Practical Full Resolution Learned Lossless Image Compression

Fabian Mentzer mentzerf@vision.ee.ethz.ch
Eirikur Agustsson aeirikur@vision.ee.ethz.ch
Michael Tschannen michaelt@nari.ee.ethz.ch
Radu Timofte timofter@vision.ee.ethz.ch
Luc Van Gool vangool@vision.ee.ethz.ch

ETH Zürich, Switzerland

Abstract

We propose the first practical learned lossless image compression system, L3C, and show that it outperforms the popular engineered codecs, PNG, WebP and JPEG2000. At the core of our method is a fully parallelizable hierarchical probabilistic model for adaptive entropy coding which is optimized end-to-end for the compression task. In contrast to recent autoregressive discrete probabilistic models such as PixelCNN, our method i) models the image distribution jointly with learned auxiliary representations instead of exclusively modeling the image distribution in RGB space, and ii) only requires three forward-passes to predict all pixel probabilities instead of one for each pixel. As a result, L3C obtains over two orders of magnitude speedups when sampling compared to the fastest PixelCNN variant (Multiscale-PixelCNN). Furthermore, we find that learning the auxiliary representation is crucial and outperforms predefined auxiliary representations such as an RGB pyramid significantly.

1. Introduction

Since likelihood-based discrete generative models learn a probability distribution over pixels, they can in theory be used for lossless image compression [40]. However, recent work on learned compression using deep neural networks has solely focused on lossy compression [4, 41, 42, 34, 1, 3, 44]. Indeed, the literature on discrete generative models [46, 45, 35, 32, 20] has largely ignored the application as a lossless compression system, with neither bitrates nor runtimes being compared with classical codecs such as PNG [31], WebP [47], JPEG2000 [38], and FLIF [39]. This is not surprising as (lossless) entropy coding using likelihood-based discrete generative models amounts to a decoding complexity essentially identical to the sampling complexity of the model, which renders many of the recent state-of-the-art autoregressive models such as PixelCNN [46], PixelCNN++ [35], and Multiscale-PixelCNN [32] impractical, requiring minutes or hours on a GPU to generate moderately large images, typically $<256 \times 256$px (see Table 2). The computational complexity of these models is mainly caused by the sequential nature of the sampling (and thereby decoding) operation, where a forward pass needs to be computed for every single (sub) pixel of the image in a raster scan order.

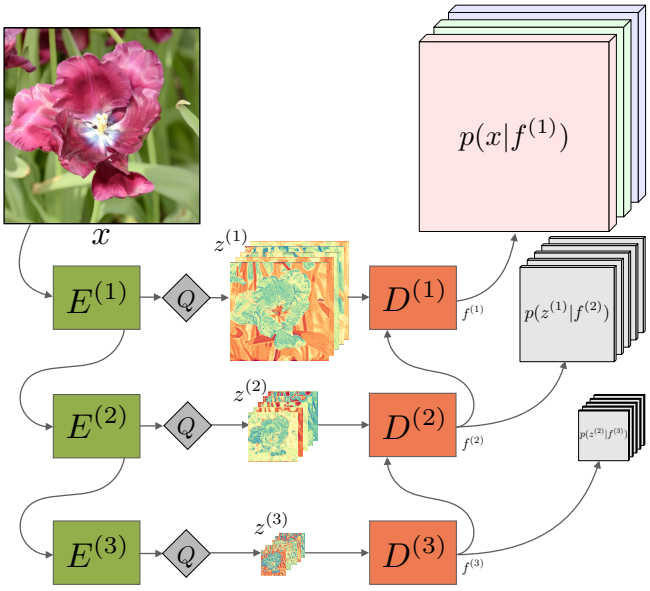

Figure 1: Overview of the architecture of L3C. The feature extractors $E^{(s)}$ compute quantized (by Q) auxiliary hierarchical feature representation $z^{(1)}, \dots, z^{(S)}$ whose joint distribution with the image x, $p(x, z^{(1)}, \dots, z^{(S)})$, is modeled using the non-autoregressive predictors $D^{(s)}$. The features $f^{(s)}$ summarize the information up to scale s and are used to predict p for the next scale.

In this paper, we address these challenges and develop a fully parallelizeable learned lossless compression system, outperforming the popular classical systems PNG, WebP and JPEG2000.

Our system (see Fig. 1 for an overview) is based on a hierarchy of fully parallel learned feature extractors and predictors which are trained jointly for the compression task. Our code is available online[1]. The role of the feature extractors is to build an auxiliary hierarchical feature representation which helps the predictors to model both the image and the auxiliary

[1] https://github.com/fab-jul/L3C-PyTorch

features themselves. Our experiments show that learning the feature representations is crucial, and heuristic (predefined) choices such as a multiscale RGB pyramid lead to suboptimal performance.

In more detail, to encode an image x, we feed it through the S feature extractors $E^{(s)}$ and predictors $D^{(s)}$. Then, we obtain the predictions of the probability distributions p, for both x and the auxiliary features $z^{(s)}$, in parallel in a single forward pass. These predictions are then used with an adaptive arithmetic encoder to obtain a compressed bitstream of both x and the auxiliary features (Sec. 3.1 provides an introduction to arithmetic coding). However, the arithmetic decoder now needs p to be able to decode the bitstream. Starting from the lowest scale of auxiliary features $z^{(S)}$, for which we assume a uniform prior, $D^{(S)}$ obtains a prediction of the distribution of the auxiliary features of the next scale, $z^{(S-1)}$, and can thus decode them from the bitstream. Prediction and decoding is alternated until the arithmetic decoder obtains the image x. The steps are visualized in Fig. A4 in the appendix.

In practice, we only need to use $S = 3$ feature extractors and predictors for our model, so when decoding we only need to perform three parallel (over pixels) forward passes in combination with the adaptive arithmetic coding.

The parallel nature of our model enables it to be orders of magnitude faster for decoding than autoregressive models, while learning enables us to obtain compression rates competitive with state-of-the-art engineered lossless codecs.

In summary, our contributions are the following:

- We propose a fully parallel hierarchical probabilistic model, learning both the feature extractors that produce an auxiliary feature representation to help the prediction task, as well as the predictors which model the joint distribution of all variables (Sec. 3).

- We show that entropy coding based on our *non-autoregressive* probabilistic model optimized for discrete log-likelihood can obtain compression rates outperforming WebP, JPEG 2000 and PNG, the latter by a large margin. We are only marginally outperformed by the state-of-the-art, FLIF, while being conceptually much simpler (Sec. 5.1).

- At the same time, our model is practical in terms of runtime complexity and orders of magnitude faster than PixelCNN-based approaches. In particular, our model is $5.31 \cdot 10^4\times$ *faster than PixelCNN++*[35] and $5.06 \cdot 10^2\times$ *faster than the highly speed-optimized MS-PixelCNN* [32] (Sec. 5.2).

2. Related Work

Likelihood-Based Generative Models As previously mentioned, essentially all likelihood-based discrete generative models can be used with an arithmetic coder for lossless compression. A prominent group of models that obtain state-of-the-art performance are variants of the auto-regressive PixelRNN/PixelCNN [46, 45]. PixelRNN and PixelCNN organize the pixels of the image distribution as a sequence and predict the distribution of each pixel conditionally on (all) previous pixels using an RNN and a CNN with masked convolutions, respectively. These models hence require a number of network evaluations equal to the number of predicted sub-pixels[2] ($3 \cdot W \cdot H$). PixelCNN++ [35] improves on this in various ways, including modeling the joint distribution of each pixel, thereby eliminating conditioning on previous channels and reducing to $W \cdot H$ forward passes. MS-PixelCNN [32] parallelizes PixelCNN by reducing dependencies between blocks of pixels and processing them in parallel with shallow PixelCNNs, requiring $O(\log WH)$ forward passes. [20] equips PixelCNN with auxiliary variables (grayscale version of the image or RGB pyramid) to encourage modeling of high-level features, thereby improving the overall modeling performance. [7, 29] propose autoregressive models similar to PixelCNN/PixelRNN, but they additionally rely on attention mechanisms to increase the receptive field.

