ProgDTD: Progressive Learned Image Compression with Double-Tail-Drop Training

Ali Hojjat[1], Janek Haberer[1], Olaf Landsiedel[1,2]
[1]Kiel University, Germany; [2]Chalmers University of Technology, Sweden
{aho, jha, ol}@informatik.uni-kiel.de

Abstract

Progressive compression allows images to start loading as low-resolution versions, becoming clearer as more data is received. This increases user experience when, for example, network connections are slow. Today, most approaches for image compression, both classical and learned ones, are designed to be non-progressive. This paper introduces ProgDTD, a training method that transforms learned, non-progressive image compression approaches into progressive ones. The design of ProgDTD is based on the observation that the information stored within the bottleneck of a compression model commonly varies in importance. To create a progressive compression model, ProgDTD modifies the training steps to enforce the model to store the data in the bottleneck sorted by priority. We achieve progressive compression by transmitting the data in order of its sorted index. ProgDTD is designed for CNN-based learned image compression models, does not need additional parameters, and has a customizable range of progressiveness. For evaluation, we apply ProgDTD to the hyperprior model, one of the most common structures in learned image compression. Our experimental results show that ProgDTD performs comparably to its non-progressive counterparts and other state-of-the-art progressive models in terms of MS-SSIM and accuracy.

1. Introduction

Image compression has been an active research field for many years and has led to numerous classical compression methods such as JPEG [41], JPEG 2000 [32], WebP [42] and BPG [5]. The rise of deep learning [21] inspired new methods that employ the neural networks' power for learned image compression [14, 23, 36, 37]. Among these, particularly the recent success of the variational image compression led to new, state-of-the-art methods, which often perform on par or even better than established, classical and deep methods [3, 4, 8, 26, 33].

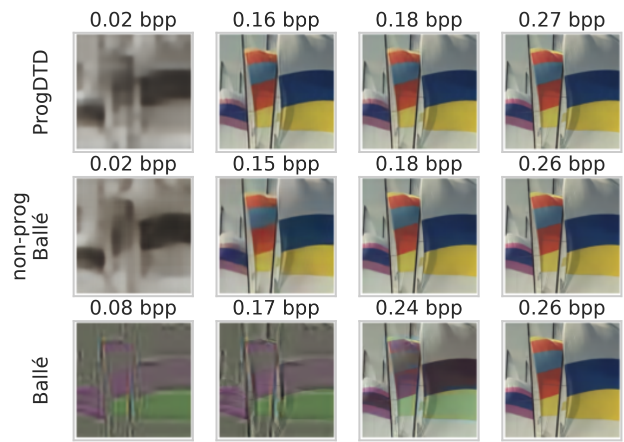


Figure 1. Qualitative comparison of reconstructed images from: *ProgDTD* (trained with $\lambda = 0.01$), *non-progressive-Ballé* (trained with $\lambda = 0.0001, 0.001, 0.005, 0.01$) and *standard-Ballé* (trained with $\lambda = 0.01$). *ProgDTD* and *non-progressive-Ballé* reconstruct images of similar quality; meanwhile, removing only a few bits in *standard-Ballé* leads to a significant degradation in quality.

Most image compression methods, both classic such as original JPEG [41], Webp [42], BPG [5] and learned ones [3, 4, 8, 26], are non-progressive. Thus, these expect the complete compressed image to be available for decoding. Such availability of an entire file is, however, a challenge in many settings: for example, slow network connections often delay the transmissions. As a result, a user or system experiences a delay until the image can be reconstructed for viewing or further processing. Progressive compression [29] addresses this problem, and the decoder can obtain an initial preview even with a small portion of the data. Later, by receiving the rest of the bits, the decoder can reconstruct a better-quality image. However, most learned approaches to image compression are non-progressive, and only a few are progressive [6, 12, 16, 24, 37].

In this paper, we introduce ProgDTD, a method to transform learned, non-progressive image compression approaches into progressive ones. Further, to showcase its efficiency, we apply ProgDTD to the Ballé [4] architecture, which is a widely adopted, state-of-the-art learned image compression model, and acts as a base architecture for many recent models [9,26]. The design of ProgDTD builds on the following observation: Previous research [20,31,43] shows that the information stored within the bottleneck of a compression model commonly varies in terms of importance, i.e., its availability impacts the reconstructed image differently.

To create a progressive compression model, we identify the most important data for each input within the bottleneck and transmit these ordered by their significance. For this, we modify the training steps of the model to enforce it to store the data in the bottleneck sorted by priority. We build on the tail-drop technique [20] and, as an example, apply it to the training of both the latent and hyper latent bottlenecks of the Ballé model [4]. We call this process double-tail-drop (DTD).

Our experimental results show that our progressive compression model trained with double-tail-drop has comparable performance in terms of MS-SSIM and accuracy and has a slight drop in PSNR compared to *non-progressive-Ballé* (ensemble of Ballé models trained for different λ) and other state-of-the-art models. A key benefit of our progressive model is that we do not add any parameters to the model. Instead, we modify the training method to sort the information in the bottleneck in order of importance. Also, ProgDTD has a customizable range of progressiveness, so we can choose the desirable bitrate range. Figure 1 shows a qualitative comparison of ProgDTD, *non-progressive-Ballé* and *standard-Ballé*. It underlines that ProgDTD and *non-progressive-Ballé* achieve similar results, whereas the *standard-Ballé* model requires the whole latent for a meaningful reconstruction of an image. Overall, the contributions of ProgDTD are as follows:

1. We introduce ProgDTD, a progressive image compression method that enables a non-progressive model to become progressive.
2. ProgDTD is a training approach that does not need additional parameters and is designed for learned image compression.
3. ProgDTD has a customizable range of progressiveness.
4. ProgDTD performs on par compared to its non-progressive version and other state-of-the-art benchmarks.

The remainder of this paper is structured as follows: In Section 2, we present related works. Then, in Section 3, we introduce the design of ProgDTD and its training. In Section 4, we evaluate ProgDTD by comparing it to the state-of-the-art. Section 5 concludes the paper.

