A. M2T - Appendix

A.1. Low discrepancy sequences

We reproduce the definition of discrepancy from Kuipers and Niederreiter [23, p.88].

Definition 1. *The* discrepancy D *of a sequence of numbers* $X = x_1, \dots, x_N$ *is*

$$D(X) = \sup_{0 \le \alpha < \beta \le 1} \left| \frac{A([\alpha, \beta); X)}{N} - (\beta - \alpha), \right| \quad (3)$$

where $A(I; X)$ *is the number of points in* X *that fall into the interval* I.

Intuitively, this measures how "dense" the densest region in X is compared to how many points we have overall. For example, a completely uniformly distributed X with N points has discrepancy $1/N$.

Definition 2. *A* low-discrepancy sequence (LDS) *is a sequence where every subsequence has low discrepancy.*

We use a LDS proposed by **Roberts [30]**, with the following construction of the i-th point $\boldsymbol{x}_i$:

$$\boldsymbol{x}_i = i\boldsymbol{\phi} \bmod [1, 1], \text{where } \boldsymbol{\phi} = [1/\rho, 1/\rho^2]. \quad (4)$$

We refer to [30] for details, but to give some intuition: in the above equation, the important property of $\boldsymbol{\phi}$ is that applying it repeatedly modulo 1 covers the space. Contrast, *e.g.*, with $\boldsymbol{\phi} = [1, 1]$, which would always map to the same point.

A.1.1 Quantizing LDS

We note that this sequence is constructed for the unit cube, and we have to quantize it to use it for our representations. We do this by constructing the quantized LDS $\hat{\boldsymbol{x}}_i$ by scaling up and rounding the underlying $\boldsymbol{x}_i$ to the integer grid $\{0, \dots, w_T - 1\} \times \{0, \dots, w_T - 1\}$ and skipping over values of i that are already used (since quantizing will lead to some values getting sampled more than once), *i.e.*, we find the smallest integer K such that quantizing $\boldsymbol{x}_1, \dots, \boldsymbol{x}_K$ covers the $w_T \times w_T$ grid (which is $K = 1381$ for $w_T = 24$).

A.1.2 Comparison to other LDS

We explore the well known sequences by **Sobol** and **Halton** (via `scipy.stats.qmc` [36]). These are defined in $[0, 1) \times [0, 1)$, so we quantize them to a $H \times W$ grid (Sec. A.1.1) We visualize these sequences for $S = 8, \alpha = 2.2$ in Fig. 9 a). We evaluate them in our models (Fig. 9 b) and see that both Sobol and Halton lead to good bits-per-pixel (bpp), but the schedule by Roberts (used in the paper) has a slight advantage. We will add this comparison to the paper.

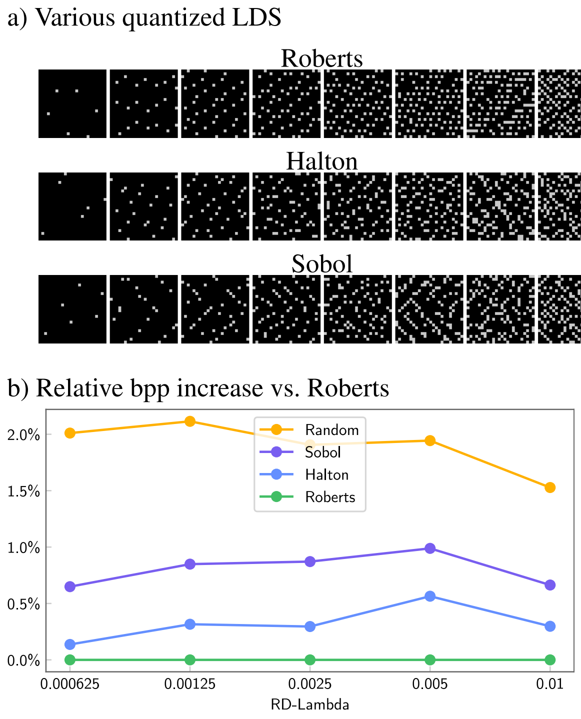

Figure 9: Comparing to Halton and Sobol.

A.2. Additional Accelerators

We show runtime on all accelerators in Fig. 10.

A.3. Implementation Details

We use the ELIC architecture for the autoencoder as detailed in [18, Supplementary Material Sec 1.1, Table 1], where we use $M = 256, N = 192$.

We use the `AdamW` optimizer from Tensorflow Addons, using `weight_decay`=0.03 · learning rate, β_1=0.9, β_2=0.98, ϵ=1E^{-9}, `global_clipnorm`=1.[4]

To improve the training stability of the GMM, we found it important to adopt the Laplace tail mass approach [7] (with weight 0.001), falling back to to the (more numerically stable) Laplacian distribution when the probability predicted by GMM vanishes.