Engineered Codecs The well-known *PNG* [31] operates in two stages: first the image is reversibly transformed to a more compressible representation with a simple autoregressive filter that updates pixels based on surrounding pixels, then it is compressed with the deflate algorithm [11]. *WebP* [47] uses more involved transformations, including the use of entire image fragments to encode new pixels and a custom entropy coding scheme. *JPEG2000* [38] includes a lossless mode where tiles are reversibly transformed before the coding step, instead of irreversibly removing frequencies. The current state-of-the-art (non-learned) algorithm is *FLIF* [39]. It relies on powerful preprocessing and a sophisticated entropy coding method based on CABAC [33] called MANIAC, which grows a dynamic decision tree per channel as an adaptive context model during encoding.

Context Models in Lossy Compression In lossy compression, context models have been studied as a way to efficiently losslessly encode the obtained image representations. Classical approaches are discussed in [24, 26, 27, 50, 48]. Recent learned approaches include [22, 25, 28], where shallow autoregressive models over latents are learned. [5] presents a model somewhat similar to L3C: Their autoencoder is similar to our fist scale, and the hyper encoder/decoder is similar to our second scale. However, since they train for lossy image compression, their autoencoder predicts RGB pixels directly. Also, they predict uncertainties σ for $z^{(1)}$ instead of a mixture of logistics. Finally, instead of learning a probability distribution for $z^{(2)}$, they assume the entries to be i.i.d. and fit a uni-

[2] A RGB "pixel" has 3 "sub-pixels", one in each channel.

variate non-parametric density model, whereas in our model, many more stages can be trained and applied recursively.

Continuous Likelihood Models for Compression The objective of continuous likelihood models, such as VAEs [19] and RealNVP [12], where $p(x')$ is a continuous distribution, is closely related to its discrete counterpart. In particular, by setting $x' = x + u$ where x is the discrete image and u is uniform quantization noise, the continuous likelihood of $p(x')$ is a lower bound on the likelihood of the discrete $q(x) = \mathbb{E}_u[p(x')]$ [40]. However, there are two challenges for deploying such models for compression. First, the discrete likelihood $q(x)$ needs to be available (which involves a non-trivial integration step). Additionally, the memory complexity of (adaptive) arithmetic coding depends on the size of the domain of the variables of the factorization of q (see Sec. 3.1 on (adaptive) arithmetic coding). Since the domain grows exponentially in the number of pixels in x, unless q is factorizable, it is not feasible to use it with adaptive arithmetic coding.

3. Method

3.1. Lossless Compression

In general, in lossless compression, some stream of symbols x is given, which are drawn independently and identically distributed (i.i.d.) from a set $\mathcal{X} = \{1, \dots, |\mathcal{X}|\}$ according to the probability mass function $\tilde{p}$. The goal is to encode this stream into a bitstream of minimal length using a "code", s.t. a receiver can decode the symbols from the bitstream. Ideally, an encoder minimizes the expected bits per symbol $\tilde{L} = \sum_{j \in \mathcal{X}} \tilde{p}(j)\ell(j)$, where $\ell(j)$ is the length of encoding symbol j (i.e., more probable symbols should obtain shorter codes). Information theory provides (e.g., [9]) the bound $\tilde{L} \geq H(\tilde{p})$ for *any* possible code, where $H(\tilde{p}) = \mathbb{E}_{j \sim \tilde{p}}[-\log \tilde{p}(j)]$ is the Shannon entropy [36].

Arithmetic Coding A strategy that almost achieves the lower bound $H(\tilde{p})$ (for long enough symbol streams) is arithmetic coding [49].[3] It encodes the entire stream into a single number $a' \in [0, 1)$, by subdividing $[0, 1)$ in each step (encoding one symbol) as follows: Let a, b be the bounds of the current step (initialized to $a = 0$ and $b = 1$ for the initial interval $[0, 1)$). We divide the interval $[a, b)$ into $|\mathcal{X}|$ sections where the length of the j-th section is $\tilde{p}(j)/(b - a)$. Then we pick the interval corresponding to the current symbol, i.e., we update a, b to be the boundaries of this interval. We proceed recursively until no symbols are left. Finally, we transmit a', which is a rounded to the smallest number of bits s.t. $a' \geq a$. Receiving a' together with the knowledge of the number of encoded symbols and $\tilde{p}$ uniquely specifies the stream and allows the receiver to decode.

[3] We use (adaptive) arithmetic coding for simplicity of exposition, but any adaptive entropy-achieving coder can be used with our method.

Adaptive Arithmetic Coding In contrast to the i.i.d. setting we just described, in this paper we are interested in losslessly encoding the pixels of a natural image, which are known to be heavily correlated and hence not i.i.d. at all. Let x_t be the sub-pixels[2] of an image x, and $\tilde{p}_{\text{img}}(x)$ the joint distribution of all sub-pixels. We can then consider the factorization $\tilde{p}_{\text{img}}(x) = \prod_t \tilde{p}(x_t | x_{t-1}, \dots, x_1)$. Now, to encode x, we can consider the sub-pixels x_t as our symbol stream and encode the t-th symbol/sub-pixel using $\tilde{p}(x_t | x_{t-1}, \dots, x_1)$. Note that this corresponds to varying the $\tilde{p}(j)$ of the previous paragraph during encoding, and is in general referred to as *adaptive* arithmetic coding (AAC) [49]. For AAC the receiver also needs to know the varying $\tilde{p}$ at every step, i.e., they must either be known a priori or the factorization must be causal (as above) so that the receiver can calculate them from already decoded symbols.

Cross-Entropy In practice, the exact $\tilde{p}$ is usually unknown, and instead is estimated by a model p. Thus, instead of using length $\log 1/\tilde{p}(x)$ to encode a symbol x, we use the sub-optimal length $\log 1/p(x)$. Then

$$\begin{aligned} H(\tilde{p}, p) &= \mathbb{E}_{j \sim \tilde{p}}\left[-\log p(j)\right] \\ &= -\sum_{j \in \mathcal{X}} \tilde{p}(j) \log p(j) \end{aligned} \tag{1}$$

is the resulting expected (sub-optimal) bits per symbol, and is called **cross-entropy** [9].

Thus, given some p, we can minimize the bitcost needed to encode a symbol stream with symbols distributed according to $\tilde{p}$ by minimizing Eq. (1). This naturally generalizes to the non-i.i.d. case described in the previous paragraph by using different $\tilde{p}(x_t)$ and $p(x_t)$ for each symbol x_t and minimizing $\sum_t H(\tilde{p}(x_t), p(x_t))$.

The following sections describe how a hierarchical causal factorization of p_{img} for natural images can be used to efficiently do learned lossless image compression (L3C).

3.2. Architecture

A high-level overview of the architecture is given in Fig. 1, while Fig. 2 shows a detailed description for one scale s. Unlike autoregressive models such as PixelCNN and PixelRNN, which factorize the image distribution autoregressively over sub-pixels x_t as $p(x) = \prod_{t=1}^{T} p(x_t | x_{t-1}, \dots, x_1)$, we jointy model all the sub-pixels and introduce a learned hierarchy of auxiliary feature representations $z^{(1)}, \dots, z^{(S)}$ to simplify the modeling task.We fix the dimensions of $z^{(s)}$ to be $C \times H' \times W'$, where the number of channels C is a hyperparameter ($C = 5$ in our reported models), and $H' = H/2^s, W' = W/2^s$ given a $H \times W$-dimensional image.[4]

[4] Considering that $z^{(s)}$ is quantized, this conveniently upper bounds the information that can be contained within each $z^{(s)}$, however, other dimensions could be explored.