2. Related Work

In this section, we introduce the required related work on learned image compressions, learned variable bitrate compression, and deep progressive image compression.

2.1. Learned image compression

Most of the learned image compression methods are based on autoencoders or an extension of them. In these types of networks, an encoder $g_a(x; \phi_g)$ compresses the input x into a latent space y. In deep image compression, authors usually call this the $AnalysisNetwork$. Next, entropy models like Huffman encoding or arithmetic encoding [22] compress and quantize the latent space y to $\hat{y}$. During training, random noise often simulates quantization loss and helps the model to adapt to quantization effects [3]. For decoding, a so called $SynthesisNetwork$, receives the bits, recovers $\hat{y}$ and another network $g_s(\hat{y}; \theta_g)$ reconstructs $\hat{x}$.

Ballé *et al.* [4] introduce a hyperprior for extracting spatial correlation in images. It assumes that correlations in each image are normal distributions with a mean of zero, and the task of the hyperprior network is to predict the standard deviation and the location of these distributions:

$$p_{\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}}(\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}) \sim N(\boldsymbol{0}, \boldsymbol{\sigma^2}) \tag{1}$$

Minnen *et al.* [26] consider these correlations as $N(\boldsymbol{\mu}, \boldsymbol{\sigma^2})$. Cheng *et al.* [8] extend this work and assume that these correlations contain multiple distributions and detect them with a GMM (Gaussian Mixture Model). In other work, Cui *et al.* [10] use asymmetric normal distributions $N(\boldsymbol{\mu}, \boldsymbol{\sigma_1^2}, \boldsymbol{\sigma_2^2})$ to simulate these correlations. Rippel and Bourdev [30], Tschannen *et al.* [38], and Agustsson *et al.* [1] adapt GANs [13] for image compression, and Nakanishi *et al.* [28] use multi-scale autoencoders to compress images at multiple levels. Nonetheless, deep compression methods commonly compress images at one specific, fixed bitrate.

2.2. Variable bitrate learned image compression

Some compression methods provide variable bitrates due to adaptive models [36] or by employing dynamic quantization [9,35]. For example, Yang *et al.* [46] propose a modulated network allowing the encoder and decoder architecture to provide variable bitrates. Cai *et al.* [6] introduce a CNN-based multi-scale decomposition transformation with content-adaptive rate allocation for variable bitrates. Yang *et al.* [45] use swimmable networks to train one compression model for different bitrates: At low bitrates, they only use parts of the network.

2.3. Progressive learned image compression

In the field of progressive, learned compression, many approaches employ RNNs. For example, Toderici *et al.*

[36] introduce an iterative LSTM based [15] compression method. First, it feeds the image to the LSTM-based compression model, calculates the residual, and feeds it again. After T repetitions, this design results in a T-step progressive compression. In further work, Toderici *et al*. [37] improve upon this by utilizing a Pixel-RNN [39]. Cai *et al*. [7] introduce a progressive compression method based on a two-level encoder-decoder. The first level encodes the input image into a basic representation with low quality, and the second level encodes the input image into a higher-quality enhancement representation. Gregor *et al*. [14] introduce a GAN-based compression method with RNNs which reconstructs the input image by feeding compressed data to the generative models. The benefit of GANs is that they can reconstruct the image even with a small part of the data and produce a more accurate reconstruction by getting more and more data. Lee *et al*. [24] proposes to create a progressive compression method based on trit-planes.

Most of these progressive methods are specifically designed neural networks for progressiveness and can be very complex. In this paper, we propose ProgDTD, which is a training method based on tail-drop [20] (see Figure 2). We adopt the tail-drop idea in ProgDTD to work with CNNs and demonstrate that with a hyperprior-based model. ProgDTD can then add progressiveness without adding any parameters or complexity and has a customizable range of progressiveness.

3. Proposed Algorithm

In this section, we introduce ProgDTD, a method to transform learned, non-progressive image compression approaches into progressive ones. To demonstrate the effectiveness of our method, we integrate it into the Ballé *et al*. [4] architecture, which is a widely adopted, state-of-the-art learned image compression model, and acts as a base architecture for many recent models [9, 26]. Our approach is not restricted to this particular architecture and is designed for learned image compression models with multiple latent representations. In Section 3.1, we briefly introduce and explain the Ballé model, then in Sections 3.2 and 3.3, we explain how we design and integrate ProgDTD into the Ballé architecture.

3.1. Model architecture

As discussed in Section 2, we formulate image compression as:

$$\boldsymbol{y} = g_a(\boldsymbol{x}; \boldsymbol{\phi}) \tag{2}$$
$$\hat{\boldsymbol{y}} = Q(\boldsymbol{y}) \tag{3}$$
$$\hat{\boldsymbol{x}} = g_s(\hat{\boldsymbol{y}}; \boldsymbol{\theta}) \tag{4}$$

where $\boldsymbol{x}, \hat{\boldsymbol{x}}, \boldsymbol{y}$, $\hat{\boldsymbol{y}}$ and Q are raw images, reconstructed images, latent representation before quantization, latent representation after quantization and quantization respectively. Furthermore, ϕ and θ are neural network-based transformations consisting of convolution, de-convolution, GDN, and IGDN [2]. As common, we approximate quantization as a uniform noise $\mathcal{U}\left(-\frac{1}{2}, \frac{1}{2}\right)$ to generate noisy codes $\hat{\boldsymbol{y}}$ during training. Ballé *et al*. define a hyperprior, by introducing side information $\boldsymbol{z}$ to capture spatial dependencies among the elements of $\boldsymbol{y}$, formulated as:

$$\boldsymbol{z} = h_a\left(\boldsymbol{y}; \boldsymbol{\phi_h}\right) \tag{5}$$
$$\hat{\boldsymbol{z}} = Q(\boldsymbol{z}) \tag{6}$$
$$p_{\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}}(\hat{\boldsymbol{y}} \mid \hat{\boldsymbol{z}}) \leftarrow h_s\left(\hat{\boldsymbol{z}}; \boldsymbol{\theta_h}\right) \tag{7}$$

where h_a and h_s denote the analysis and synthesis in the auxiliary autoencoder. Further, $p_{\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}}(\hat{\boldsymbol{y}} \mid \hat{\boldsymbol{z}})$ is the estimated distribution conditioned on $\hat{\boldsymbol{z}}$. Like Ballé *et al*., we use a normal distribution with a zero mean to extract spatial dependencies:

$$p_{\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}}(\hat{\boldsymbol{y}}|\hat{\boldsymbol{z}}) \sim N(\boldsymbol{0}, \boldsymbol{\sigma^2}) \tag{8}$$

