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.

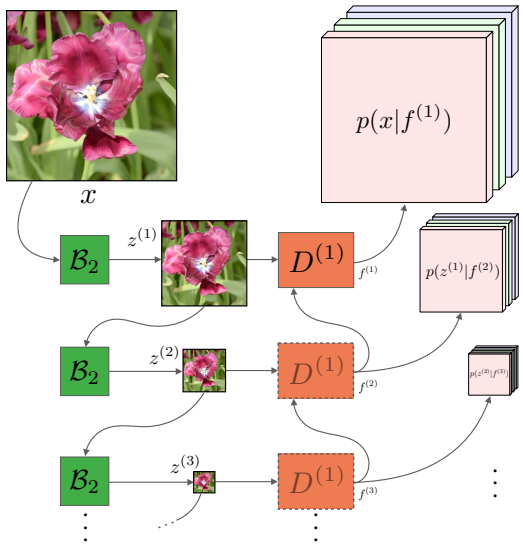

Figure A2: Architecture for the RGB Shared baseline. Note that we train only one predictor $D^{(1)}$.

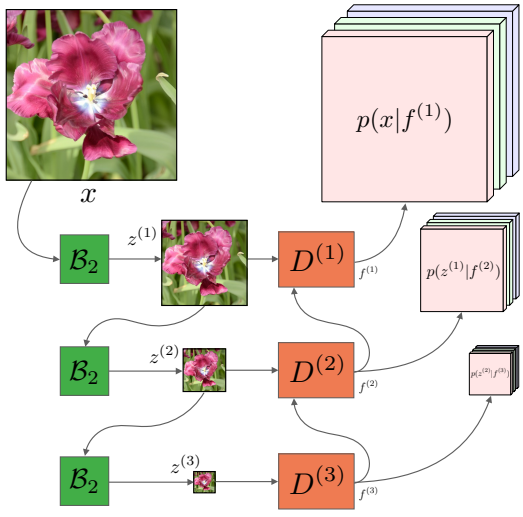

Figure A3: Architecture for the RGB baseline. Multiple predictors are trained.

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.

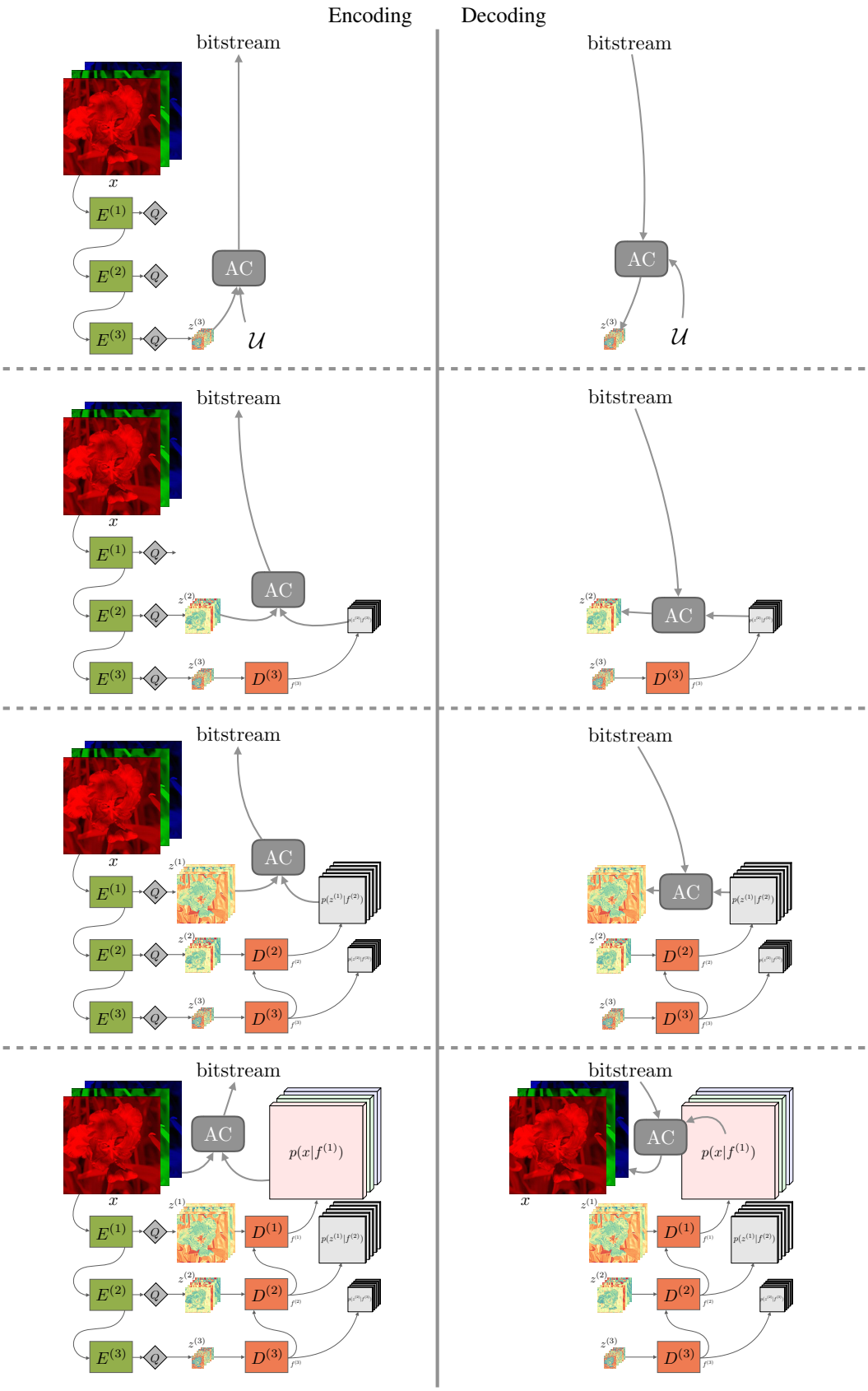

Figure A4: Visualizing encoding and decoding: At every step, the arithmetic coder (AC) takes a probability distribution and a $z^{(s)}$.