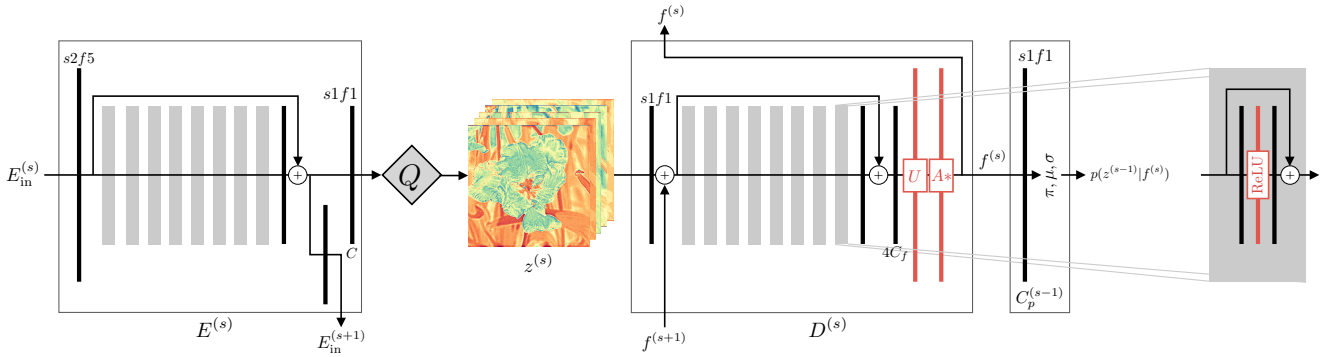

Figure 2: Architecture details for a single scale s. For $s = 1$, $E^{(1)}_{\text{in}}$ is the RGB image x normalized to $[-1, 1]$. All vertical **black** lines are convolutions, which have $C_f = 64$ filters, except when denoted otherwise beneath. The convolutions are stride 1 with 3×3 filters, except when denoted otherwise above (using $sSfF$ = stride s, filter f). We add the features $f^{(s+1)}$ from the predictor $D^{(s+1)}$ to those of the first layer of $D^{(s)}$ (a skip connection between scales). The gray blocks are residual blocks, shown once on the right side. C is the number of channels of $z^{(s)}$, $C_p^{(s-1)}$ is the final number of channels, see Sec. 3.4. Special blocks are denoted in red: U is pixelshuffling upsampling [37]. $A*$ is the "atrous convolution" layer described in Sec. 3.2. We use a heatmap to visualize $z^{(s)}$, see Sec. A.4.

Specifically, we model the joint distribution of the image x and the feature representations $z^{(s)}$ as

$$p(x, z^{(1)}, \dots, z^{(S)}) = p(x|z^{(1)}, \dots, z^{(S)}) \prod_{s=1}^{S} p(z^{(s)}|z^{(s+1)}, \dots, z^{(S)})$$

where $p(z^{(S)})$ is a uniform distribution. The feature representations can be hand designed or learned. Specifically, on one side, we consider an RGB pyramid with $z^{(s)} = \mathcal{B}_{2^s}(x)$, where $\mathcal{B}_{2^s}$ is the bicubic (spatial) subsampling operator with subsampling factor 2^s. On the other side, we consider a learned representation $z^{(s)} = F^{(s)}(x)$ using a feature extractor $F^{(s)}$. We use the hierarchical model shown in Fig. 1 using the composition $F^{(s)} = Q \circ E^{(s)} \circ \cdots \circ E^{(1)}$, where the $E^{(s)}$ are feature extractor blocks and Q is a scalar differentiable quantization function (see Sec. 3.3). The $D^{(s)}$ in Fig. 1 are predictor blocks, and we parametrize $E^{(s)}$ and $D^{(s)}$ as convolutional neural networks.

Letting $z^{(0)} = x$, we parametrize the conditional distributions for all $s \in \{0, \dots, S\}$ as

$$p(z^{(s)}|z^{(s+1)}, \dots, z^{(S)}) = p(z^{(s)}|f^{(s+1)}),$$

using the predictor features $f^{(s)} = D^{(s)}(f^{(s+1)}, z^{(s)})$.[5] Note that $f^{(s+1)}$ summarizes the information of $z^{(S)}, \dots, z^{(s+1)}$.

The predictor is based on the super-resolution architecture from EDSR [23], motivated by the fact that our prediction task is somewhat related to super-resolution in that both are dense prediction tasks involving spatial upsampling. We mirror the predictor to obtain the feature extractor, and follow [23] in not using BatchNorm [16]. Inspired by the "atrous spatial pyramid pooling" from [6], we insert a similar layer at the end of $D^{(s)}$: In $A*$, we use three atrous convolutions in parallel, with rates 1, 2, and 4, then concatenate the resulting feature maps to a $3C_f$-dimensional feature map.

[5] The final predictor only sees $z^{(S)}$, i.e., we let $f^{(S+1)} = 0$.

3.3. Quantization

We use the scalar quantization approach proposed in [25] to quantize the output of $E^{(s)}$: Given levels $\mathbb{L} = \{\ell_1, \dots, \ell_L\} \subset \mathbb{R}$, we use nearest neighbor assignments to quantize each entry $z' \in z^{(s)}$ as

$$z = Q(z') := \arg\min_j \|z' - \ell_j\|, \tag{2}$$

but use differentiable "soft quantization"

$$\tilde{Q}(z') = \sum_{j=1}^{L} \frac{\exp(-\sigma_q \|z' - \ell_j\|)}{\sum_{l=1}^{L} \exp(-\sigma_q \|z' - \ell_l\|)} \ell_j \tag{3}$$

to compute gradients for the backward pass, where σ_q is a hyperparameter relating to the "softness" of the quantization. For simplicity, we fix $\mathbb{L}$ to be $L = 25$ evenly spaced values in $[-1, 1]$.

3.4. Mixture Model

For ease of notation, let $z^{(0)} = x$ again. We model the conditional distributions $p(z^{(s)}|z^{(s+1)}, \dots, z^{(S)})$ using a generalization of the discretized logistic mixture model with K components proposed in [35], as it allows for efficient training: The alternative of predicting logits per (sub-)pixel has the downsides of requiring more memory, causing sparse gradients (we only get gradients for the logit corresponding to the ground-truth value), and does not model that neighbouring values in the domain of p should have similar probability.

Let c denote the channel and u, v the spatial location. For all scales, we assume the entries of $z^{(s)}_{cuv}$ to be independent across u, v, given $f^{(s+1)}$. For RGB ($s = 0$), we define

$$p(x|f^{(1)}) = \prod_{u,v} p(x_{1uv}, x_{2uv}, x_{3uv}|f^{(1)}), \tag{4}$$

where we use a weak autoregression over RGB channels to define the joint probability distribution via a mixture p_m (dropping the indices uv for shorter notation):

$$p(x_1, x_2, x_3|f^{(1)}) = p_m(x_1|f^{(1)}) \cdot p_m(x_2|f^{(1)}, x_1) \cdot p_m(x_3|f^{(1)}, x_2, x_1). \quad (5)$$

We define p_m as a mixture of logistic distributions p_l (defined in Eq. (10) below). To this end, we obtain mixture weights[6] π^k_{cuv}, means μ^k_{cuv}, variances σ^k_{cuv}, as well as coefficients λ^k_{cuv} from $f^{(1)}$ (see further below), and get

$$\begin{aligned} p_m(x_{1uv}|f^{(1)}) &= \sum_k \pi^k_{1uv}\, p_l(x_{1uv}|\tilde{\mu}^k_{1uv}, \sigma^k_{1uv}) \\ p_m(x_{2uv}|f^{(1)}, x_{1uv}) &= \sum_k \pi^k_{2uv}\, p_l(x_{2uv}|\tilde{\mu}^k_{2uv}, \sigma^k_{2uv}) \\ p_m(x_{3uv}|f^{(1)}, x_{1uv}, x_{2uv}) &= \sum_k \pi^k_{3uv}\, p_l(x_{3uv}|\tilde{\mu}^k_{3uv}, \sigma^k_{3uv}), \end{aligned} \quad (6)$$

where we use the conditional dependency on previous x_{cuv} to obtain the updated means $\tilde{\mu}$, as in [35, Sec. 2.2],

$$\begin{aligned} \tilde{\mu}^k_{1uv} &= \mu^k_{1uv} & \tilde{\mu}^k_{2uv} &= \mu^k_{2uv} + \lambda^k_{\alpha uv}\, x_{1uv} \\ \tilde{\mu}^k_{3uv} &= \mu^k_{3uv} + \lambda^k_{\beta uv}\, x_{1uv} + \lambda^k_{\gamma uv}\, x_{2uv}. \end{aligned} \quad (7)$$

Note that the autoregression over channels in Eq. (5) is only used to update the means μ to $\tilde{\mu}$.

