College of Engineering

ELEC 491 – Electrical Engineering Design Project

Final Report

Flexible-Rate Bidirectional Video Compression with Motion Refinement

Participant Information:

Eren Çetin – 64277

Project Advisor(s)
Ahmet Murat Tekalp
Ogün Kırmemiş

19.01.2022

Abstract

Video content has been becoming more prevalent in the last decade by capturing 80% of the online data. As this is the case, more efficient video compression methods are helpful to provide a better service for people who stream online video content with a limited bandwidth. For that reason, our aim is to develop a learned bidirectional video compression framework that achieves superior or competitive rate-distortion performance compared to other works in the literature of video compression. To achieve the desired results, we employ additional modules such as bidirectional motion prediction, motion refinement, learned frame fusion and achieve flexible bitrate using a single model with learned quantization parameters. Testing our network on UVG dataset, a common benchmark, we achieve competitive or superior results at high bitrates when we compare our results with other learned video compression networks such as DVC [1], Scale-Space Flow [2], RLVC [3], and LHBDC [4] in terms of PSNR and MS-SSIM scores. On the other hand, the model achieves slightly worse rate-distortion performance at low bitrates compared to LHBDC [4] and the traditional SVT-HEVC codec at very slow preset in terms of PSNR and MS-SSIM. In the following report, further details of the proposed network are provided extensively with visual results that demonstrate the effect of main modules.

Table of Contents

1. Introduction

1.1. Concept

Compression is an important part of our life, allowing us to use internet and many other data sources that would be impossible to reach if we had to store or download raw data without compression. Thanks to Claude Shannon, information theory has brought the age of information with his celebrated paper defining the information in 1948. He defined the information in terms of "bits" and provided entropy as a quantitative tool to measure the amount of information. That way, information can be thought as an increasing function of uncertainty that relies in a signal.

In addition to defining the information in terms of "bits", he demonstrated that there exists a maximum rate of transmission over a channel, which we call the bandwidth of a channel. Accordingly, it is possible to transmit error-free signals as long as transmission rate is less than the channel capacity. Although inventions in communication have been allowing to reach closer to the theoretical limit, the available bandwidth has always been a bottleneck in information transfer, limiting many applications including video livestreaming or playing online games.

As this limit cannot be overcome, one possible solution is compressing the information that is intended to be transmitted. However, Shannon has once again demonstrated a limit for the compression rate that cannot be overcome. This lower limit is the entropy of the signal of interest. Since this limit cannot be exceeded for lossless compression, the aim in applications of lossless image compression and video compression is to reach as close as possible to the lower bound.

In contrast to lossless compression, the human visual system is less sensitive to high frequency content which makes it logical to consider lossy compression as a favorable option to get rid of high frequency content on an image or video. That way, the lower bound can be eliminated while the distortion on the image or video is minimized with respect to human perception. Because it allows a greater compression rate and minimizes the distortion on images, lossy compression is widely adopted in today's applications that do not require a perfectly reconstructed image. Today, almost all image and video content on the internet are examples of lossy compression as lossless compression is not necessary in many applications such as social media and video streaming.

As the raw data consists of redundancies such as temporal or spatial redundancies, certain transform operations can reduce the entropy of data and allow a more efficient compression with entropy coding. Since the data after transform must be quantized for compression, the quantization prevents the one-to-one mapping between encoding and decoding; thus, induces loss of information during lossy compression. For the rest of this report, we will be only concerned with lossy

compression as it has a wider field of application for image and video compression.

1.2. Image Compression

Image compression is an important step towards video compression as videos are composed of many temporally linked images. Image compression is possible because of the spatial correlation between individual pixels on an image. Consider a scene where the picture is divided between sky and ground. As the sky would be mostly blue and have less change in a close neighborhood, it might be enough to send very few symbols for a large region. On the other hand, in images which contain highly non-uniform shapes and structures (such as a city scene with many color variations), the correlation between pixels is reduced, thus, we require more bits to build the image back.

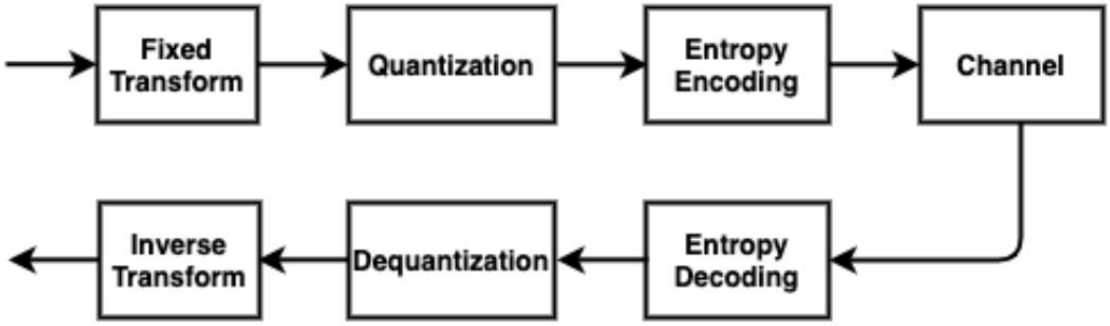

Figure 1. The framework of lossy image compression.