Previous works (*e.g.* [27, 28]) center the representation with the mean prediction of the entropy model during quantization, which couples entropy modelling and the reconstruction. For our models, this poses a problem, since during training, we do not know the mean before doing one pass through the model, so we would have to double the forward passes to do this. To simplify the setup, we simply drop this. As an important benefit, dropping the centering also allows us to employ a patched inference scheme

[4] https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/AdamW

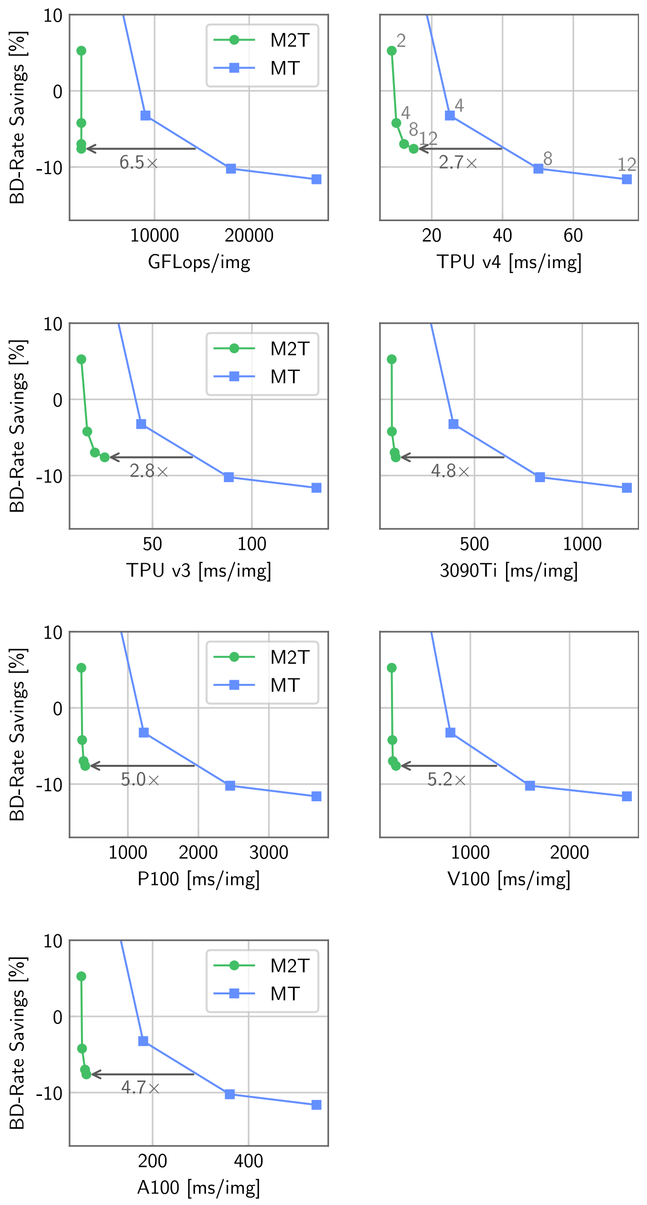

Figure 10: Speed on all tested platforms as well as FLOPS.

described next without creating block artifacts in the reconstruction.

A.4. Algorithms

We show pseudo-code for how our masked transformers can be used on the sender side in neural compression in Als 1 2.

A.5. Samples

We show samples from the entropy model in Fig. 11. For Fig. 7 in the main text, we take the average over 50 samples per step.

A.6. Flax Implementation

We show the flax implementation of M2T in Fig. 12.

Algorithm 1 MT Sender

```
Require: Input $y$ of shape $(b', w_T^2, c)$, seed $s$.
  $c \leftarrow ones(S, C) \cdot$ mask token                ▷ Current Input
  masker $\leftarrow$ make_masker(seed=$s$)
  for $i \in \{1, \ldots,$ masker.num_steps$\}$ do
      params $\leftarrow$ transformer($c$)
      $m \leftarrow$ masker.get_mask(params)
      bit_stream $\leftarrow$ bitencode($y[:, m, :]$, params$[:, m, :]$)
      $c \leftarrow y[:, m, :]$                              ▷ Uncover input
  end for
```

Algorithm 2 M2T Sender

```
Require: Input $y$ of shape $(b', P^2, C)$, seed $s$.
  masker $\leftarrow$ make_masker(seed=$s$)
  $y_{\text{perm}} \leftarrow$ permute(masker, $y$)
  t $\leftarrow$ 0                                          ▷ Total Uncovered
  $c \leftarrow []$                                         ▷ Current Input
  for $i \in \{1, \ldots,$ masker.num_steps$\}$ do
      $m_{\text{len}} \leftarrow$ len(masker.get_mask(params))
      $c$.extend(ones(b, $m_{\text{len}}$, C) $\cdot$ mask_token
      params, cache $\leftarrow$ transformer($c$, cache)
      bit_stream $\leftarrow$ bitencode(
          $y_{\text{perm}}[:, t : t + m_{\text{len}}, :]$, params)
      $c$.extend($y_{\text{perm}}[:, t : t + m_{\text{len}}, :]$)        ▷ Uncover input
  end for
```

