *dynamic discretisation* since the buckets are dynamic dependent on the values of the latents we seek to code.

It is worth considering the merits and drawbacks of these two contrasting approaches. Static discretisation has the advantage that it can be used with any factorisation of the prior - all that has to be done is sampling and then bucket estimation. It does require this bucket estimation step to be performed however, which adds an extra step of computation to the coding process, although it only has to be performed once. There is also no guarantee that the estimated buckets will be appropriate for all latents that may be encountered. For example if the prior is multi-modal then it is possible that some modes will not be seen during the bucket estimation step, which will result in poor compression rates at test time if these modes are observed.

In contrast, dynamic discretisation has the advantage that it will always accurately reflect the prior since it is exact[11], and does not rely on an approximation of the

[11]Note that the process of assigning the buckets for a given density may not be exact, given that

marginals. It is also much simpler to calculate than the static discretisation, since usually the CDFs for the conditional prior can be used directly to solve for the bucket locations. However, the dynamic discretisation must be calculated for each latent to be coded, which adds to the overhead at run-time.

Algorithm 5: HiLLoC encode

Data : $\mathcal{D}$
Model : $p_{\boldsymbol{\theta}}(\mathbf{z}_{1:L}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}), q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x})$
Require: ANS stack with sufficient data encoded
while $\mathcal{D} \neq \emptyset$ **do**
  pick $\mathbf{x}^0 \in \mathcal{D}$;
  decode $\mathbf{z}_L^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_L|\mathbf{x}^0)$;
  for $l = L-1$ *to* 1 **do**
    decode $\mathbf{z}_l^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}^0, \mathbf{x}^0)$;
  encode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}^0)$;
  for $l = 1$ *to* $L-1$ **do**
    encode $\mathbf{z}_l^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}^0)$;
  encode $\mathbf{z}_L^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_L)$;
  $\mathcal{D} \leftarrow \mathcal{D} \setminus \mathbf{x}^0$;
Send : ANS stack (serialised into bitstream), $N := |\mathcal{D}|$

Algorithm 6: HiLLoC decode

Model : $p_{\boldsymbol{\theta}}(\mathbf{z}_{1:L}), p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}), q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x})$
Require: ANS stack with data encoded (deserialised), N
$\mathcal{D} \leftarrow \emptyset$;
for $n = 1$ *to* N **do**
  decode $\mathbf{z}_L^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_L)$;
  for $l = L-1$ *to* 1 **do**
    decode $\mathbf{z}_l^0$ with $p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}^0)$;
  decode $\mathbf{x}^0$ with $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}_{1:L}^0)$;
  for $l = 1$ *to* $L-1$ **do**
    encode $\mathbf{z}_l^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}^0, \mathbf{x}^0)$;
  encode $\mathbf{z}_L^0$ with $q_{\boldsymbol{\phi}}(\mathbf{z}_L|\mathbf{x}^0)$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathbf{x}^0$;
Output : $\mathcal{D}$

Note that in the above considerations, we made the assumption that latent samples came from the prior, which for BB-ANS will not be true - they will be

not all CDFs have analytic forms. However, this is a negligible source of error compared to the Monte Carlo integration used for static discretisation.

sampled from the posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$. However, as discussed in Section 2.3.3.1, since the posterior is generally close to the prior for a trained model, we can ignore this difference in practice.

An important consideration though, is that we must use the same discretisation when coding the latents according to both the prior and posterior. Thus if the prior and posterior have different structures, we cannot use dynamic discretisation, which is another drawback of the method. As an example, consider a prior and a posterior structured as:

$$p_{\boldsymbol{\theta}}(\mathbf{z}_{1:L}) = p_{\boldsymbol{\theta}}(\mathbf{z}_L) \prod_{l=1}^{L-1} p_{\boldsymbol{\theta}}(\mathbf{z}_l|\mathbf{z}_{l+1:L}) \tag{2.14}$$

$$q_{\boldsymbol{\phi}}(\mathbf{z}_{1:L}|\mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z}_1|\mathbf{x}) \prod_{l=2}^{L} q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l-1:1}) \tag{2.15}$$

The first step of BB-ANS would be to decode $\mathbf{z}_1$ according to the posterior, but we would not be able to calculate the discretisation required, as the prior over $\mathbf{z}_1$ is conditioned on all the other latents, which we do not have values for yet.

As such, dynamic discretisation enforces that we have the same structure for the posterior as the prior. This is generally referred to in the context of hierarchical latent variable models as having a top-down posterior $q_{\boldsymbol{\phi}}(\mathbf{z}) = q_{\boldsymbol{\phi}}(\mathbf{z}_L) \prod_{l=1}^{L-1} q_{\boldsymbol{\phi}}(\mathbf{z}_l|\mathbf{z}_{l+1:L})$, as opposed to a bottom-up posterior as defined in Equation 2.15.

We show an example graphical model for Bit-Swap and HiLLoC, which demonstrates the difference between top-down and bottom-up posteriors, as well as the Markov nature of the Bit-Swap model requirement, in Figure 2.4. We also summarise the resulting HiLLoC codec in Algorithms 5 and 6.

2.5.4 Vectorised lossless compression

We did not address the exact nature of how the ANS entropy coding was performed in the proof-of-concept experiments in Section 2.4, but the implementation was a simple Python implementation which ran in loops over the dimensions of the data being coded.

This is sufficient for the small scale experiments which we ran earlier, but is too

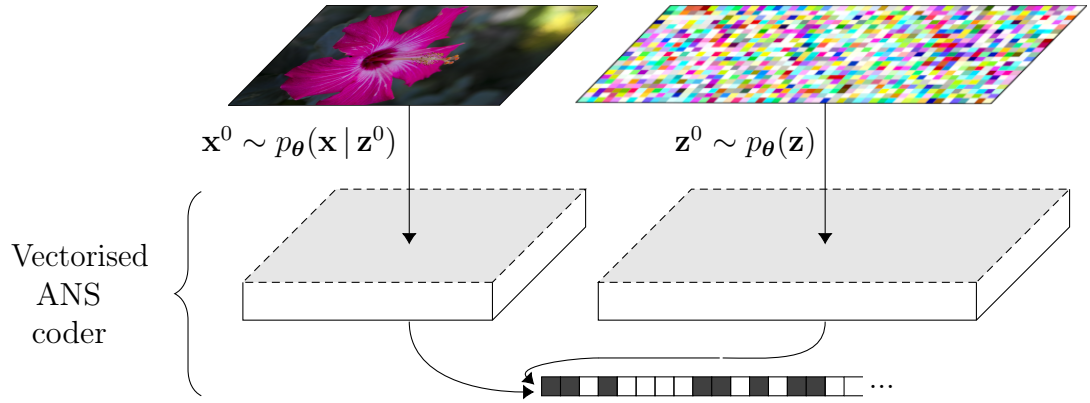


Figure 2.5: Visualisation of the process of pushing images and latents from a VAE to the vectorised ANS stack. The ANS stack head is shaped such that images and latents can be pushed and popped in parallel, without reshaping. Beneath the shaped top of the stack is the flat message stream output by ANS.

slow to run on high-dimensional images and latents that we seek to code as we use larger models and look to compress larger images. As such, we rewrite the ANS compression to be implemented in a vectorised fashion. This is not fully parallel, but instead using primarily Numpy [Harris et al., 2020] operations on vectors rather than scalars, which is significantly faster than using loops in the Python interpreter.

As such, we effectively reshape the *top* of the ANS stack (which is the location to which data is *pushed* or *popped*, to use the standard language of stacks) to be the same shape as the data we are seeking to code. Each dimension of the data is then coded via ANS on each individual ANS sub-stack on the reshaped stack. After the coding is performed these sub-stacks can be concatenated into a regular ANS stack or bitstream. To ensure the overhead is low while doing this, the BitKnit technique [Giesen, 2015] is used. BitKnit uses the fact that the values at the top of the sub-stacks are not uniformly distributed to reshape the stack head more efficiently than simply concatenating. As discussed in Giesen [2015], elements of the top of the sub-stacks have a probability mass roughly

$$p(h) \propto 1/h. \tag{2.16}$$

Equivalently, the *length* of h is approximately uniformly distributed. More detailed discussion and an empirical demonstration of this is given by Bloom [2014]. An

efficient way to form the final output message at the end of decoding, is to fold the stack head vector by repeatedly encoding half of it onto the other half, until only a scalar remains, using the above distribution for the encoding. We implement this technique and use it for our experiments. The number of (vectorised) encode steps required is logarithmic in the size (i.e. the number of elements) of the stack head.

Some of the overhead from vectorisation also comes at the *start* of encoding, when, in existing implementations, the elements of the stack head vector are initialised to copies of a fixed constant. Information from these copies ends up in the message and introduces redundancy which scales with the size of the head. This overhead can be removed by initialising the stack head to a vector of length 1 and then growing the length of the stack head vector gradually as more random data is added to the stack, by *decoding* new stack head vector elements according to the distribution (2.16).

We visualise the vectorised coding process in Figure 2.5. Note that we use a different shaped head for the latent and the image, since in theory the respective coding can be performed in parallel.

2.6 Larger Scale Experiments

We now present experiments to verify that the HiLLoC method is indeed a strong image compression system, which can effectively compress images from different sources and of different sizes.

We implement HiLLoC with a ResNet VAE (RVAE), as discussed in Section 2.5.1. We give a detailed description of the model architecture in Section 1.3.4.2.

In all experiments we used an RVAE with 24 stochastic hidden layers. The RVAE utilises skip connections, which have been shown to be important to be able to effectively train deep models [He et al., 2016, Maaløe et al., 2019]. As discussed in Section 2.5.2, if we were using the Bit-Swap method we would not be able to use skip connections, and training such deep models would be more challenging.

We trained the RVAE on the ImageNet 32 training set, then evaluated the RVAE ELBO and HiLLoC compression rate on the ImageNet 32 test set. To test generalisation, we also evaluated the ELBO and compression rate on the tests sets of

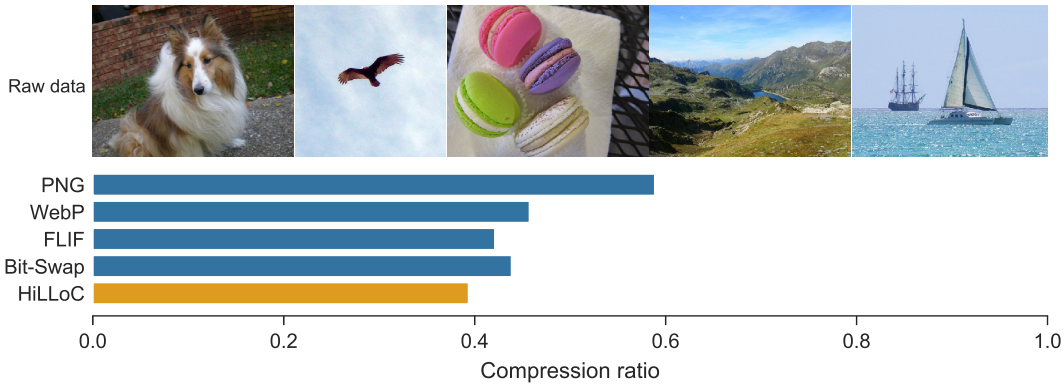


Figure 2.6: A selection of images from the ImageNet dataset and the compression rates achieved on the dataset by PNG, WebP, FLIF, Bit-Swap and the HiLLoC codec (with ResNet VAE) presented in this thesis.

ImageNet 64, CIFAR-10 and full size ImageNet. For ImageNet 32, ImageNet 64 and CIFAR-10 we report compression rates on the entire test set, and so do not require using an auxiliary codec and or patching as described in Section 2.5.2. For full size ImageNet, we do use the described method, since we only compress 2000 random images from the test set (thus the initial bits cost is amortised over fewer images). The results are shown in Table 2.2.

2.6.1 Scaling up the model size

Table 2.2: Compression performance of HiLLoC with RVAE compared to other codecs. Rates measured in bits/dimension (raw data is 8 bits/dimension). For HiLLoC we display compression rate and theoretical performance (ELBO). All HiLLoC results are obtained from the same model, trained on ImageNet 32.

		ImageNet 32	ImageNet 64	CIFAR-10	ImageNet
Generic	PNG	6.39	5.71	5.87	4.71
	WebP	5.29	4.64	4.61	3.66
	FLIF	4.52	4.19	4.19	3.37
Discrete flow	IDF[12]	4.18	3.90	3.34	-
	IDF generalised[13]	4.18	3.94	3.60	-
	LBB[14]	3.88	3.70	3.12	-
VAE	Bit-Swap	4.50	-	3.82	3.51[15]
	HiLLoC	4.20	3.90	3.56	3.15
	HiLLoC (ELBO)	(4.18)	(3.89)	(3.55)	(3.14)

Table 2.2 shows that HiLLoC achieves competitive compression rates on all benchmarks, and state of the art on full size ImageNet images. We compare HiLLoC to generic codecs, as well as learned results using both methods based on discrete flows and VAEs (which HiLLoC falls under). The discrete flow models use similar flow models to those discussed in this thesis, but with additional mechanisms to account for the fact that we need to code discrete data (whereas flows naturally parameterise continuous densities). In particular, IDF [Hoogeboom et al., 2019] uses a quantisation operation and the straight-through-estimator, whereas LBB [Ho et al., 2019b] uses a local latent variable model to quantise, and utilises bits-back coding.

The fact that HiLLoC can achieve state of the art compression on ImageNet relative to the baselines, even under a change of distribution, is striking. This provides strong evidence of its efficacy as a general method for lossless compression of natural images. Naively, one might expect a degradation of performance relative to the original test set when changing the test distribution—even more so when the resolution changes. However, in the settings we studied, the *opposite* was true, in that the average per-pixel ELBO (and thus the compressed message length) was *lower* on all other datasets compared to the ImageNet 32 validation set.

In the case of CIFAR, we conjecture that the reason for this is that its images are simpler and contain more redundancy than ImageNet. This theory is backed up by the performance of standard compression algorithms which, as shown in Table 2.2, also perform better on CIFAR images than they do on ImageNet 32. We find the compression rate improvement on larger images more surprising. We hypothesise that this is because pixels at the edge of an image are harder to model because they have less context to reduce uncertainty. The ratio of edge pixels to interior pixels is lower for larger images, thus we might expect less uncertainty per pixel in a larger image.

To demonstrate the effect of vectorisation we timed ANS of single images at

[12]Integer discrete flows, retrieved from Hoogeboom et al. [2019].

[13]Integer discrete flows trained on ImageNet 32. ImageNet 64 images are split into four 32×32 patches. Retrieved from Hoogeboom et al. [2019].

[14]Local bits back, retrieved from Ho et al. [2019b].

[15]For Bit-Swap, full size ImageNet images were cropped so that their side lengths were multiples of 32.

Table 2.3: Runtime of vectorised vs. serial ANS implementations. Times are given to the nearest second for a variety of image sizes, and were computed on a desktop with 6 CPU cores and a GTX 1060 GPU.