Considering the framework for the lossy image compression as described in Figure 1, we initially map the pixel domain representation of our image into a latent representation. The operation provides a one-to-one mapping; however, the quantization step causes information loss as the operation is not reversible. As the quantization steps become larger, the amount of distortion in our decoded image also increases. After the image is quantized, an entropy coder is used to compress the latent representation in a lossless manner. Subsequently, the decoder decodes the latent representation, dequantizes it and transforms it back into pixel domain.

1.3. Video Compression

Today, over 80% of the data on the internet is composed of video data [5]. This percentage is only expected to increase with newer technologies and more demand for entertainment through tech companies such as Netflix, Google, and Amazon. For that reason, the research for more efficient video compression methods has been in rise to transmit video content with less lagging and store it in a smaller space.

In addition to the spatial correlation within the frames of a video, the frames have a high

correlation as videos mostly do not have interruptions and have a smooth transition between frames. For that reason, the video compression techniques also make use of the temporal correlation by applying inter prediction techniques. To exploit the temporal correlation, video compression models utilize motion estimation and compensation components as the motion information connects two consequent frames to each other.

The importance of research on learned video compression comes from the capability of performing nonlinear transforms with deep neural networks. Although traditional video codecs are still in employment, they can only perform linear transforms that can fall short for decorrelation of information and entropy reduction. Furthermore, the combinatorial nature of video codecs causes a great problem for defining the optimal video codec. As learned video codecs can be optimized in an end-to-end manner with a single rate-distortion loss, the optimality is less of a concern.

The problem that we have to address in video compression is a joint optimization problem which considers several components of an autoencoder architecture. Autoencoder is composed of an encoder and a decoder which has similar tasks as the traditional video codecs. While the encoder transforms the pixel domain image representation into a latent representation with lower entropy, the decoder transforms it back to the pixel domain representation while keeping the quality as high as possible.

For video compression, we encode and decode the motion vectors and the residual frames which allow us to make use of the temporal correlation and reduce the entropy compared to single frame image compression. To perform this operation, we utilize two separate autoencoders to compress the motion vectors and the residual frames. In addition to the encoder and the decoder, the entropy coding requires accurate estimation of entropy parameters in order to reduce the bitrate as much as possible. Thus, each autoencoder architecture requires a prior network to estimate the mean and scale of the latent representations. The bitrate is estimated with the entropy of our latent representations as entropy coding is capable of achieving close to minimum bound bitrates. As the coding performance of our network depends on the rate-distortion performance, we optimize the whole model using an end-to-end approach with a single loss function and simultaneous optimization of all components.

1.4. Objectives

The objective of this project is to build a learned bidirectional video compression network that can be optimized in an end-to-end manner and can achieve a competitive rate-distortion performance compared to other works in the literature. As unidirectional video compression yields a worse rate-distortion performance both in case of traditional codecs and learned codecs, our aim is to build a bidirectional video compression network instead of a unidirectional compression network that can

also achieve different bitrates by using a single network for all levels. That way we aim to reduce the training cost and achieve a better generalization.

Furthermore, a major aim of this project is to improve over the bidirectional video compression framework proposed by Yilmaz and Tekalp [4] with the use of special modules such as motion prediction and motion refinement modules. In addition to these modules, we aim to achieve arbitrary rate-distortion trade-offs using the gain unit proposed by Cui et al. [6] for image compression.

2. Related Work

Our model builds on the previous work that was presented in both image compression and video compression domains. In following sections, the details of these prior works are explained while comparing them with our proposed network.

2.1. Joint Autoregressive and Hierarchical Priors for Learned Image Compression

For image compression, one important work is provided by Minnen et al. [7]. With their work, Minnen et al. [7] propose a model that brings an improvement to the learned image compression domain by modeling the probability distribution of image latent representations with a mean and a scale value. Since the probability modeling has a high importance for entropy coding, the model can lower the bitrate since the distribution can be better estimated.

