

Full Resolution Image Compression with Recurrent Neural Networks

George Toderici
Google Research

gtoderici@google.com

Damien Vincent
damienv@google.com

Nick Johnston
nickj@google.com

Sung Jin Hwang
sjhwang@google.com

David Minnen
dminnen@google.com

Joel Shor
joelshor@google.com

Michele Covell
covell@google.com

Abstract

This paper presents a set of full-resolution lossy image compression methods based on neural networks. Each of the architectures we describe can provide variable compression rates during deployment without requiring retraining of the network: each network need only be trained once. All of our architectures consist of a recurrent neural network (RNN)-based encoder and decoder, a binarizer, and a neural network for entropy coding. We compare RNN types (LSTM, associative LSTM) and introduce a new hybrid of GRU and ResNet. We also study “one-shot” versus additive reconstruction architectures and introduce a new scaled-additive framework. We compare to previous work, showing improvements of 4.3%–8.8% AUC (area under the rate-distortion curve), depending on the perceptual metric used. As far as we know, this is the first neural network architecture that is able to outperform JPEG at image compression across most bitrates on the rate-distortion curve on the Kodak dataset images, with and without the aid of entropy coding.

1. Introduction

Image compression has traditionally been one of the tasks which neural networks were suspected to be good at, but there was little evidence that it would be possible to train a single neural network that would be competitive across compression rates and image sizes. [17] showed that it is possible to train a single recurrent neural network and achieve better than state of the art compression rates for a given quality regardless of the input image, but was limited to 32×32 images. In that work, no effort was made to capture the long-range dependencies between image patches.

Our goal is to provide a neural network which is competitive across compression rates on images of arbitrary size. There are two possible ways to achieve this: 1) design a stronger patch-based residual encoder; and 2) design an entropy coder that is able to capture long-term dependencies

between patches in the image. In this paper, we address both problems and combine the two possible ways to improve compression rates for a given quality.

In order to measure how well our architectures are doing (i.e., “quality”), we cannot rely on typical metrics such as Peak Signal to Noise Ratio (PSNR), or L_p differences between compressed and reference images because the human visual system is more sensitive to certain types of distortions than others. This idea was exploited in lossy image compression methods such as JPEG. In order to be able to measure such differences, we need to use a human visual system-inspired measure which, ideally should correlate with how humans perceive image differences. Moreover, if such a metric existed, and were differentiable, we could directly optimize for it. Unfortunately, in the literature there is a wide variety of metrics of varying quality, most of which are non-differentiable. For evaluation purposes, we selected two commonly used metrics, PSNR-HVS [7] and MS-SSIM [19], as discussed in Section 4.

1.1. Previous Work

Autoencoders have been used to reduce the dimensionality of images [9], convert images to compressed binary codes for retrieval [13], and to extract compact visual representations that can be used in other applications [18]. More recently, variational (recurrent) autoencoders have been directly applied to the problem of compression [6] (with results on images of size up to 64×64 pixels), while non-variational recurrent neural networks were used to implement variable-rate encoding [17].

Most image compression neural networks use a fixed compression rate based on the size of a bottleneck layer [2]. This work extends previous methods by supporting variable rate compression while maintaining high compression rates beyond thumbnail-sized images.

2. Methods

In this section, we describe the high-level model architectures we explored. The subsections provide additional details about the different recurrent network components in our experiments. Our compression networks are comprised of an encoding network E , a binarizer B and a decoding network D , where D and E contain recurrent network components. The input images are first encoded, and then transformed into binary codes that can be stored or transmitted to the decoder. The decoder network creates an estimate of the original input image based on the received binary code. We repeat this procedure with the residual error, the difference between the original image and the reconstruction from the decoder.

Figure 1 shows the architecture of a single iteration of our model. While the network weights are shared between iterations, the states in the recurrent components are propagated to the next iteration. Therefore residuals are encoded and decoded in different contexts in different iterations. Note that the binarizer B is stateless in our system.