	Image size		
	32×32	64×64	128×128
serial ANS	99	380	1460
vectorised ANS	2	5	11

different, fixed, sizes, using a fully vectorised and a fully serial implementation. The results are given in Table 2.3, which clearly shows a speedup of nearly three orders of magnitude for all image sizes. We find that the run times for encoding and decoding are roughly linear in the number of pixels, and the time to compress an average sized ImageNet image of 500×374 pixels (with vectorised ANS) is around 29s on a desktop computer with 6 CPU cores and a GTX 1060 GPU. Although this is still a long time, the vast majority of time is spent in neural network inference, since we use a very deep model and are not exploiting batch parallelism. Reducing this inference time will be motivation for the following chapter of this thesis.

2.7 Conclusion

In this chapter we demonstrated that it is possible to achieve lossless compression with low overheads using latent variable models. To do this we utilised the bits-back coding method, which describes at a high level how to use a latent variable model to compress data at the rate of the negative ELBO. To materialise bits-back coding into an actual codec, we showed that the choice of entropy coder is critical. In particular, using asymmetric numeral systems (ANS), rather than the previously used arithmetic coding (AC), reduces the overhead incurred from entropy coding significantly, and makes the resulting coding rate suitable for practical usage. We initially demonstrated that our resulting algorithm, bits-back coding with ANS (BB-ANS) can achieve close to the theoretical bits-back coding compression rate in a small-scale, proof-of-concept experiment.

We then explored how to expand BB-ANS beyond small-scale experiments to effective compression of more complex and realistic datasets. To this end we used

a more powerful, hierarchical latent variable model which raises many challenges not present in the model used in the small-scale experiments. Namely how to discretise the hierarchical latent space, how to initialise the ANS chain and ensuring entropy coding itself is fast. We term the resulting codec that resolves these issues as Hierarchical Latent Lossless Compression (HiLLoC).

We then verified HiLLoC experimentally, showing that it can achieve compression rates competitive with state-of-the-art, hand-crafted image codecs on realistic image datasets, even when tested on different datasets to the training data. As such, this is a step towards replacing mainstream image codecs with learned compression techniques.

One point worth considering is under what conditions learned codecs such as HiLLoC fail. We find that generally HiLLoC performs relatively well when tasked with coding realistic images (as demonstrated by our range of experiments). However, if presented with something sufficiently different from the training set it will fail to be effective. As a simple example, asking HiLLoC to compress a noise image will result in the HiLLoC message size being much larger than the raw image.

One reason for this is simply that there can not exist universal lossless compressors - that is, there is a simple argument to show by the pigeonhole principle that no lossless codec can decrease all file sizes. But more importantly, learned codecs tend to fail less gracefully than generic codecs. We suspect that this is a natural trade-off, as specialisation in lossless compression must come at the expense of some generalisation. One mitigation for these failure modes is to have an auxiliary codec that we fall back to if we estimate the log-likelihood to be sufficiently low under the PGM underlying HiLLoC.

In the next chapter we will turn our attention to the models that make up the learned component of codecs such as HiLLoC, and examine how to make them more computationally efficient. Since, for a codec to be viable as a replacement for existing generic codecs, this is an important consideration.

Chapter 3

Binary neural networks for probabilistic generative models

The work presented in this chapter was published in [Bird et al., 2021b].

In the previous chapter we described BB-ANS, a lossless compression system built by combining a class of probabilistic generative model (PGM) and ANS. We considered the computational efficiency of ANS and how it can be improved, but we did not consider the efficiency of the PGM itself.

In this chapter, we detail a method that improves the computational efficiency of PGMs by implementing the neural networks as *binary neural networks*, that is neural networks with binary-valued weights.

3.1 Introduction

Although the application of binary neural networks for classification is relatively well-studied [Courbariaux et al., 2015, M., 2018, Gu et al., 2018], there has been no research that we are aware of that has examined whether binary neural networks can be used effectively in *unsupervised* learning problems. Indeed, many of the deep generative models that are popular for unsupervised learning do have high parameter counts and are computationally expensive [Vaswani et al., 2017, Maaløe et al., 2019, Ho et al., 2019a]. These models would stand to benefit significantly, from an efficiency standpoint, from converting the weights and activations to binary

values, which we call *binarisation* for brevity.

The need for computational efficiency in PGMs is particularly strong for the use of PGMs in compression, which we explored in Chapter 2. In compression use cases run-time is a key consideration of the codec, for example codecs for video streaming are required to decode in real-time. Additionally, having a very large model can defeat the whole purpose of compression to begin with, since both parties must have access to the same codec, in other words the communication of the model may outweigh the benefits it brings if the model is prohibitively large. Furthermore, the nature of compression is that of a communication between a sender and receiver - often in practice this is manifest in a client-server relationship. It is feasible that the server may have high amounts of compute on hand, but the client devices are often compute-constrained such as smartphones and personal computers. In such scenarios, having lightweight codecs is crucial.

In this chapter we will detail our method to effectively binarise certain classes of deep PGMs. Our specific contributions are:

- Describe a new normalisation technique, which we call binary weight normalisation, that is appropriate for binarising generative models. We also show that fast binary kernels can still be used even when normalising.

- Motivate theoretically and practically which components of deep PGMs can be effectively binarised. In particular, we specify that residual layers are natural candidates for binarisation, whereas layers that feed directly into the model output tend to be less amenable to binarisation.

- We demonstrate our methods and theories empirically. We validate that binary weight normalisation is more effective than binary batch normalisation for our models of interest, as well as examining the trade-offs between models with binary and real-valued activations. Further, we show that it is possible to increase the parameter count of the model using cheap binary weights, to increase model performance without sacrificing too much computational performance.

3.2 Binary neural networks

Chapter 2 created codecs that utilise neural networks as the core of the flexible, learned component. In this section we will describe how neural networks can be implemented with binary-valued weights $\mathbf{w}_{\mathbb{B}}$ rather than the usual real-valued[1] weights and activations [Courbariaux et al., 2015, Hubara et al., 2016, Rastegari et al., 2016, M., 2018, Gu et al., 2018]. Such *binary neural networks* have improved computational efficiency at the expense of flexibility (in terms of their space of functions they can model). In this thesis, we use the convention of binary values being in $\mathbb{B} \coloneqq \{-1, 1\}$.

3.2.1 Benefits and disadvantages of binary neural networks

The primary motivation for using binary neural networks is to decrease the memory and computational requirements of the model. Clearly binary weights require less memory to be stored: 32× less than the usual 32-bit floating-point weights.

Binary neural networks also admit significant speed-ups. A reported 2× speed-up can be achieved by a layer with binary weights and real-valued inputs [Rastegari et al., 2016]. This can be made an additional 29× faster if the inputs to the layer are also constrained to be binary [Rastegari et al., 2016]. With both binary weights and inputs, linear operators such as convolutions can be implemented using the inexpensive XNOR and bit-count binary operations. A simple way to ensure binary inputs to a layer is to have a binary activation function before the layer [Hubara et al., 2016, Rastegari et al., 2016].

The primary disadvantage of binary neural networks is their loss of flexibility versus real-valued weights. Theoretically we can say that a binary neural network with a given architecture contain a strictly smaller hypothesis class than the equivalent network with real-valued weights. This is due to the fact that the real-valued weights contain the binary values in the range of values they can take (or at least, they can be constructed to do so - the details will be the particular implementation of floating-point numbers being used).

[1] We use real-valued throughout this thesis to be synonymous with "implemented with floating-point precision".

Another disadvantage is that binary neural networks can be more difficult to train than networks with real-valued weights. We discuss this in greater detail in Section 3.2.2.

3.2.2 Optimisation of binary neural networks

Taking a trained model with real-valued weights and binarising the weights has been shown to lead to significant worsening of performance [Alizadeh et al., 2019]. So instead the binary weights are optimised. It is common to not optimise the binary weights directly, but instead optimise a set of underlying real-valued weights $\mathbf{w}_{\mathbb{R}}$ which can then be binarised in some fashion for inference. In this thesis we will adopt the convention of binarising the underlying weights using the sign function (see Equation 3.2). We also use the sign function as the activation function when we use binary activations (see Equation 3.5, where $\boldsymbol{\alpha}_{\mathbb{R}}$ are the real-valued pre-activations). We define the sign function as:

$$\text{sign}(x) \coloneqq \begin{cases} -1, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{3.1}$$

Since the derivative of the sign function is zero almost everywhere[2], the gradients of the underlying weights $\mathbf{w}_{\mathbb{R}}$ and through binary activations are zero almost everywhere. This makes gradient-based optimisation challenging. To overcome this issue, the straight-through estimator (STE) [Bengio et al., 2013] can be used. When computing the gradient of the loss $\mathcal{L}$, the STE replaces the gradient of the sign function (or other discrete output functions) with an approximate surrogate. A straightforward and widely used surrogate gradient is the identity function, which we use to calculate the gradients of the real-valued weights $\boldsymbol{w}_{\mathbb{R}}$ (see Equation 3.3).

It has been shown useful to clip the gradients when their magnitude becomes too large [Courbariaux et al., 2015, Alizadeh et al., 2019]. Therefore we use a clipped identity function for the gradients of the pre-activations (see Equation 3.6). This avoids saturating a binary activation.

[2]Apart from at 0, where it is non-differentiable.

	Weights		Activations	
Forward pass:	$\boldsymbol{w}_{\mathbb{B}} = \text{sign}(\boldsymbol{w}_{\mathbb{R}})$	(3.2)	$\boldsymbol{\alpha}_{\mathbb{B}} = \text{sign}(\boldsymbol{\alpha}_{\mathbb{R}})$	(3.5)
Backward pass:	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_{\mathbb{R}}} \coloneqq \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_{\mathbb{B}}}$	(3.3)	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_{\mathbb{R}}} \coloneqq \frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}_{\mathbb{B}}} 1_{\lvert\boldsymbol{\alpha}_{\mathbb{R}}\rvert \leq 1}$	(3.6)
After update:	$\boldsymbol{w}_{\mathbb{R}} \leftarrow \text{clip}(\boldsymbol{w}_{\mathbb{R}}; [-1, 1])$	(3.4)	–	

Table 3.1: The optimisation procedure when using the straight-through estimator. Note that functions are applied element-wise on vectors in this table.

Lastly, the loss value only depends on the sign of the real-valued weights. Therefore, the values of the weights are generally clipped to be in $[-1, 1]$ after each gradient update (see Equation 3.4). This restricts the magnitude of the weights and thus makes it easier to flip the sign.

Note that there are other techniques shown to benefit training binary neural networks in certain circumstances, for example using two-stage training [Alizadeh et al., 2019]. We do try some of these methods later in the thesis when using binary neural networks, but empirically we do not find them to be useful for our particular experiments. As such, only the previously listed techniques are the ones that we do find essential for stable training.

3.3 Using binary neural networks in probabilistic generative models

In this section we will describe the techniques required to successfully implement portions of PGMs with binary neural networks.

3.3.1 Binary weight normalisation

It is important to apply some kind of normalisation after a binary layer. Binary weights are often large in magnitude relative to the usual real-valued weights, and can result in large outputs which can destabilise training [Sari et al., 2020]. Previous

binary neural network implementations have largely used batch normalisation [Ioffe and Szegedy, 2015], which can be executed efficiently using a shift-based implementation [Hubara et al., 2016]. Batch normalisation involves normalising a batch at a given step in the network by the empirical mean and standard deviation per dimension. An affine transformation, parameterised by a shift and scale, is then applied.

However, it is common in generative modelling to use weight normalisation [Salimans and Kingma, 2016] instead of batch normalisation. For example, it is used in the Flow++ [Ho et al., 2019a] and state-of-the-art hierarchical VAE models [Kingma et al., 2016, Maaløe et al., 2019]. Weight normalisation factors a vector of weights $\mathbf{w}$ into a vector of the same dimension $\mathbf{v}$ and a magnitude g, both of which are learned. The weight vector is then expressed as:

$$\mathbf{w} = \mathbf{v} \cdot \frac{g}{||\mathbf{v}||} \tag{3.7}$$

Where $||\cdot||$ denotes the Euclidean norm. This implies that the norm of $\mathbf{w}$ is g. Note that the learned parameters are $\mathbf{v}, g$, and $\mathbf{w}$ is the version of the weights used in the model.

Now suppose we wish to binarise the parameters of a weight normalised layer. We are only able to binarise $\mathbf{v}$, since binarising the magnitude g (and a possible bias b) could result in large outputs of the layer. However, g and b do not add significant compute or memory requirements, as they are applied elementwise and are much smaller than the binary weight vector.

Denote $\mathbf{v}_{\mathbb{R}} \in \mathbb{R}^n$ as the set of real-valued parameters which we will optimise. Let $\mathbf{v}_{\mathbb{B}} = \text{sign}(\mathbf{v}_{\mathbb{R}})$ be the binarised weight vector. Since every element of $\mathbf{v}_{\mathbb{B}}$ is one of ± 1, we know that $||\mathbf{v}_{\mathbb{B}}|| = \sqrt{n}$ [3]. We then have:

$$\mathbf{w}_{\mathbb{R}} = \mathbf{v}_{\mathbb{B}} \cdot \frac{g}{\sqrt{n}} \tag{3.8}$$

Where we have used $\mathbf{w}_{\mathbb{R}}$ to denote that the resulting weight vector is real-valued.

We refer to this as *binary weight normalisation*, or BWN. Importantly, this is

[3] $||\mathbf{v}_{\mathbb{B}}|| = \sqrt{\sum_i (v_{\mathbb{B},i})^2} = \sqrt{\sum_i 1} = \sqrt{n}$

faster to compute than the usual weight normalisation (Equation 3.7), since we do not have to calculate the norm of $\mathbf{v}_{\mathbb{B}}$. The binary weight normalisation requires only $O(1)$ FLOPs to calculate the scaling for $\mathbf{v}_{\mathbb{B}}$, whereas the regular weight normalisation requires $O(n)$ FLOPs to calculate the scaling for $\mathbf{v}$. For a model of millions of parameters, this can be a significant speed-up. Binary weight normalisation also has a more straightforward backward pass, since we do not need to take gradients of the $1/||\mathbf{v}||$ term.

Furthermore, convolutions $\mathcal{F}$ and other linear transformations can be implemented using cheap binary operations when using binary weights, $\mathbf{w}_{\mathbb{B}}$, as discussed in Section 3.2.1[4]. However, after applying binary weight normalisation, the weight vector is real-valued, $\mathbf{w}_{\mathbb{R}}$, which would seem to prevent the speed-ups being obtained. Fortunately, due to the properties of linear transformations, we can apply the normalisation factor $\alpha = g/\sqrt{n}$ either before or after applying the convolution to input $\mathbf{x}$.

$$\mathcal{F}(\mathbf{x}, \mathbf{v}_{\mathbb{B}} \cdot \alpha) = \mathcal{F}(\mathbf{x}, \mathbf{v}_{\mathbb{B}}) \cdot \alpha \tag{3.9}$$

So if we wish to utilise fast binary operations for the binary convolution layer, we need to apply binary weight normalisation *after* the convolution. This means that the weights are binary for the convolution operation itself. This couples the convolution operation and the weight normalisation, and we refer to the overall layer as a binary weight normalised convolution, or BWN convolution. Note that the above process applies equally well to other linear transformations.

