7. Supplementary Material

A. Images used in figures

Video used in Figures 1 and 2 by Ambrose Productions, and Figure 4 by TravelTip. Both [CC BY-SA 3.0 https://creativecommons.org/licenses/by/3.0/legalcode], via YouTube.

B. Architectural details

In this section we detail the architecture and hyperparameters of the autoencoder and code-model that we use in our experiments.

B.1. Autoencoder

As decribed in section 4.2, we design our autoencoder by extending the (2D) model of [28] to use 3D convolutions. The exact model is depicted in Figure 9.

B.1.1 Quantization

As explained in Section 4.2, the encoder network outputs continuous latent variables $\tilde{\mathbf{z}} \in \mathbb{R}^{B\times T\times K\times H/s\times W/s}$, which are then quantized using a learned codebook $\mathcal{C} = \{c_1, \ldots, c_L\}$.

Quantization involves computing $\mathbf{q}_{\mathbf{z}|\mathbf{x}} \in \mathbb{R}^{B\times T\times K\times H/s\times W/s\times L}$ that defines the probability for each codebook center i at each position j in $\mathbf{z}$ (note that this is one-hot over the L axis). We assume independence between all elements of $\mathbf{z}$ given $\mathbf{x}$, and we use the codebook distance to compute $\mathbf{q}_{\mathbf{z}|\mathbf{x}}$:

$$q^{ij}_{\mathbf{z}|\mathbf{x}} = q(z_j = i|\mathbf{x}) = \frac{e^{-\tau|\tilde{z}_j - c_i|}}{\sum_{k=1}^{L} e^{-\tau|\tilde{z}_j - c_k|}} \tag{5}$$

Where for notational simplicity, we are using a single index j to index over all (B, T, K, H, W) dimensions of $\tilde{\mathbf{z}}$.

Note that as $\tau \to \infty$, $q(z_j = i|\mathbf{x})$ will put more and more probability mass on a single (the closest) center and will eventually be deterministic. This is desirable, as we want a deterministic encoder. In practice, we use a $\tau = 10^7$ which we observe to always give us one-hot vectors for 32-bit precision floats. In the backward pass, we use the gradient of a softmax with a $\tau = 1$ for numerical stability.

On the decoder side, the one-hot probabilities $\mathbf{q}_{\mathbf{z}|\mathbf{x}}$ are embedded using the same codebook $\mathcal{C}$ to obtain the scalar tensor $\hat{\mathbf{z}}$ approximating $\tilde{\mathbf{z}}$ that is then decoded to predict $\mathbf{x}$.

B.2. Autoregressive Code-Model

The code model takes as input the one-hot probability tensor $\mathbf{q}_{\mathbf{z}|\mathbf{x}}$ output by the encoder, and predicts the probability for each entry j in $\mathbf{z}$ in an autoregressive manner:

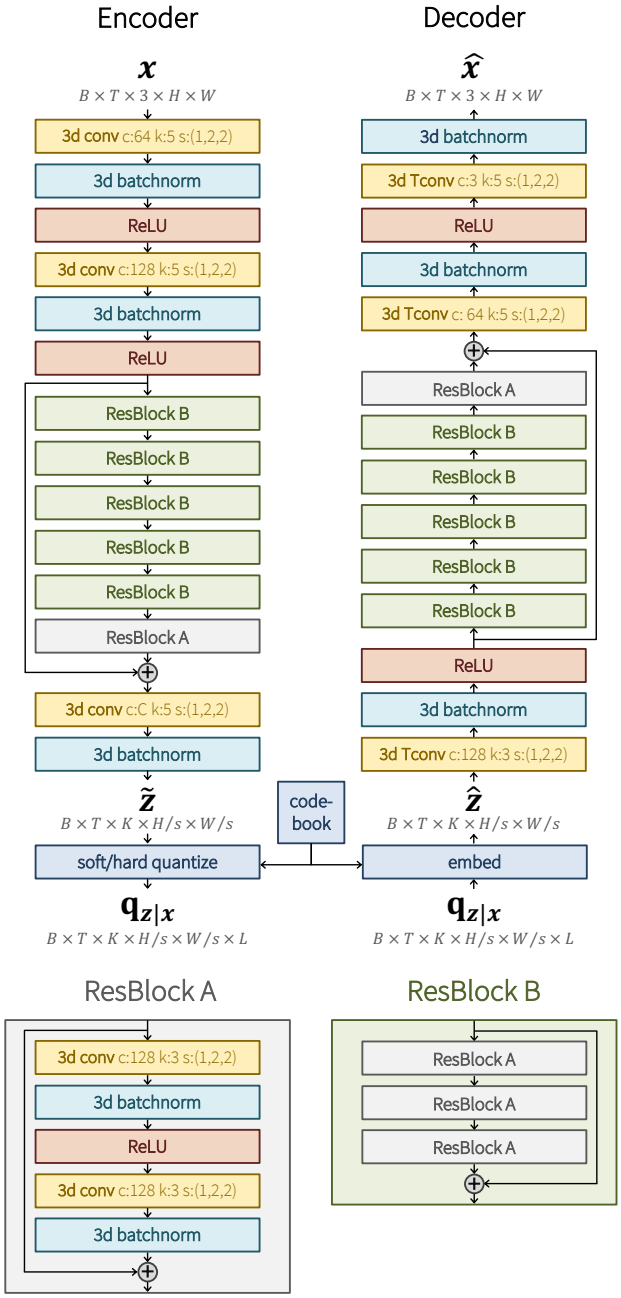

Figure 9: Architecture of our autoencoder. *Tconv* denotes transposed convolution. For (transposed) convolutional layers c denotes the number of output channels, k denotes the kernel size and s denotes the stride. These are either expressed as (x, y, z) triplets or as a single number that is used for each dimension. "Same"-padding is used for all (transposed) convolution layers. $\mathbf{q}_{\mathbf{z}|\mathbf{x}}$ refers to the tensor of one-hot probabilities $q(z_j = i|\mathbf{x})$.

$$p^{ij}_{\mathbf{z}}(\mathbf{q}_{\mathbf{z}|\mathbf{x}}) = p(z_j = i|\mathbf{z}_{<j}) \tag{6}$$

We use a 4 layer PixelCNN [37] architecture with a kernel size of 5x5 and 8 hidden channels ($h = 8$). We embed the one-hot probabilities of $\mathbf{q}_{\mathbf{z}|\mathbf{x}}$ using a learnable scalar embedding. We experimented with using the encoder codebook as the prior embedding, but we found that it did not make any difference in performance in practice.

B.2.1 Conditioning

For the frame-conditioned and GRU-conditioned model (see Figure 3), we inject the conditioning variable into each of the autoregressive blocks, right before applying the gated nonlinearity.

This conditioning input is a featuremap, and its number of channels should match the number of channels in the ARMBlock. The gated nonlinearity requires two times the number of hidden channels h. As there is a nonlinearity for both the horizontal and vertical stack, this would require $4h$ channels. Since the filters in PixelCNN are fully connected along the channel dimension, the required number of output channels for the conditioning featuremap is $(4h)K$.

We use a (conventional) convolutional layer to preprocess the conditioning input and to upsample the number of channels to match the size of each autoregressive block in the PixelCNN. The prior architecture is depicted in Figure 10.

B.2.2 Encoder and Code-Model gradients

During training, the code model is updated to minimize the rate loss $\mathcal{L}_{\text{rate}} = \text{CE}[q(\mathbf{z}|\mathbf{x}), p(\mathbf{z})]$. The rate loss is a sum over the elementwise cross-entropy between q and p (which is summed over each class i of the codebook and each element j of $\mathbf{z}$).

$$\mathcal{L}_{\text{rate}} = -\sum_{\mathbf{z}} q(\mathbf{z}|\mathbf{x}) \log p(\mathbf{z}) \tag{7}$$

$$= -\sum_{ij} q^{ij}_{\mathbf{z}|\mathbf{x}} \log p^{ij}_{\mathbf{z}}(\mathbf{q}_{\mathbf{z}|\mathbf{x}}) = \sum_{ij} \mathcal{L}^{ij}_{\text{rate}} \tag{8}$$