We can compactly represent a single iteration of our networks as follows:

$$\begin{aligned} b_t &= B(E_t(r_{t-1})), & \hat{x}_t &= D_t(b_t) + \gamma \hat{x}_{t-1}, & (1) \\ r_t &= x - \hat{x}_t, & r_0 &= x, & \hat{x}_0 &= 0 \end{aligned} \quad (2)$$

where D_t and E_t represent the decoder and encoder with their states at iteration t respectively, b_t is the progressive binary representation; \hat{x}_t is the progressive reconstruction of the original image x with $\gamma = 0$ for “one-shot” reconstruction or 1 for additive reconstruction (see Section 2.2); and r_t is the residual between x and the reconstruction \hat{x}_t . In every iteration, B will produce a binarized bit stream $b_t \in \{-1, 1\}^m$ where m is the number of bits produced after every iteration, using the approach reported in [17]. After k iterations, the network produces $m \cdot k$ bits in total. Since our models are fully convolutional, m is a linear function of input size. For image patches of 32×32 , $m = 128$.

The recurrent units used to create the encoder and decoder include two convolutional kernels: one on the input vector which comes into the unit from the previous layer and the other one on the state vector which provides the recurrent nature of the unit. We will refer to the convolution on the state vector and its kernel as the “hidden convolution” and the “hidden kernel”.

In Figure 1, we give the spatial extent of the input-vector convolutional kernel along with the output depth. All convolutional kernels allow full mixing across depth. For example, the unit D -RNN#3 has 256 convolutional kernels that operate on the input vector, each with 3×3 spatial extent and full input-depth extent (128 in this example, since the depth of D -RNN#2 is reduced by a factor of four as it goes through the “Depth-to-Space” unit).

The spatial extents of the hidden kernels are all 1×1 , except for in units D -RNN#3 and D -RNN#4 where the hidden kernels are 3×3 . The larger hidden kernels consistently resulted in improved compression curves compared to the 1×1 hidden kernels exclusively used in [17].

During training, a L_1 loss is calculated on the weighted residuals generated at each iteration (see Section 4), so our total loss for the network is:

$$\beta \sum_t |r_t| \quad (3)$$

In our networks, each $32 \times 32 \times 3$ input image is reduced to a $2 \times 2 \times 32$ binarized representation per iteration. This results in each iteration representing $1/8$ bit per pixel (bpp). If only the first iteration is used, this would be 192:1 compression, even before entropy coding (Section 3).

We explore a combination of recurrent unit variants and reconstruction frameworks for our compression systems. We compare these compression results to the results from the deconvolutional network described in [17], referred to in this paper as the Baseline network.

2.1. Types of Recurrent Units

In this subsection, we introduce the different types of recurrent units that we examined.

LSTM: One recurrent neural-network element we examine is a LSTM [10] with the formulation proposed by [20]. Let x_t , c_t , and h_t denote the input, cell, and hidden states, respectively, at iteration t . Given the current input x_t , previous cell state c_{t-1} , and previous hidden state h_{t-1} , the new cell state c_t and the new hidden state h_t are computed as

$$[f, i, o, j]^T = [\sigma, \sigma, \sigma, \tanh]^T ((Wx_t + Uh_{t-1}) + b), \quad (4)$$

$$c_t = f \odot c_{t-1} + i \odot j, \quad (5)$$

$$h_t = o \odot \tanh(c_t), \quad (6)$$

where \odot denotes element-wise multiplication, and b is the bias. The activation function σ is the sigmoid function $\sigma(x) = 1/(1 + \exp(-x))$. The output of an LSTM layer at iteration t is h_t .

The transforms W and U , applied to x_t and h_{t-1} , respectively, are convolutional linear transformations. That is, they are composites of Toeplitz matrices with padding and stride transformations. The spatial extent and depth of the W convolutions are as shown in Figure 1. As pointed out earlier in this section, the U convolutions have the same depths as the W convolutions. For a more in-depth explanation, see [17].

Associative LSTM: Another neural network element we examine is the Associative LSTM [5]. Associative LSTM extends LSTM using holographic representation. Its new

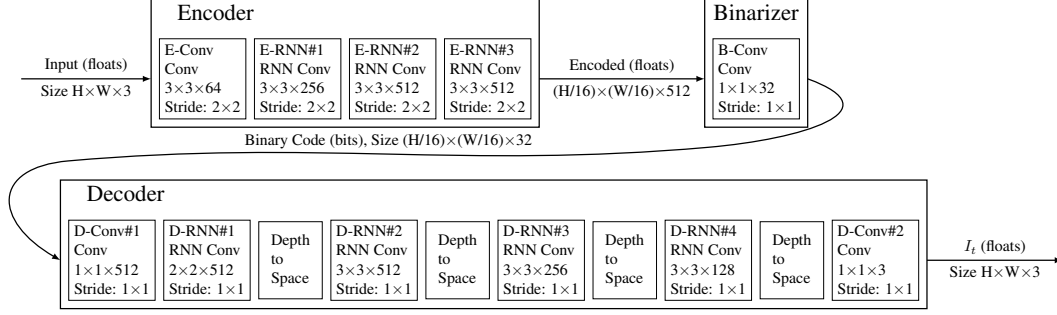


Figure 1. A single iteration of our shared RNN architecture.