3.3.1.1 Initialisation of BWN Layers

An important aspect of weight normalised layers is the initialisation. Since we are normalising the weights, and not the output of a layer (like in batch normalisation), at initialisation a weight normalised layer has an unknown output scale. To remedy this, it is usual to use data-dependent initialisation [Salimans and Kingma, 2016], in which some data points are used to set the initial g and b values such that the layer output is approximately unit normal distributed.

[4]This applies when the inputs are real-valued or binary, but the speed-ups are far greater for binary inputs

This can be applied straightforwardly to BWN layers when training the model end-to-end, that is initialising the model at random and training til convergence. It is common, though, when training binary neural networks for classification, to use two-stage training [Alizadeh et al., 2019]. This initialises the underlying weights $\mathbf{v}_{\mathbb{R}}$ of binary layers with the values from a trained model with real-valued weights.

Consider what would happen if we were to try and initialise all the components of a BWN layer with those from a trained layer with real-valued weights. The g and b can be transferred directly, and it is logical to initialise the underlying weights $\mathbf{v}_{\mathbb{R}}$ with the trained $\mathbf{v}$ values. So the magnitude of the overall weight vector $\mathbf{w}$ would remain the same in the BWN layer as in the floating-point layer, since we normalise the $\mathbf{v}_{\mathbb{B}}$ vector and apply the same g, b. This initialisation seems reasonable, but fails in practice. We speculate that the reason that this fails is that, although the magnitude of the weight vector remains the same after transfer, the *direction* can be very different, since the sign function will change the direction of $\mathbf{v}_{\mathbb{R}}$[5]. Since we apply the weight vector by taking products, the output from the initialised binary layer is very different from the trained layer.

A more considered approach is to only initialise the underlying weights $\mathbf{v}_{\mathbb{R}}$ with the values from the trained network, and initialise g and b with data-dependent initialisation as normal. This way, the data-dependent initialisation can compensate for the change of direction that occurs in the binarisation of $\mathbf{v}$. This method does train, but slower than initialising at random and training end-to-end. The only difference between training end-to-end and using this reduced form of two-stage training is the initial values $\mathbf{v}_0$ of the real-valued weights $\mathbf{v}_{\mathbb{R}}$ underlying $\mathbf{v}_{\mathbb{B}}$. In the random initialisation these are sampled from a Gaussian:

$$\mathbf{v}_0 \sim \mathcal{N}(\mathbf{v}_{\mathbb{R}}; 0, 0.05^2) \tag{3.10}$$

We can even normalise the trained real-valued weights such that they have the same mean and variance as the Gaussian (within a weight tensor). This still results in worse performance from the two-stage training. As a result, we simply use the

[5]Note that this effect is stronger in higher dimensional spaces.

random initialisation in our experiments.

3.3.2 Choosing layers to quantise

We aim to binarise deep generative models, in which it is common to utilise residual layers extensively. Residual layers are functions with skip connections:

$$\mathbf{g}_{\text{res}}(\mathbf{x}) = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x}) + \mathbf{x} \tag{3.11}$$

Indeed, the models we target in this thesis, the ResNet VAE and Flow++ models, have the majority of their parameters within residual layers. Therefore they are natural candidates for binarisation, since binarising them would result in a large decrease in the computational cost of the model. To binarise them we implement $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{x})$ in Equation 3.11 using binary weights and possibly activations.

Note that although we have already used the ResNet VAE in our HiLLoC codec in Chapter 3, we choose also the Flow++ model since it is roughly representative of the models used in discrete flows [Hoogeboom et al., 2019, Ho et al., 2019b], which are a viable learned alternative to HiLLoC.

The motivation for using residual layers is that they can be used to add more representative capability to a model without suffering from the *degradation problem* [He et al., 2016]. That is, residual layers can easily learn the identity function by driving the weights to zero. So, if sufficient care is taken with initialisation and optimisation, adding residual layers to the model should not degrade performance, helping to precondition the problem.

Degradation of performance is of particular concern when using binary layers. Binary weights and activations are both less expressive than their real-valued counterparts, and more difficult to optimise. These disadvantages of binary layers are more pronounced for generative modelling than for classification. Generative models need to be very expressive, since we wish to model complex data such as images. Optimisation can also be difficult, since the likelihood of a data point is highly sensitive to the distribution statistics output by the model, and can easily diverge.

This sensitivity of the objective to the output of the model is motivation for

another choice we make in which layers to binarise - we do not binarise the "output layers", i.e. the layers responsible for outputting the distribution statistics themselves. Fortunately, for most models of interest these output layers account for only a small fraction of the total parameters. Further motivation for this choice is that these output layers tend not to be residual layers, due to the restriction that residual layers have the same input and output shape (so that the skip connection, i.e. input, can be added to the output), and our input will often be of different shape than the "hidden layers" (i.e not output layers).

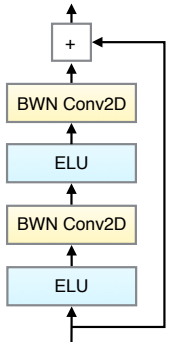
Crucially, if we were to use a residual binary layer *without* weight normalisation, then the layer would not be able to learn the identity function, as the binary weights cannot be set to zero. This would remove the primary motivation to use binary residual layers. In contrast, using a binary weight normalised layer in the residual layer, the gain g and bias b can be set to zero to achieve the identity function. As such, we binarise the ResNet VAE and Flow++ models by implementing the residual layers using BWN layers.

3.3.3 Deep generative models with binary weights

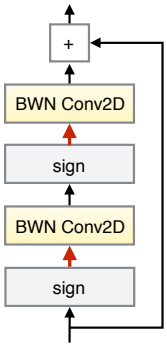
We now describe the binarised versions of the ResNet VAE and Flow++ model, using the techniques and considerations from Section 3.3. Note that, for both the ResNet VAE and Flow++ models, we still retain a valid probabilistic model after binarising the weights. In both cases, the neural networks are simply used to output distribution parameters, which define a normalised density for any set of parameter values.

3.3.3.1 ResNet VAE

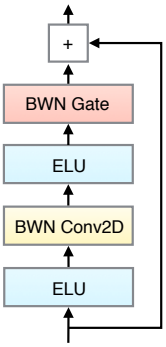
As per Section 3.3.2, we wish to binarise the residual layers of the ResNet VAE. The residual layers are constructed as convolutional residual blocks, consisting of two 3×3 convolutions and non-linearities, with a skip connection. This is shown in Figure 3.1(a)-(b). To binarise the block, we change the convolutions to BWN convolutions, as described in Section 3.3.1. We can either use real-valued activations or binary activations. Binary activations allow the network to be executed much



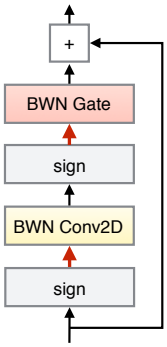
(a) RVAE
(32-bit activations)



(b) RVAE
(1-bit activations)



(c) Flow++
(32-bit activations)



(d) Flow++
(1-bit activations)

Figure 3.1: The residual blocks used in the binarised ResNet VAE and Flow++ models, using both binary and floating-point activations. The BWN Gate layer is a binary weight normalised 1×1 convolution followed by a gated linear unit. We display the binary valued tensors with thick red arrows.

faster, but are less expressive. We use the ELU function as the real-valued activation, and the sign function as the binary activation.

3.3.3.2 Flow++

As with the ResNet VAE, in the Flow++ model the residual layers are structured as stacks of convolutional residual blocks. To binarise the residual blocks, we change both the 3×3 convolution and the gated 1×1 convolution in the residual block to be BWN convolutions. The residual block design is shown in Figure 3.1(c)-(d). We have the option of using real-valued or binary activations.

Table 3.2: Results for binarised ResNet VAE and Flow++ model on CIFAR-10 and ImageNet 32 test sets. The modelling loss is the negative ELBO, reported in bits per dimension. We give the percentage of the model parameters that are binary and the overall size of the model parameters. The weights and activations refer to those within the residual layers of the model, which are the targets for binarisation.

	Precision		**Modelling loss**		**# Parameters**	**% Binary**	**Memory cost**
	Weights	Activations	CIFAR	ImageNet 32			
ResNet VAE	32-bit	32-bit	3.45	4.25	56M	0%	255 MB
	1-bit	32-bit	3.60	4.47	56M	97.1%	13 MB
	1-bit	1-bit	3.73	4.58	56M	97.1%	13 MB
increased width	1-bit	32-bit	3.56	4.42	96M	97.7%	20 MB
	1-bit	1-bit	3.68	4.52	96M	97.7%	20 MB
no residual	N.A.	N.A.	3.78	4.69	1.6M	0%	6 MB
Flow++	32-bit	32-bit	3.21	4.05	34M	0%	129 MB
	1-bit	32-bit	3.29	4.18	34M	90.1%	14 MB
	1-bit	1-bit	3.43	4.30	34M	90.1%	14 MB
no residual	N.A.	N.A.	3.54	4.47	2.2M	0%	9 MB

3.4 Experiments

We run experiments with the ResNet VAE and the Flow++ model, to demonstrate the effect of binarising the models. We train and evaluate on the CIFAR-10 and ImageNet 32 datasets. For both models we use the Adam optimiser [Kingma and Ba, 2015], which has been demonstrated to be effective in training binary neural networks [Alizadeh et al., 2019].

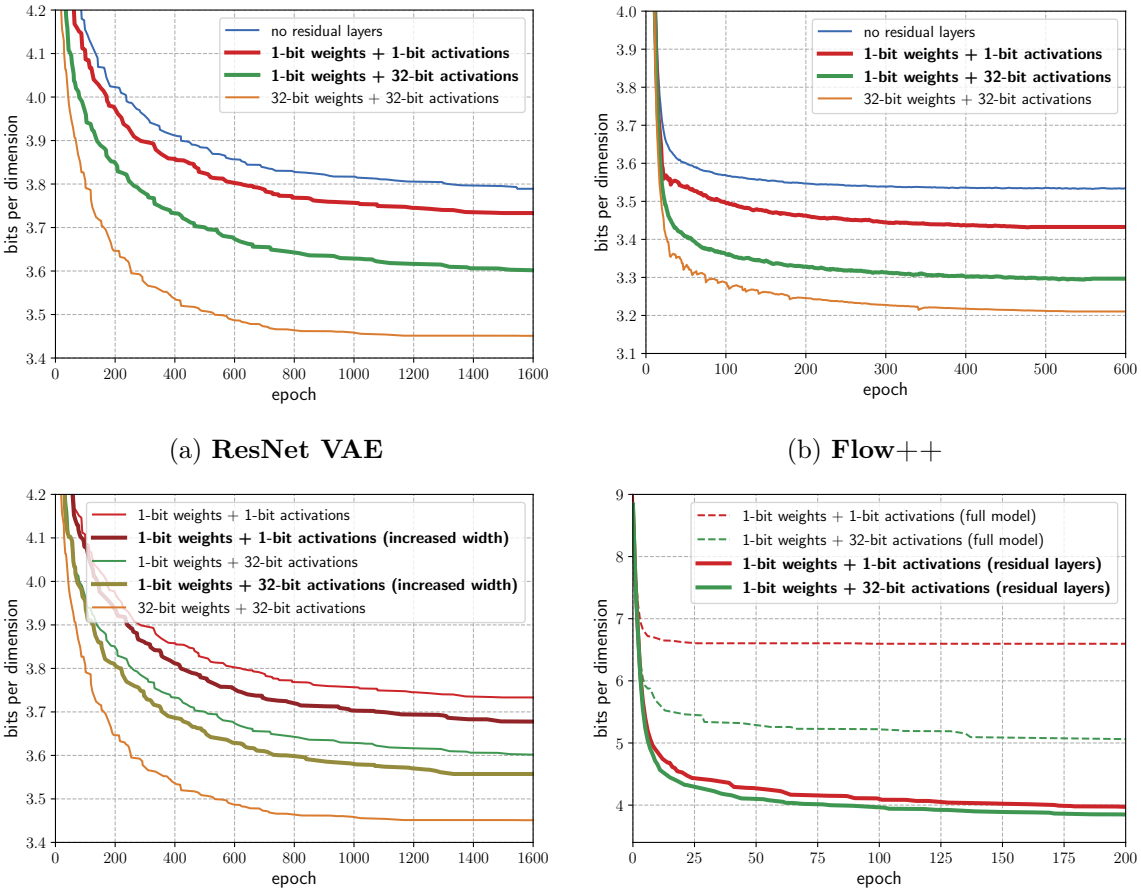
For the ResNet VAE, we decrease the number of latent variables per latent layer and increase the width of the residual channels, as compared to the original implementation. We found that increasing the ResNet blocks in the first latent layer slightly increased modelling performance. Furthermore, we chose not to model the posterior using IAF layers [Kingma et al., 2016], since we want to keep the model class as general as possible.

For the Flow++ model, we decrease the number of components in the mixture of logistics for each coupling layer and increase the width of the residual channels, as compared to the original implementation. For simplicity, we also remove the attention mechanism from the model, since the ablations the authors performed showed that this had only a small effect on the model performance.

Note that we do not use any techniques to try and boost the test performance of our models, such as importance sampling or using weighted averages of the model parameters. These are often used in generative modelling, but since we are trying to establish the relative performance of models with various degrees of binarisation, we prefer to keep the experiments and comparisons simpler.

3.4.1 Density modelling

We display results in Table 3.2. We can see that the models with binary weights and real-valued activations perform only slightly worse than those with real-valued weights, for both the ResNet VAE and the Flow++ models. For the models with binary weights, we observe better performance when using real-valued activations than with the binary activations. These results are as expected given that binary values are by definition less expressive than real values. All models with binary



(a) **ResNet VAE** (b) **Flow++**

(c) **ResNet VAE** (increased channel width) (d) **ResNet VAE** (ablations)

Figure 3.2: Test loss values during training of the ResNet VAE and Flow++ models on the CIFAR dataset. Subfigures (a) and (b): models with binary weights and either binary or real-valued activations. Compared to the model with real-valued weights and activations, and a baseline with the residual layers set to the identity. Subfigures (c) and (d): the effect of increasing the width of the residual channels, and ablations.

weights perform better than a baseline model with the residual layers set to the identity, indicating that the binary layers do learn. We display samples from the binarised models in Figure 3.4.

Importantly, we see that the model size is significantly smaller when using binary weights - 94% smaller for the ResNet VAE and 90% smaller for the Flow++ model.

These results demonstrate the fundamental trade-off that using binary layers in generative models allows. By using binary weights the size of the model can be drastically decreased, but there is a slight degradation in modelling performance. The model can then be made much faster by using binary activations as well as

weights, but this decreases performance further.