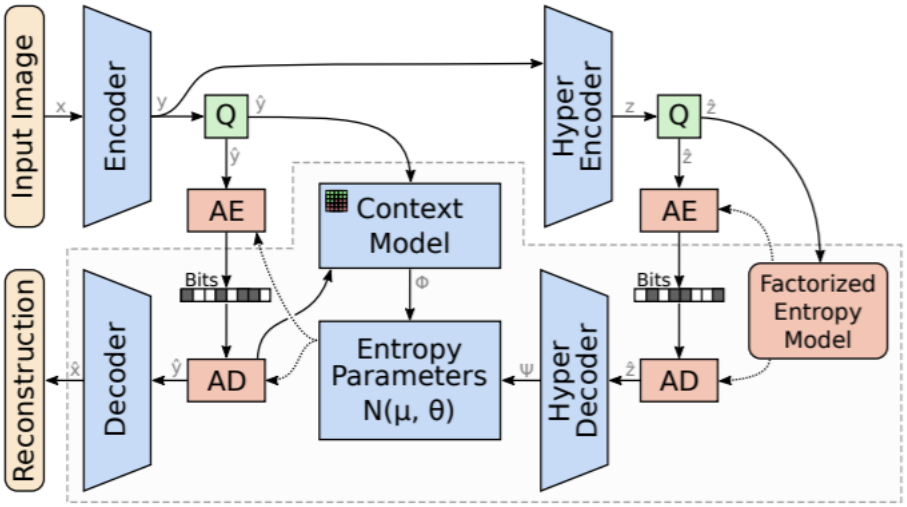

Figure 2. The architecture of "Joint Autoregressive and Hierarchical Priors" network proposed by Minnen et al. [7]

Furthermore, as depicted in Figure 2, the model proposes a causal context model to exploit the spatial correlation further. That way, the entropy parameters (mean and scale) are estimated better to encode the latent representation with a more precise probability estimation.

As the model provides well estimation of entropy parameters, the architecture is utilized in our

model for the compression of motion vectors and residual frames thinking as if they were images that are compressed with this model. Thus, this network has an important place in our proposed video compression network.

2.2. Deep Video Compression Framework (DVC)

DVC model [1] is a pillar stone in the literature of end-to-end optimized deep video compression networks. The model uses unidirectional motion vectors to use the temporal correlation between frames and provide a low delay learned video codec. These motion vectors are composed of two channels where the channels represent the shift in a pixel in horizontal direction and vertical direction.

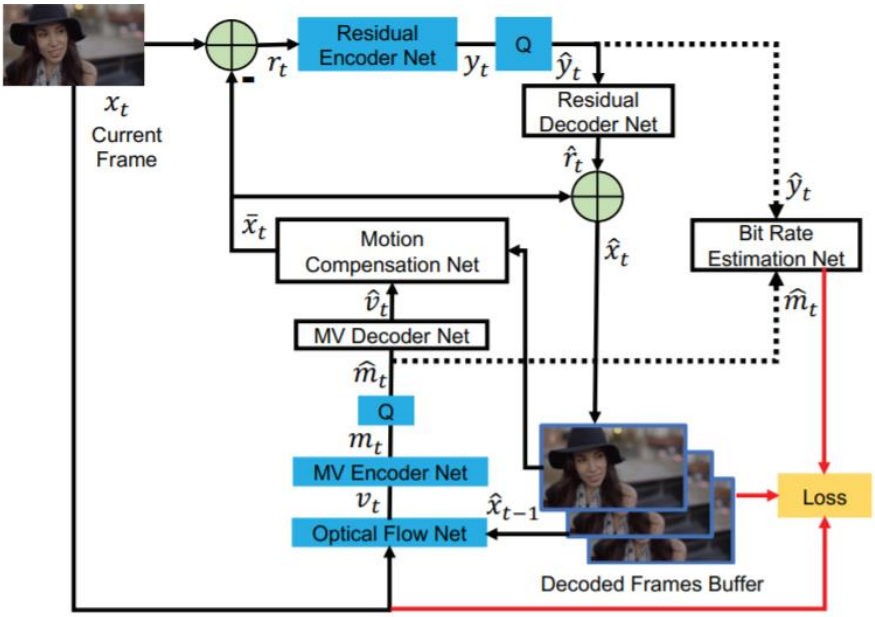

Figure 3. The architecture of DVC network proposed by Lu et al. [1]

Same as our model, the initial keyframes of the group of pictures is coded using a learned image compression network while the rest of the frames of the group of pictures are compressed using the network provided in Figure 3. First of all, the motion vectors are calculated using a learned motion estimation network and the motion vectors are encoded into a latent representation. The latent representation is quantized and passed to the decoder network that reverts the latent representation back into motion vectors that were estimated by the encoder. Later on, a motion compensation network performs bilinear warping on the last reference frame that is available to decoder with the estimated and transmitted motion vectors. The warped frames are also processed by a motion compensation network that aims to reduce the warping artifacts. That way, the motion compensated frame is acquired. Finally, the residual frame is calculated by subtracting the ground truth frame from the motion compensated frame and compressed with a similar architecture as the motion compression network. The residual frame is added back to the motion compensated frame at the decoder side and

the output frame is achieved. After compressing rest of the frames using the proposed network, the loss is calculated over all frames where the loss is the rate-distortion loss.