states are computed as

$$[f, i, o, j, r_i, r_o]^T = [\sigma, \sigma, \sigma, \text{bnd}, \text{bnd}, \text{bnd}]^T ((Wx_t + Uh_{t-1}) + b), \quad (7)$$

$$c_t = f \odot c_{t-1} + r_i \odot i \odot j, \quad (8)$$

$$h_t = o \odot \text{bnd}(r_o \odot c_t), \quad (9)$$

$$\tilde{h}_t = (\text{Re } h_t, \text{Im } h_t). \quad (10)$$

The output of an Associative LSTM at iteration t is \tilde{h}_t . The input x_t , the output \tilde{h}_t , and the gate values f, i, o are real-valued, but the rest of the quantities are complex-valued. The function $\text{bnd}(z)$ for complex z is z if $|z| \leq 1$ and is $z/|z|$ otherwise. As in the case of non-associative LSTM, we use convolutional linear transformations W and U .

Experimentally, we determined that Associative LSTMs were effective only when used in the decoder. Thus, in all our experiments with Associative LSTMs, non-associative LSTMs were used in the encoder. **Gated Recurrent Units:** The last recurrent element we investigate is the Gated Recurrent Unit [3] (GRU). The formulation for GRU, which has an input x_t and a hidden state/output h_t , is:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}), \quad (11)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1}), \quad (12)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(Wx_t + U(r_t \odot h_{t-1})). \quad (13)$$

As in the case of LSTM, we use convolutions instead of simple multiplications. Inspired by the core ideas from ResNet [8] and Highway Networks [16], we can think of GRU as a computation block and pass residual information around the block in order to speed up convergence. Since GRU can be seen as a doubly indexed block, with one index being iteration and the other being space, we can formulate a residual version of GRU which now has two residual connections. In the equations below, we use h_t^o to denote the output of our formulation, which will be distinct from the

hidden state h_t :

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tanh(Wx_t + U(r_t \odot h_{t-1})) + \alpha_h W_h h_{t-1}, \quad (14)$$

$$h_t^o = h_t + \alpha_x W_{ox} x_t. \quad (15)$$

where we use $\alpha_x = \alpha_h = 0.1$ for all the experiments in this paper.

This idea parallels the work done in Higher Order RNNs [15], where linear connections are added between iterations, but not between the input and the output of the RNN.

2.2. Reconstruction Framework

In addition to using different types of recurrent units, we examine three different approaches to creating the final image reconstruction from our decoder outputs. We describe those approaches in this subsection, along with the changes needed to the loss function.

One-shot Reconstruction: As was done in [17], we predict the full image after each iteration of the decoder ($\gamma = 0$ in (1)). Each successive iteration has access to more bits generated by the encoder which allows for a better reconstruction. We call this approach ‘‘one-shot reconstruction’’. Despite trying to reconstruct the original image at each iteration, we only pass the previous iteration’s residual to the next iteration. This reduces the number of weights, and experiments show that passing both the original image and the residual does not improve the reconstructions.

Additive Reconstruction: In additive reconstruction, which is more widely used in traditional image coding, each iteration only tries to reconstruct the residual from the previous iterations. The final image reconstruction is then the sum of the outputs of all iterations ($\gamma = 1$ in (1)).

Residual Scaling: In both additive and ‘‘one shot’’ reconstruction, the residual starts large, and we expect it to decrease with each iteration. However, it may be difficult for the encoder and the decoder to operate efficiently across a wide range of values. Furthermore, the rate at which the residual shrinks is content dependent. In some patches (e.g., uniform regions), the drop-off will be much more dramatic than in other patches (e.g., highly textured patches).