Note that in our experiments all models have the same inference speed, since to our knowledge there are no readily usable implementations of efficient binary convolutional kernels - the only ones we are aware of are research code which are not easily transferable in modern machine learning frameworks.

3.4.2 Increasing the residual channels

Binary models are less expensive in terms of memory and compute. This raises the question of whether binary models could be made *larger* in parameter count than the model with real-valued weights, with the aim of trying to improve performance for a fixed computational budget. We examine this by increasing the number of channels in the residual layers (from 256 to 336) of the ResNet VAE. This increases the number of binary weights by approximately 40 million, but leaves the number of real-valued weights roughly constant[6]. The results are shown in Table 3.2 and Figure 3.2(c). We can see the increase the binary parameter count does have a noticeable improvement in performance. The model size increases from 13 MB to 20 MB, which is still an order of magnitude smaller than the model with real-valued weights (255 MB). It is an open question as to how much performance could be improved by increasing the size of the binary layers even further. The barrier to this approach currently is training, since we need to maintain and optimise a set of real-valued weights during training. These get prohibitively large as we increase the model size significantly.

One interesting avenue is to train the binary weights directly, rather than using real-valued weights as a proxy [Gupta et al., 2015, Li et al., 2017], which could potentially alleviate the memory bottleneck during training.

3.4.3 Ablations

We perform ablations to verify our hypothesis from Section 3.3.2 that we should only binarise the residual layers of the generative models. We attempt to binarise

[6]There will be a slight increase, since we use real-valued weights to map to and from the residual channels.

all layers in the ResNet VAE using BWN layers, using both binary and real-valued activations. The results are shown in Figure 3.2(d). As expected, the loss values attained are significantly worse than when binarising only the residual layers.

To examine the performance of the binary weight normalisation (BWN), we perform an ablation against using the more widely used batch normalisation [Ioffe and Szegedy, 2015]. We simply place a batch normalisation operation after every layer, instead of using BWN. Note that this still permits the use of fast binary operations, since the weights and activations are binary valued. We train the Flow++ model with binary weights and both binary and real-valued activations, comparing the two normalisation schemes. The results are shown in Figure 3.3. We can see that BWN is slightly better for the model with binary activations, and significantly better for the model with real-valued activations. Importantly, we have found BWN to be more stable than batch normalisation, which can often result in training instabilities. Indeed, to obtain the results we present when using batch normalisation, training was restarted many times. Binary batch normalisation also generally requires two-stage training, which we use to improve the stability of training, but it is still less stable than using BWN. BWN is also both faster to compute and simpler, not relying on retaining running averages of batch statistics.

It is also worth noting, that it is not possible to train these binary weighted generative models without any form of normalisation, since training is too unstable. This is not surprising, since the binary weights themselves are large in magnitude and can result (in particular with binary activations) in very large layer outputs.

3.5 Conclusion

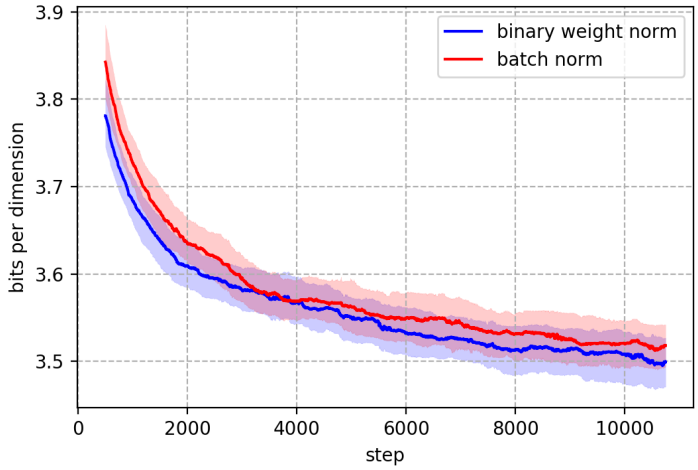
In this chapter we considered the problem of computational efficiency in probabilistic generative models (PGMs), a widely-used class of models with many applications - for example as use as the backbone of learned lossless codecs, as per Chapter 2. Despite their wide usage, such models have had little attention with regards to their computational efficiency, with work on efficiency instead much more widely studied on models for supervised learning.

To explore increasing computational efficiency in PGMs, we focused on using binary-valued weights in the neural networks (termed binary neural networks) of PGMs. Binary neural networks have been shown to be effective at drastically decreasing computational requirements in neural networks for supervised learning, but have never been explored for PGMs.

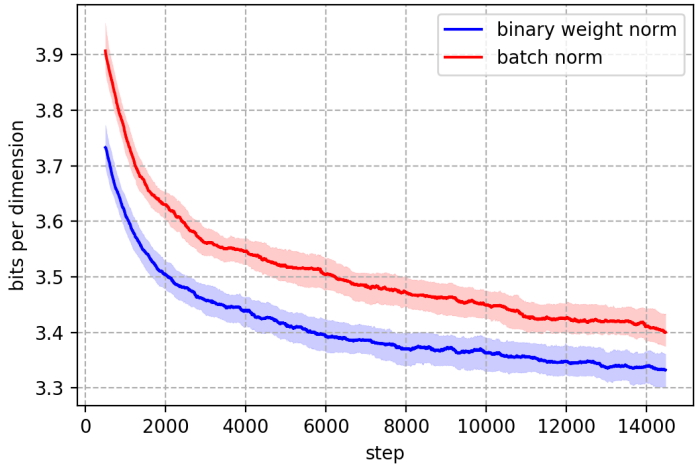
Using binary neural networks in the class of PGMs that we are interested in is not as immediately obvious as the for the classification examples common in the binary neural network literature, where in some cases the entire network can be implemented in binary weights (which we refer to as binarising). We showed that such attempts do not work in for the PGMs we are interested in. Instead, we showed that the choices of which layers of the neural network to binarise is crucial, and architecture dependent. We identified a particular set of layers which, by their design, are natural candidates for binarisation - the residual layers. These layers also often make up the majority of models parameters, which makes storing and executing them efficiently particularly valuable.

Furthermore, we showed that a popular form of normalisation used in PGMs, weight normalisation, has a very simple and practically effective analogue in the case of binary neural networks, which we term binary weight normalisation (BWN).

We verified that both of our contributions, namely the choice of which layers to binarise and using our normalisation scheme, result in an effective method to reduce the computational demands of PGMs. We showed this by taking two powerful PGMs, that are representative of a wide class of models, and showing that we could reduce their space (and potentially time) requirements by a significant margin with a relatively small impact on modelling performance.



(a) **Flow++** (1-bit weights + 1-bit activations)



(b) **Flow++** (1-bit weights + 32-bit activations)

Figure 3.3: Training loss values achieved when using binary weight normalisation and batch normalisation for the training of binary weighted Flow++ models. We run training with 5 different seeds and plot the mean as the solid line and one standard deviation within the shaded region.

(a) ResNet VAE (32-bit weights, 32-bit activations)

(b) Flow+ (32-bit weights, 32-bit activations)

(c) ResNet VAE (1-bit weights, 32-bit activations)

(d) Flow++ (1-bit weights, 32-bit activations)

(e) ResNet VAE (1-bit weights, 1-bit activations)

(f) Flow++ (1-bit weights, 1-bit activations)

Figure 3.4: Samples from the ResNet VAE (left) and Flow++ (right) models trained on CIFAR. We provide samples from the models with (a)/(b) real-valued weights and activations, (c)/(d) binary weights and real-valued activations, (e)/(f) binary weights and activations.

Chapter 4

Scene Compression

The work presented in this chapter was published in [Bird et al., 2021a].

In Chapter 2 we established a method for lossless compression, BB-ANS, which utilises probabilistic generative models (PGMs) and demonstrated its usefulness on the task of image compression. In Chapter 3 we showed that PGMs themselves can be implemented effectively by using binary weights, which drastically reduces the model size and can also theoretically be used to decrease the run-time of such models. In this chapter we will somewhat combine these concepts, by studying how to compress generative models for 3D scene data. Since such models can be used to reconstruct a 3D scene, this amounts to performing lossy compression of the 3D scene data itself, which is a relatively unexplored topic in the literature.

4.1 Introduction

The ability to render 3D scenes from arbitrary viewpoints can be seen as a big step in the evolution of digital multimedia, and has applications such as mixed reality media, graphic effects, design, and simulations. Often such renderings are based on a number of high resolution images of some original scene, and it is clear that to enable many applications, the data will need to be stored and transmitted efficiently over low-bandwidth channels (e.g. to a mobile phone for augmented reality).

Traditionally, the need to compress this data is viewed as a separate need from rendering. For example, light field images (LFI) consist of a set of images taken

from multiple viewpoints. To compress the original views, often standard video compression methods such as HEVC [Sullivan et al., 2012] are repurposed [Jiang et al., 2017, Barina et al., 2019]. Since the range of views is narrow, light field images can be effectively reconstructed by blending a smaller set of representative views [Astola and Tabus, 2018, Jiang et al., 2017, Zhao et al., 2018, Bakir et al., 2018, Jia et al., 2019]. Blending based approaches, however, may not be suitable for the more general case of arbitrary-viewpoint 3D scenes, where a very diverse set of original views may increase the severity of occlusions, and thus would require storage of a prohibitively large number of views to be effective.

A promising avenue for representing more complete 3D scenes is through neural representation functions, which have shown a remarkable improvement in rendering quality [Mildenhall et al., 2020, Sitzmann et al., 2019, Liu et al., 2020, Schwarz et al., 2020]. In such approaches, views from a scene are rendered by evaluating the representation function at sampled spatial coordinates and then applying a differentiable rendering process. Such methods are often referred to as implicit representations, since they do not explicitly specify the surface locations and properties within the scene, which would be required to apply some conventional rendering techniques like rasterisation [Akenine-Möller et al., 2019]. However, finding the representation function for a given scene requires training a neural network. This makes this class of methods difficult to use as a rendering method in the existing framework, since it is computationally infeasible on a low-powered end device like a mobile phone, which are often on the receiving side. Due to the data processing inequality, it may also be inefficient to compress the original views (the training data) rather than the trained representation itself, because the training process may discard some information that is ultimately not necessary for rendering (such as redundancy in the original views or noise). We give a more thorough background on the above methods (both blending and implicit representations) used to model 3D scene data in Section 4.2.

We propose to apply neural representation functions to the scene compression problem by compressing the representation function itself. We use the NeRF model [Mildenhall et al., 2020], a method which has demonstrated the ability to produce high-quality renders of novel views, as our representation function. To reduce

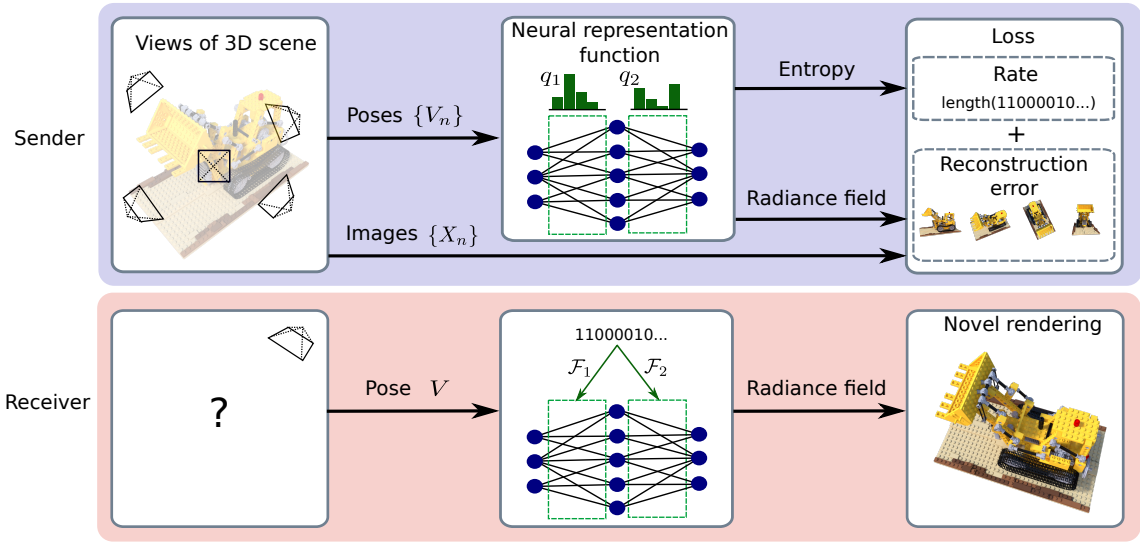


Figure 4.1: Overview of cNeRF. The sender trains an entropy penalised neural representation function on a set of views from a scene, minimising a joint rate-distortion objective. The receiver can use the compressed model to render novel views.

redundancy of information in the model, we build upon the model compression approach of Oktay et al. [2020], applying an entropy penalty to the set of discrete reparameterised neural network weights. The *compressed NeRF* (cNeRF) describes a radiance field, which is used in conjunction with a differentiable neural renderer to obtain novel views (see Figure 4.1). To verify the proposed method, we construct a strong baseline method based on the approaches seen in the field of light field image compression. cNeRF consistently outperforms the baseline method, producing simultaneously superior renders and lower bitrates. We further show that cNeRF can be improved in the low bitrate regime when compressing multiple scenes at once. To achieve this, we introduce a novel parameterisation which shares parameters across models and optimise jointly across scenes.

4.2 Models for 3D scene data

So far in Chapter 3 and Chapter 3, we have considered classes of models that approximate the *unconditional* probability of data, i.e. $p_{\boldsymbol{\theta}}(\mathbf{x})$ where $\mathbf{x}$ is some from data medium such as an image. However, not all tasks are appropriately modelled in this way. One such example is the task of rendering a 3-dimensional (3D) scene from

an arbitrary viewpoint.

As a motivating example, we can image a virtual reality application where we wish to "look around" in some digitally-rendered scene. Since we wish the views to change smoothly as we alter the viewpoint, we need to be able to render an appropriate image from any possible viewpoint.

We can see this is a *conditional* modelling task, that is we have to map from some viewpoint/pose[1] $\mathbf{v}$ to an image $\mathbf{x}$. In this context, the act of storing or transmitting a "scene" refers to storing or transmitting some system that allows the receiver to render the scene from any viewpoint. Thus when we now speak of "compressing a scene" we mean reducing the size of this system.

One observation we can make is that, since we need to be able to render from an arbitrary viewpoint, this makes lossless compression challenging. If we were to do so, it would mean that we would require that the system can render an extremely large[2] set of views without any reconstruction error. Although theoretically possible, we do not know of any effective system for doing this.

Thus we consider instead *lossy compression* for a scene. One consequence is that we are not constrained to probabilistic models, which are a requirement to permit lossless compression via entropy coding. Instead, we can simply model the function mapping from poses $\mathbf{v}$ to images $\mathbf{x}$ as a regression task, minimising some reconstruction error for a given model size. Indeed, there have been recent advancements in learning such rendering functions, which we now review.