2.3. Scale-Space Flow (SSF)

Similar to DVC [1], the Scale-Space Flow (SSF) model proposed by Agustsson et al. [2] separates the motion and residual information and encodes them in separate autoencoder architectures. However, the back warping operation that is utilized by the DVC [1] network yields motion compensation artifacts that reduce the frame quality. This effect was avoided by employing a motion compensation network to reduce the artifacts. With SSF model [2], the back warping operation is replaced with scale-space warping. Scale-space warping adds a third channel to the motion vectors called the scale channel and allows this third channel to resemble the uncertainty that is present in difficult to predict areas of the frame. Using this channel, the scale-space warping operation blurs the regions where the motion compensation would yield worse artifacts and blur these regions in order to improve the frame quality while also reducing the entropy in the residual frames.

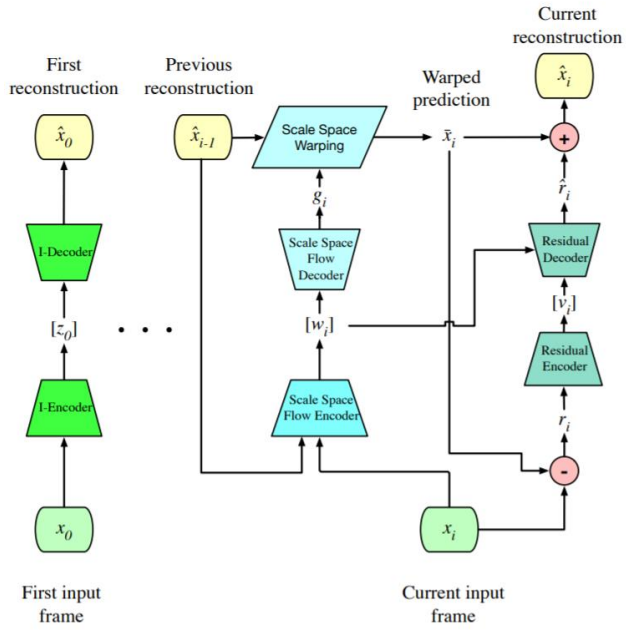

Figure 4. The architecture of the Scale-Space Flow network proposed by Agustsson et al. [2]

As depicted in Figure 4, the compression framework starts by coding the first frame of the group of pictures using keyframe compression (I-compression) as a reference frame. Afterwards, the scale-space flow autoencoder network takes the reference frame and the current input frame to construct a scale-space flow by compressing the latent representations. Using the scale-space flow warping, the reference frame is warped to acquire the motion compensated (warped) prediction frame and the residual frame. Finally, the residual frame is compressed in a separate autoencoder network and the reconstructed residual frame is added back to the warped prediction to achieve the reconstructed

current frame.

The framework is similar to our network since the network does not compress the vectors that contain the motion vectors but constructs the motion vectors using an autoencoder architecture after giving reference and current frames as input.

2.4. Asymmetric Gained Deep Image Compression with Continuous Rate Adaptation

The work of Cui et al. [6] does not provide a complete architecture for video or image compression, however they demonstrate how usage of gain and inverse gain units can help in achieving continuous rate-distortion curves and avoid training multiple networks.

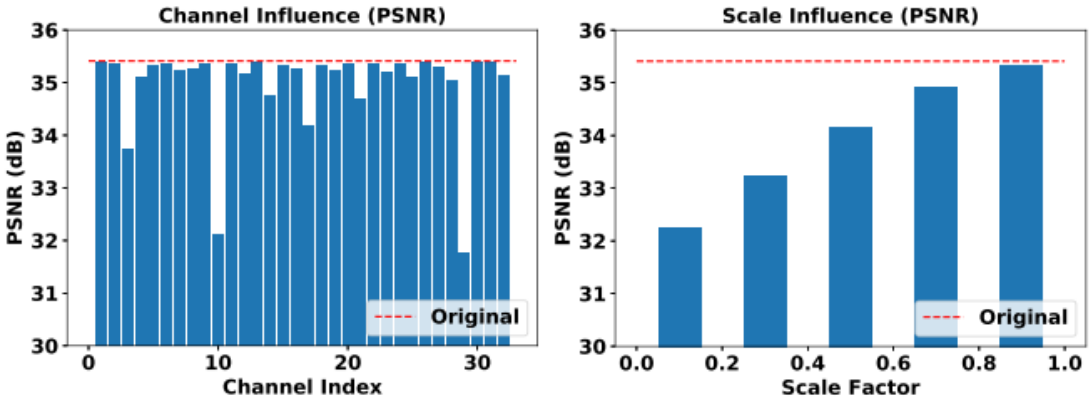

Figure 5. The difference between channel influences on quality of reconstructed frame [6].

Since the latent representation of a frame or motion vector has to be quantized before entropy coding, the quantization bin size has to be deduced. Instead of training separate networks to learn the quantization bin size inherently through the convolution layers, Cui et al. [6] propose learning channel-wise quantization parameters for different bitrate levels. They demonstrate that separate channels have varying relative importance on frame quality in Figure 5. Thus, they propose that we can scale the channels with different parameters before the quantization step and learn the scale parameters during the training.