We use the RD-Loss (rate-distortion loss function) as:

$$\mathcal{L} = \mathcal{R}(\hat{\boldsymbol{y}}) + \mathcal{R}(\hat{\boldsymbol{z}}) + \lambda \cdot \mathcal{D}(\boldsymbol{x}, \hat{\boldsymbol{x}}) \tag{9}$$

where λ controls the rate-distortion trade-off and different λ values corresponded to different bitrates. $\mathcal{R}(\hat{\boldsymbol{y}})$ and $\mathcal{R}(\hat{\boldsymbol{z}})$ denote the consumed bitrate in each of the bottlenecks. $\mathcal{D}(\boldsymbol{x}, \hat{\boldsymbol{x}})$ denotes the distortion term [8].

3.2. Rateless autoencoder

In ProgDTD, we adjust the training steps to store data in the bottleneck sorted by significance. Then by transmitting the filters of the bottleneck ordered by their significance, we inherently achieve progressive compression. We achieve

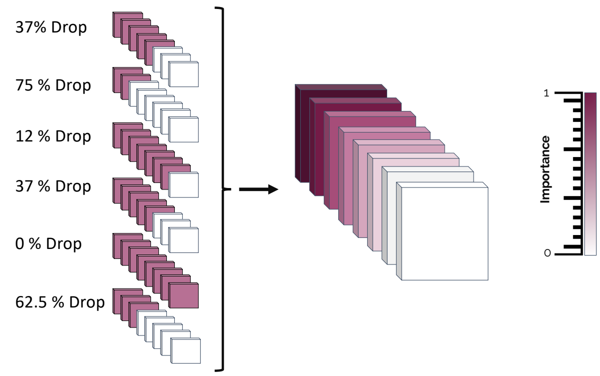


Figure 2. Tail-Drop Training: This image shows an example of the information distribution in the bottleneck $\mathcal{B}[8, M, M]$. If we train the model with the tail-drop, the training procedure puts the data in order of their importance. The colors show the importance of each filter.

Algorithm 1: Double-tail-drop training. The red text marks our extensions over the original algorithm.

Data: Bottleneck of the latent: $\mathcal{B}_{lat}[C, M_l, M_l]$,
Bottleneck of the hyperprior: $\mathcal{B}_{hp}[C, M_h, M_h]$,
Number of batches: N_{batch},
Number of epochs: N_{epoch},
Batch size: $N_{batch-size}$
Result: Progressively trained network
```
for 1 ≤ e_i ≤ N_epoch do
    for 1 ≤ b_i ≤ N_batch do
        for 1 ≤ x_i ≤ N_batch-size do
            B_lat = ImageAnalysis(D[x_i]);
            B_hp = HyperAnalysis(B_lat);
            /* dropping from Tail */
            K ← Generate a sample from U(u_1, u_2);
            B_hp[K : C, M_h, M_h] = 0;
            B_lat[K : C, M_l, M_l] = 0;
            B_hp, P_hp = HyperBottleneck(B_lat);
            σ = HyperSynthesis(B_hp);
            B_lat, P_lat=ImageBottleneck(B_lat, σ);
            P_hp[K : C, M_h, M_h] = 1;
            P_lat[K : C, M_l, M_l] = 1;
            RecImage = ImageAnalysis(B_lat);
        end
    end
    L = RDLoss(Rec, P_lat, P_hp);
    Back-propagation;
    Updating the model parameters θ;
end
```

this goal by modifying the training steps of the model to enforce it to store the data in the bottleneck sorted by priority. We build on the tail-drop technique [20], extend it, and, as a case study, apply it to the training of both the latent and hyper latent bottlenecks of the Ballé model [4]. We call this approach double-tail-drop (DTD).

We define the bottleneck $\mathcal{B}$ with C filters of size $M * M$ as $\mathcal{B}_{[C,M,M]}$. Further, $\mathcal{B}_{[0:K,M,M]}$ denotes that we keep the first K filters of the bottleneck and drop the others. For simplicity, we define the loss function of an autoencoder with K filter in the bottleneck as:

$$\mathcal{L}(\theta, \phi, K) \tag{10}$$

We define our loss function as the following multi-objective optimization problem:

$$\forall_{K\in\{0,1,2,\ldots,N\}} \min \mathcal{L}(\theta, \phi, K) \tag{11}$$

A naïve solution is weighted sum optimization to reduce the problem to a single objective function as follows:

$$\min_{\theta,\phi} \sum_{K=1}^{N} \omega_L \mathcal{L}(\theta, \phi, K) \tag{12}$$

Koike-Akino *et al.* [20] propose that stochastic tail-drop regularization during training can be interpreted as a weight ω_L for MLP neural networks. During training, for each batch, they draw a random number K from the uniform distribution $\mathcal{U}(0, 1)$ and then drop $K\%$ last filters from the bottleneck iteratively for T times. In other words, they calculate the loss T times for each batch. The complexity of this training is in order of $\mathcal{O}(epoch * batch * T * \mathcal{C}_{loss-complexity})$. Their approach is specifically designed for MLP-based neural networks, is computationally heavy, and is not directly applicable to multi-stage autoencoders with auxiliary autoencoders and RD-Loss [3, 4, 26]. In the next section, we close this gap.

