Supplementary Material

A. NECST architecture and hyperparameters

A.1. MNIST

For MNIST, we used the static binarized version as provided in (Burda et al. (2015)) with train/validation/test splits of 50K/10K/10K respectively.

- encoder: MLP with 1 hidden layer (500 hidden units), ReLU activations
- decoder: 2-layer MLP with 500 hidden units each, ReLU activations. The final output layer has a sigmoid activation for learning the parameters of $p_{\text{noisy_enc}}(y|x;\phi,\epsilon)$
- n_bits: 100
- n_epochs: 200
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.001

A.2. Omniglot

We statically binarize the Omniglot dataset by rounding values above 0.5 to 1, and those below to 0. The architecture is the same as that of the MNIST experiment.

- n_bits: 200
- n_epochs: 500
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.001

A.3. Random bits

We randomly generated length-100 bitstrings by drawing from a Bern(0.5) distribution for each entry in the bitstring. The train/validation/test splits are: 5K/1K/1K. The architecture is the same as that of the MNIST experiment.

- n_bits: 50
- n_epochs: 200
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.001

A.4. SVHN

For SVHN, we collapse the ”easier” additional examples with the more difficult training set, and randomly partition 10K of the roughly 600K dataset into a validation set.

- encoder: CNN with 3 convolutional layers + fc layer, ReLU activations
- decoder: CNN with 4 deconvolutional layers, ReLU activations.
- n_bits: 500
- n_epochs: 500
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.001

The CNN architecture for the encoder is as follows:

1. conv1 = n_filters=128, kernel_size=2, strides=2, padding=”VALID”
2. conv2 = n_filters=256, kernel_size=2, strides=2, padding=”VALID”
3. conv3 = n_filters=512, kernel_size=2, strides=2, padding=”VALID”
4. fc = 4*4*512 $\rightarrow$ n_bits, no activation

The decoder architecture follows the reverse, but with a final deconvolution layer as: n_filters=3, kernel_size=1, strides=1, padding=”VALID”, activation=ReLU.

A.5. CIFAR10

We split the CIFAR10 dataset into train/validation/test splits.

- encoder: CNN with 3 convolutional layers + fc layer, ReLU activations
- decoder: CNN with 4 deconvolutional layers, ReLU activations.
- n_bits: 500
- n_epochs: 500
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.001

The CNN architecture for the encoder is as follows:

1. conv1 = n_filters=64, kernel_size=3, padding="SAME"
2. conv2 = n_filters=32, kernel_size=3, padding="SAME"
3. conv3 = n_filters=16, kernel_size=3, padding="SAME"
4. fc = 4*4*16 $\rightarrow$ n_bits, no activation

Each convolution is followed by batch normalization, a ReLU nonlinearity, and 2D max pooling. The decoder architecture follows the reverse, where each deconvolution is followed by batch normalization, a ReLU nonlinearity, and a 2D upsampling procedure. Then, there is a final deconvolution layer as: n_filters=3, kernel_size=3, padding="SAME" and one last batch normalization before a final sigmoid nonlinearity.

A.6. CelebA

We use the CelebA dataset with standard train/validation/test splits with minor preprocessing. First, we align and crop each image to focus on the face, resizing the image to be $(64, 64, 3)$.

- encoder: CNN with 5 convolutional layers + fc layer, ELU activations
- decoder: CNN with 5 deconvolutional layers, ELU activations.
- n_bits: 1000
- n_epochs: 500
- batch size: 100
- L2 regularization penalty of encoder weights: 0.001
- Adam optimizer with lr=0.0001

The CNN architecture for the encoder is as follows:

1. conv1 = n_filters=32, kernel_size=4, strides=2, padding="SAME"
2. conv2 = n_filters=32, kernel_size=4, strides=2, padding="SAME"
3. conv3 = n_filters=64, kernel_size=4, strides=2, padding="SAME"
4. conv4 = n_filters=64, kernel_size=4, strides=2, padding="SAME"
5. conv5 = n_filters=256, kernel_size=4, strides=2, padding="VALID"
6. fc = 256 $\rightarrow$ n_bits, no activation

The decoder architecture follows the reverse, but without the final fully connected layer and the last deconvolutional layer as: n_filters=3, kernel_size=4, strides=2, padding="SAME", activation=sigmoid.