4.2.1 Blending approaches to novel view synthesis

If the range of views is narrow, i.e. the difference in location and viewing direction is narrow, then light field images can be effectively reconstructed by *blending* a smaller set of representative views [Astola and Tabus, 2018, Jiang et al., 2017, Zhao et al., 2018, Bakir et al., 2018, Jia et al., 2019].

[1] Note that "view" and "pose" are somewhat synonymous, and both refer to the location and viewing direction of the camera that has taken the image, which we assume to be a perfect pinhole camera.

[2] We don't use the term infinite, since we can imagine some minimum distance between rendered viewpoints which makes the set of possible view finite.

4.2.1.1 Local light field fusion

We now describe the specific blending based approach that we will use as a benchmark in this thesis, Local Light Field Fusion (LLFF) [Mildenhall et al., 2019].

LLFF is a learned approach in which a novel view is rendered by promoting nearby views to multi-plane images (MPIs), which are then blended together. The model consists of two learned components: a component which promotes images with dimension $H \times W \times 3$ to MPIs with dimension $H \times W \times D \times 3$ where D is the number of planes in the MPIs. A 3D convolutional neural network then takes the MPIs for the neighbouring views and predicts blending weights and opacities with which to recombine the blended MPI into a 2D image.

4.2.2 Scene representation functions

Blending based approaches, as described in Section 4.2.1 may not be suitable for the more general case of arbitrary-viewpoint 3D scenes, where a very diverse set of original views may increase the severity of occlusions.

A promising avenue for representing more complete 3D scenes is through neural representation functions, which have shown a remarkable improvement in rendering quality [Mildenhall et al., 2020, Sitzmann et al., 2019, Liu et al., 2020, Schwarz et al., 2020]. In such approaches, views from a scene are rendered by evaluating the representation function at sampled spatial coordinates and then applying a differentiable rendering process. Such methods are often referred to as *implicit representations*, since they do not explicitly specify the surface locations and properties within the scene, which would be required to apply some conventional rendering techniques like rasterisation [Akenine-Möller et al., 2019]. However, finding the representation function for a given scene generally requires training a neural network.

4.2.2.1 Neural radiance fields

We now describe one of the most successful approaches utilising a scene representation function, which we will use in this thesis, neural radiance fields (NeRF) [Mildenhall et al., 2020]. This approach uses a neural network to model a *radiance field*. The

radiance field itself is a learned mapping $g_{\boldsymbol{\theta}}\colon \mathbb{R}^5 \to (\mathbb{R}^3, \mathbb{R}^+)$, where the input is a 3D spatial coordinate $\boldsymbol{p} = (x, y, z) \in \mathbb{R}^3$ and a 2D viewing direction $\boldsymbol{d} = (\omega, \phi) \in \mathbb{R}^2$. The NeRF model also makes use of a positional encoding into the frequency domain, applied elementwise to spatial and directional inputs

$$\boldsymbol{\gamma}(p) = (\sin(2^0\pi p), \cos(2^0\pi p), ..., \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)) \tag{4.1}$$

This type of encoding has been shown to be important for implicit models, which take as input low dimensional data which contains high frequency information [Tancik et al., 2020, Sitzmann et al., 2020].

The network output is an RGB value $\boldsymbol{c} = (r, g, b) \in \mathbb{R}^3$ and a density element $\sigma \in \mathbb{R}^+$. To render a particular view, the RGB values are sampled along the relevant rays and accumulated according to their density elements. In particular, the color $\boldsymbol{c}(\boldsymbol{r})$ of a ray $\boldsymbol{r} = \{\boldsymbol{o} + t\boldsymbol{d} : t \geq 0\}$, in direction $\boldsymbol{d}$ from the camera origin $\boldsymbol{o}$, is computed as

$$\boldsymbol{c}(\boldsymbol{r}) = \sum_{i=1}^{K} T_i(1 - \exp(-\sigma_i\delta_i))\boldsymbol{c}_i, \quad \text{where } T_i = \exp\Big(-\textstyle\sum_{j=1}^{i-1} \sigma_j\delta_j\Big), \tag{4.2}$$

where $(\boldsymbol{c}_i, \sigma_i)$ is the output of the mapping evaluated at $(\boldsymbol{p}_i, \boldsymbol{d})$, where $\boldsymbol{p}_i = \boldsymbol{o} + t_i\boldsymbol{d}$, t_i is the distance of sample i from the origin along the ray, and $\delta_i = t_{i+1} - t_i$ is the distance between samples. The color $\boldsymbol{c}(\boldsymbol{r})$ can be interpreted as the expected color of the point along the ray in the scene closest to the camera, if the points in the scene are distributed along the ray according to an inhomogeneous Poisson process. Since in a Poisson process with density σ_i, the probability that there are no points in an interval of length δ_i is $\exp(-\sigma_i\delta_i)$. Thus T_i is the probability that there are no points between t_1 and t_i, and $(1 - \exp(-\sigma_i\delta_i))$ is the probability that there is a point between t_i and t_{i+1}. The rendered view $\hat{\mathbf{x}}$ comprises pixels whose colors $\boldsymbol{c}(\boldsymbol{r})$ are evaluated at rays emanating from the same camera origin $\boldsymbol{o}$ but having slightly different directions $\boldsymbol{d}$, depending on the camera pose $\mathbf{v}$.

The learned radiance field mapping $g_{\boldsymbol{\theta}}$ is parameterised with two multi-layer

perceptrons (MLPs)[3], which Mildenhall et al. [2020] refer to as the "coarse" and "fine" networks, with parameters $\boldsymbol{\theta}_c$ and $\boldsymbol{\theta}_f$ respectively. The input locations to the coarse network are obtained by sampling regularly along the rays, whereas the input locations to the fine network are sampled conditioned on the radiance field of the coarse network. The networks are trained by minimising the Euclidean distance from their renderings to the ground truth image:

$$\mathcal{L} = \underbrace{\sum_{n=1}^{N} \|\hat{\mathbf{x}}_n^c(\boldsymbol{\theta}_c; \mathbf{v}_n) - \mathbf{x}_n\|_2^2}_{\mathcal{L}_c(\boldsymbol{\theta}_c)} + \underbrace{\sum_{n=1}^{N} \left\|\hat{\mathbf{x}}_n^f(\boldsymbol{\theta}_f; \mathbf{v}_n, \boldsymbol{\theta}_c) - \mathbf{x}_n\right\|_2^2}_{\mathcal{L}_f(\boldsymbol{\theta}_f; \boldsymbol{\theta}_c)} \tag{4.3}$$

Where $||\cdot||_2$ is the Euclidean norm and the $\hat{\mathbf{x}}_n$ are the rendered views. Note that the rendered view from the fine network $\hat{\mathbf{x}}_n^f$ relies on both the camera pose $\mathbf{v}_n$ and the coarse network to determine the spatial locations to query the radiance field. We drop the explicit dependence of $\mathcal{L}_f$ on $\boldsymbol{\theta}_c$ in the rest of the thesis to avoid cluttering the notation. During training, generally only a batch of pixels is rendered rather than the full image.

4.3 Entropy penalised neural representation functions

In this section we will describe our method to perform scene compression via an entropy penalised neural representation function.

4.3.1 Compressed neural radiance fields

To achieve a compressed representation of a scene, we propose to compress the neural scene representation function itself. In this thesis we use the NeRF model as our representation function. To compress the NeRF model, we build upon the model compression approach of Oktay et al. [2020] and jointly train for rendering as well as compression in an end-to-end trainable manner. It is possible to use other

[3]Often also referred to as feed-forward or dense networks, but we shall use MLP in this thesis.

approaches based on model distillation, quantisation or pruning to compress the model. However, the approach of Oktay et al. [2020] is appealing as it allows us to smoothly vary the trade-off between rate and distortion, whereas in many competing model compression approaches this is not possible.

The model compression work of Oktay et al. [2020] reparameterises the model weights Θ into a latent space as Φ. The latent weights are decoded by a learned function $\mathcal{F}$, i.e. $\Theta = \mathcal{F}(\Phi)$. The latent weights Φ are modeled as samples from a learned prior q, such that they can be entropy coded according to this prior. To minimise the rate, i.e. length of the bit string resulting from entropy coding these latent weights, a differentiable approximation of the information content $I(\boldsymbol{\Phi}) = -\log_2(q(\boldsymbol{\Phi}))$ of the latent weights is penalised. The continuous Φ are quantised before being applied in the model, with the straight-through estimator [Bengio et al., 2013] used to obtain surrogate gradients of the loss function.

Following Ballé et al. [2017], uniform noise is added when learning the continuous prior $q(\boldsymbol{\Phi} + \boldsymbol{u})$ where $u_i \sim U(-\frac{1}{2}, \frac{1}{2})\ \forall\ i$. This uniform noise is a stand-in for the quantisation, and results in a good approximation for the self-information through the negative log-likelihood of the noised continuous latent weights. After training, the quantised weights $\tilde{\Phi}$ are obtained by rounding, $\tilde{\Phi} = \lfloor \Phi \rceil$, and transmitted along with discrete probability tables obtained by integrating the density over the quantisation intervals. The continuous weights Φ and any parameters in q itself can then be discarded.

We now consider how to apply this model compression technique to NeRF, with the resulting combination referred to subsequently as cNeRF. Combining the rate-distortion trade-off with the fine and coarse networks of NeRF, as described in Section 4.2.2.1, results in the full objective that we seek to minimise:

$$\mathcal{L}(\Phi, \Psi) = \underbrace{L_c(\mathcal{F}_c(\tilde{\Phi}_c)) + L_f(\mathcal{F}_f(\tilde{\Phi}_f))}_{\text{Distortion}} + \lambda \underbrace{\sum\nolimits_{\boldsymbol{\phi} \in \Phi} I(\boldsymbol{\phi})}_{\text{Rate}} \tag{4.4}$$

where Ψ denotes the parameters of $\mathcal{F}$ as well any parameters in the prior distribution q, and we have explicitly split Φ into the coarse Φ_c and fine Φ_f components such that $\Phi = \{\Phi_c, \Phi_f\}$. λ is a trade-off parameter that balances between rate and distortion.

A rate–distortion (RD) plot can be traced by varying λ to explore the performance of the compressed model at different bitrates. The distortion is generally the Euclidean distance between the rendering and the true image, as defined in Section 4.2.

4.3.2 Compressing a single scene

When training cNeRF to render a single scene, we have to choose how to parameterise and structure $\mathcal{F}$ and the prior distribution q over the network weights. Since the networks are MLPs, the model parameters for a layer l consist of the kernel weights and biases $\{\mathbf{W}_l, \mathbf{b}_l\}$. We compress only the kernel weights $\mathbf{W}_l$, leaving the bias uncompressed since it is much smaller in size. The quantised kernel weights $\tilde{\mathbf{W}}_l$ are mapped to the model weights by $\mathcal{F}_l$, i.e. $\mathbf{W}_l = \mathcal{F}_l(\tilde{\mathbf{W}}_l)$. $\mathcal{F}_l$ is constructed as an affine scalar transformation, which is applied elementwise to $\tilde{\mathbf{W}}_l$:

$$\mathcal{F}_l(\tilde{W}_{l,ij}) = \alpha_l \tilde{W}_{l,ij} + \beta_l \tag{4.5}$$

We take the prior to be factored over the layers, such that we learn a prior per linear kernel q_l. Within each kernel, we take the weights in $\tilde{\mathbf{W}}_l$ to be i.i.d. from the univariate distribution q_l, parameterised by a small MLP, as per the approach of Ballé et al. [2017]. Note that the parameters of this MLP can be discarded after training (once the probability mass functions have been built).

Figure 4.2: Renderings of the synthetic Lego scene and real Fern scene from the uncompressed NeRF model, at 32 bits per parameter (bpp), and from cNeRF with $\lambda \in \{0.0001, 0.01\}$.

4.3.3 Compressing multiple scenes

While the original NeRF model is trained for a single scene, we hypothesise that better rate–distortion performance can be achieved for multiple scenes, especially if they share information, by training a joint model. For a dataset of M scenes, we parameterise the kernel weights of model m, layer l as:

$$\begin{aligned}\mathbf{W}_l^m &= \mathcal{F}_l^m(\tilde{\mathbf{W}}_l^m, \tilde{\mathbf{S}}_l) \\ &= \alpha_l^m \tilde{\mathbf{W}}_l^m + \beta_l^m \mathbf{1} + \gamma_l \tilde{\mathbf{S}}_l \end{aligned} \tag{4.6}$$

Where $\mathbf{1}$ is a matrix of ones. Compared to Equation 4.5, we have added a shift, parameterised as a scalar linear transformation of a discrete shift $\tilde{\boldsymbol{S}}_l$, that is shared across all models $m \in \{1, ..., M\}$. $\tilde{\mathbf{S}}_l$ has the same dimensions as the kernel $\mathbf{W}_l^m$, and as with the discrete latent kernels, $\tilde{\mathbf{S}}_l$ is coded by a learned probability distribution. The objective for the multi-scene model becomes:

$$\mathcal{L}(\Phi, \Psi) = \sum_{m=1}^{M} \left[L_c^m(\mathcal{F}_c^m(\tilde{\Phi}_c^m, \tilde{\Phi}_c^s)) + L_f^m(\mathcal{F}_f^m(\tilde{\Phi}_f^m, \tilde{\Phi}_f^s)) + \lambda \sum_{\boldsymbol{\phi} \in \Phi^m} I(\boldsymbol{\phi}) \right] + \lambda \sum_{\boldsymbol{\phi} \in \Phi^s} I(\boldsymbol{\phi}) \tag{4.7}$$

where Φ^s is the set of all discrete shift $\tilde{S}$ parameters, and the losses, latent weights and affine transforms are indexed by scene and model m. Note that this parameterisation has *more* parameters than the total of the M single scene models, which at first appears counter-intuitive, since we wish to reduce the overall model size. It is constructed as such so that the multi-scene parameterisation contains the M single scene parameterisations - they can be recovered by setting the shared shifts to zero. If the shifts are set to zero then their associated probability distributions can collapse to place all their mass at zero. So we expect that if there is little benefit to using the shared shifts then they can be effectively ignored, but if there is a benefit to using them then they can be utilised. As such, we can interpret this parameterisation as inducing a soft form of parameter sharing.

Figure 4.3: Rendering of the Horn scene from compressed NeRF (left) and HEVC + LLFF (right). Despite using more than twice the space as cNeRF (2,002KB, compared to 995KB), HEVC + LLFF shows obvious artifacts, such as the highlighted ghosting effect around the edge of the head.