The gain and inverse gain units are used to scale the latent representation by multiplying each channel with a different parameter for the respective bitrate level and change the quantization bin size effectively. The scaling parameters of the gain and inverse gain units are paired with each bitrate level and rate-distortion trade-off value. After achieving a latent representation from the encoder, the latent representation is multiplied by the channel-wise scaling vector (gain vector) of the gain module. Then, the multiplied latent representation is quantized and passed to the decoder. The decoder performs an inverse scaling operation by multiplying the quantized and reconstructed latent representation with

its own learned channel-wise scaling parameters. Subsequently, the decoder decodes the latent representation. These scaling parameters are learned throughout the training and can be thought of as vectors which multiply the channels of latent representations.

The work of Cui et al. [6] has an important place in our network as it allows the adoption of flexible rate in our model without training separate instances. Although they propose the use of gain and inverse gain units in image compression, our model successfully integrates these components into video compression framework by using them in autoencoders of both motion compression and residual compression modules.

2.5. End-to-End Rate-Distortion Optimized Learned Hierarchical Bi-Directional Video Compression

Yilmaz and Tekalp [4] design a learned hierarchical bidirectional video compression network (LHBDC) to demonstrate the superior results that can be achieved with hierarchical bidirectional video coding frameworks compared to sequential video compression frameworks. They propose encoding videos with group of picture size of 8 frames and $K = 3$ hierarchical levels. Their proposed method compresses first reference frames as keyframes while compressing every other frame in between the initial reference frames as bidirectional predicted frames.

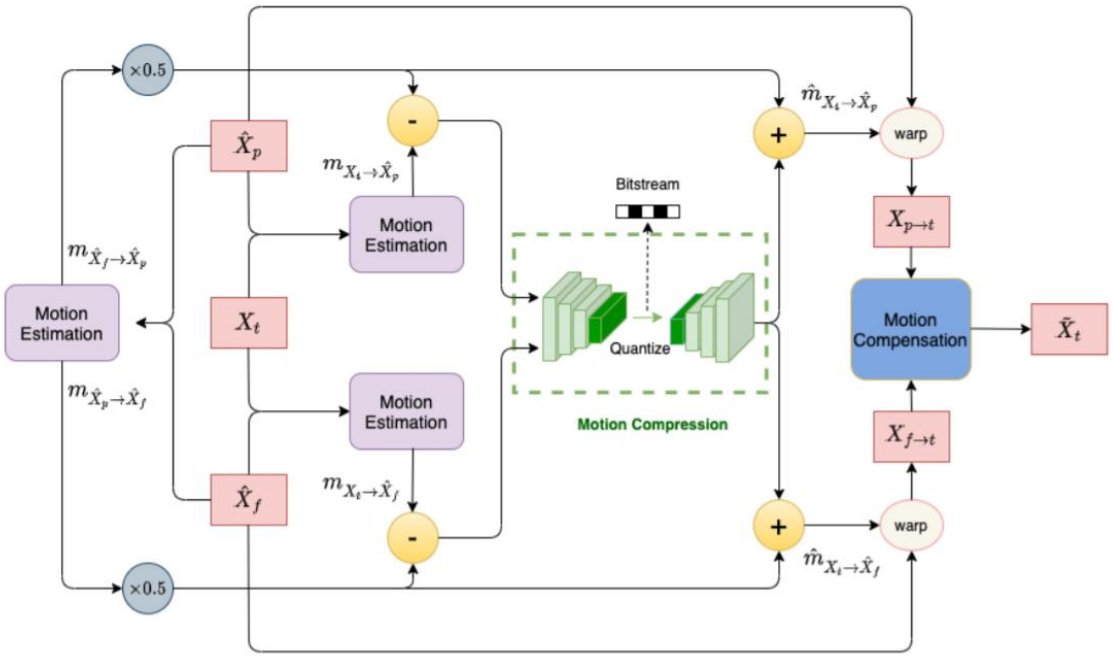

Figure 6. The architecture of LHBDC proposed by Yilmaz and Tekalp [4].

After compressing the initial keyframes at hierarchical level $K = 1$ using the version of learned still-image compression network proposed by Cheng et al. [8] without attention layers, the rest of the frames (7 frames) are coded bidirectionally using the network in Figure 6 with the initial keyframes

as reference frames. The bidirectional predicted frames take the closest decoded past and future frames as reference frames for the backward and forward motion estimation and compensation.