3.3. ProgDTD

In this section, we introduce our double-tail-drop algorithm and how we integrate it into multi-stage autoencoders [3, 4, 26]. Then, we discuss the progressive behavior of our model and explain how we extend the RD-Loss function to support double-tail-drop.

3.3.1 Double-tail-drop

We extend the concept of tail-drop [20] to enable progressive image compression. During training, for each input image, we draw a random number K uniformly from $\mathcal{U}(u_1, u_2)$ and drop $K\%$ of the tail of the bottleneck for that specific image. If we do the tail dropping for each image of each batch, we distribute the knowledge like a CDF of uniform distributions $\mathcal{U}(u_1, u_2)$. Our experimentation results show that the uniform distribution performs best among the common distributions. Furthermore, when we calculate the loss for each batch, we drop different percentages of each image in the batch. This allows us to optimize for all of the images in the batch simultaneously by calculating the gradient signals in a way that considers different tail drops to optimize the loss function (Eq. 11).

Figure 2 shows an example of the tail-drop for a batch of size 6 and 8 filters in the bottleneck. With tail-drop training, we force the model to update gradient signals so that the first filters have more important information compared to the later filters. This is because the first filters are involved more frequently in the model training and dropped less often, whereas the final filters have less knowledge due to being dropped more frequently. The complexity of ProgDTD is in order of $\mathcal{O}(epoch * T * \mathcal{C}_{loss-complexity})$.

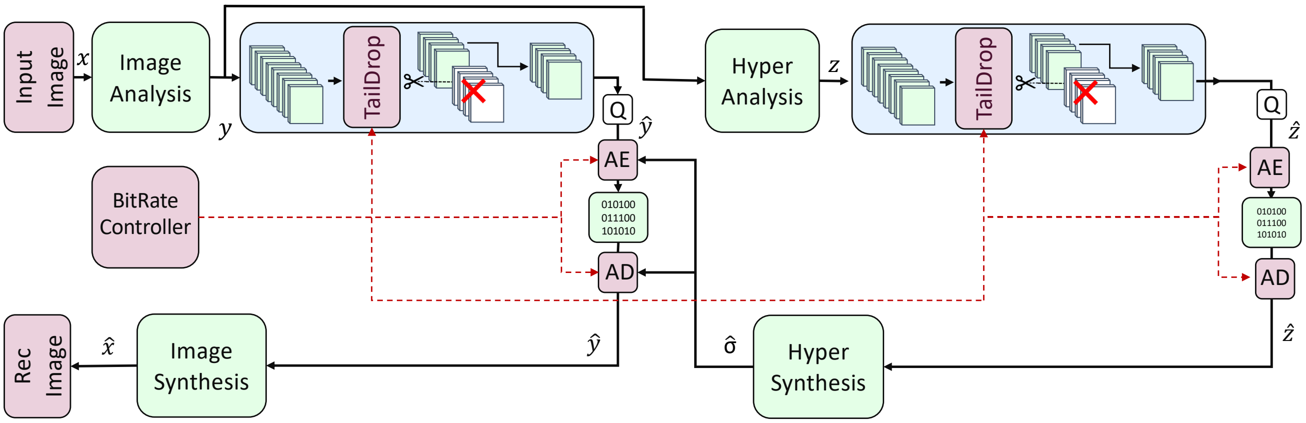


Figure 3. The network architecture of the hyperprior model [4] with double-tail-drop. Quantization is represented by Q, and arithmetic encoding and decoding are denoted by AE and AD, respectively. With the bitrate controller, we can determine the bitrate that we want to have.

3.3.2 Double-tail-drop for ImageAnalysis and HyperAnalysis networks

ProgDTD is designed for CNN-based compression networks. To properly evaluate the benefits of ProgDTD, we employ the hyperprior model [4], the fundamental architecture for most of today's learned image techniques. Figure 3 shows the position of the double-tail-drop in the Ballé (hyperprior) model. We employ two different tail-drop blocks: one for the latent and one for the hyperlatent. As shown in Algorithm 1, we drop a similar percentage in both tail-drop blocks. Therefore we only employ one random number generator. For example, if we keep 30 percent of the latent bottleneck $\mathcal{B}_{lat}$, we also keep 30 percent of the hyperlatent bottleneck $\mathcal{B}_{hp}$. Nevertheless, we can apply tail-drop with different random number generators, but based on our results, the single generator performs better. The reason is that we need to train the model in a way that puts more important information in both bottlenecks simultaneously, which means if we want to send $K\%$ percent of both bottlenecks, it can select the most important parts of the hyperprior and the latent at the same time. Algorithm 1 shows the procedure of tail-drop training.

3.3.3 Exploring progressive behavior

As noted in the previous sections, we need to sample from $\mathcal{U}(u_1, u_2)$ during the training of ProgDTD, and by setting the range (u_1, u_2), we specify the range of scalability. ProgDTD has the following limitation: If we set a wide range of progressiveness, we lose some performance. For example, if we set $(u_1 = 0, u_2 = 1)$, our model can produce scalable results starting from less than 0.01 bpp. However, if we set $(u_1 = 0.3, u_2 = 1)$, we can achieve better performance with a narrower range of progressiveness. Thus, by considering the application of the compression model, we can specify the target bitrate range and train the model on that specific range. In our evaluation in Section 4, we further discuss and evaluate this effect, see Figures 6b, 5a, 5b.

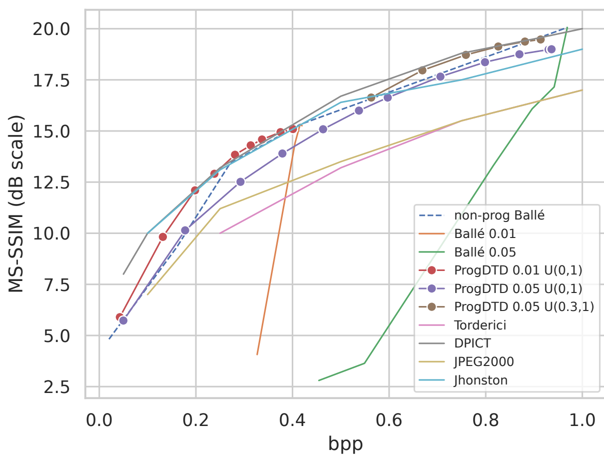
After training our model with ProgDTD, we send data in the bottlenecks according to its index. During training, we drop values by setting them to zero. Further, on the decoder side, we know the positions of the missing filters and thus initialize these as zeros before feeding them with the received filters into the decoder. Overall, this technique leads to a significant reduction in the size of the compressed images.