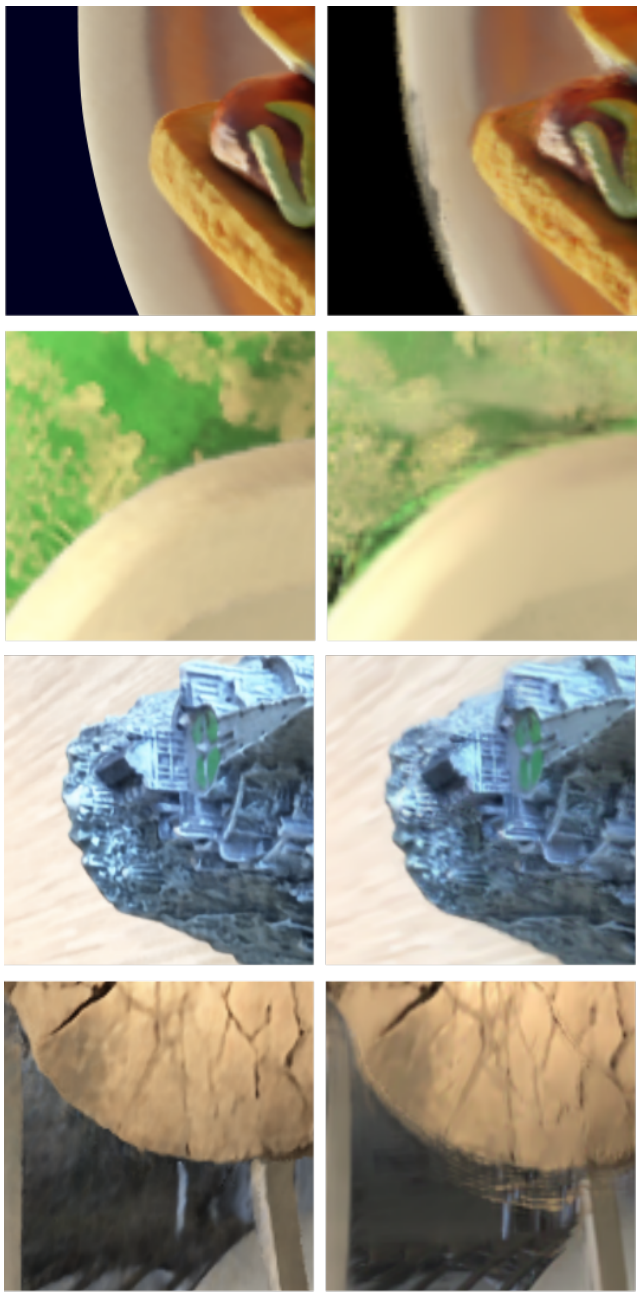


Figure 4.4: A comparison of four (zoomed in) renderings from cNeRF with $\lambda = 0.0001$ and HEVC + LLFF with QP=30. HEVC + LLFF shows obvious artifacts such as ghosting around edges and an overall less crisp rendering.

4.4 Experiments

4.4.1 Datasets

To demonstrate the effectiveness of our method, we evaluate on two sets of scenes used by Mildenhall et al. [2020]:

- *Synthetic.* Consisting of 800×800 pixel views taken from either the upper hemisphere or entire sphere around an object rendered using the Blender software package. There are 100 views taken to be in the train set and 200 in the test set.

- *Real.* Consisting of a set of forward facing 1008×756 pixel photos of a complex scene. The number of images varies between 20 and 62 per scene, with 1/8 of the images taken as the test images.

Since we are interested in the ability of the receiver to render novel views, all distortion results (for any choice of perceptual metric) presented are given on the test sets.

4.4.2 Architecture and optimisation

We maintain the same architecture for the NeRF model as Mildenhall et al. [2020], consisting of 13 linear layers and ReLU activations. For cNeRF we use Adam [Kingma and Ba, 2015] to optimise the latent weights Φ and the weights contained in the decoding functions $\mathcal{F}$. For these parameters we use initial learning rate of 5×10^{-4} and a learning rate decay over the course of learning, as per Mildenhall et al. [2020]. For the parameters of the learned probability distributions q, we find it beneficial to use a lower learning rate of 5×10^{-5}, such that the distributions do not collapse prematurely. We initialise the latent linear kernels using the scheme of Glorot and Bengio [2010], and the decoders $\mathcal{F}$ near the identity.

4.4.3 Building a baseline

To understand how effective cNeRF is, it is necessary to compare to a reasonable baseline method. Since we are, to our knowledge, the first to study this problem of scene compression, no such baseline yet exists.

To build our baseline method, we follow the general methodology exhibited in light field compression and take the compressed representation of the scene to be a compressed subset of the views. The receiver then decodes these views, and renders novel views conditioned on the reconstructed subset. We use the video codec HEVC to compress the subset of views, as is done by Jiang et al. [2017]. To render novel views conditioned on the reconstructed set of views, we choose the Local Light Field Fusion (LLFF) approach of Mildenhall et al. [2019]. LLFF is a state-of-the-art learned approach in which a novel view is rendered by promoting nearby views to multiplane images, which are then blended. We refer to the full baseline subsequently as HEVC + LLFF.

There exist a number of design choices and hyperparameters with this baseline, which we refer to subsequently as HEVC + LLFF. The number of views we compress from a scene has a direct impact on both the compressed size and the quality of the reconstructions. More compressed views corresponds to a higher compressed size, although the marginal compressed size of an image may decrease given that most images will be coded as residual frames with HEVC. On the other hand, more compressed views will generally result in higher quality renderings from LLFF, since it essentially interpolates known views to render new ones. Another factor is the order that we choose to compress the views using HEVC, since there is (usually) no explicit time ordering in the views themselves. Intuitively, if we pick the compression ordering such that the views appear more like a natural video, then we expect HEVC compression to improve.

We alter the quantisation parameter (QP) of HEVC to explore the rate distortion frontier, fixing the rest of the hyperparameters to sensible values. The QP determines the level of quantisation used in the coding procedure, with higher values leading to smaller compressed size and lower quality. We study the effects of the QP, the number of transmitted images and the compression order in our ablations in Section 4.5.1.

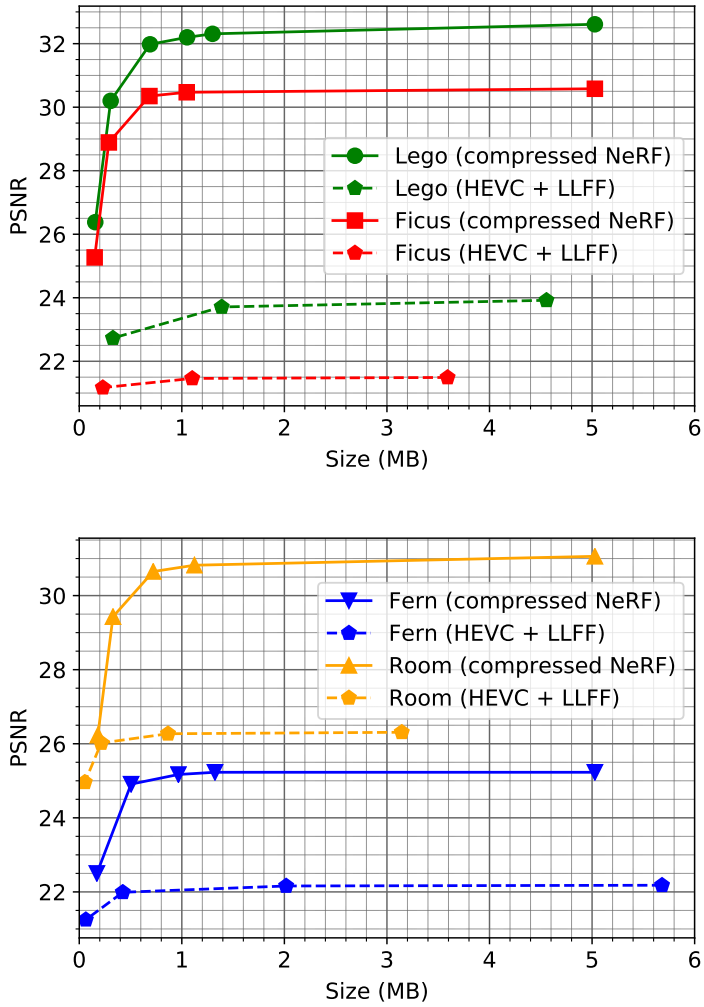


Figure 4.5: Rate–distortion curves for both the cNeRF and HEVC + LLFF approaches, on two synthetic (left) and two real (right) scenes. We include uncompressed NeRF, which is the rightmost point on the curve and a fixed size across scenes. We truncate the curves for HEVC + LLFF, since increasing the bitrate further does not improve PSNR. See Figure 4.9 for the full curves.

	NeRF	**cNeRF** ($\lambda = 1e^{-4}$)			**HEVC + LLFF** (QP=30)	
Scene	PSNR	PSNR	Size (KB)	Reduction	PSNR	Size (KB)
Chair	33.51	32.28	621	8.32×	24.38	5,778
Drums	24.85	24.85	870	5.94×	20.25	5,985
Ficus	30.58	30.35	701	7.37×	21.49	3,678
Hotdog	35.82	34.95	916	5.65×	30.18	2,767
Lego	32.61	31.98	707	7.31×	23.92	4,665
Materials	29.71	29.17	670	7.71×	22.49	3,134
Mic	33.68	32.11	560	9.23×	28.95	3,032
Ship	28.51	28.24	717	7.21×	24.95	5,881
Room	31.06	30.65	739	7.00×	26.27	886
Fern	25.23	25.17	990	5.22×	22.16	2,066
Leaves	21.10	20.95	1,154	4.48×	18.15	3,162
Fortress	31.65	31.15	818	6.32×	26.57	1,149
Orchids	20.18	20.09	1,218	4.24×	17.87	2,357
Flower	27.42	27.21	938	5.51×	23.46	1,009
T-Rex	27.24	26.72	990	5.22×	22.30	1,933
Horns	27.80	27.28	995	5.20×	20.71	2,002

Table 4.1: Results comparing the uncompressed NeRF model,cNeRF and HEVC + LLFF baseline. We pick λ and QP to give a reasonable trade-off between bitrate and PSNR. The reduction column is the reduction in the size of cNeRF as compared to the uncompressed NeRF model, which has a size of 5,169KB. Note that for all scenes, cNeRF achieves both a higher PSNR and a lower bitrate than HEVC + LLFF.

	cNeRF ($\lambda = 1e^{-4}$)		**HEVC + LLFF** (QP=30)	
Scene	PSNR(Y)	PSNR(UV)	PSNR(Y)	PSNR(UV)
Chair	36.64	45.21	33.41	42.46
Drums	26.68	37.17	21.72	34.72
Ficus	31.77	43.50	24.05	37.00
Hotdog	36.31	42.70	31.98	41.62
Lego	29.89	40.56	24.94	35.92
Materials	26.82	38.53	16.23	35.04
Mic	33.36	48.02	29.87	48.64
Ship	28.27	38.46	26.48	38.48
Room	32.59	44.94	27.05	42.50
Fern	25.16	40.33	22.15	38.06
Leaves	21.17	36.09	18.43	34.52
Fortress	31.60	44.70	27.31	41.67
Orchids	20.43	34.65	18.17	32.11
Flower	27.91	38.25	24.27	33.90
T-Rex	26.77	42.64	22.42	40.07
Horns	27.68	42.65	22.57	40.13

Table 4.2: PSNR values comparing cNeRF to HEVC + LLFF. The images are rendered in the YUV color encoding and the PSNR is computed in the Y channel and average of the UV channels. cNeRF is superior to HEVC + LLFF in PSNR(Y) and PSNR(UV) for all scenes except PSNR(UV) for Mic and Ship. Note that the bitrates are the same as for Table 4.1.

	cNeRF ($\lambda = 1e^{-4}$)			**HEVC + LLFF** (QP=30)		
Scene	MS-SSIM(Y)	MS-SSIM(RGB)	LPIPS	MS-SSIM(Y)	MS-SSIM(RGB)	LPIPS
Chair	0.997	0.997	0.014	0.989	0.989	0.026
Drums	0.953	0.952	0.070	0.910	0.909	0.109
Ficus	0.986	0.986	0.023	0.925	0.924	0.069
Hotdog	0.992	0.990	0.041	0.983	0.981	0.070
Lego	0.983	0.980	0.034	0.951	0.945	0.080
Materials	0.971	0.970	0.047	0.833	0.832	0.172
Mic	0.992	0.992	0.022	0.988	0.988	0.026
Ship	0.891	0.889	0.201	0.879	0.883	0.174
Room	0.979	0.977	0.087	0.950	0.948	0.168
Fern	0.934	0.932	0.187	0.862	0.862	0.239
Leaves	0.909	0.905	0.204	0.843	0.842	0.244
Fortress	0.966	0.962	0.090	0.918	0.914	0.165
Orchids	0.852	0.849	0.220	0.720	0.721	0.320
Flower	0.945	0.941	0.138	0.911	0.906	0.171
T-Rex	0.962	0.961	0.101	0.905	0.903	0.182
Horns	0.953	0.951	0.169	0.886	0.884	0.238

Table 4.3: MS-SSIM and LPIPS values comparing cNeRF to HEVC + LLFF. MS-SSIM is given in both the Y channel (from the YUV color encoding) and RGB. LPIPS is given for RGB, as it is only defined for such. In all cases, cNeRF is superior to HEVC + LLFF in MS-SSIM and LPIPS. Note that the bitrates are the same as for Table 4.1

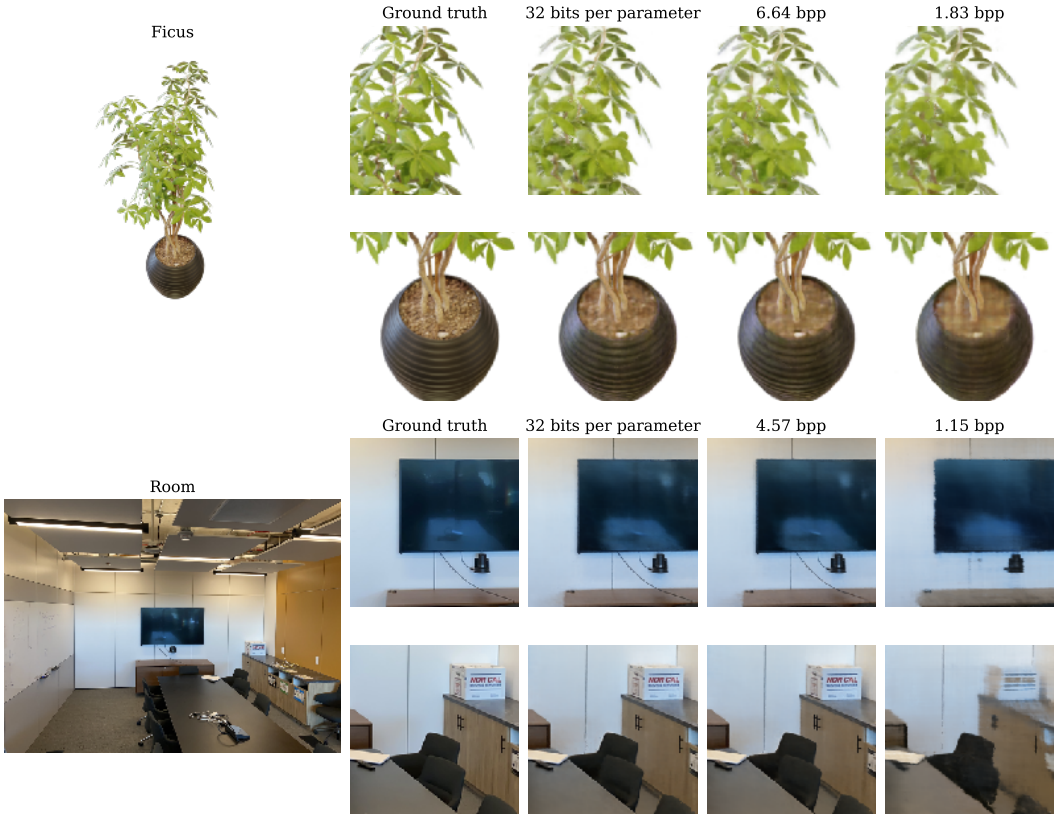


Figure 4.6: Renderings of the synthetic Ficus scene and real Room scene from the uncompressed NeRF model, at 32 bits per parameter (bpp), and from cNeRF with $\lambda \in \{0.0001, 0.01\}$.

