Efficient Image Compression

James Camacho
MIT / AI
jamesc03@mit.edu

Linda He
Harvard / Applied Mathematics
lindahe@college.harvard.edu

Abstract

With high-dimensional spaces such as images or video, perfect communication becomes prohibitively expensive. A lossy compressed version is cheaper to transmit and good enough for most purposes. Traditional algorithms such as JPEG use a Fourier transform to pick out the most important features for transmission. In this paper, we explore using auto- and raster-encoders to automatically and efficiently compress images instead. We find they significantly outperform JPEG on the MNIST dataset, and discuss potential future improvements to their speed and cost via reinforcement learning.

1 Introduction

Over 75% of internet traffic comes in the form of video (Cisco, 2018), and YouTube alone runs a $30bn business from their distribution (Alphabet Inc., 2024). At these massive scales, every bit of transmission is important. In this paper, we explore more optimal compression, in part inspired from language models. As video is rather compute-intensive to train, we restrict ourselves to the MNIST image dataset.

Image and text have often been treated as separate domains, but there is much that can be applied between the two. Inspired by quantization and fractalization from image compression, Witten *et al.* proposed several lossy text encoding schemes as far back as 1992 (Witten et al., 2000). Recent advances in image generation such as denoising models (Ho et al., 2020) have similarly found applications in text generation by interpreting language tokens as pixels (Kou et al., 2024). Going the other direction, the popular transformer architecture of language models have been applied to vision (Dosovitskiy et al., 2021), and has seen state-of-the-art success in video production (Liu et al., 2024).

This "vision transformer" architecture bears a striking resemblance to JPEG which is no coincidence. Generation is the inverse of compression, and more generally "being able to compress well is closely related to acting intelligently" (Hutter, 2020). Unfortunately, this enforces a tradeoff between the algorithm's speed and size. For example, an unintelligent compressor may store images verbatim, or a generator may simply sample from its training dataset. Smarter algorithms will do much better, but require more computation.

It would be interesting to explore this compute-compression tradeoff in more depth, but this paper will focus mostly on the tradeoff between quality and compression.

2 Data

We use the MNIST dataset with separate train and test sets. MNIST has 28×28 greyscale images with values in the range $[0, 1]$, quantized to 8 bits. A sample image can be seen in Figure 2.

3 Methods

3.1 JPEG

We use JPEG as a baseline to compare against. The steps in JPEG compression are roughly:

1. Colorspace Conversion: RGB images are converted to YCbCr format (a holdout from þe olde days of black-and-white television). Our images are already in greyscale, i.e. solely the Y channel. If they had color, the Cb and Cr colors would be downsampled.

2. Discrete Cosine Transform (DCT): The image is divided into 8×8 pixel blocks, and a DCT is applied to each block, converting the spatial image data into the frequency domain.

3. Quantization: The low frequencies contain most of the important features of an image, so the higher frequencies are more aggressively quantized. The degree of quantization is controlled by a "quality" setting.

4. Entropy Coding: The quantized DCT coefficients are then compressed using lossless entropy coding techniques like Huffman coding or arithmetic coding.

The purpose of the DCT is to extract "important" features. Convolutional neural networks are known to automatically extract important features (Erhan et al., 2009), so our first experiment will be against them. The entropy coding looks very similar to how language models are trained, so our second experiment we will train a model in an LLM fashion.

3.2 Auto-Encoders

Auto-encoders consist of an encoder and a decoder network trained together. Our encoder consists of three convolutional layers and a linear layer to control the latent dimension h, and the decoder has the reverse process (see Figure 1). By default, PyTorch stores floats using four bytes, so the encoding takes $4h$ bytes.

To train, we use the mean-squared error between the original and decoded images, running through two epochs of the MNIST training set with the Adam optimizer at $\gamma = 10^{-3}$. We keep a separate dataset for testing. Sample reconstructions can be seen in Figure 2. As expected, the quality improves as the latent dimension increases, though the reconstructions remain blurry. This is an artifact of our loss function. Minimizing the squared error,

$$L_2 = \sum (x - \text{correct})^2$$

is equivalent to maximizing the log-likelihood under a Gaussian prior for the error function:

$$L_2 = \log \prod e^{(x-\text{correct})^2}.$$

The actual distribution of error is decidedly not Gaussian (Figure 3), so this biases the reconstruction towards a higher entropy than necessary. A common fix is to add a small adversarial loss (Makhzani et al., 2016), but since our quality metric is the L_2 loss, we opt to keep the bias. We will use this bias to improve our raster-encoder.