For the other scales ($s > 0$), the formulation only changes in that we use no autoregression at all, i.e., $\tilde{\mu}_{cuv} = \mu_{cuv}$ for all c, u, v. No conditioning on previous channels is needed, and Eqs. (4)-(6) simplify to

$$p(z^{(s)}|f^{(s+1)}) = \prod_{c,u,v} p_m(z^{(s)}_{cuv}|f^{(s+1)}) \quad (8)$$

$$p_m(z^{(s)}_{cuv}|f^{(s+1)}) = \sum_k \pi^k_{cuv}\, p_l(x_{cuv}|\mu^k_{cuv}, \sigma^k_{cuv}). \quad (9)$$

For all scales, the individual logistics p_l are given as

$$p_l(z|\mu, \sigma) = \big(\text{sigmoid}((z + b/2 - \mu)/\sigma) - \text{sigmoid}((z - b/2 - \mu)/\sigma)\big). \quad (10)$$

Here, b is the bin width of the quantization grid ($b = 1$ for $s = 0$ and $b = 1/12$ otherwise). The edge-cases $z = 0$ and $z = 255$ occurring for $s = 0$ are handled as described in [35, Sec. 2.1].

For all scales, we obtain the parameters of $p(z^{(s-1)}|f^{(s)})$ from $f^{(s)}$ with a 1×1 convolution that has $C_p^{(s-1)}$ output channels (see Fig. 2). For RGB, this final feature map must contain the three parameters π, μ, σ for each of the 3 RGB channels and K mixtures, as well as $\lambda_\alpha, \lambda_\beta, \lambda_\gamma$ for every mixture, thus requiring $C_p^{(0)} = 3 \cdot 3 \cdot K + 3 \cdot K$ channels. For $s > 0$, $C_p^{(s)} = 3 \cdot C \cdot K$, since no λ are needed. With the parameters, we can obtain $p(z^{(s)}|f^{(s+1)})$, which has dimensions $3\times H\times W\times 256$ for RGB and $C\times H'\times W'\times L$ otherwise (visualized with cubes in Fig. 1).

We emphasize that in contrast to [35], our model is *not* autoregressive over pixels, i.e., $z^{(s)}_{cuv}$ are modelled as independent across u, v given $f^{(s+1)}$ (also for $z^{(0)} = x$).

3.5. Loss

We are now ready to define the loss, which is a generalization of the discrete logistic mixture loss introduced in [35]. Recall from Sec. 3.1 that our goal is to model the true joint distribution of x and the representations $z^{(s)}$, i.e., $\tilde{p}(x, z^{(1)}, \dots, z^{(s)})$ as accurately as possible using our model $p(x, z^{(1)}, \dots, z^{(s)})$. Thereby, the $z^{(s)} = F^{(s)}(x)$ are defined using the learned feature extractor blocks $E^{(s)}$, and $p(x, z^{(1)}, \dots, z^{(s)})$ is a product of discretized (conditional) logistic mixture models with parameters defined through the $f^{(s)}$, which are in turn computed using the learned predictor blocks $D^{(s)}$. As discussed in Sec. 3.1, the expected coding cost incurred by coding $x, z^{(1)}, \dots, z^{(s)}$ w.r.t. our model $p(x, z^{(1)}, \dots, z^{(s)})$ is the cross entropy $H(\tilde{p}, p)$.

We therefore directly minimize $H(\tilde{p}, p)$ w.r.t. the parameters of the feature extractor blocks $E^{(s)}$ and predictor blocks $D^{(s)}$ over samples. Specifically, given N training samples $x_1, \dots, x_N$, let $F_i^{(s)} = F^{(s)}(x_i)$ be the feature representation of the i-th sample. We minimize

$$\begin{aligned} &\mathcal{L}(E^{(1)}, \dots, E^{(S)}, D^{(1)}, \dots, D^{(S)}) \\ &= -\sum_{i=1}^N \log\Big(p\big(x_i, F_i^{(1)}, \dots, F_i^{(S)}\big)\Big) \\ &= -\sum_{i=1}^N \log\Big(p\big(x_i|F_i^{(1)}, \dots, F_i^{(S)}\big) \cdot \prod_{s=1}^S p\big(F_i^{(s)}|F_i^{(s+1)}, \dots, F_i^{(S)}\big)\Big) \\ &= -\sum_{i=1}^N \Big(\log p(x_i|F_i^{(1)}, \dots, F_i^{(S)}) + \sum_{s=1}^S \log p(F_i^{(s)}|F_i^{(s+1)}, \dots, F_i^{(S)})\Big). \end{aligned} \quad (11)$$

Note that the loss decomposes into the sum of the cross-entropies of the different representations. Also note that this loss corresponds to the negative log-likelihood of the data w.r.t. our model which is typically the perspective taken in the generative modeling literature (see, e.g., [46]).

Propagating Gradients through Targets We emphasize that in contrast to the generative model literature, we learn the representations, propagating gradients to both $E^{(s)}$

[6] Note that in contrast to [35] we do not share mixture weights π^k across channels. This allows for easier marginalization of Eq. (5).

[bpsp]	Method	Open Images	DIV2K	RAISE-1k
Ours	L3C	2.604	3.097	2.747
Learned Baselines	RGB Shared	2.918 +12%	3.657 +18%	3.170 +15%
	RGB	2.819 +8.2%	3.457 +12%	3.042 +11%
Non-Learned Approaches	PNG	3.779 +45%	4.527 +46%	3.924 +43%
	JPEG 2000	2.778 +6.7%	3.331 +7.5%	2.940 +7.0%
	WebP	2.666 +2.3%	3.234 +4.4%	2.826 +2.9%
	FLIF	2.473 −5.1%	3.046 −1.7%	2.602 −5.3%

Table 1: Compression performance of our method (L3C) and learned baselines (RGB Shared and RGB) to previous (non-learned) approaches, in bits per sub-pixel (bpsp). We emphasize the difference in percentage to our method for each other method in *green* if L3C outperforms the other method and in *red* otherwise.

and $D^{(s)}$, since each component of our loss depends on $D^{(s+1)}, \dots, D^{(S)}$ via the parametrization of the logistic distribution and on $E^{(s)}, \dots, E^{(1)}$ because of the differentiable Q. Thereby, our network can autonomously learn to navigate the trade-off between a) making the output $z^{(s)}$ of feature extractor $E^{(s)}$ more easily estimable for the predictor $D^{(s+1)}$ and b) putting enough information into $z^{(s)}$ for the predictor $D^{(s)}$ to predict $z^{(s-1)}$.

3.6. Relationship to MS-PixelCNN

When the auxiliary features $z^{(s)}$ in our approach are restricted to a non-learned RGB pyramid (see baselines in Sec. 4), this is somewhat similar to MS-PixelCNN [32]. In particular, [32] combines such a pyramid with upscaling networks which play the same role as the predictors in our architecture. Crucially however, they rely on combining such predictors with a shallow PixelCNN and upscaling one dimension at a time ($W \times H \to 2W \times H \to 2W \times 2H$). While their complexity is reduced from $O(WH)$ forward passes needed for PixelCNN [46] to $O(\log WH)$, their approach is in practice still two orders of magnitude slower than ours (see Sec. 5.2). Further, we stress that these similarities only apply for our RGB baseline model, whereas our best models are obtained using learned feature extractors trained jointly with the predictors.

4. Experiments

Models We compare our main model (L3C) to two learned baselines: For the *RGB Shared* baseline (see Fig. A2) we use bicubic subsampling as feature extractors, i.e., $z^{(s)} = \mathcal{B}_{2^s}(x)$, and only train one predictor $D^{(1)}$. During testing, we obtain multiple $z^{(s)}$ using $\mathcal{B}$ and apply the single predictor $D^{(1)}$ to each. The *RGB* baseline (see Fig. A3) also uses bicubic subsampling, however, we train $S = 3$ predictors $D^{(s)}$, one for each scale, to capture the different distributions of different RGB scales. For our main model, *L3C*, we additionally learn $S = 3$ feature extractors $E^{(s)}$.[7] Note that the only difference to the RGB baseline is that the representations $z^{(s)}$ are learned. We train all these models until they converge at 700k iterations.