To perform bidirectional compression, the network initially estimates the motion vectors from the past and future decoded reference frames to the current frame. The estimation is performed by SPyNet [9] pretrained motion estimation network. Simultaneously, the motion vectors between the past and future decoded reference frames are also estimated with the same network to be used for the prediction of motion vectors. Making a linear motion assumption, Yilmaz and Tekalp [4] assumes that the half of the motion vectors between the past and future reference frames should yield a close prediction for the motion vectors between the past reference frame and the current frame. Calling the halved motion vectors between the past and future reference frames as predictions, they subtract these predictions from the motion vectors between the past reference frame and the current frame and repeat the same operation for the motion vectors between the future reference frame and the current frame. That way, they aim to compress the residual motion vectors which are the subtracted deviations from the predicted motion vectors. The residual motion vectors are then subsampled using a cubic filter to use less bits. The subsampled vectors are then compressed using a network that is a version of network by Minnen et al. [7] with residual blocks.

After reconstructing the residual motion vectors at the decoder side, these vectors are interpolated using a bicubic filter and added back to the predicted motion vectors and used to warp the reference frames to acquire the current frame. In order to utilize the bidirectional motion information, the two warped frames are fused using a motion compensation mask that is constructed with a U-Net architecture and warped frames as inputs. The fused frame is the final motion compensated frame that allows to compute the residual frame by subtracting it from the current frame. Finally, the residual frame is compressed with an autoencoder network that is similar to the motion compression network. The residual frame is added back to the motion compensated frame after it is reconstructed at the decoder to form the final reconstructed current frame.

As our model is designed in collaboration with Yilmaz and Tekalp, we adopt some components from their previous work with many adjustments. First of all, our framework uses a different keyframe compression network. Although the hierarchical structure of our bidirectional compression framework is same as the LHBDC, we do not assume linear motion and thus employ a non-linear motion prediction network. Furthermore, we perform motion refinement on top of the motion prediction and do not use the motion residual with an explicit subtraction operation. As our motion compression network has a similar input-output relation to the SSF [2] model which has frames as inputs and flow information as output, we have a significant difference from the LHBDC network.

Finally, our motion compression module also transfers the frame fusion mask to the decoder side as extra information whereas the LHBDC computes the mask using an additional network.

3. System Design

In order to build our video compression model, many trials are performed with different units such as deformable convolutions instead of bilinear warping with optical flow. In addition, the trials included working with a composite video compression framework that utilized both predicted-frames and bidirectional-frames that make use of unidirectional and bidirectional motion information, respectively. However, our trials with given variations did not yield satisfactory results; thus, our final trial with bidirectional video compression framework with motion refinement has been chosen as our best performing network.

As bidirectional video compression networks make use of motion information both in forward and backward directions in time, they are capable of achieving superior gains over unidirectional video compression networks. Resulting from this fact, our model achieves a competitive rate-distortion performance compared to other works in the literature. In the following sections, the basic building blocks of the model will be illustrated in detail with visual performance evaluations and architectural details.

3.1. Overview

Our model is composed of four main building blocks. These blocks are motion prediction, motion compression, learned frame fusion and residual compression modules which allow us to achieve a high compression rate with a low distortion cost. As each module is composed of differentiable operations, our model is suitable for end-to-end training using a single loss function.

Similar to work by Yilmaz and Tekalp [4], our work is trained and tested for a group of pictures of 8 frames and $K = 3$ hierarchical levels as displayed in Figure 7. In the proposed framework, the first frame of each group of pictures is coded as a keyframe, thus its compression does not make use of temporal correlation with previous or future frames.

For the keyframe compression, our model utilizes the learned image compression model proposed by Minnen et al. [7]. However, we do not use the context model that is proposed as it brings a significant slowdown because of its sequential operation over the pixels of an image. The keyframe compression model uses a hyperprior network which learns the entropy parameters of a frame and uses Gaussian distribution for probability modelling. That way, the model is capable of using arithmetic coding after determining the probability distribution over the pixels of an image.

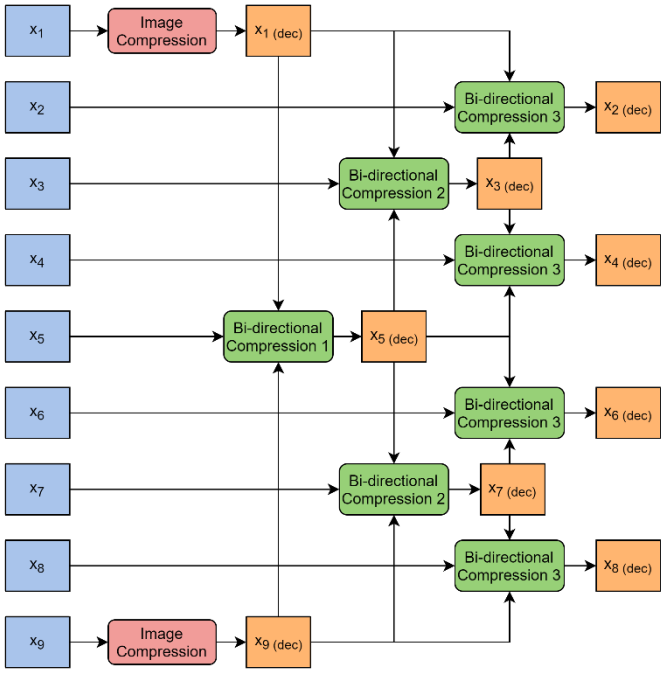

Figure 7. Coding scheme of the proposed model in a single group of pictures. In a group of pictures, only the first frame is intra-coded while the rest of the frames are coded in a bidirectional manner.