Note that unlike [28] we do not do any detaching of the gradient. As a result, the derivative of the rate loss w.r.t. the encoder parameters θ_q is affected by the code-model in the following way:

$$\frac{\partial \mathcal{L}^{ij}_{\text{rate}}}{\partial \theta_q} = -\frac{\partial q^{ij}_{\mathbf{z}|\mathbf{x}}}{\partial \theta_q} \left(\frac{q^{ij}_{\mathbf{z}|\mathbf{x}}}{p^{ij}_{\mathbf{z}}(\mathbf{q}_{\mathbf{z}|\mathbf{x}})} \frac{\partial p^{ij}_{\mathbf{z}}(\mathbf{q}_{\mathbf{z}|\mathbf{x}})}{\partial q^{ij}_{\mathbf{z}|\mathbf{x}}} + \log p^{ij}_{\mathbf{z}}(\mathbf{q}_{\mathbf{z}|\mathbf{x}}) \right) \tag{9}$$

Thus, the encoder is trained not only to minimize the distortion, but also to minimize the rate (e.g. to predict latents that are easily predictable by the code-model). This can also been seen by reversing the paths of the forward arrows in Figure 2.

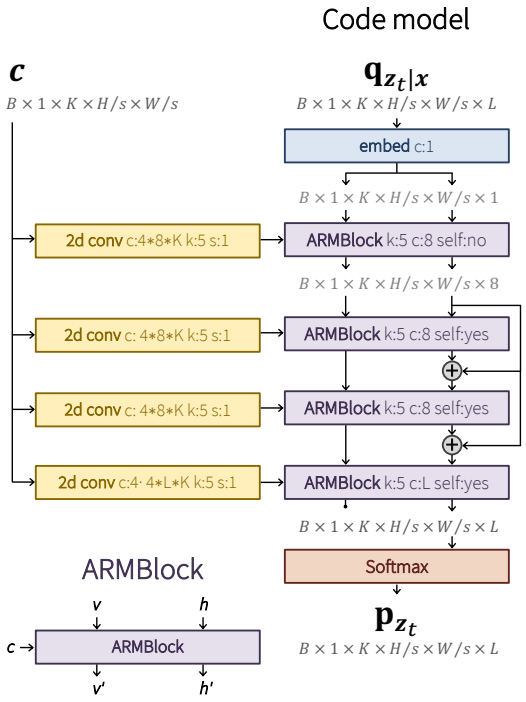

Figure 10: Architecture of our prior / code-model. ARMBlock refers to a block in PixelCNN [37] with horizontal stack h, vertical stack v and conditioning input c (Figure 2 of [37]). $\mathbf{c}$ represents the conditioning input that is used in our frame-conditioned and GRU-conditioned code-model.

C. Evaluation procedure

C.1. Traditional codec baselines

We use FFMPEG[1] 2.8.15-0 to obtain the performance for the H.264/AVC and H.265/HEVC baselines. We use the default settings unless reported otherwise.

C.2. Data preprocessing

We build our evaluation datasets by extracting the png frames from the raw source videos using FFMPEG. Because some videos are in yuv colorspace, the conversion to rgb could in theory lead to some distortion, though this is imperceptible in practice. We use the same dataloading pipeline to evaluate our neural networks and the FFMPEG baselines as to avoid differences in ground-truth data.

C.3. Rate-Distortion

For the FFMPEG baselines, we divide the total filesize by the total number of pixels to obtain bpp. For our neural network, we use the rate loss (converted into bpp) as a