As discussed in Section 2, most of the existing deep progressive models have limited steps in their scalability of progressiveness [7, 36, 37, 40]. In contrast, in our approach, we achieve $C \times (u_2 - u_1)$ steps of scalability with $\mathcal{B}_{lat}[C, M, M]$ as a bottleneck and $\mathcal{U}(u_1, u_2)$ as a random number generator. For example, in our implementation, which we adapted from Ballé [4], we have 196 progressive steps with $\mathcal{U}(u_1 = 0, u_2 = 1)$ and 129 progressive steps with $\mathcal{U}(u_1 = 0.3, u_2 = 1)$ (see Figures 6b, 5a, 5b).

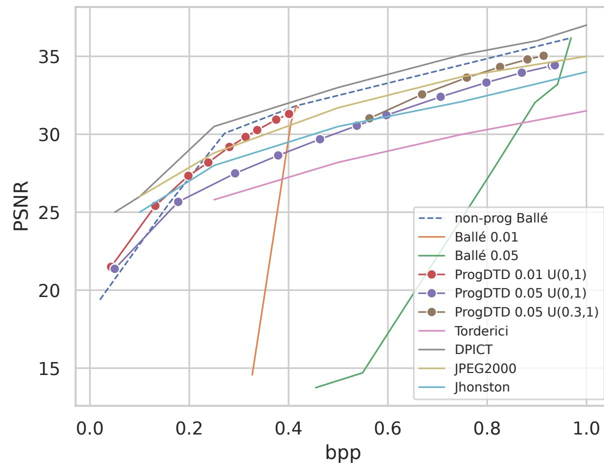
3.3.4 Rate Distortion loss function

Since we do not send the dropped values to the decoder, we must change the bitrate calculator to consider this dropping during training. As shown in the Algorithm 1, we remove the effect of dropped values by setting $\mathcal{P}_{lat}[K : C, M_h, M_h]$ and $\mathcal{P}_{hp}[K : C, M_h, M_h]$ to one.

Then we can rewrite Eq. 9 as:

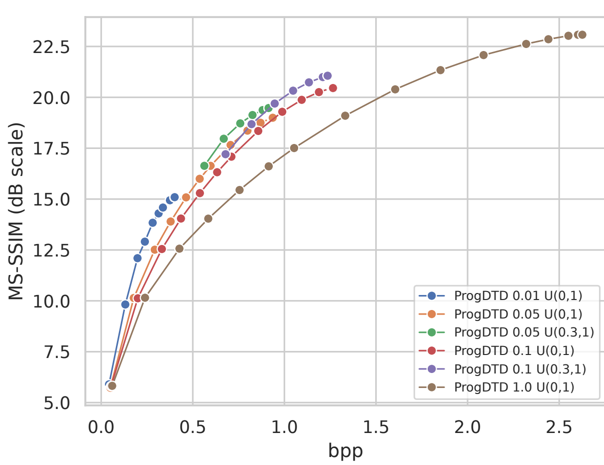


(a) RD curves (MS-SSIM)

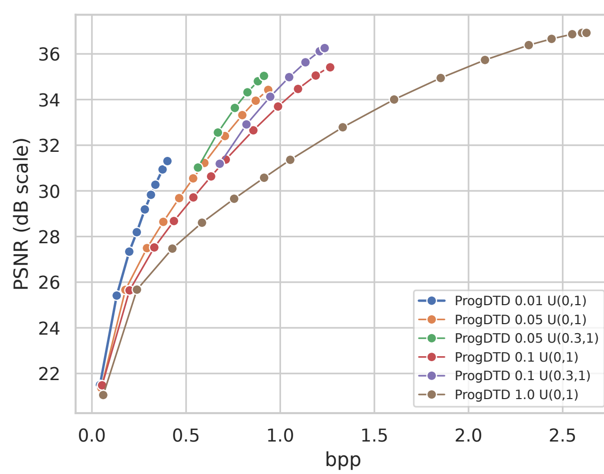


(b) RD curves (PSNR)

Figure 4. RD performance (MS-SSIM and PSNR) comparison of the proposed *ProgDTD* (trained with $\lambda = 0.01, 0.05$ and also with $\mathcal{U}(0,1)$ and $\mathcal{U}(0.3,1)$ as the random number generator), *non-progressive-Ballé* (trained with $\lambda = 0.0001, 0.001, 0.005, 0.01, 0.05$) and *standard-Ballé* (trained with $\lambda = 0.01, 0.05$), DPICT (2022) [24], JPEG2000 [32], Johnston (2018) *et al.* [16] and Torderici (2015) *et al.* [36] on the KODAK dataset. Note that the *non-progressive-Ballé* (dashed line) is a collection of Ballé models which have been trained with different λ, and it is **not** progressive.