Datasets We train our models on 213 487 images randomly selected from the *Open Images Train* dataset [21]. We downscale the images to 768 pixels on the longer side to remove potential artifacts from previous compression, discarding images where rescaling does not result in at least $1.25\times$ downscaling. Further, following [5] we discard high saturation/non-photographic images, i.e., images with mean $S > 0.9$ or $V > 0.8$ in the HSV color space. We evaluate on 500 randomly selected images from *Open Images Test* and the 100 images from the commonly used super-resolution dataset *DIV2K* [2], both preprocessed like the training set. Finally, we evaluate on *RAISE-1k* [10], a "real-world image dataset" with 1000 images: To show how our network generalizes to arbitrary image sizes, we randomly resize these images s.t. the longer side is $500 - 2000$ pixels.

In order to compare to the PixelCNN literature, we additionally train L3C on the ImageNet32 and ImageNet64 datasets [8], each containing 1 281 151 training images and 50 000 validation images, of 32×32 resp. 64×64 pixels.

Training We use the RMSProp optimizer [15], with a batch size of 30, minimizing Eq. (11) directly (no regularization). We train on 128×128 random crops, and apply random horizontal flips. We start with a learning rate $\lambda = 1 \cdot 10^{-4}$ and decay it by a factor of 0.75 every 5 epochs. On ImageNet32/64, we increase the batch size to 120 and decay λ every epoch, due to the smaller images.

Architecture Ablations We find that adding BatchNorm [17] slightly degrades performance. Furthermore, replacing the stacked atrous convolutions $A*$ with a single convolution, slightly degrades performance as well. By stopping

[7] We chose $S = 3$ because increasing S comes at the cost of slower training, while yielding negligible improvements in bitrate. For an image of size $H \times W$, the last bottleneck has $5 \times H/8 \times W/8$ dimensions, each quantized to $L = 25$ values. Encoding this with a uniform prior amounts to $\approx 4\%$ of the total bitrate. For the RGB Shared baseline, we apply $D^{(1)}$ 4 times, as only one encoder is trained.

	Method	32×32px	320×320px
BS=1	L3C (Ours)	0.0168 s	0.0291 s
	PixelCNN++ [35]	47.4 s*	$\approx$ 80 min‡
BS=30	L3C (Ours)	0.000624 s	0.0213 s
	PixelCNN++	11.3 s*	$\approx$ 18 min‡
	PixelCNN [46]	120 s†	$\approx$ 8 hours‡
	MS-PixelCNN [32]	1.17 s†	$\approx$ 2 min‡

Table 2: Sampling times for our method (L3C), compared to the PixelCNN literature. The results in the first two rows were obtained with batch size (BS) 1, the other times with BS=30, since this is what is reported in [32]. [$*$]: Times obtained by us with code released of PixelCNN++ [35], on the same GPU we used to evaluate L3C (Titan X Pascal). [$\dagger$]: times reported in [32], obtained on a Nvidia Quadro M4000 GPU (no code available). [$\ddagger$]: To put the numbers into perspective, we compare our runtime with *linearly interpolated* runtimes for for the other approaches on 320×320 crops.

gradients from propagating through the targets of our loss, we get significantly worse performance – in fact, the optimizer does not manage to pull down the cross-entropy of any of the learned representations $z^{(s)}$ significantly.

We find the choice of σ_q for Q has impacts on training: [25] suggests setting it s.t. $\tilde{Q}$ resembles identity, which we found to be good starting point, but found it beneficial to let σ_q be slightly smoother (this yields better gradients for the encoder). We use $\sigma_q = 2$.

Additionally, we explored the impact of varying C (number of channels of $z^{(s)}$) and the number of levels L and found it more beneficial to increase L instead of increasing C, i.e., it is beneficial for training to have a finer quantization grid.

Other Codecs We compare to FLIF and the lossless mode of WebP using the respective official implementations [39, 47], for PNG we use the implementation of Pillow [30], and for the lossless mode of JPEG 2000 we use the Kakadu implementation [18]. See Sec. 2 for a description of these codecs.

5. Results

5.1. Compression

Table 1 shows a comparison of our approach (L3C) and the learned baselines to the other codecs, on our testsets, in terms of bits per sub-pixel (bpsp)[8] All of our methods outperform the widely-used PNG, which is at least 43% larger on all datasets. We also outperform WebP and JPEG 2000 everywhere by a smaller margin of up to 7.5%. We note that FLIF still marginally outperforms our model but remind the reader of the many hand-engineered highly specialized techniques involved in FLIF (see Section 2). In contrast, we use a simple convolutional feed-forward neural network architecture. The

[8]We follow the likelihood-based generative modelling literature in measuring bpsp; X bits per pixel (bpp) $= X/3$ bpsp, see also footnote 2.

[bpsp]	ImageNet32	Learned
L3C (ours)	4.76	✓
PixelCNN [46]	3.83	✓
MS-PixelCNN [32]	3.95	✓
PNG	6.42	
JPEG 2000	6.35	
WebP	5.28	
FLIF	5.08	

Table 3: Comparing bits per sub-pixel (bpsp) on the 32×32 images from ImageNet32 of our method (L3C) vs. PixelCNN-based approaches and classical approaches.

RGB baseline with $S = 3$ learned predictors outperforms the RGB Shared baseline on all datasets, showing the importance of learning a predictor for each scale. Using our main model (L3C), where we additionally learn the feature extractors, we outperform both baselines: The outputs are at least 12% larger everywhere, showing the benefits of learning the representation.

5.2. Comparison with PixelCNN

While PixelCNN-based approaches are not designed for lossless image compression, they learn a probability distribution over pixels and optimize for the same log-likelihood objective. Since they thus can in principle be used inside a compression algorithm, we show a comparison here.

Sampling Runtimes Table 2 shows a speed comparison to three PixelCNN-based approaches (see Sec. 2 for details on these approaches). We compare time spent when *sampling* from the model, to be able to compare to the PixelCNN literature. Actual decoding times for L3C are given in Sec. 5.3.

While the runtime for PixelCNN [46] and MS-PixelCNN [32] is taken from the table in [32], we can compare with L3C by assuming that PixelCNN++ is not slower than PixelCNN to get a conservative estimate[9], and by considering that MS-PixelCNN reports a $105\times$ speedup over PixelCNN. When comparing on 320×320 crops, we thus observe massive speedups compared to the original PixelCNN: $>1.63 \cdot 10^5\times$ for batch size (BS) 1 and $>5.31 \cdot 10^4\times$ for BS 30. We see that on 320×320 crops, L3C is *at least* $5.06 \cdot 10^2\times$ faster than MS-PixelCNN, the fastest PixelCNN-type approach. Furthermore, Table 2 makes it obvious that the PixelCNN based approaches are not practical for lossless compression of high-resolution images.

We emphasize that it is impossible to do a completely fair comparison with PixelCNN and MS-PixelCNN due to the unavailability of their code and the different hardware. Even if the same hardware was available to us, differences in frameworks/framework versions (PyTorch vs. Tensorflow) can not

[9]PixelCNN++ is in fact around $3\times$ faster than PixelCNN due to modelling the joint directly, see Sec. 2.

be accounted for. See also Sec. A.3 for notes on the influence of the batch size.

Bitcost To put the runtimes reported in Table 2 into perspective, we also evaluate the bitcost on ImageNet32, for which PixelCNN and MS-PixelCNN were trained, in Table 3. We observe our outputs to be 20.6% larger than MS-PixelCNN and 24.4% larger than the original PixelCNN, but smaller than all classical approaches. However, as shown above, this increase in bitcost is traded against orders of magnitude in speed. We obtain similar results for ImageNet64, see Sec. A.2.

5.3. Encoding / Decoding Time

To encode/decode images with L3C (and other methods outputting a probability distribution), a pass with an entropy coder is needed. We implemented a relatively simple pipeline to encode and decode images with L3C, which we describe in the supplementary material, in Section A.1. The results are shown in Tables 4 and A1. As noted in Section A.1, we did not optimize our code for speed, yet still obtain practical runtimes. We also note that to use other likelihood-based methods for lossless compression, similar steps are required. While our encoding time is in the same order as for classical approaches, our decoder is slower than that of the other approaches. This can be attributed to more optimized code and offloading complexity to the encoder – while in our approach, decoding essentially mirrors encoding. However, combining encoding and decoding time we are either faster (FLIF) or have better bitrate (PNG, WebP, JPEG 2000).