B. Additional experimental details and results

B.1. WebP/JPEG-Ideal Channel Code System

For the BSC channel, we can compute the theoretical channel capacity with the formula $C = 1 - H_b(\epsilon)$, where ϵ denotes the bit-flip probability of the channel and H_b denotes the binary entropy function. Note that the communication rate of C is achievable in the asymptotic scenario of infinitely long messages; in the finite bit-length regime, particularly in the case of short blocklengths, the highest achievable rate will be much lower.

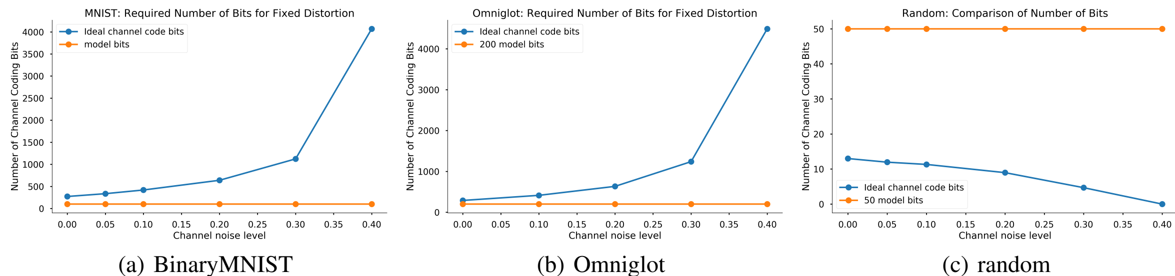
For each image, we first obtain the target distortion d per channel noise level by using a fixed bit-length budget with NECST. Next we use the JPEG compressor to encode the image at the distortion level d. The resulting size of the compressed image $f(d)$ is used to get an estimate $f(d)/C$ for the number of bits used for the image representation in the ideal channel code scenario. While measuring the compressed image size $f(d)$, we ignore the header size of the JPEG image, as the header is similar for images from the same dataset.

The plots compare $f(d)/C$ with m, the fixed bit-length budget for NECST.

B.2. Fixed Rate: JPEG-LDPC system

We first elaborate on the usage of the LDPC software. We use an optimized C implementation of LDPC codes to run our decoding time experiments:
(`http://radfordneal.github.io/LDPC-codes/`).
The procedure is as follows:

- `make-pchk` to create a parity check matrix for a regular LDPC code with three 1's per column, eliminating cycles of length 4 (default setting). The number of parity check bits is: `total number of bits allowed` - `length of codeword`.
- `make-gen` to create the generator matrix from the parity check matrix. We use the default `dense` setting to create a dense representation.
- `encode` to encode the source bits into the LDPC encoding
- `transmit` to transmit the LDPC code through a `bsc` channel, with the appropriate level of channel noise specified (e.g. 0.1)



(a) BinaryMNIST (b) Omniglot (c) random

Figure 5. Theoretical m required for JPEG + ideal channel code to match NECST's distortion.

- `extract` to obtain the actual decoded bits, or `decode` to directly obtain the bit errors from the source to the decoding.

LDPC-based channel codes require larger blocklengths to be effective. To perform an end-to-end experiment with the JPEG compression and LDPC channel codes, we form the input by concatenating multiple blocks of images together into a grid-like image. In the first step, the fixed rate of m is scaled by the total number of images combined, and this quantity is used to estimate the target $f(d)$ to which we compress the concatenated images. In the second step, the compressed concatenated image is coded together by the LDPC code into a bit-string, so as to correct for any errors due to channel noise.

Finally, we decode the corrupted bit-string using the LDPC decoder. The plots compare the resulting distortion of the compressed concatenated block of images with the average distortion on compressing the images individually using NECST. Note that, the experiment gives a slight disadvantage to NECST as it compresses every image individually, while JPEG compresses multiple images together.

We report the average distortions for sampled images from the test set.