(a) RD curves (MS-SSIM) of ProgDTD



(b) RD curves (PSNR) of ProgDTD

Figure 5. RD performance (MS-SSIM and PSNR) comparison of the proposed *ProgDTD* (trained with $\lambda = 0.01, 0.05, 0.1, 1.0$ and also with $\mathcal{U}(0,1)$ and $\mathcal{U}(0.3,1)$ as the random number generator) on the KODAK dataset.

$$
\begin{aligned}
R + \lambda \cdot D = & \mathbb{E}_{\boldsymbol{x}\sim p_{\boldsymbol{x}}} \left[-\log_2 p\prime_{\hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}}) \right] \\
& + \mathbb{E}_{\boldsymbol{x}\sim p_{\boldsymbol{x}}} \left[-\log_2 p\prime_{\hat{\boldsymbol{z}}}(\hat{\boldsymbol{z}}) \right] \\
& + \lambda \cdot \mathbb{E}_{\boldsymbol{x}\sim p_{\boldsymbol{x}}} \| \boldsymbol{x} - \hat{\boldsymbol{x}} \|_2^2
\end{aligned} \tag{13}
$$

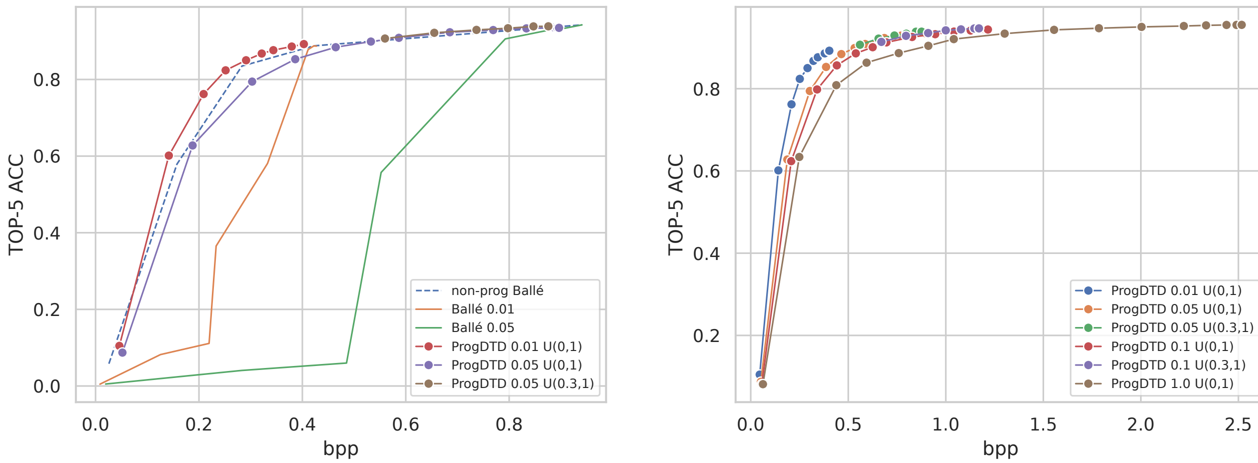
where $p\prime_{\hat{\boldsymbol{z}}}(\hat{\boldsymbol{z}})$ and $p\prime_{\hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}})$ are a modified version of $p_{\hat{\boldsymbol{z}}}(\hat{\boldsymbol{z}})$ and $p_{\hat{\boldsymbol{y}}}(\hat{\boldsymbol{y}})$.

4. Evaluation

In this section, we evaluate the performance of ProgDTD. We begin with a brief discussion of implementation details and datasets. Next, we present parameters before focusing on the performance evaluation. We conclude this section with a discussion of the results.

4.1. Implementation and Datasets

For the model architecture, we build on the Ballé model [4] and integrate ProgDTD into the implementation of the



(a) RD curves (ACC)

(b) RD curves (ACC) of ProgDTD

Figure 6. RA (Rate-Accuracy) performance of *ProgDTD* with same configuration like 4a, 4b, 5b, 5a. Note that the *non-progressive-Ballé* (dashed line) is a collection of Ballé models which have been trained with different λ, and it is **not** progressive.

Ballé model by Facebook Research [27]. As datasets, we use the Vimeo90k dataset [44] for training, and both the KODAK [19] and ImageNet [11] datasets for evaluation. For training, we extract 30,000 images with 256×256 patches from the Vimeo dataset, and we use the Adam [17] optimizer with a batch size of 4 and a learning rate of 0.0001. We train the model for 150 epochs using MSE as the loss function. As typical, we evaluate ProgDTD on the KODAK dataset, which consists of 24 images of resolution 512×768. We resize the images to 512×512 and then extract four patches with a size of 256×256 each. Finally, we calculate the bitrates by bits per pixel and evaluate our model using PSNR and MS-SSIM as metrics. As image compression is often combined with other tasks such as classification, we – in addition – evaluate the classification accuracy of EfficientNet-B0 [34] on the ImageNet [11] dataset, which gets the reconstructed images from ProgDTD as inputs.

4.2. Evaluation Setup

To evaluate our approach, we train our model with $\lambda = 0.01, 0.05$ and the Ballé model with $\lambda = 0.0001, 0.001, 0.005, 0.01, 0.05$ and utilize $\mathcal{U}(0,1)$ and $\mathcal{U}(0.3,1)$ as the ranges for random number generation in our filters. First, we compare ProgDTD with *non-progressive-Ballé*, trained for different lambdas, to show that ProgDTD achieves comparable performance across different bitrates. Second, since ProgDTD provides progressive image compression, we compare our model and the standard Ballé model with the same scenario of sending each filter by its index. We call this *standard-Ballé* and show that it fails to achieve good performance when reducing the bitrate.

4.3. RD-curves with MS-SSIM and PSNR

As discussed in the design section, ProgDTD has a trade-off that needs to be evaluated: Prioritizing a broader range of progressiveness will come at the cost of losing some performance. Conversely, if we limit the range of progressiveness, we achieve better performance. Figure 4a presents MS-SSIM and shows that $ProgDTD^{\lambda=0.05}_{\mathcal{U}(0,1)}$, which has been trained with $\lambda = 0.05$ and $\mathcal{U}(0,1)$, has a performance closed to *non-progressive-Ballé*. In other words, $ProgDTD^{\lambda=0.05}_{\mathcal{U}(0,1)}$ provides progressive compression and achieves a compression quality with a slight drop in MS-SSIM when compared to the standard Ballé which has been trained separately for different λ. We can compensate for this decline in performance by limiting the range of the random number generator $\mathcal{U}(u_1, u_2)$ and thereby limiting the degree of progressiveness. For example, Figure 4a shows that $ProgDTD^{\lambda=0.05}_{\mathcal{U}(0.3,1)}$ reaches the same performance as *non-progressive-Ballé*.