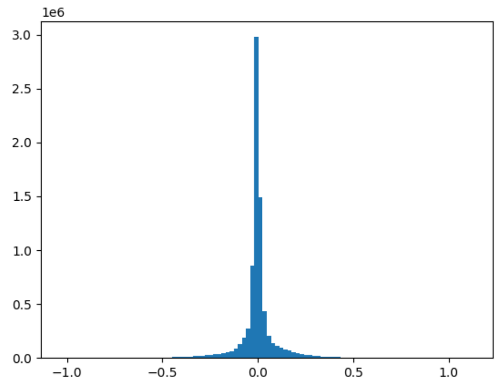

Figure 3: Error histogram, 100 bins.

3.3 Raster-Encoders

In þe olde days, television was broadcast pixel by pixel, row by row, until an entire frame was drawn. This "raster" pattern flattens images into one-dimension, which we use to train models in a language-model style.

Originally we converted the grayscale images to RGB and pulled apart the individual bits, so a $28 \times 28 \times 8$ bit grayscale image became a sequence of $T = 18816$ tokens. This is too large for attention, being over a gigabyte per head dimension ($4T^2 \approx 1.4$GB), so we used the state-space model Mamba (Gu and Dao, 2023). It uses a scan algorithm and takes $O(T)$ memory and $O(T \log T)$ time, as opposed to $O(T^2)$ for attention.

Unfortunately, we ran into software/hardware issues when scaling up, and didn't want to implement Mamba from scratch, so we switched to transformers. To reduce the sequence length, we kept our images grayscale and used bytes rather than bits for our tokens, giving a sequence of 784 tokens with 256 one-hot dimensions (Figure 4).

To compress, we can use arithmetic encoding on the output probabilities. Let $p(x)$ be the probability the next-byte prediction is x and $q(x)$ the distribution of bytes encountered (i.e. in the training set). We want to minimize the compression length,

$$\mathbb{E}_q[-\log p] = \sum q \log p,$$

i.e. the traditional cross-entropy loss. Since the MNIST dataset is relatively small, the training data is unlikely to perfectly match the underlying image distribution. To fix this, we can inject extra entropy into q while minimizing the expected error. We wish to minimize

$$L_2 = \mathbb{E}_q[(x - \text{correct})^2] = \sum (x - \text{correct})^2 q(x)$$

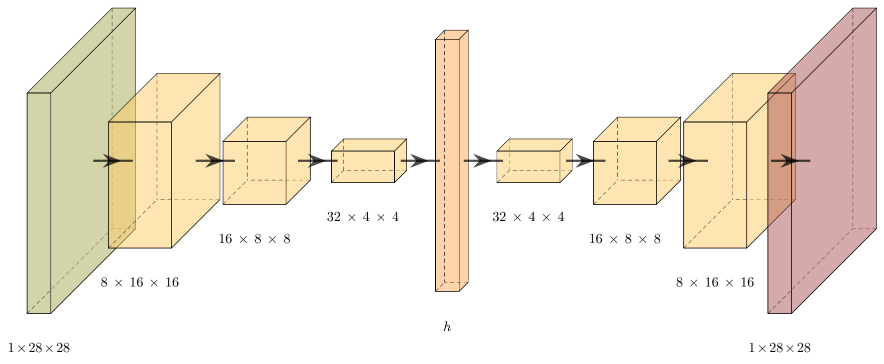

Figure 1: Auto-encoder network.

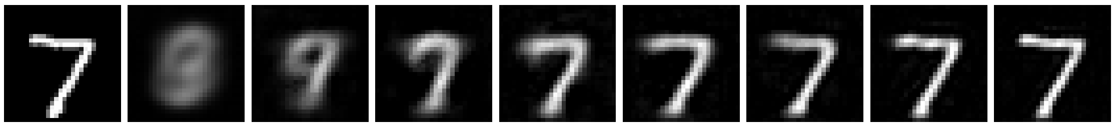

Figure 2: Reconstructions of a sample image (left). To the right, $h = 0, 1, 2, 4, 8, 16, 32, 64$.

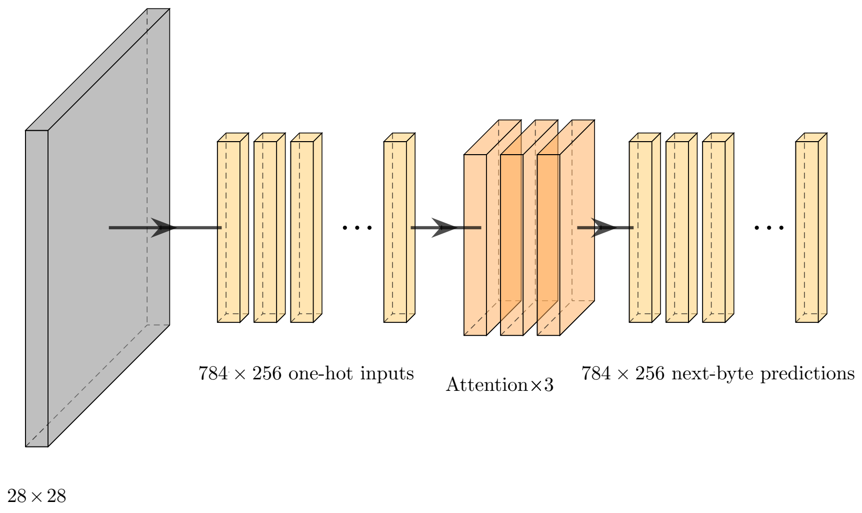

Figure 4: Raster-encoder network.