Unfortunately, we were unable to run the end-to-end for some scenarios and samples due to errors in the decoding (LDPC decoding, invalid JPEGs etc.).

B.3. VAE-LDPC system

For the VAE-LDPC system, we place a uniform prior over all the possible latent codes and compute the KL penalty term between this prior $p(y)$ and the random variable $q_{\text{noisy_enc}}(y|x;\phi,\epsilon)$. The learning rate, batch size, and choice of architecture are data-dependent and fixed to match those of NECST as outlined in Section C. However, we use half the number of bits as allotted for NECST so that during LDPC channel coding, we can double the codeword length in order to match the rate of our model.

B.4. Interpolation in Latent Space

We show results from latent space interpolation for two additional datasets: SVHN and celebA. We used 500 bits for SVHN and 1000 bits for celebA across channel noise levels of [0.0, 0.1, 0.2, 0.3, 0.4, 0.5].

B.5. Markov chain image generation

We observe that we can generate diverse samples from the data distribution after initializing the chain with both: (1) examples from the test set and (2) random Gaussian noise $x_0 \sim \mathcal{N}(0, 0.5)$.

B.6. Downstream classification

Following the setup of (Grover & Ermon (2019)), we used standard implementations in `sklearn` with default parameters for all 8 classifiers with the following exceptions:

1. KNN: `n_neighbors=3`
2. DF: `max_depth=5`
3. RF: `max_depth=5, n_estimators=10, max_features=1`
4. MLP: `alpha=1`
5. SVC: `kernel=linear, C=0.025`

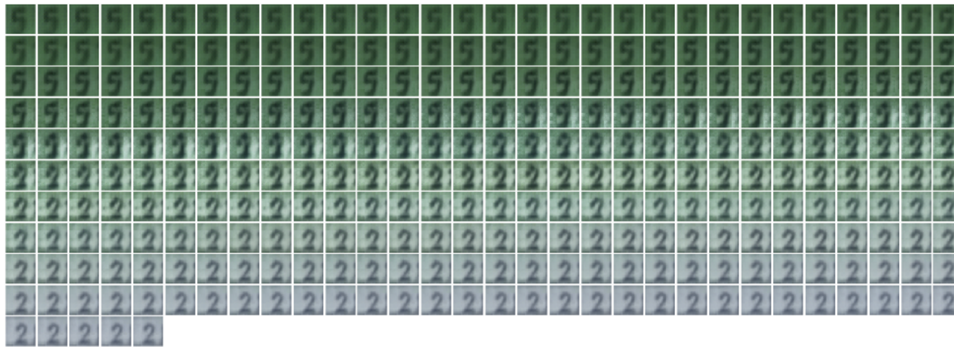
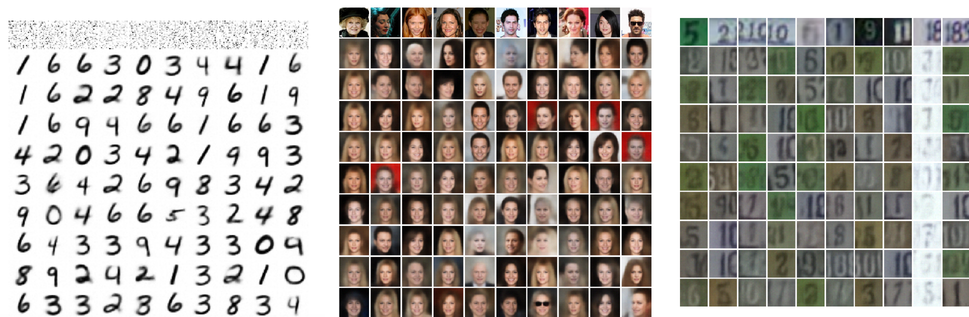


Figure 6. Latent space interpolation of 5 → 2 for SVHN, 500 bits at noise=0.1

Figure 7. Latent space interpolation for celebA, 1000 bits at noise=0.1



(a) MNIST, initialized from random noise (b) celebA, initialized from data (c) SVHN, initialized from data

Figure 8. Markov chain image generation after 9000 timesteps, sampled per 1000 steps