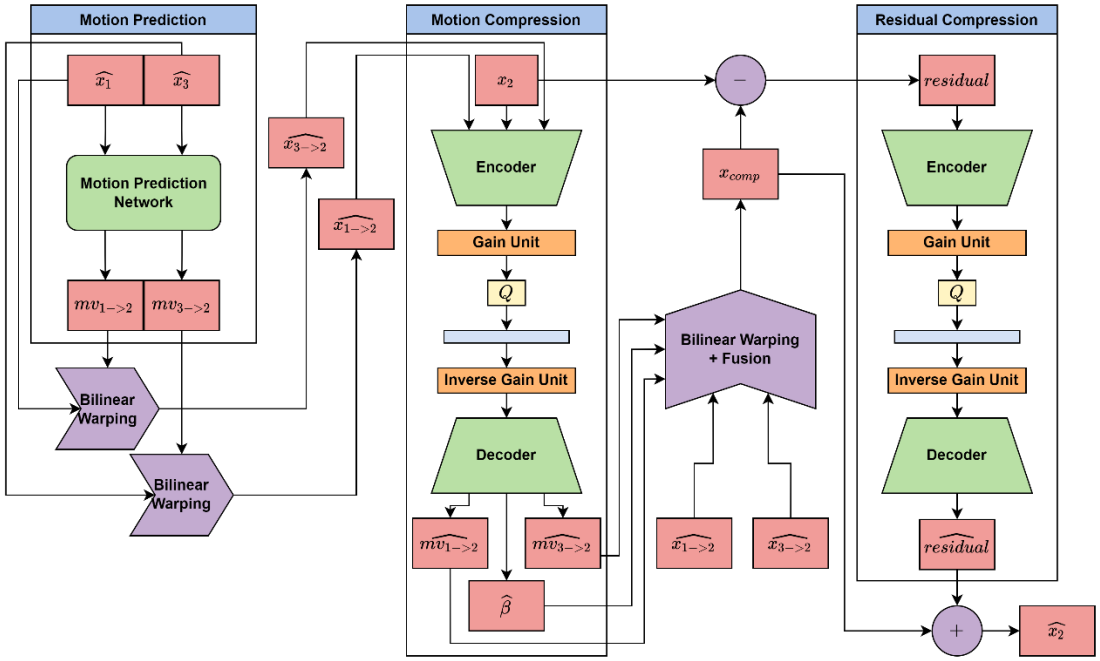

Figure 8. Overview of the proposed network architecture

After compressing the first frames of two consequent group of pictures using the keyframe compressor described above, other frames are compressed using our bidirectional compression model depicted in Figure 8. The keyframes are used as reference frames of the middle frame that relies in the hierarchical level $K = 1$, for the prediction of motion vectors and current frame. For all other frames of the group of pictures, we take past and future frames that are in one lower hierarchical level

as the backward and forward reference frames. For each frame, our model uses same parameters in the inference time. Because there exist seven frames in a group of pictures other than the keyframe, our model has to run seven times to encode and decode these seven frames. The separate components that bring increased compression efficiency to our model are described in following sections from Section 3.2 to Section 3.7.

3.2. Motion Vector Prediction

To reduce the temporal redundancy further and make use of the correlation between frames, we utilize a motion prediction network that has a U-Net architecture as depicted in Figure 9. Because of its architecture, the network is capable of learning a multiscale representation of frames and predicting the motion vectors more accurately.

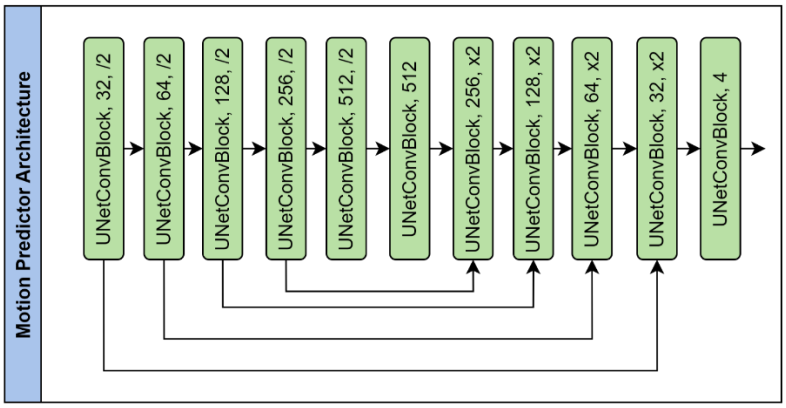

Figure 9. Motion vector prediction module architecture.