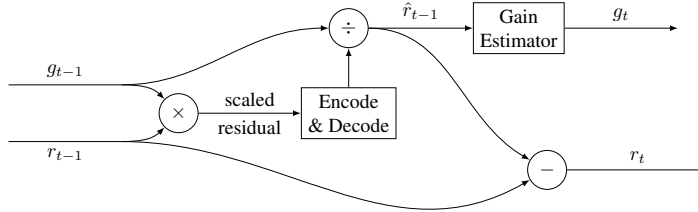


Figure 2. Adding content-dependent, iteration-dependent residual scaling to the additive reconstruction framework. Residual images are of size $H \times W \times 3$ with three color channels, while gains are of size 1 and the same gain factor is applied to all three channels per pixel.

To accommodate these variations, we extend our additive reconstruction architecture to include a content-dependent, iteration-dependent gain factor. Figure 2 shows the extension that we used. Conceptually, we look at the reconstruction of the previous residual image, r_{t-1} , and derive a gain multiplier for each patch. We then multiply the target residual going into the current iteration by the gain that is given from processing the previous iteration’s output. Equation 1 becomes:

$$g_t = G(\hat{x}_t), \quad b_t = B(E_t(r_{t-1} \odot \text{ZOH}(g_{t-1}))), \quad (16)$$

$$\hat{r}_{t-1} = D_t(b_t) \odot \text{ZOH}(g_{t-1}), \quad (17)$$

$$\hat{x}_t = \hat{x}_{t-1} + \hat{r}_{t-1}, \quad r_t = x - \hat{x}_t, \quad (18)$$

$$g_0 = 1, \quad r_0 = x. \quad (19)$$

where \odot is element-wise division and ZOH is spatial upsampling by zero-order hold. $G(\cdot)$ estimates the gain factor, g_t , using a five-layer feed-forward convolutional network, each layer with a stride of two. The first four layers give an output depth of 32, using a 3×3 convolutional kernel with an ELU nonlinearity [4]. The final layer gives an output depth of 1, using a 2×2 convolutional kernel, with an ELU nonlinearity. Since ELU has a range of $(-1, \infty)$ a constant of 2 is added to the output of this network to obtain g_t in the range of $(1, \infty)$.

3. Entropy Coding

The entropy of the codes generated during inference are not maximal because the network is not explicitly designed to maximize entropy in its codes, and the model does not necessarily exploit visual redundancy over a large spatial extent. Adding an entropy coding layer can further improve the compression ratio, as is commonly done in standard image compression codecs. In this section, the image encoder is a given and is only used as a binary code generator.

The lossless entropy coding schemes considered here are fully convolutional, process binary codes in progressive order and for a given encoding iteration in raster-scan order. All of our image encoder architectures generate binary codes of the form $c(y, x, d)$ of size $H \times W \times D$, where H and W are integer fractions of the image height and

width and D is $m \times$ the number of iterations. We consider a standard lossless encoding framework that combines a conditional probabilistic model of the current binary code $c(y, x, d)$ with an arithmetic coder to do the actual compression. More formally, given a context $T(y, x, d)$ which depends only on previous bits in stream order, we will estimate $P(c(y, x, d) | T(y, x, d))$ so that the expected ideal encoded length of $c(y, x, d)$ is the cross entropy between $P(c | T)$ and $\hat{P}(c | T)$. We do not consider the small penalty involved by using a practical arithmetic coder that requires a quantized version of $\hat{P}(c | T)$.

3.1. Single Iteration Entropy Coder

We leverage the PixelRNN architecture [14] and use a similar architecture (BinaryRNN) for the compression of binary codes of a single layer. In this architecture (shown on Figure 3), the estimation of the conditional code probabilities for line y depends directly on some neighboring codes but also indirectly on the previously decoded binary codes through a line of states S of size $1 \times W \times k$ which captures both some short term and long term dependencies. The state line is a summary of all the previous lines. In practice, we use $k = 64$. The probabilities are estimated and the state is updated line by line using a 1×3 LSTM convolution.