[1] https://ffmpeg.org/

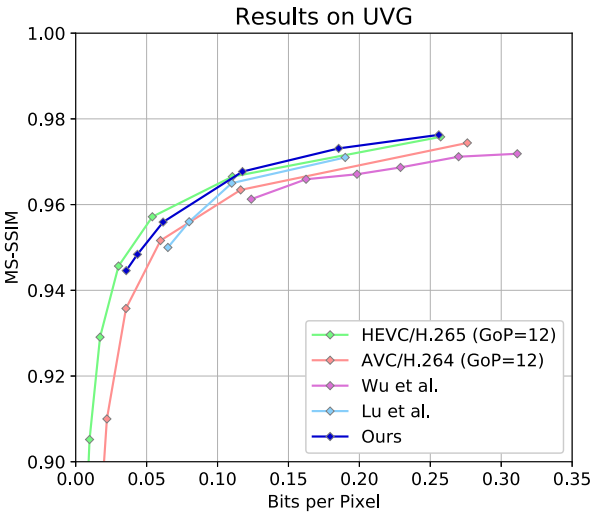

Figure 11: Rate/distortion results on UVG for classical and learned compression methods. H.264 and H.265 results were obtained with restricted FFMPEG settings (Group of Pictures set to 12).

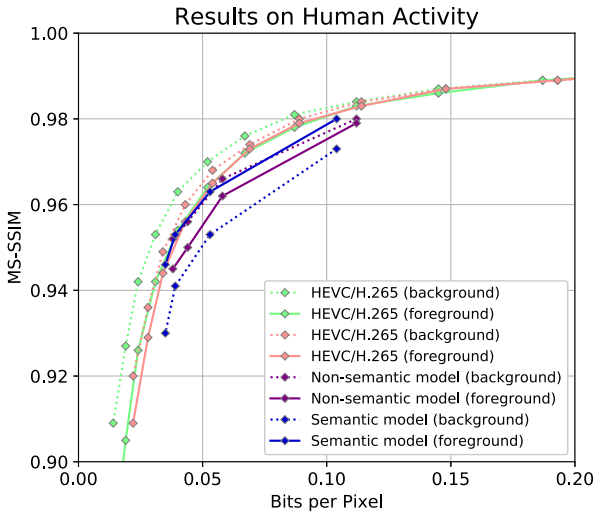

Figure 12: Semantic compression. Background is easier for all models, HEVC, AVC, and non-semantic model, except for our learned semantic compresison.

proxy for rate. By definition, the rate loss gives the expected bitrate under adaptive arithmetic coding, and expected bpp was shown to be highly correlated to actual bpp [27].

We calculate MS-SSIM [39] using our own implementation which we benchmarked against the implementation in tensorflow[2]. We use the same power factors that are initially proposed in [39].

D. Additional results

D.1. Comparison to other methods

When comparing neural networks to traditional codecs, it is common practice to evaluate those codecs under restrictive settings. For example, group of pictures (GoP) is often set to a value that is similar to the number of frames used to evaluate the neural networks [40, 27]. Furthermore, encoding `preset` will be set to `fast` (which will result in worse compression performance) [40, 27]. In our evaluation (presented in Figure 6) we instead use the FFMPEG default values of `GoP=25` and `preset=medium`.

In Figure 11 we compare our end-to-end method to other learned compression methods and use baseline codecs with the restrictive setting of `GoP=12`, which is used in [40, 27]. The figure shows that our model has a better rate-distortion performance than H.265/HEVC under these restrictive settings for bitrates higher than 1.2 bpp.

D.2. Semantic Compression

The results for semantic compression are reported in Figure 7a. To avoid clutter, background performance for H.264/AVC and H.265/HEVC are omitted there. Figure 12 shows the full results including background performance for traditional codecs.

D.3. Adaptive Compression

Quantitative rate-distortion performance for adaptive compression is reported in Figure 7b. In Figure 13 we show a qualitative sample. Notice the clear block artifacts that can be observed around the road markings for H.265/HEVC. For our generic model, we do not observe such articacts, though we can see that edges are somewhat blurry around the line markings. In our adapted model, the road markings are significantly improved.

We note that the expanding perspective motion observed in road-driving footage is a great example of a predictable pattern in the data that a neural network could learn to exploit, while it would be difficult to manually engineer algorithms that use these patterns.

[2] https://www.tensorflow.org/versions/r1.13/api_docs/python/tf/image/ssim_multiscale

(a) HEVC/H.265 (0.025 BPP)

(b) Generic model (0.030 BPP)

(c) Adapted model (0.025 BPP)

Figure 13: Qualitative results for adaptive compression.