for a fixed entropy

$$H(q) = -\sum q \log q.$$

Lagrange multipliers give

$$q \propto e^{-\beta(x-\text{correct})^2},$$

which is the Gaussian blurring we saw from the auto-encoder section. The choice of inverse-temperature β does matter. Low values make q essentially uniform, while higher ones provide almost no smoothing. The former makes every pixel take nearly the full eight bits to compress, while the latter has too much surprisal for a few pixels. We find the best results for $\beta \approx 32$.

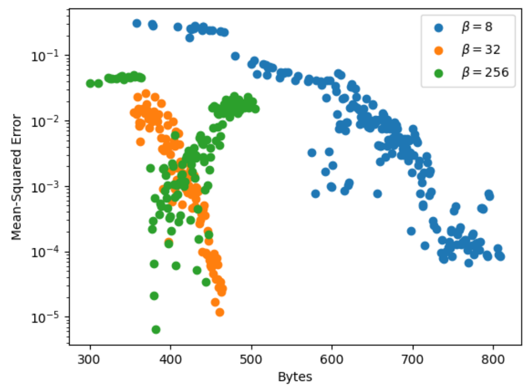

Figure 5: Raster-encoding for several smoothing parameters. Increasing β leads to better compression until reversing around $\beta = 64$, when the surprisal becomes too expensive to allow switching colors often enough.

Since a decoder would not have access to the original image, it can only compute probabilities using previously decoded pixels. Thus, we have to encode the pixels raster-wise, feeding back in the chosen value after each step. We choose values to minimize the following cost function:

$$\text{cost}(x) = \text{quality} \cdot (x - \text{correct})^2 - \log p(x).$$

A higher quality ensures the error is smaller, at the cost of using more bits. Since the decoder does not have access to future pixel values, we also mask the encoder's attention to only look at previous pixels. This has the added benefit of removing artifacts present with full attention (Figure 6).

4 Results

We find the auto-encoders outperform JPEG for small compression lengths, but flatten out at around 400 bytes. This is likely because the convolutions only capture local features. Raster-encoders, which only capture global features, begin outperforming JPEG *after* 400 bytes (Figure 7). This implies a combination of the two may create a better scheme, such as using vision transformers or switching the raster pattern to a space-filling curve.

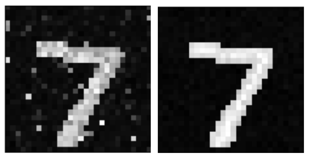

Figure 6: Full attention (left) vs. masked attention (right).

5 Further Work

5.1 Reinforcement Learning

We explored a reinforcement learning (RL) agent-based approach to improve image compression results. In this method, the RL agent determines the discrete allocation of bits (0s and 1s) for compressing the image – the decision-making process is guided by rewards and feedback from the environment, which balances the accuracy of the encoding with the number of bits used.

The model contains the following steps:

1. **Feature Extraction:** The RL agent first inputs the image into an encoder, which is a CNN that extracts features from the image and transforms them into a latent space representation.

2. **Latent Space to Hidden Space Conversion:** The latent space representation is then passed through a decoder: This decoder integrates a DeepMamba model (consists of multiple Mamba layers, each performing specific transformations to refine the hidden space representation) that processes the latent space to output a hidden space. The DeepMamba model.

3. **Action Decision:** Given the hidden space and the current encoding, the RL agent determines the next action. The agent can choose to append either a 0 bit or a 1 bit to the current encoding or terminate the encoding process. This decision is made using a Q-learning policy.