The end-to-end probability estimation includes 3 stages. First, the initial convolution is a 7×7 convolution used to increase the receptive field of the LSTM state, the receptive field being the set of codes $c(i, j, \cdot)$ which can influence the probability estimation of codes $c(y, x, \cdot)$. As in [14], this initial convolution is a masked convolution so as to avoid dependencies on future codes. In the second stage, the line LSTM takes as input the result z_0 of this initial convolution and processes one scan line at a time. Since LSTM hidden states are produced by processing the previous scan lines, the line LSTM captures both short- and long-term dependencies. For the same reason, the input-to-state LSTM transform is also a masked convolution. Finally, two 1×1 convolutions are added to increase the capacity of the network to memorize more binary code patterns. Since we attempt to predict binary codes, the Bernoulli-distribution parameter can be directly estimated using a sigmoid activation in the

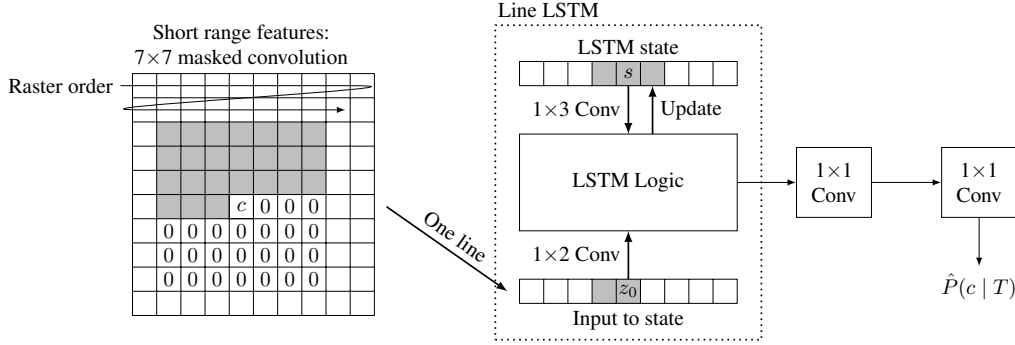


Figure 3. Binary recurrent network (BinaryRNN) architecture for a single iteration. The gray area denotes the context that is available at decode time.

last convolution.

We want to minimize the number of bits used after entropy coding, which leads naturally to a cross-entropy loss. In case of $\{0, 1\}$ binary codes, the cross-entropy loss can be written as:

$$\sum_{y,x,d} -c \log_2(\hat{P}(c | T)) - (1-c) \log_2(1 - \hat{P}(c | T)) \quad (20)$$

3.2. Progressive Entropy Coding

When dealing with multiple iterations, a baseline entropy coder would be to duplicate the single iteration entropy coder as many times as there are iterations, each iteration having its own line LSTM. However, such an architecture would not capture the redundancy between the iterations. We can augment the data that is passed to the line LSTM of iteration #k with some information coming from the previous layers: the line LSTM in Figure 3 receives not just z_0 like in the single iteration approach but also z_1 estimated from the previous iteration using a recurrent network as shown on Figure 4. Computing z_1 does not require any masked convolution since the codes of the previous layers are fully available.

4. Results

Training Setup: In order to evaluate the recurrent models we described, we used two sets of training data. The first dataset is the “ 32×32 ” dataset gathered in [17]. The second dataset takes a random sample of 6 million 1280×720 images on the web, decomposes the images into non-overlapping 32×32 tiles and samples 100 tiles that have the worst compression ratio when using the PNG compression algorithm. By selecting the patches that compress the least under PNG, we intend to create a dataset with “hard-to-compress” data. The hypothesis is that training on such patches should yield a better compression model. We refer to this dataset as the “High Entropy (HE)” dataset.

All network architectures were trained using the Tensorflow [1] API, with the Adam [11] optimizer. Each network was trained using learning rates of $[0.1, \dots, 2]$. The L_1 loss

(see Equation 3) was weighted by $\beta = (s \times n)^{-1}$ where s is equal to $B \times H \times W \times C$ where $B = 32$ is the batch size, $H = 32$ and $W = 32$ are the image height and width, and $C = 3$ is the number of color channels. $n = 16$ is the number of RNN unroll iterations.

Evaluation Metrics: In order to assess the performance of our models, we use a perceptual, full-reference image metric for comparing original, uncompressed images to compressed, degraded ones. It is important to note that there is no consensus in the field for which metric best represents human perception so the best we can do is sample from the available choices while acknowledging that each metric has its own strengths and weaknesses. We use Multi-Scale Structural Similarity (MS-SSIM) [19], a well-established metric for comparing lossy image compression algorithms, and the more recent Peak Signal to Noise Ratio - Human Visual System (PSNR-HVS) [7]. We apply MS-SSIM to each of the RGB channels independently and average the results, while PSNR-HVS already incorporates color information. MS-SSIM gives a score between 0 and 1, and PSNR-HVS is measured in decibels. In both cases, higher values imply a closer match between the test and reference images. Both metrics are computed for all models over the reconstructed images after each iteration. In order to rank models, we use an aggregate measure computed as the area under the rate-distortion curve (AUC).