5.4. Sampling Representations

We stress that we study image compression and not image generation. Nevertheless, our method produces models from which x and $z^{(s)}$ can be sampled. Therefore, we visualize the output when sampling part of the representations from our model in Fig. 3: the top left shows an image from the Open Images test set, when we store all scales (losslessly). When we store $z^{(1)}, z^{(2)}, z^{(3)}$ but not x and instead sample from $p(x|f^{(1)})$, we only need 39.2% of the total bits without noticeably degrading visual quality. Sampling $z^{(1)}$ and x leads to some blur while reducing the number of stored bits to 9.21% of the full bitcost. Finally, only storing $z^{(3)}$ (containing $64 \times 64 \times 5$ values from $\mathbb{L}$ and 2.85% of the full bitcost) and sampling $z^{(2)}$, $z^{(1)}$, and x produces significant artifacts. However, the original image is still recognizable, showing the ability of our networks to learn a hierarchical representation capturing global image structure.

Codec	Encoding [s]	Decoding [s]	[bpsp]	GPU	CPU
L3C (Ours)	0.242	0.374	2.646	✓	✓
PNG	0.213	$6.09 \cdot 10^{-5}$	3.850		✓
JPEG 2000	$1.48 \cdot 10^{-2}$	$2.26 \cdot 10^{-4}$	2.831		✓
WebP	0.157	$7.12 \cdot 10^{-2}$	2.728		✓
FLIF	1.72	0.133	2.544		✓

Table 4: Encoding and Decoding times compared to classical approaches, on 512×512 crops, as well as bpsp and required devices.

Figure 3: Effect of generating representations instead of storing them, given different $z^{(s)}$ of a 512×512 image from the Open Images test set. Below each generated image, we show the required bitcost and which scales are stored.

6. Conclusion

We proposed and evaluated a fully parallel hierarchical probabilistic model with auxiliary feature representations. Our L3C model outperforms PNG, JPEG 2000 and WebP on all datasets. Furthermore, it significantly outperforms the RGB Shared and RGB baselines which rely on predefined heuristic feature representations, showing that learning the representations is crucial. Additionally, we observed that using PixelCNN-based methods for losslessly compressing full resolution images takes two to five orders of magnitude longer than L3C.

To further improve L3C, future work could investigate weak forms of autoregression across pixels and/or dynamic adaptation of the model network to the current image. Moreover, it would be interesting to explore domain-specific applications, e.g., for medical image data.

Acknowledgments The authors would like to thank Sergi Caelles for the insightful discussions and feedback. This work was partly supported by ETH General Fund (OK) and Nvidia through the hardware grant.

References

[1] E. Agustsson, F. Mentzer, M. Tschannen, L. Cavigelli, R. Timofte, L. Benini, and L. V. Gool. Soft-to-Hard Vector Quantization for End-to-End Learning Compressible Representations. In *NIPS*, 2017. 1

[2] E. Agustsson and R. Timofte. NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study. In *CVPR Workshops*, 2017. 6

[3] E. Agustsson, M. Tschannen, F. Mentzer, R. Timofte, and L. Van Gool. Generative Adversarial Networks for Extreme Learned Image Compression. *arXiv preprint arXiv:1804.02958*, 2018. 1

[4] J. Ballé, V. Laparra, and E. P. Simoncelli. End-to-end Optimized Image Compression. *ICLR*, 2016. 1

[5] J. Ballé, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston. Variational Image Compression with a Scale Hyperprior. *ICLR*, 2018. 2, 6

[6] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking Atrous Convolution for Semantic Image Segmentation. *arXiv preprint arXiv:1706.05587*, 2017. 4

[7] X. Chen, N. Mishra, M. Rohaninejad, and P. Abbeel. PixelSNAIL: An Improved Autoregressive Generative Model. In *ICML*, 2018. 2

[8] P. Chrabaszcz, I. Loshchilov, and F. Hutter. A downsampled variant of ImageNet as an alternative to the CIFAR datasets. *arXiv preprint arXiv:1707.08819*, 2017. 6

[9] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012. 3

[10] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter, and G. Boato. RAISE: A Raw Images Dataset for Digital Image Forensics. In *ACM MMSys*. ACM, 2015. 6

[11] P. Deutsch. DEFLATE compressed data format specification version 1.3. Technical report, 1996. 2

[12] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using Real NVP. *ICLR*, 2017. 3

[13] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *PCS*, 2015. 12

[14] F. Giesen. Interleaved entropy coders. *arXiv preprint arXiv:1402.3392*, 2014. 12

[15] G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning Lecture 6a Overview of mini-batch gradient descent. 6

[16] S. Ioffe. Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models. In *NIPS*, 2017. 4

[17] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 2015. 6

[18] Kakadu JPEG2000 implementation. http://kakadusoftware.com. 7

[19] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *ICLR*, 2014. 3

[20] A. Kolesnikov and C. H. Lampert. PixelCNN Models with Auxiliary Variables for Natural Image Modeling. In *ICML*, 2017. 1, 2

[21] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, S. Kamali, M. Malloci, J. Pont-Tuset, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017. 6

[22] M. Li, W. Zuo, S. Gu, D. Zhao, and D. Zhang. Learning Convolutional Networks for Content-weighted Image Compression. In *CVPR*, 2018. 2

[23] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee. Enhanced Deep Residual Networks for Single Image Super-Resolution. In *CVPR Workshops*, 2017. 4

[24] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h. 264/avc video compression standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):620–636, 2003. 2

[25] F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. Van Gool. Conditional Probability Models for Deep Image Compression. In *CVPR*, 2018. 2, 4, 7

[26] B. Meyer and P. Tischer. TMW - a new method for lossless image compression. In *PCS*, 1997. 2

[27] B. Meyer and P. E. Tischer. Glicbawls - Grey Level Image Compression by Adaptive Weighted Least Squares. In *Data Compression Conference*, 2001. 2

[28] D. Minnen, J. Ballé, and G. D. Toderici. Joint Autoregressive and Hierarchical Priors for Learned Image Compression. In *NeurIPS*. 2018. 2

[29] N. Parmar, A. Vaswani, J. Uszkoreit, Ł. Kaiser, N. Shazeer, and A. Ku. Image Transformer. *ICML*, 2018. 2

[30] Pillow Library for Python. https://python-pillow.org. 7

[31] Portable Network Graphics (PNG). http://libpng.org/pub/png/libpng.html. 1, 2

[32] S. Reed, A. Oord, N. Kalchbrenner, S. G. Colmenarejo, Z. Wang, Y. Chen, D. Belov, and N. Freitas. Parallel Multiscale Autoregressive Density Estimation. In *ICML*, 2017. 1, 2, 6, 7, 12

[33] I. E. Richardson. *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons, 2004. 2

[34] O. Rippel and L. Bourdev. Real-Time Adaptive Image Compression. In *ICML*, 2017. 1

[35] T. Salimans, A. Karpathy, X. Chen, and D. P. Kingma. PixelCNN++: A PixelCNN Implementation with Discretized Logistic Mixture Likelihood and Other Modifications. *ICLR*, 2017. 1, 2, 4, 5, 7

[36] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 1948. 3

[37] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. *CVPR*, 2016. 4

[38] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58, 2001. 1, 2

[39] J. Sneyers and P. Wuille. FLIF: Free lossless image format based on MANIAC compression. In *ICIP*, 2016. 1, 2, 7

[40] L. Theis, A. v. d. Oord, and M. Bethge. A note on the evaluation of generative models. *ICLR*, 2016. 1, 3

[41] L. Theis, W. Shi, A. Cunningham, and F. Huszar. Lossy Image Compression with Compressive Autoencoders. *ICLR*, 2017. 1

[42] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell. Full Resolution Image Compression with Recurrent Neural Networks. In *CVPR*, 2017. 1

[43] R. Torfason, F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. Van Gool. Towards Image Understanding from Deep Compression without Decoding. *ICLR*, 2018. 13

[44] M. Tschannen, E. Agustsson, and M. Lucic. Deep Generative Models for Distribution-Preserving Lossy Compression. In *NeurIPS*. 2018. 1