Next, we evaluate PSNR. Figure 4b shows that ProgDTD can not reach the performance of *non-progressive-Ballé* in terms of PSNR. This is due to the following reason: As ProgDTD utilizes the tail-drop approach for training, the last filters of the bottleneck are less involved in the training and therefore receive less gradient to optimize. This leads to the consequence that we can not use the full capacity of the bottleneck. Since PSNR focuses on the actual values, and we do not use the full capacity of the bottleneck, this explains the decline in PSNR when compared to the performance *non-progressive-Ballé*,

As discussed in Section 3, ProgDTD achieves progressive compression by sending filters based on their index. In Figures 4a and 4b we compare our model with *standard-*

Ballé (trained separately for $\lambda = 0.01, 0.05$), which shows the result of the standard Ballé model when we drop filters based on their index. Our results show that – unlike *standard-Ballé* – our model has a graceful decrease in terms of MS-SSIM and PSNR and outperforms *standard-Ballé*, showing the effect of double-tail-drop training. To appropriately evaluate the performance of ProgDTD, we compare it further to selected state-of-the-art models like DPICT (2022) [24], JPEG2000 [32], Johnston (2018) *et al*. [16] and Torderici (2015) *et al*. [36]. Figure 4a shows that our results are comparable to or better than those of the aforementioned models.

To visually evaluate the reconstruction performance of ProgDTD and compare it to state of the art, we feed one sample from the KODAK dataset to *ProgDTD*, *non-progressive-Ballé* and *standard-Ballé*. Figure 1 plots the reconstructed image for multiple compression levels. It shows that *ProgDTD* and *non-progressive-Ballé* have a similar reconstruction result, meanwhile, the *standard-Ballé* model does not have any good reconstruction as soon as we drop from the bottleneck.

4.4. RD-curve with accuracy

One of the use cases of image and video compression is for AI tasks. For example, suppose we have an edge device that does not have the required hardware capability to run a classification task but can compress and send data to a powerful server over a network connection with variable bandwidth [18, 25, 47, 48]. In such a case of variable bandwidth (e.g., cellular connectivity) and especially when classification tasks have a deadline, we argue that progressive image compression provides substantial benefits.

In this scenario, we care about classification accuracy instead of reconstruction metrics like PSNR and MS-SSIM, which we discussed before. We feed the reconstructed image to EfficientNet-B0 [34] and calculate the top-5 accuracy to evaluate this. Figure 6a shows that $ProgDTD^{\lambda=0.01}_{\mathcal{U}(0,1)}$ and $ProgDTD^{\lambda=0.05}_{\mathcal{U}(0.3,1)}$ with the smaller scalability ranges outperform *non-progressive-Ballé*, while $ProgDTD^{\lambda=0.05}_{\mathcal{U}(0,1)}$ with the larger scalability range from 0 bpp to 1 bpp, has on par performance compared to *non-progressive-Ballé*. Moreover, ProgDTD outperforms *standard-Ballé* since *standard-Ballé* does not deal well with dropping parts of the bottleneck.

4.5. Discussion

Overall, our evaluation results show that ProgDTD achieves a good and often even a comparable performance compared to its non-progressive counterpart. However, despite its benefits, the design of ProgDTD is limited. We train the model by utilizing tail-drop, and by dropping from the tail of the bottleneck, we are forcing the model to put the most important information in the first filters. Because of this, the last filters in the bottleneck are less involved in the model training, and they will get less gradient to optimize. In other words, with ProgDTD, we do not use the full capacity of the bottleneck, as shown in Figures 5b, 5a, and 6b. If we train our model with a high λ, we will get an increased range of scalability, but at the same time, we lose some performance in the smaller bitrates. In contrast, if we train with a smaller λ, we achieve better performance at the same bpp, but we have limited our progressiveness.

4.6. Efficiency of ProgDTD

As we mentioned, ProgDTD is a training method that transforms non-progressive compression models to progressive ones and does not add any parameters or complexity to the base model. Therefore, the encoding and decoding times remain the same. Like other learned progressive compression models [24, 36], we only add decoding time by iterating the decoder when we want to load an image progressively. However, when receiving incomplete data due to time-critical constraints, by running the decoder only once after the cut-off time, we can have a preview of the image even with a small portion of the data, which would not work for a non-progressive model, as shown in Figures 4a, 4b,6a.

5. Conclusions

This paper presents ProgDTD, a training approach that transforms non-progressive image compression methods into progressive ones. The idea behind ProgDTD is based on the fact that information stored within the bottleneck of a compression model can have varying levels of importance. ProgDTD modifies the training steps to prioritize data storage in the bottleneck based on its importance and transmits the data based on their level of importance to achieve progressive compression. This technique is designed for CNN-based learned image compression models and requires no additional parameters. We use the hyperprior model, a fundamental structure of learned image compression methods, to evaluate the effectiveness of ProgDTD. Our experimental results show that ProgDTD performs comparably to its non-progressive counterparts and other state-of-the-art progressive models in terms of MS-SSIM and accuracy. For our future work, we will explore the performance of ProgDTD on other learned compression models to verify that ProgDTD works with any CNN-based model.

Acknowledgements

This project has received funding from the Federal Ministry for Digital and Transport under the CAPTN-Förde 5G project grant no. 45FGU139_H and Federal Ministry for Economic Affairs and Climate Action under the Marispace-X project grant no. 68GX21002E.