We collect these metrics on the widely used Kodak Photo CD dataset [12]. The dataset consists of 24 768 \times 512 PNG images (landscape/portrait) which were never compressed with a lossy algorithm.

Architectures: We ran experiments consisting of $\{\text{GRU, Residual GRU, LSTM, Associative LSTM}\} \times \{\text{One Shot Reconstruction, Additive Reconstruction, Additive Rescaled Residual}\}$ and report the results for the best performing models after 1 million training steps.

It is difficult to pick a “winning” architecture since the two metrics that we are using don’t always agree. To further complicate matters, some models may perform better at low bit rates, while others do better at high bit rates. In order to

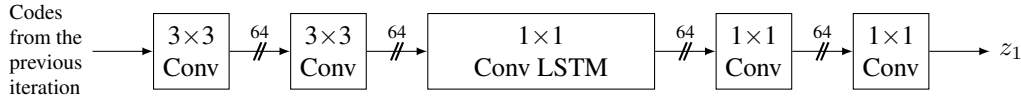


Figure 4. Description of neural network used to compute additional line LSTM inputs for progressive entropy coder. This allows propagation of information from the previous iterations to the current.

be as fair as possible, we picked those models which had the largest area under the curve, and plotted them in [Figure 5](#) and [Figure 6](#).

The effect of the High Entropy training set can be seen in [Table 1](#). In general models benefited from being trained on this dataset rather than on the 32×32 dataset, suggesting that it is important to train models using “hard” examples. For examples of compressed images from each method, we refer the reader to the supplemental materials.

When using the 32×32 training data, GRU (One Shot) had the highest performance in both metrics. The LSTM model with Residual Scaling had the second highest MS-SSIM, while the Residual GRU had the second highest PSNR-HVS. When training on the High Entropy dataset, The One Shot version of LSTM had the highest MS-SSIM, but the worst PSNR-HVS. The GRU with “one shot” reconstruction ranked 2nd highest in both metrics, while the Residual GRU with “one shot” reconstruction had the highest PSNR-HVS.

We depict the results of compressing image 5 from the Kodak dataset in [Figure 7](#). We invite the reader to refer to the supplemental materials for more examples of compressed images from the Kodak dataset.

Entropy Coding: The progressive entropy coder is trained for a specific image encoder, and we compare a subset of our models. For training, we use a set of 1280×720 images that are encoded using one of the previous image encoders (resulting in a $80 \times 45 \times 32$ bitmap or $1/8$ bits per pixel per RNN iteration).

[Figure 5](#) and [Figure 6](#) show that all models benefit from this additional entropy coding layer. Since the Kodak dataset has relatively low resolution images, the gains are not very significant – for the best models we gained between 5% at 2 bpp, and 32% at 0.25 bpp. The benefit of such a model is truly realized only on large images. We apply the entropy coding model to the Baseline LSTM model, and the bit-rate saving ranges from 25% at 2 bpp to 57% at 0.25 bpp.

5. Discussion

We presented a general architecture for compressing with RNNs, content-based residual scaling, and a new variation of GRU, which provided the highest PSNR-HVS out of the models trained on the high entropy dataset. Because our class of networks produce image distortions that are not well captured by the existing perceptual metrics, it is difficult to declare a best model. However, we provided