4.4.4 Results

4.4.4.1 Single scene compression

To explore the frontier of achievable rate–distortion points for cNeRF, we evaluate at a range of entropy weights λ for four scenes – two synthetic (Lego and Ficus) and two real (Fern and Room). To explore the rate–distortion frontier for the HEVC + LLFF baseline we evaluate at a range of QP values for HEVC. We give a more thorough description of the exact specifications of the HEVC + LLFF baseline and the ablations we perform to select the hyperparameter values in Section 4.5.1. We show the results in Figure 4.5. We also plot the performance of the uncompressed NeRF model – demonstrating that by using entropy penalisation the model size can be reduced substantially with a relatively small increase in distortion. For

these scenes we plot renderings at varying levels of compression in Figure 4.2 and Figure 4.6. The visual quality of the renderings does not noticeably degrade when compressing the NeRF model down to bitrates of roughly 5-6 bits per parameter (the precise bitrate depends on the scene). At roughly 1 bit per parameter, the visual quality has degraded significantly, although the renderings are still sensible and easily recognisable. We find this to be a surprising positive result, given that assigning a single bit per parameter is extremely restrictive for such a complex regression task as rendering. Indeed, to our knowledge no binary neural networks have been demonstrated to be effective on such tasks.

Although the decoding functions $\mathcal{F}$ (Equation 4.5) are just relatively simple scalar affine transformations, we do not find any benefit to using more complex decoding functions. With the parameterisation given, most of the total description length of the model is in the coded latent weights, not the parameters of the decoders or entropy models. We give a full breakdown in Table 4.5.

Figure 4.5 shows that cNeRF clearly outperforms the HEVC + LLFF baseline, always achieving lower distortions at a (roughly) equivalent bitrate. Reconstruction quality is reported as peak signal-to-noise ratios (PSNR). The results are consistent with earlier demonstrations that NeRF produces much better renderings than the LLFF model [Mildenhall et al., 2020]. However, it is still interesting to see that this difference persists even at much lower bitrates. To evaluate on the remaining scenes, we select a single λ value for cNeRF and QP value for HEVC + LLFF. We pick the values to demonstrate a reasonable trade-off between rate and distortion. The results are shown in Table 4.1. For every scene the evaluated approaches verify that cNeRF achieves a lower distortion at a lower bitrate. We can see also that cNeRF is consistently able to reduce the model size significantly without seriously impacting the distortion. Further, we evaluate the performance of cNeRF and HEVC + LLFF for other perceptual quality metrics in Table 4.2 and 4.3. Although cNeRF is trained to minimise the squared error between renderings and the true images (and therefore maximise PSNR), cNeRF also outperforms HEVC + LLFF in both MS-SSIM [Wang et al., 2003] and LPIPS [Zhang et al., 2018]. This is significant, since the results of Mildenhall et al. [2020] indicated that for SSIM and LPIPS, the LLFF model

had a similar performance to NeRF when applied to the real scenes. We display a comparison of renderings from cNeRF and HEVC + LLFF in Figure 4.4.

4.4.4.2 Multi-scene compression

For the multi-scene case we compress one pair of synthetic scenes and one pair of real scenes. We train the multi-scene cNeRF using a single shared shift per linear kernel, as per Equation 4.6. To compare the results to the single scene models, we take the two corresponding single scene cNeRFs, sum the sizes and average the distortions. We plot the resulting rate–distortion frontiers in Figure 4.7. The results demonstrate that the multi-scene cNeRF improves upon the single scene cNeRFs at low bitrates, achieving higher PSNR values with a smaller compressed size. This meets our expectation, since the multi-scene cNeRF can share parameters via the shifts (Equation 4.6) and so decrease the code length of the scene-specific parameters. At higher bitrates we see no benefit to using the multi-scene parameterisation, and in fact see slightly worse performance. This indicates that in the unconstrained rate setting, there is no benefit to using the shared shifts, and that they may slightly harm optimisation.

4.5 Ablations

4.5.1 HEVC + LLFF specification

There are many hyperparameters to select for the HEVC + LLFF baseline. The first we consider is the number of images to compress with HEVC. If too many images are compressed with HEVC then at some point the performance of LLFF will saturate and an unnecessary amount of space will be used. On the other hand, if too few images are compressed with HEVC, then LLFF will find it difficult to blend these (de)compressed images to form high quality renders. To illustrate this effect, we run an ablation on the Fern scene where we vary the number of images we compress with HEVC, rendering a held out set of images conditioned on the reconstructions. The results are displayed in Figure 4.8. We can clearly see the saturation point at around

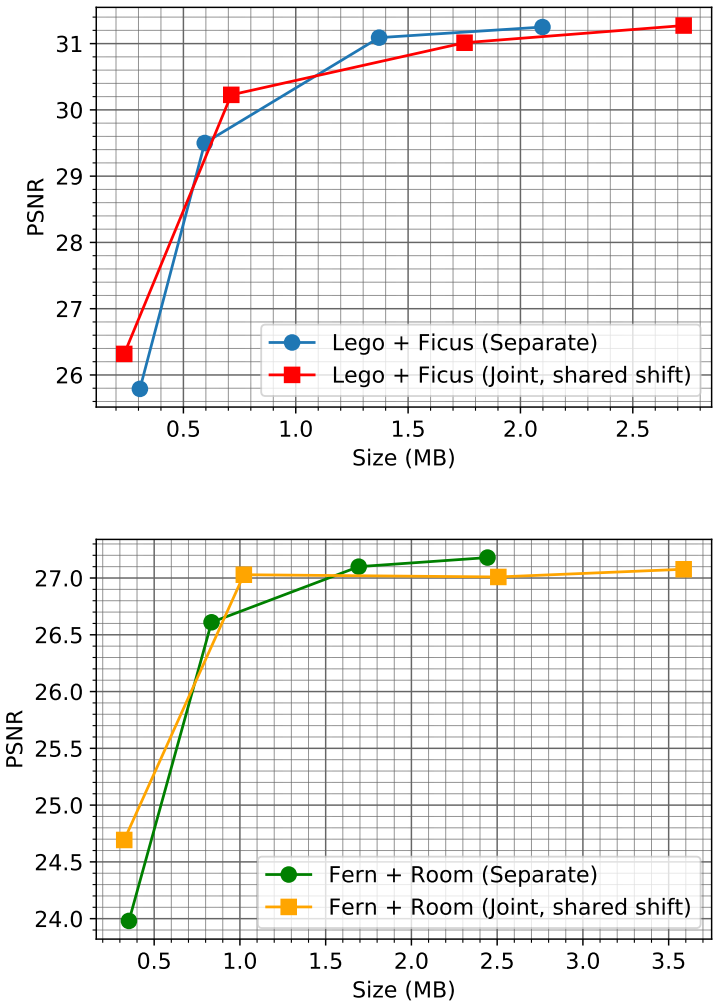


Figure 4.7: Rate–distortion curves for comparing the multi-scene model with a single shared shift to the single scene models. The models are shown for two synthetic (left) and two real scenes (right).

10 images, beyond which there is no benefit to compressing extra images. Thus when picking the number of images to compress for new scenes, we do not use more than 4 per test image (which corresponds to compressing 12 images in our ablation).

The second effect we study is the order in which images are compressed with HEVC, which affects the performance as HEVC is a video codec and thus sensitive to image ordering. It stands to reason that the more the sequence of images resemble a natural video, the better coding will be. As such, we consider two orderings: firstly the "snake scan" ordering, in which images are ordered vertically by their camera pose, going alternately left to right then right to left. The second is the "lozenge"

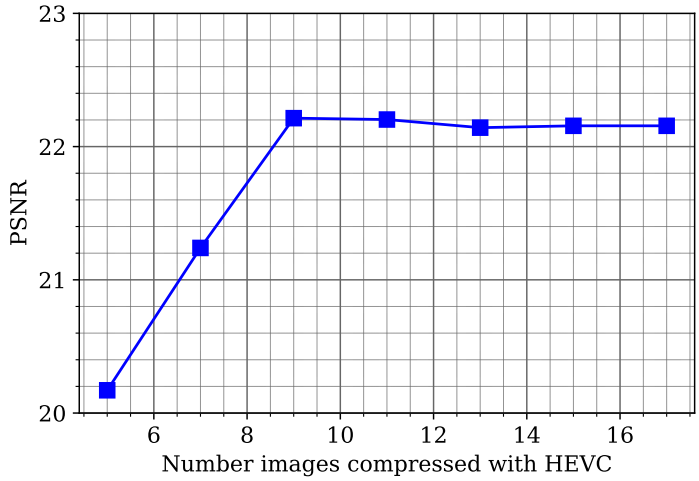


Figure 4.8: Test performance of the HEVC + LLFF baseline across different number of images compressed with HEVC. The decompressed images are used by LLFF to reconstruct the test views.

ordering [Jiang et al., 2017], in which images are ordered by the camera pose in a spiral outwards from their centre. Both orderings appear sensible since they always step from a given camera pose to an adjacent pose. We compare results compressing and reconstructing a set of images using HEVC across a range of Quantisation Parameter (QP) values for the Fern scene in Table 4.4. The difference between the two orderings is very small. Since snake scan is simpler to implement, we use this in all our experiments.

The effect of changing QP is demonstrated in Figure 4.9, and we select QP=30 for the experiments in which we choose one rate–distortion point to evaluate, since it achieves almost the same performance as QP=20 and QP=10 with considerably less space.

QP	Snake scan	Lozenge
10	50.9	50.8
20	42.5	42.4
30	33.9	33.8
40	26.7	26.8
50	22.6	22.5

Table 4.4: PSNR values from compressing and reconstructing a set of images from the Fern scene, with two different orderings on the images.

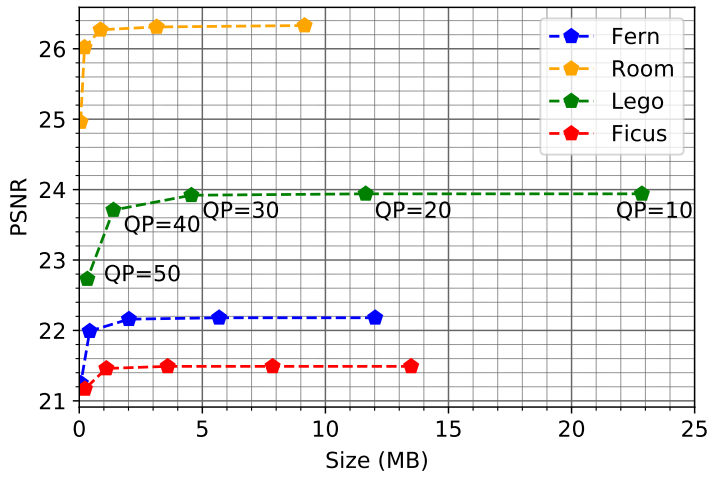


Figure 4.9: Full rate–distortion curves for HEVC + LLFF, with labels showing the effect of the QP parameter. To avoid clutter, only the Lego QP labels are given, and the other scenes are similarly ordered from QP=10 on the right to QP=50 on the left.

Entropy weight λ	Rate (KB)	Overhead (KB)
1×10^{-2}	119	23
1×10^{-3}	293	27
1×10^{-4}	673	34
1×10^{-5}	1061	42

Table 4.5: Breakdown of the cNeRF size across four entropy weights trained on the Lego scene. The size is divided into the size of the coded latent weights (the rate) and everything else (the overhead). The overhead consists of description lengths of the probability tables built from the prior q, the parameters of the decoding functions $\mathcal{F}$ and any bias parameters.

4.6 Conclusion

In this chapter we considered the problem of how to compress the data from a 3D scene. Although there is growing amounts of 3D scene data being generated and transmitted, this is still a relatively unstudied topic within learned compression. In particular we focused on a codec which can be used to render a scene from an arbitrary viewpoint, which is a significantly more challenging problem than communicating a fixed set of renderings, which has many parallels with video compression.

To compress such a system, we utilised recent advances in scene representation functions - learned models which can be used to render a scene from any angle. We use the Neural Radiance Field (NeRF) model, which has been demonstrated to be

an effective model for a range of scenes. By using a scene representation function, we showed that the task of compressing the scene is condensed into compressing the representation function itself. Since in NeRF the model is parameterised via a neural network, this becomes a task of compressing a neural network.

To this end, we introduced compressed NeRF (cNeRF), which uses a state-of-the-art method in learned model compression to compress the neural network within NeRF. Since the model is optimised for a given rate-distortion trade-off, we can change the trade-off parameter to satisfy a range of constraints on bit-rate.

To explore how effective cNeRF is, we constructed a baseline that does not utilise scene representation functions. Specifically, we combined a well-known video codec, HEVC, with a state-of-the-art method for blending views to render from a new viewpoint. We then demonstrated that cNeRF always achieves a superior rate-distortion trade-off to this baseline across a range of real and synthetic scene data. Furthermore, we explored how cNeRF performs at a variety of rate-distortion trade-offs. Surprisingly, our results showed that cNeRF can produce visually appealing reconstructions even at extremely low bit-rates.

Conclusion

This thesis has the intention of making advances in the field of learned compression and the associated methods. We now conclude with a consideration of how the research presented works towards this goal.

In Chapter 2 we presented a novel method to use latent variable models as the backbone of a lossless compression system. By pairing the well-known bits-back coding argument with the modern ANS entropy coder, resulting in a method we call BB-ANS [Townsend et al., 2019], we showed that we could losslessly compress data at close to the theoretical compression rate as given by the bits-back coding. As a proof-of-concept, we demonstrated that a small latent variable model can be used to compress the simple MNIST dataset as well as a suite of generic benchmark codecs.