References

[1] Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. Generative adversarial networks for extreme learned image compression. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 221–231, 2019. 2

[2] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. Density modeling of images using a generalized normalization transformation. *arXiv preprint arXiv:1511.06281*, 2015. 3

[3] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016. 1, 2, 4

[4] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*, 2018. 1, 2, 3, 4, 5, 6

[5] Bpg image format. https://bellard.org/bpg/. Accessed: 2023-02-14. 1

[6] Chunlei Cai, Li Chen, Xiaoyun Zhang, and Zhiyong Gao. Efficient variable rate image compression with multi-scale decomposition network. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(12):3687–3700, 2018. 1, 2

[7] Chunlei Cai, Li Chen, Xiaoyun Zhang, Guo Lu, and Zhiyong Gao. A novel deep progressive image compression framework. In *2019 Picture Coding Symposium (PCS)*, pages 1–5. IEEE, 2019. 3, 5

[8] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned image compression with discretized gaussian mixture likelihoods and attention modules. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7939–7948, 2020. 1, 2, 3

[9] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Variable rate deep image compression with a conditional autoencoder. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3146–3154, 2019. 2, 3

[10] Ze Cui, Jing Wang, Shangyin Gao, Tiansheng Guo, Yihui Feng, and Bo Bai. Asymmetric gained deep image compression with continuous rate adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10532–10541, 2021. 2

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 7

[12] Enmao Diao, Jie Ding, and Vahid Tarokh. Drasic: Distributed recurrent autoencoder for scalable image compression. In *2020 Data Compression Conference (DCC)*, pages 3–12. IEEE, 2020. 1

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020. 2

[14] Karol Gregor, Frederic Besse, Danilo Jimenez Rezende, Ivo Danihelka, and Daan Wierstra. Towards conceptual compression. *Advances In Neural Information Processing Systems*, 29, 2016. 1, 3

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 3

[16] Nick Johnston, Damien Vincent, David Minnen, Michele Covell, Saurabh Singh, Troy Chinen, Sung Jin Hwang, Joel Shor, and George Toderici. Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4385–4393, 2018. 1, 6, 8

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 7

[18] Jong Hwan Ko, Taesik Na, Mohammad Faisal Amir, and Saibal Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018. 8

[19] Eastman kodak (1993). kodak lossless true color image suite (photocd pcd0992). https://r0k.us/graphics/kodak. 7

[20] Toshiaki Koike-Akino and Ye Wang. Stochastic bottleneck: Rateless auto-encoder for flexible dimensionality reduction. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 2735–2740. IEEE, 2020. 2, 3, 4

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017. 1

[22] Glen Langdon and Jorma Rissanen. Compression of black-white images with arithmetic coding. *IEEE Transactions on Communications*, 29(6):858–867, 1981. 2

[23] Jooyoung Lee, Seunghyun Cho, and Seung-Kwon Beack. Context-adaptive entropy model for end-to-end optimized image compression. *arXiv preprint arXiv:1809.10452*, 2018. 1

[24] Jae-Han Lee, Seungmin Jeon, Kwang Pyo Choi, Youngo Park, and Chang-Su Kim. Dpict: Deep progressive image compression using trit-planes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16113–16122, 2022. 1, 3, 6, 8

[25] Hongshan Li, Chenghao Hu, Jingyan Jiang, Zhi Wang, Yonggang Wen, and Wenwu Zhu. Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution. In *2018 IEEE 24th international conference on parallel and distributed systems (ICPADS)*, pages 671–678. IEEE, 2018. 8

[26] David Minnen, Johannes Ballé, and George D Toderici. Joint autoregressive and hierarchical priors for learned image compression. *Advances in neural information processing systems*, 31, 2018. 1, 2, 3, 4

[27] Matthew Muckley, Jordan Juravsky, Daniel Severo, Mannat Singh, Quentin Duval, and Karen Ullrich. Neuralcompression. https://github.com/facebookresearch/NeuralCompression, 2021. 7

[28] Ken M Nakanishi, Shin-ichi Maeda, Takeru Miyato, and Daisuke Okanohara. Neural multi-scale image compression. In *Computer Vision–ACCV 2018: 14th Asian Conference on*

Computer Vision, Perth, Australia, December 2–6, 2018, Revised Selected Papers, Part VI 14, pages 718–732. Springer, 2019. 2

[29] J-R Ohm. Advances in scalable video coding. *Proceedings of the IEEE*, 93(1):42–56, 2005. 1

[30] Oren Rippel and Lubomir Bourdev. Real-time adaptive image compression. In *International Conference on Machine Learning*, pages 2922–2930. PMLR, 2017. 2

[31] Matthias Scholz, Martin Fraunholz, and Joachim Selbig. Nonlinear principal component analysis: neural network models and applications. In *Principal manifolds for data visualization and dimension reduction*, pages 44–67. Springer, 2008. 2

[32] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The jpeg 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58, 2001. 1, 6, 8

[33] Rige Su, Zhengxue Cheng, Heming Sun, and Jiro Katto. Scalable learned image compression with a recurrent neural networks-based hyperprior. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 3369–3373. IEEE, 2020. 1

[34] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019. 7, 8

[35] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017. 2

[36] George Toderici, Sean M O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. Variable rate image compression with recurrent neural networks. *arXiv preprint arXiv:1511.06085*, 2015. 1, 2, 3, 5, 6, 8

[37] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5306–5314, 2017. 1, 3, 5

[38] Michael Tschannen, Eirikur Agustsson, and Mario Lucic. Deep generative models for distribution-preserving lossy compression. *Advances in neural information processing systems*, 31, 2018. 2

[39] Aäron Van Den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International conference on machine learning*, pages 1747–1756. PMLR, 2016. 3

[40] Gregory K Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991. 5

[41] Gregory K Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992. 1

[42] Webp. https://developers.google.com/speed/webp/docs/compression. Accessed: 2023-02-14. 1

[43] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987. 2

[44] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. *International Journal of Computer Vision*, 127:1106–1125, 2019. 7

[45] Fei Yang, Luis Herranz, Yongmei Cheng, and Mikhail G Mozerov. Slimmable compressive autoencoders for practical neural image compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4998–5007, 2021. 2

[46] Fei Yang, Luis Herranz, Joost Van De Weijer, José A Iglesias Guitián, Antonio M López, and Mikhail G Mozerov. Variable rate deep image compression with modulated autoencoder. *IEEE Signal Processing Letters*, 27:331–335, 2020. 2

[47] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 476–488, 2020. 8

[48] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291, 2018. 8