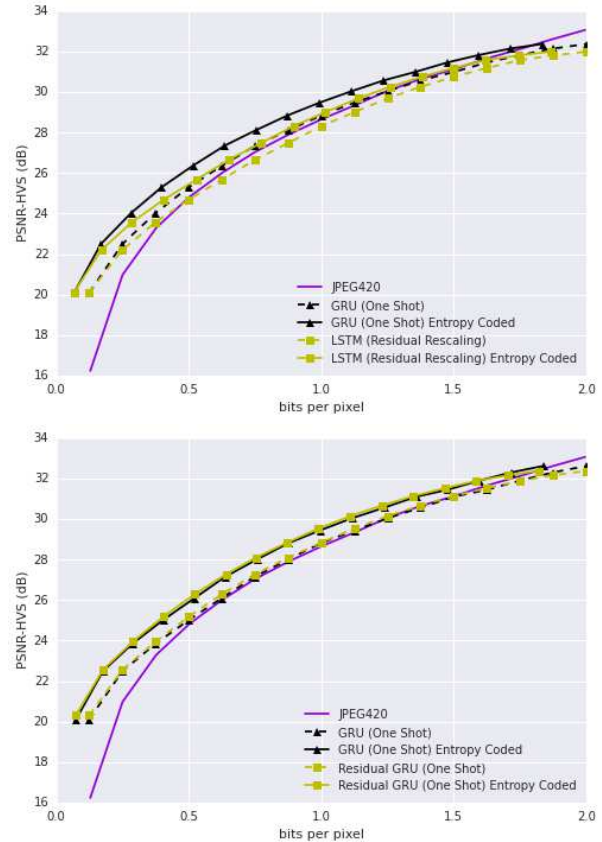


Figure 6. Rate distortion curve on the Kodak dataset given as PSNR-HVS vs. bit per pixel (bpp). Dotted lines: before entropy coding, Plain lines: after entropy coding. Top: Two top performing models trained on the 32×32 dataset. Bottom: Two top performing models trained on the High Entropy dataset.

a set of models which perform well according to these metrics, and on average we achieve better than JPEG performance on both MS-SSIM AUC and PSNR-HVS AUC, both with and without entropy coding. With that said, our models do benefit from the additional step of entropy coding due to the fact that in the early iterations the recurrent encoder models produce spatially correlated codes. Additionally, we are open sourcing our best Residual GRU model and our Entropy Coder training and evaluation in <https://github.com/tensorflow/models/tree/master/comp-resession>.

The next challenge will be besting compression methods derived from video compression methods, such as WebP

Table 1. Performance on the Kodak dataset measured as area under the curve (AUC) for the specified metric, up to 2 bits per pixel. All models are trained up for approximately 1,000,000 training steps. No entropy coding was used. After entropy coding, the AUC will be higher for the network-based approaches.

Trained on the 32×32 dataset.					
Model	Rank	MS-SSIM AUC	Rank	PSNR-HVS AUC	
GRU (One Shot)	1	1.8098	1	53.15	
LSTM (Residual Scaling)	2	1.8091	4	52.36	
LSTM (One Shot)	3	1.8062	3	52.57	
LSTM (Additive Reconstruction)	4	1.8041	6	52.22	
Residual GRU (One Shot)	5	1.8030	2	52.73	
Residual GRU (Residual Scaling)	6	1.7983	8	51.25	
Associative LSTM (One Shot)	7	1.7980	5	52.33	
GRU (Residual Scaling)	8	1.7948	7	51.37	
Baseline [17]		1.7225		48.36	

Trained on the High Entropy dataset.					
LSTM (One Shot)	1	1.8166	8	48.86	
GRU (One Shot)	2	1.8139	2	53.07	
Residual GRU (One Shot)	3	1.8119	1	53.19	
Residual GRU (Residual Scaling)	4	1.8076	7	49.61	
LSTM (Residual Scaling)	5	1.8000	4	51.25	
LSTM (Additive)	6	1.7953	5	50.67	
Associative LSTM (One Shot)	7	1.7912	3	52.09	
GRU (Residual Scaling)	8	1.8065	6	49.97	
Baseline LSTM [17]		1.7408		48.88	