4. **Environment Feedback:** After each action, the environment evaluates the accuracy of the

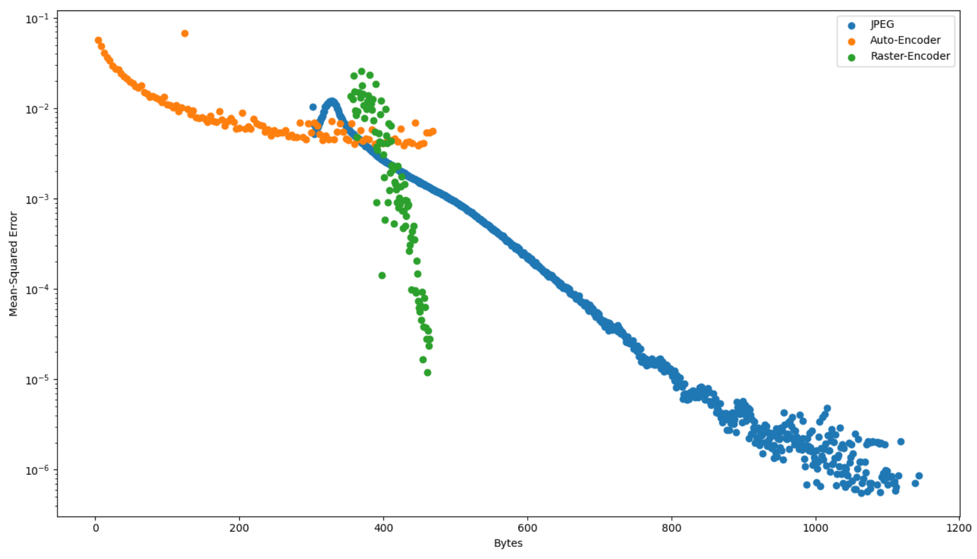

Figure 7: Compression results.

updated encoding. This evaluation is done by comparing the reconstructed image (generated from the new encoding) to the original image using L2 error loss (mean squared error) with a deduction of how many bits are used in the encoding.

5. **Iterative Optimization:** The loop continues iteratively: the agent updates the encoding, receives feedback, and refines its actions to optimize the balance between encoding accuracy and the number of bits used.

Results: Based on our experimental result, the RL agent converges to outputting one bit encoding. RL is very tricky to work with/very unstable, we discuss in the next section ways we can improve this approach.

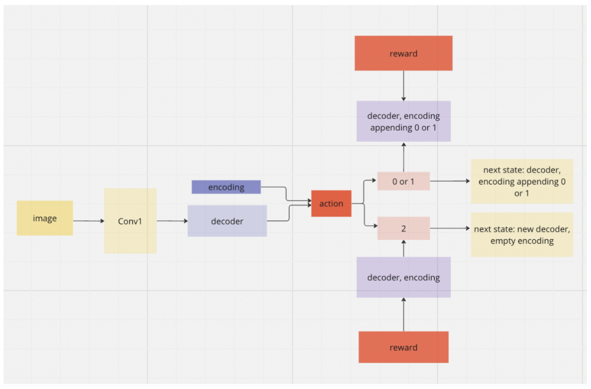
reward
decoder, encoding appending 0 or 1
encoding
action
0 or 1
next state: decoder, encoding appending 0 or 1
image
Conv1
decoder
2
next state: new decoder, empty encoding
decoder, encoding
reward

Figure 8: Reinforcement learning schematic.

5.2 Other Approaches

One potential approach to refining image compression involves employing the mixture-of-experts method. Specifically, given the promising results of raster encoding, we propose training a reinforcement learning (RL) agent to dynamically select the most appropriate pre-trained language model (LLM) for a given image. This approach leverages a diverse set of LLMs, each trained with different parameters and layers of Mamba and GPT-4 models, to optimize the encoding process. The RL agent's task is to decide, based on the characteristics of the input image, whether to apply a dense encoding or a sparse encoding strategy. By intelligently choosing the appropriate LLM model, the agent can balance the trade-offs between encoding density and computational efficiency.

Contributions

James Camacho wrote the sections on auto- and raster-encoding, as well as the introduction. Linda He wrote the sections on JPEG and reinforcement learning.

References

Alphabet Inc. 2024. Alphabet 2024 q1 earnings report.

Cisco. 2018. Global device growth and traffic profiles.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. *Preprint*, arXiv:2010.11929.

Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. 2009. Visualizing higher-layer features of a deep network. *Technical Report, Univeristé de Montréal*.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *Preprint*, arXiv:2312.00752.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Preprint*, arXiv:2006.11239.

Marcus Hutter. 2020. Universal artificial intelligence, aixi, and agi. Video interview.

Siqi Kou, Lanxiang Hu, Zhezhi He, Zhijie Deng, and Hao Zhang. 2024. Cllms: Consistency large language models. *Preprint*, arXiv:2403.00835.

Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, Lifang He, and Lichao Sun. 2024. Sora: A review on background, technology, limitations, and opportunities of large vision models. *Preprint*, arXiv:2402.17177.

Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. 2016. Adversarial autoencoders. *Preprint*, arXiv:1511.05644.

Ian Witten, Timothy Bell, Alistair Moffat, Craig Nevill-Manning, Cowlemon Tony, and Harold Thimbleby. 2000. Semantic and generative models for lossy text compression. *The Computer Journal*, 37.