[45] A. van den Oord, N. Kalchbrenner, L. Espeholt, k. kavukcuoglu, O. Vinyals, and A. Graves. Conditional Image Generation with PixelCNN Decoders. In *NIPS*, 2016. 1, 2

[46] A. Van Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel Recurrent Neural Networks. In *ICML*, 2016. 1, 2, 5, 6, 7, 12

[47] WebP Image format. `https://developers.google.com/speed/webp/`. 1, 2, 7

[48] M. J. Weinberger, J. J. Rissanen, and R. B. Arps. Applications of Universal Context Modeling to Lossless Compression of Gray-Scale images. *IEEE Transactions on Image Processing*, 5(4):575–586, 1996. 2

[49] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. 3

[50] X. Wu, E. Barthel, and W. Zhang. Piecewise 2D autoregression for predictive image coding. In *ICIP*, 1998. 2

A. Practical Full Resolution Learned Lossless Image Compression – Appendix

Decoding Time	Obtaining CDF for Decoder [s]	Arithmetic Decoding [s]
$s = 3, 64 \times 64$	-	0.00179
$s = 2, 128 \times 128$	0.00737	0.00759
$s = 1, 256 \times 256$	0.0219	0.0234
$s = 0, 512 \times 512$	0.143	0.169
Total	0.172	0.202

Table A1: We show the time to obtain CDF, including all forward passes through the different stages, as well as the time required by the arithmetic decoder. We measured on a Titan X (Pascal), and took the average over 500 crops of 512×512 pixels. For $s = 3$, we assume a uniform prior, and thus do not need to calculate a CDF.

A.1. Encoding and Decoding Details

Table A1 shows the time required to decode each scale s. We first obtain the CDF as a matrix on the CPU to be able to use the arithmetic decoder (see below), and then do a pass with the arithmetic decoder. We did not optimize either part for speed, as noted in Sec. A.1.2.

The following shows detailed steps, using again $z^{(0)} = x$. The steps are also visualized in Fig. A4.

Encoding

1. Forward pass through network to obtain $\forall s: \; z^{(s)}, f^{(s)}$.

2. Encode $z^{(S)}$ assuming a uniform prior, i.e., assuming each of the L symbols is equally likely. This requires $\log_2(L)$ bits per symbol.

3. Update the means μ predicted for the RGB scale ($s = 0$) to $\tilde{\mu}$, given the input x (see Eq. (7)).

4. In practice, the division into intervals $[a, b)$ required for arithmetic coding described in Sec. 3.1 is most efficiently done by having access to the cumulative distribution function (CDF) of the symbol to encode. Thus, for the RGB scale ($s = 0$), we obtain the CDF analogously to Eq. (6):

$$\begin{aligned} C(x_{1uv}|f^{(1)}) &= \sum_k \pi^k_{1uv} \; C_l(x_{1uv}|\tilde{\mu}^k_{1uv}, \sigma^k_{1uv}) \\ C(x_{2uv}|f^{(1)}, x_{1uv}) &= \sum_k \pi^k_{2uv} \; C_l(x_{2uv}|\tilde{\mu}^k_{2uv}, \sigma^k_{2uv}) \\ C(x_{3uv}|f^{(1)}, x_{1uv}, x_{2uv}) &= \sum_k \pi^k_{3uv} \; C_l(x_{3uv}|\tilde{\mu}^k_{3uv}, \sigma^k_{3uv}). \end{aligned} \tag{12}$$

And, analogously to Eq. (9), the CDF for $s > 0$ for each channel c is

$$C(z^{(s)}_{cuv}|f^{(s+1)}) = \sum_k \pi^k_{cuv} C_l(z^{(s)}_c|\mu^k_{cuv}, \sigma^k_{cuv}). \tag{13}$$

C_l in Eqs. (12), (13) is the CDF of the logistic distribution,

$$C_l(z|\mu, \sigma) = \text{sigmoid}((z - \mu)/\sigma).$$

For each s, c the CDF $C(z^{(s)}_c|f^{(s+1)})$ is a $H' \times W' \times L$-dimensional matrix, where $L = 257$ for RGB and $L = 26$ otherwise, and $H' = H/2^s, W' = W/2^s$.

5. For each $s \in \{S+1, \dots, 0\}$, encode each channel c of $z^{(s)}$ with the predicted $C(z^{(s)}_c|f^{(s+1)})$, using adaptive arithmetic coding (see Sec. 3.1). To be able to uniquely decode, the sub-bitstream for $z^{(s)}$ always starts with a triplet encoding its dimensions C, H', W' as UINT16. The final bitstream is the concatenation of all sub-bitstreams.

Decoding

1. Obtain the final $z^{(S)}$ from the bitstream, which was encoded with a uniform prior.

2. Feed $z^{(S)}$ to $D^{(S)}$ to obtain $f^{(S)}$, and thereby also $C(z^{(S-1)}_{cuv}|f^{(S)})$ for all c. Since the decoder now has access to the same CDF as the encoder, we can decode $z^{(S-1)}$ from the bitstream with our adaptive arithmetic decoder.

3. Analogously, we repeat the previous step to obtain $z^{(S)}, \dots, z^{(1)}$, as well as $f^{(S)}, \dots, f^{(1)}$ using the accompanying CDFs.

4. Given $f^{(1)}$, which contains all parameters for the RGB scale (i.e., we know $\forall k, c, u, v$: $\pi^k_{cuv}, \mu^k_{cuv}, \sigma^k_{cuv}$ as well as $\lambda^k_{\alpha uv}, \lambda^k_{\alpha uv}, \lambda^k_{\alpha uv}$, see Sec. 3.4), we can obtain the CDF for the first channel of x (x_1, red channel), $C(x_1|f^{(1)})$, and decode this first channel from the bitstream. Now we know x_1, and with $\mu^k_{2uv}, \lambda^k_{\alpha uv}$ we can obtain $\tilde{\mu}^k_2$ via Eq. (7). With this, we also know the CDF of the next channel, $C(x_2|f^{(1)}, x_1)$, and can decode x_2 from the bitstream. In the same fashion, we can then obtain $\tilde{\mu}^k_3$, then $C(x_3|f^{(1)}, x_1, x_2)$, and thus x_3.

5. Concatenating the channels x_1, x_2, x_3, we finally obtain the decoded image x.

A.1.1 Hardware Used

Our timings were obtained on a machine with a Titan X (Pascal) GPU and Intel Xeon E5-2680 v3 CPU.

A.1.2 Notes on Code Optimization

The encoder can be run in parallel over all scales, as all CDFs are known after one forward pass. Further, we do not need to know the CDF for all symbols, but only for the symbols z we encode and $z+1$, since this specifies the interval $[a, b)$. The decoder is sequential in the scales since $z^{(s)}$ is required to predict the distribution of $z^{(s-1)}$. Still, for $s > 0$, the decoding of the channels of the $z^{(s)}$ could be parallelized, as the channels are modelled fully independently. However, we did not implement either of these improvements, keeping the code simple.

For both encoder and decoder, the CDFs must be available to the CPU, as the arithmetic coder runs there. However, the CDFs are huge tensors for real-world images ($H \times W \times 257$ for RGB, which amounts to 257MB for each channel of a 512×512 image). To save the expensive copying from GPU to CPU, we implemented our own CUDA kernel to store the claculated C directly into "managed memory", which can be accessed from both CPU and GPU. However, we did not optimize this CUDA kernel for speed.

Finally, while state-of-the-art adaptive entropy coders typically require on the order of milliseconds per MB (see [13] and in particular [14] for benchmarks on adaptive entropy coding), we implemented a simple arithmetic coding module to obtain the times in our tables. Please see the code[1] for details.

A.2. Comparison on ImageNet64

We show a bpsp comparison on ImageNet64 in Table A2. Similar to what we observed on ImageNet32 (see Section 5.2), our outputs are 23.8% larger than MS-PixelCNN and 19.4% larger than the original PixelCNN, but smaller than all classical approaches. We note again that increase in bitcost is traded against orders of magnitude in speed.

We also note that the gap between classical approaches and PixelCNN becomes smaller compared to ImageNet32.