JPEG		
YCbCr 4:4:4		51.28
YCbCr 4:2:0		52.61

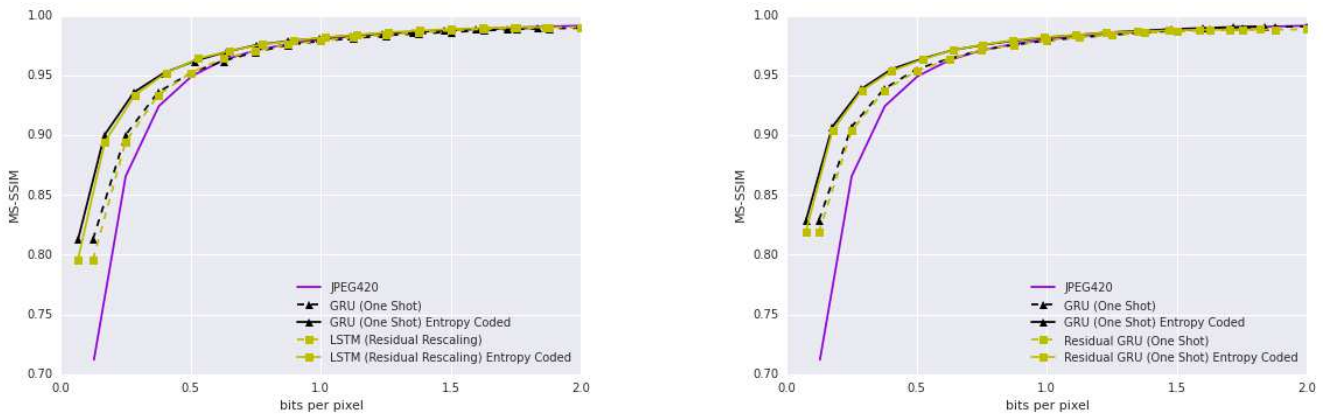


Figure 5. Rate distortion curve on the Kodak dataset given as MS-SSIM vs. bit per pixel (bpp). Dotted lines: before entropy coding, Plain lines: after entropy coding. Left: Two top performing models trained on the 32x32 dataset. Right: Two top performing models trained on the High Entropy dataset.

(which was derived from VP8 video codec), on large images since they employ tricks such as reusing patches that were already decoded. Additionally training the entropy coder (BinaryRNN) and the patch-based encoder jointly and

on larger patches should allow us to choose a trade-off between the efficiency of the patch-based encoder and the predictive power of the entropy coder. Lastly, it is important to emphasize that the domain of perceptual differences is



Figure 7. Comparison of compression results on Kodak Image 5. The top row is target at 0.25 bpp, the bottom row at 1.00 bpp. The left column is JPEG 420 and the right column is our Residual GRU (One Shot) method. The bitrates for our method are before entropy coding. In the first row (0.25 bpp) our results are more able to capture color (notice color blocking on JPEG). In the second row (1.00 bpp) our results don't incur the mosquito noise around objects (one example is highlighted with an orange circle). Results at 1 bpp may be difficult to see on printed page. Additional results available in the supplemental materials.

in active development. None of the available perceptual metrics truly correlate with human vision very well, and if they do, they only correlate for particular types of distortions. If one such metric were capable of correlating with human raters for all types of distortions, we could incorporate it directly into our loss function, and optimize directly for it.

References

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 5
- [2] J. Ballé, V. Laparra, and E. P. Simoncelli. End-to-end optimization of nonlinear transform codes for perceptual quality. In *Picture Coding Symposium*, 2016. 1
- [3] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 3
- [4] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015. 4
- [5] I. Danihelka, G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves. Associative long short-term memory. In *ICML 2016*, 2016. 2
- [6] K. Gregor, F. Besse, D. Jimenez Rezende, I. Danihelka, and D. Wierstra. Towards Conceptual Compression. *ArXiv e-prints*, 2016. 1

- [7] P. Gupta, P. Srivastava, S. Bhardwaj, and V. Bhateja. A modified psnr metric based on hvs for quality assessment of color images. *IEEEExplore*, 2011. 1, 5
- [8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. 3
- [9] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. 1
- [10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997. 2
- [11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 5
- [12] E. Kodak. Kodak lossless true color image suite (PhotoCD PCD0992). 5
- [13] A. Krizhevsky and G. E. Hinton. Using very deep autoencoders for content-based image retrieval. In *European Symposium on Artificial Neural Networks*, 2011. 1
- [14] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016. 4
- [15] R. Soltani and H. Jiang. Higher order recurrent neural networks. *arXiv preprint arXiv:1605.00064*, 2016. 3
- [16] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. In *International Conference on Machine Learning: Deep Learning Workshop*, 2015. 3
- [17] G. Toderici, S. M. O’Malley, S. J. Hwang, D. Vincent, D. Minnen, S. Baluja, M. Covell, and R. Sukthankar. Variable rate image compression with recurrent neural networks. *ICLR 2016*, 2016. 1, 2, 3, 5, 7
- [18] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. *Journal of Machine Learning Research*, 2012. 1
- [19] Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multiscale structural similarity for image quality assessment. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 1398–1402. Ieee, 2003. 1, 5
- [20] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. 2