Previous attempts at implementing bits-back coding have suffered from unavoidable overheads [Frey, 1997], which made them impractical. In contrast, BB-ANS has no significant overheads. Thus it is an important step into furthering research into learned lossless compression, as it opens up the widely used class of latent variable models as possible lossless compressors. Many of the improvements into better latent variable models (as measured by modelling performance) can now be translated into improvements in learned lossless compression, simply by using BB-ANS. For example, a new class of *variational diffusion models* [Kingma et al., 2021] has been developed since we published our papers on BB-ANS. These new models were immediately applied to lossless compression using a derivative scheme of BB-ANS, demonstrating state-of-the-art compression rates.

We also showed in Chapter 2 how to make steps towards making BB-ANS an effective generic image codec. We implemented BB-ANS using a large, hierarchical latent variable model, which results in strong compression performance. Further, by

training the model on ImageNet [Deng et al., 2009] we demonstrate that the resulting codec can be robust to small shifts in image distribution, and different image sizes than those used during training. Clearly this is an important step towards developing learned, generic compressors. The hierarchical latent variable model introduces some practical considerations when used in conjunction with BB-ANS. To address these, we then showed how to handle the discretisation of the hierarchical latent space, how to initialise the bits-back chain without incurring a significant overhead and how to use a vectorised ANS coder to increase coding speed on large images. We termed the resulting method HiLLoC, and demonstrated that it can compress data from the CIFAR [Krizhevsky, 2009] and ImageNet datasets at rates superior to the best generic codecs available at the time, and also superior to Bit-Swap [Kingma et al., 2019], the main competing method that utilises bits-back coding.

In Chapter 3 we considered how to make the underlying models used in learned compression more computationally efficient. The neural network based architectures that are prevalent in modern methods can have run-times and space requirements that make them prohibitive for use in certain applications. Compression can be one such application, since codecs have to be reasonable in time and space requirements in order to be viable to run in a widespread fashion, for example utilising edge devices such as personal phones.

In order to make the models used in learned compression more computationally efficient, we investigated the possibility of implementing them using binary neural networks. Binary neural networks are a research topic that has been explored recently as they drastically reduce the space requirements of neural networks (versus using the usual range of floating-point precision such as 32-bit), and can also be used to achieve large speed-ups via the implementation of efficient kernels that exploit the binary precision.

However, there has not been, to our knowledge, any research into applying binary neural networks for unsupervised learning, which encompasses the class of methods used for learned compression. As such, we explored implementing two modern, powerful probabilistic generative models - Flow++ [Ho et al., 2019a] and the ResNet VAE [Kingma et al., 2016] - using binary neural networks. We showed

that a simple and fast variant of weight normalisation, which we term BWN, is superior to the previously developed batch normalisation used for binary neural networks [Hubara et al., 2016]. We also motivate implementing only certain parts of the neural network architecture. In particular we demonstrate, theoretically and empirically, that only implementing the residual layers [He et al., 2016] is an effective measure. We demonstrate that on CIFAR and ImageNet, our proposed methods result in a greater than 90% saving in space requirement, with only a small amount of performance degradation.

Lastly, in Chapter 4 we considered how to apply learned compression to a different data domain, namely that of images of 3D scenes. The applications for such 3D data, such as virtual and augmented reality, are relatively new. As such, research into learned compression for 3D data is largely unexplored. Given the growth of technologies that utilise this data, and that the data itself can have very large space requirements, research into effective compression of 3D scene data has the potential to be valuable.

In order to investigate learned compression for 3D scene data, we explored the possibility of compressing a neural scene representation function, in particular using the successful NeRF method [Mildenhall et al., 2020]. Since the learned component of NeRF is a neural network, and the NeRF model can render arbitrary views from a scene, the act of compressing the scene data is translated into compressing the neural network component of NeRF.

The uncompressed NeRF model does not provide perfect reconstructions of rendered views, thus the compressed version also has some reconstruction error. We are therefore performing lossy compression on the scene data, not lossless as we explored earlier in this thesis. We utilised a recent method in model compression [Oktay et al., 2020] to compress the neural network, which allows the rate-distortion trade-off to be optimised explicitly. We refer to the overall approach as cNeRF, and demonstrated its effectiveness on two sets of scene data, one synthetic and one real. Our approach is able to reach bitrates of around 1 bit per parameter (of the neural network used in NeRF), whilst still providing visually appealing renderings.

To validate our method, we constructed a baseline method that does not use a

scene representation function, instead blending neighbouring views to construct the rendering at a specified viewpoint. This relies on the communication of a diverse range of baseline views, which we do via the HEVC codec [Sullivan et al., 2012]. We also use the modern, learned LLFF method [Mildenhall et al., 2019] to blend the baseline views. We showed in our experiments that cNeRF consistently outperforms the baseline in terms of rate-distortion trade-off.

There are many possible avenues for future work to further the results and methodologies of this thesis, and we highlight a select few below:

- **Improved latent variable models for lossless compression**: we utilised the ResNet VAE [Kingma et al., 2016] model with HiLLoC, which when we developed the method showed near state-of-the-art performance. Since then there has been much research into latent variable probabilistic generative models, and one of the benefits of BB-ANS and HiLLoC is that improvements in the underlying models can be realised as improvements in lossless compression performance using these models. As such, further improvements to the models may push learned lossless compression to be an attractive alternative to more traditional codecs. A prominent example of this is the recently developed class of variational diffusion models [Kingma et al., 2021], which were immediately realised into a state-of-the-art lossless codec using BB-ANS.

- **Improved quantisation schemes for generative models**: we explored using binary precision to for neural networks used in probabilistic generative models. Although binary precision does offer very large space savings and potential execution time speed-ups, it is an extreme quantisation. To mitigate this fact, we find it necessary to not implement the full neural network using binary weights, but just certain components (in our case the residual layers). As such, it would be interesting to explore using less extreme quantisation schemes, for example 4 or 8-bit precision [Wang et al., 2018, Sun et al., 2020], or a level of quantisation which is learned. This might allow quantisation of greater portions of the generative models without degradation of performance.

- **A unified model for scene data compression**: our method, cNeRF, for

compressing data from a 3D scene relied on the powerful NeRF method, which uses a scene representation function to render an arbitrary view from a scene. However, the downside of the method is that it has to be trained for each scene, so cannot generalise to unseen scenes quickly. Ideally, we would like a generic method to have the rendering capabilities of NeRF and also the ability to generalise to new scenes without training (or perhaps just a fast fine-tuning phase). Such a method could be used for the compression of scene data without onerous training requirements, and indeed there has been some recent research into such methods [Wang et al., 2021, Müller et al., 2022].

The purpose of this thesis is to work towards learned compression methods that are practical and applicable to compressing the large amounts of data being produced in the modern world. Hopefully, with further efforts in this area, these methods can move from research projects to reality for end-users.

Bibliography

T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering.* CRC Press, 2019.

M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal. An empirical study of binary neural networks' optimisation. *International Conference on Learning Representations (ICLR)*, 2019.

P. Astola and I. Tabus. WaSP: Hierarchical warping, merging, and sparse prediction for light field image compression. *European Workshop on Visual Information Processing (EUVIP)*, 2018.

J. Ba, J. Kiros, and G. Hinton. Layer normalization. *ArXiv e-prints*, arXiv:1607.06450, 2016.

N. Bakir, W. Hamidouche, O. Déforges, K. Samrouth, and M. Khalil. Light field image compression based on convolutional neural networks and linear approximation. *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018.

J. Ballé, V. Laparra, and E. P. Simoncelli. End-to-end optimized image compression. *International Conference on Learning Representations (ICLR)*, 2017.

D. Barina, T. Chlubna, M. Solony, D. Dlabaja, and P. Zemcik. Evaluation of 4D light field compression methods. *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 2019.

Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv e-prints*, arXiv:1308.3432, 2013.

T. Bird, J. Ballé, S. Singh, and P. A. Chou. 3D scene compression through entropy penalized neural representation functions. *Picture Coding Symposium (PCS)*, 2021a.

T. Bird, F. H. Kingma, and D. Barber. Reducing the computational cost of deep generative models with binary neural networks. *International Conference on Learning Representations (ICLR)*, 2021b.

C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

C. Bloom. Understanding ANS - 9 . `http://cbloomrants.blogspot.com/2014/02/02-10-14-understanding-ans-9.html`, 2014. Accessed: 2019-05-22.

D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *International Conference on Learning Representations (ICLR)*, 2016.

M. Courbariaux, Y. Bengio, and J. David. BinaryConnect: Training deep neural networks with binary weights during propagations. *Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

Y. Dauphin, A. Fan, M. Auli, and D. Grangier. Language modeling with gated convolutional networks. *International Conference on Machine Learning (ICML)*, 2017.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: a large-scale hierarchical image database. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

L. Dinh, D. Krueger, and Y. Bengio. NICE: Non-linear independent components estimation. *Workshop contribution at International Conference on Learning Representations (ICLR)*, 2015.

L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real NVP. *International Conference on Learning Representations (ICLR)*, 2017.

J. Duda. Asymmetric numeral systems. *ArXiv e-prints*, arXiv:0902.027, 2009.

P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi: 10.1109/TIT.1975.1055349.

B. Frey. *Bayesian networks for pattern classification, data compression, and channel coding*. PhD thesis, University of Toronto, 1997.

K. Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3-4), 1975.

J.-L. Gailly and M. Adler. GNU gzip. `https://www.gnu.org/software/gzip/`, 1992. Accessed: 2022-11-22.

F. Giesen. rANS in practice. `https://fgiesen.wordpress.com/2015/12/21/rans-in-practice/`, 2015. Accessed: 2019-05-22.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

S. W. Golomb. Run-length encodings. *IEEE Trans Info Theory*, 12(3):399, 1966.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

J. Gu, C. Li, B. Zhang, J. Han, X. Cao, J. Liu, and D. Doermann. Projection convolutional neural networks for 1-bit CNNs via discrete back propagation. *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *International Conference on Machine Learning (ICML)*, 2015.

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer,

M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

G. Hinton and D. van Camp. Keeping neural networks simple by minimizing the description length of the weights. *Conference on Computational Learning Theory (COLT)*, 1993.

J. Ho, X. Chen, A. Srinivas, Y. Duan, and P. Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. *International Conference on Machine Learning (ICML)*, 2019a.

J. Ho, E. Lohn, and P. Abbeel. Compression with flows via local bits-back coding. *Conference on Neural Information Processing Systems (NeurIPS)*, 2019b.

M. Hoffman and M. Johnson. ELBO surgery: yet another way to carve up the variational evidence lower bound. *Workshop in Advances in Approximate Bayesian Inference, NeurIPS*, 2016.

E. Hoogeboom, J. W. T. Peters, R. van den Berg, and M. Welling. Integer discrete flows and lossless compression. *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. *Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning (ICML)*, 2015.

C. Jia, X. Zhang, S. Wang, S. Wang, and S. Ma. Light field image compression using generative adversarial network-based view synthesis. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

X. Jiang, M. Le Pendu, and C. Guillemot. Light field compression using depth image based view synthesis. *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, 2017.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

D. P. Kingma and M. Welling. Auto-encoding variational bayes. *International Conference on Learning Representations (ICLR)*, 2014.

D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. *Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

D. P. Kingma, T. Salimans, B. Poole, and J. Ho. Variational diffusion models. *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

F. H. Kingma, P. Abbeel, and J. Ho. Bit-Swap: recursive bits-back coding for lossless compression with hierarchical latent variables. *International Conference on Machine Learning (ICML)*, 2019.

P. Kingma, D. and P. Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. *Conference on Neural Information Processing Systems (NeurIPS)*, 1989.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein. Training quantized nets: A deeper understanding. *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt. Neural sparse voxel fields. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

M. M. Training wide residual networks for deployment using a single bit for each weight. *International Conference on Learning Representations (ICLR)*, 2018.

L. Maaløe, M. Fraccaro, V. Liévin, and O. Winther. BIVA: a very deep hierarchy of latent variables for generative modeling. *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

D. Mackay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

B. Mildenhall, P. Srinivasan, R. Ortiz-Cayon, N. K. Kalantari, R. Ramamoorthi, R. Ng, and A. Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*, 2019.

B. Mildenhall, P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. *European Conference on Computer Vision (ECCV)*, 2020.

T. Müller, A. Evans, C. Schied, and A. Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. doi: 10.1145/3528223.3530127. URL `https://doi.org/10.1145/3528223.3530127`.

K. P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, 2022.

D. Oktay, J. Ballé, S. Singh, and A. Shrivastava. Scalable model compression by entropy penalized reparameterization. *International Conference on Learning Representations (ICLR)*, 2020.

M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. *European Conference on Computer Vision (ECCV)*, 2016.

D. J. Rezende and S. Mohamed. Variational inference with normalizing flows. *International Conference on Machine Learning (ICML)*, 2015.

D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic back-propagation and variational inference in deep latent gaussian models. *International Conference on Machine Learning (ICML)*, 2014.

R. Salakhutdinov and I. Murray. On the quantitative analysis of deep belief networks. *International Conference on Machine Learning (ICML)*, 2008.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

E. Sari, M. Belbahri, and V. P. Nia. How does batch normalization help binary training? *ArXiv e-prints*, arXiv:1909.09139, 2020.

K. Schwarz, Y. Liao, M. Niemeyer, and A. Geiger. GRAF: Generative radiance fields for 3D-aware image synthesis. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948.

V. Sitzmann, M. Zollhöfer, and G. Wetzstein. Scene representation networks: Continuous 3D-structure-aware neural scene representations. *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

J. Sneyers and P. Wuille. FLIF: Free lossless image format based on MANIAC compression. *IEEE International Conference on Image Processing (ICIP)*, 2016.

C. K. Sønderby, T. Raiko, L. Maaløe, S. K. Sønderby, and O. Winther. Ladder variational autoencoders. *Conference on Neural Information Processing Systems (NeurIPS)*, 2016.

G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2012.

X. Sun, N. Wang, C. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, E. M. Kaoutar, V. Srinivasan, and K. Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

J. Townsend, T. Bird, and D. Barber. Practical lossless compression with latent variables using bits back coding. *International Conference on Learning Representations (ICLR)*, 2019.

J. Townsend, T. Bird, J. Kunze, and D. Barber. HiLLoC: Lossless image compression with hierarchical latent variable models. *International Conference on Learning Representations (ICLR)*, 2020.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

C. S. Wallace. Classification by minimum-message-length inference. In *Proceedings of the International Conference on Advances in Computing and Information (ICCI)*, pages 72–81, 1990.

N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

Q. Wang, Z. Wang, K. Genova, P. P. Srinivasan, H. Zhou, J. T. Barron, R. Martin-Brualla, N. Snavely, and T. Funkhouser. IBRNet: Learning multi-view image-based rendering. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multiscale structural similarity for image quality assessment. *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers*, 2003.

Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. doi: 10.1109/MC.1984.1659158.

I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The unreasonable effectiveness of deep features as a perceptual metric. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Z. Zhao, S. Wang, C. Jia, X. Zhang, S. Ma, and J. Yang. Light field image compression based on deep learning. *IEEE International Conference on Multimedia and Expo (ICME)*, 2018.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi: 10.1109/TIT.1978.1055934.