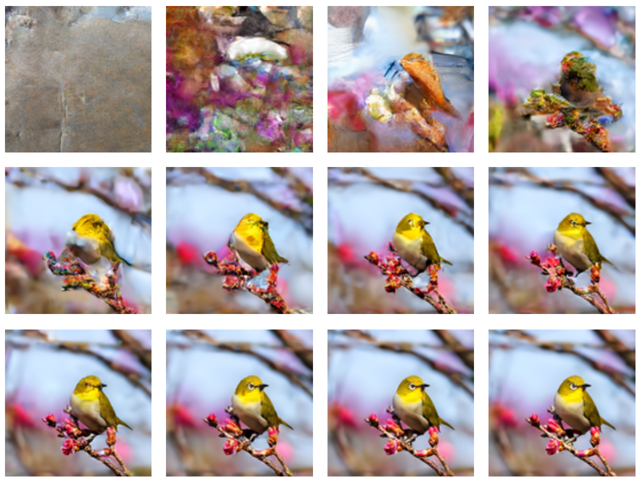

Figure 11: Raw samples.

```
import functools
from typing import Optional
from flax.linen import attention
from flax.linen.linear import DenseGeneral
from flax.linen.module import compact, merge_param
from jax import lax
import jax.numpy as jnp

# Forked from flax.
class PermutedMHA(attention.MultiHeadDotProductAttention):
  @compact
  def __call__(
      self, inputs_q, inputs_kv, mask=None, deterministic: Optional[bool] = None):
    if self.dropout_rate > 0.0:
      raise NotImplementedError('Dropout removed for simplicity.')
    if len(mask.shape) != 2:
      raise NotImplementedError('Only supporting 2D masks at the moment!')
    features = self.out_features or inputs_q.shape[-1]
    qkv_features = self.qkv_features or inputs_q.shape[-1]
    assert qkv_features % self.num_heads == 0
    head_dim = qkv_features // self.num_heads
    dense = functools.partial(
        DenseGeneral, axis=-1, dtype=self.dtype, param_dtype=self.param_dtype,
        features=(self.num_heads, head_dim), kernel_init=self.kernel_init, bias_init=self.bias_init,
        use_bias=self.use_bias, precision=self.precision)
    query, key, value = (
        dense(name='query')(inputs_q), dense(name='key')(inputs_kv), dense(name='value')(inputs_kv))
    if self.decode:
      assert mask is not None
      is_initialized = self.has_variable('cache', 'cached_key')
      cached_key = self.variable('cache', 'cached_key', jnp.zeros, key.shape, key.dtype)
      cached_value = self.variable('cache', 'cached_value', jnp.zeros, value.shape, value.dtype)
      cache_index = self.variable('cache', 'cache_index', lambda: jnp.array(0, dtype=jnp.int32))
      if is_initialized:
        *batch_dims, max_length, num_heads, depth_per_head = cached_key.value.shape
        *_, count, query_num_heads, query_depth_per_head = query.shape
        if query_num_heads != num_heads or query_depth_per_head != depth_per_head:
          raise ValueError("Invalid dimensions")
        cur_index = cache_index.value
        start_indices = (0,) * len(batch_dims) + (cur_index, 0, 0)
        key = lax.dynamic_update_slice(cached_key.value, key, start_indices)
        value = lax.dynamic_update_slice(cached_value.value, value, start_indices)
        cached_key.value = key
        cached_value.value = value
        # We just slice the mask, since it's q_length, kv_length, and we slide through Q.
        mask = lax.dynamic_slice(mask, (cur_index, 0), (count, max_length))
        cache_index.value = cache_index.value + count
    x = self.attention_fn(
        query, key, value,
        mask=mask, dropout_rng=None, dropout_rate=self.dropout_rate,
        broadcast_dropout=self.broadcast_dropout,
        deterministic=True, dtype=self.dtype, precision=self.precision)
    return DenseGeneral(
        features=features, axis=(-2, -1), kernel_init=self.kernel_init,
        bias_init=self.bias_init, use_bias=self.use_bias, dtype=self.dtype,
        param_dtype=self.param_dtype, precision=self.precision, name='out')(x)
```

Figure 12: Flax attention implementation for M2T.

A.7. Raw Data

We provide the raw data for the figures of the main text in Table 2, 3.

MT		M2T	
0	1	2	3
0.058108	27.079653	0.059169	27.034135
0.094242	28.468065	0.097261	28.408479
0.153969	29.985175	0.162184	29.978115
0.247314	31.652337	0.257729	31.644149
0.380635	33.393239	0.385378	33.372010

Table 2: Raw data for Fig. 1 (Rate-distorion on Kodak).

MT		M2T	
0	1	2	3
0.043276	29.986596	0.043651	29.888452
0.067407	31.379083	0.069709	31.270813
0.105127	32.825032	0.110558	32.813387
0.159596	34.256665	0.165478	34.246637
0.234286	35.674083	0.235964	35.657870

Table 3: Raw data for Fig. 8 (Rate-distortion on CLIC)