A.3. Note on Comparing Times for 32×32 Images

In Table 2, we report run times for batch size 30 to be able to compare with the run times reported in [32]. However, this comparison is biased against us, as can be seen in Table A3: Since our network is fairly small, we can process up to 480 images of size 32×32 in parallel. We observe that the time to sample one image drops as the batch size increases, indicating that for BS=30, some overhead dominates.

Figure A1: Heatmap visualization of the first three channels for each of the representations $z^{(1)}, z^{(2)}, z^{(3)}$, each containing values in $\mathbb{L} = \{1, \ldots, 25\}$, as indicated by the scale underneath.

[bpsp]	ImageNet64	Learned
L3C (ours)	4.42	✓
PixelCNN [46]	3.57	✓
MS-PixelCNN [32]	3.70	✓
PNG	5.74	
JPEG 2000	5.07	
WebP	4.64	
FLIF	4.54	

Table A2: Comparing bits per sub-pixel (bpsp) on the 64×64 images from ImageNet64 of our method (L3C) vs. PixelCNN-based approaches and classical approaches.

Batch Size	Time per image [s]
30	$6.24 \cdot 10^{-4}$
60	$4.31 \cdot 10^{-4}$
120	$3.16 \cdot 10^{-4}$
240	$2.52 \cdot 10^{-4}$
480	$2.42 \cdot 10^{-4}$

Table A3: Effect of varying the batch size.

A.4. Visualizing Representations

We visualize the representations $z^{(1)}, z^{(2)}, z^{(3)}$ in Fig. A1. It can be seen that the global image structure is preserved over scales, with representations corresponding to smaller s modeling more detail. This shows potential for efficiently performing image understanding tasks on partially decoded images similarly as described in [43] for lossy learned compression: instead of training a feature extractor for a given task on x, one could directly use the features $z^{(s)}$ from our network.

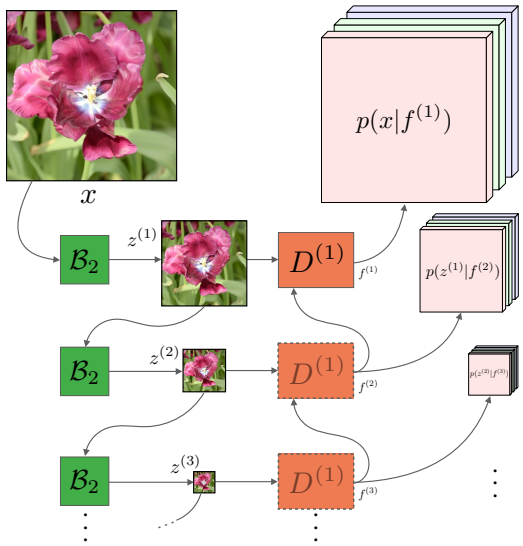

Figure A2: Architecture for the RGB Shared baseline. Note that we train only one predictor $D^{(1)}$.

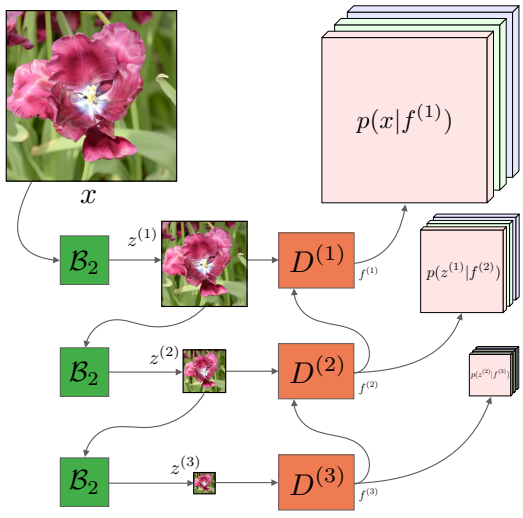

Figure A3: Architecture for the RGB baseline. Multiple predictors are trained.

A.5. Architectures of Baselines

Figs. A2, A3 show the architectures for the RGB Shared and RGB baselines. The dots in Fig. A2 indicate that the model could in theory be applied more since $D^{(1)}$ is used for every scale.

A.6. Encoding and Decoding Visualized

We visualize the steps needed to encode the different $z^{(s)}$ in Fig. A4 on the next page.

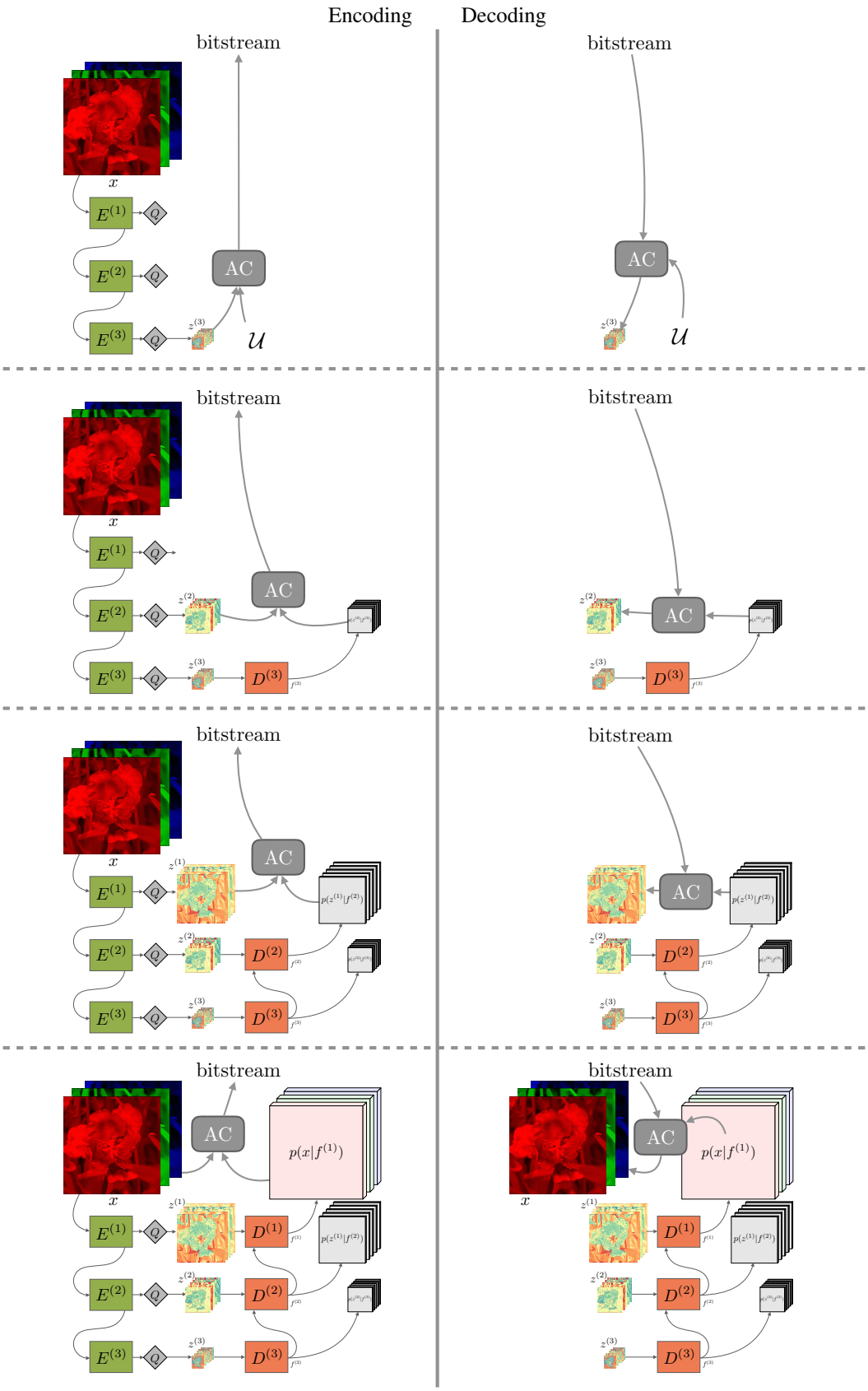

Figure A4: Visualizing encoding and decoding: At every step, the arithmetic coder (AC) takes a probability distribution and a $z^{(s)}$.