The motion vector prediction module takes two reference frames that are previously decoded and predicts two motion vectors that are estimated from the past reference frame to current frame and from the future reference frame to current frame. That way, we reduce the temporal redundancy by not transmitting the predictions since the coarse parts of the motion vectors are predicted by the prediction module which is present in both encoder and decoder. Since both the encoder and the decoder are aware of the prediction, we can transmit the finer details in motion vectors alone at the motion refinement module.

After predicting the coarse motion vectors which are exemplified in Figure 10, the reference frames are bilinear warped towards the current frame. These predicted frames are later on passed to the motion refinement and compression module that performs both compression and refinement.

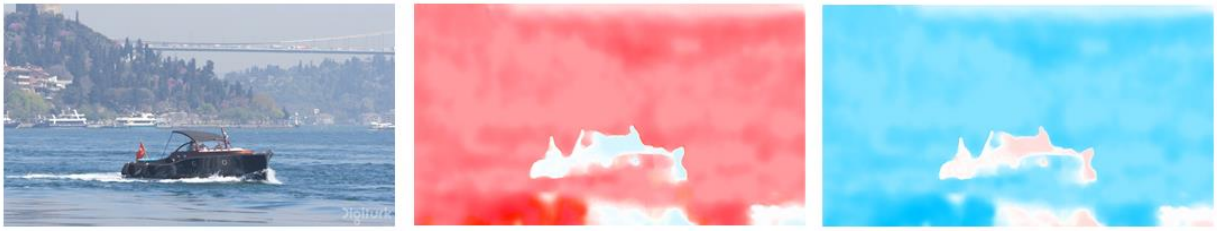

Figure 10. a) The ground truth current image. b) The motion vectors in forward direction from the past reference frame to current reference frame. c) The motion vectors in backward direction from the future reference frame to current reference frame. (Red color depicts motion vectors in the -x direction while blue color represents motion vectors in the +x direction.)

3.3. Motion Refinement

After predicting the coarse motion vectors with the motion prediction module, the finer details in the motion vectors are transmitted together with the refining motion compression module present in Figure 11. The middle layers of the autoencoder architecture has 128 filters to transform the frames into a latent representation at the encoder and form the motion vectors and the fusion mask later at the decoder. This module performs both the refinement and motion compression in a similar manner to the Scale-Space Flow model proposed by Agustsson et al. [2]. The module has the same architecture as the keyframe compression network proposed by Minnen et al. [7] except the context model which is not present in our model. In addition, our model utilizes residual blocks in order to reduce the problem of vanishing gradients and boost the optimization process.

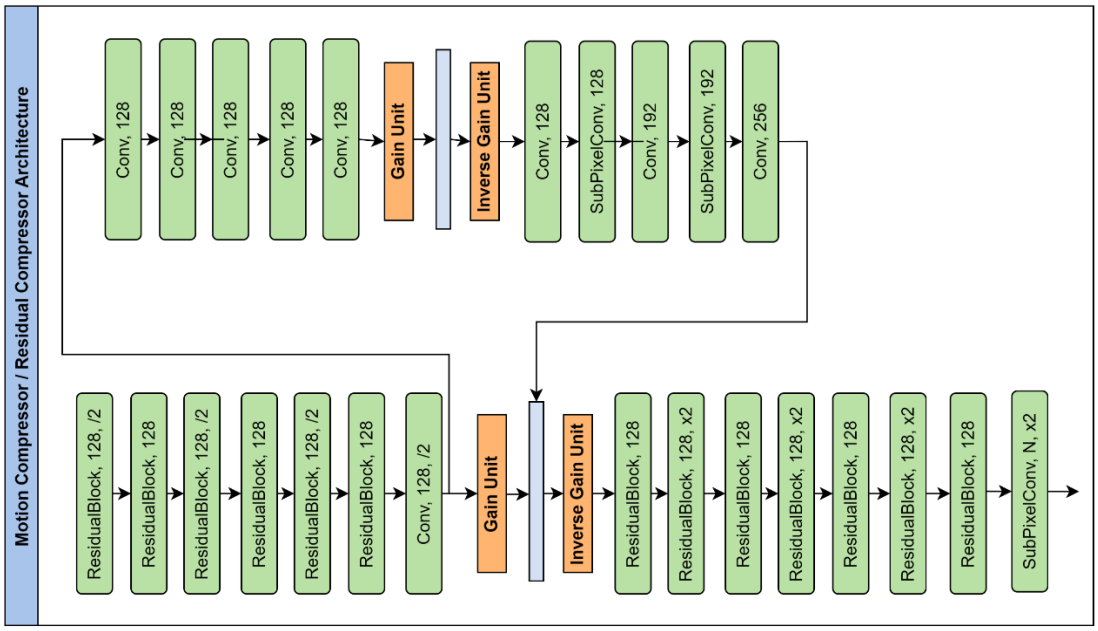

Figure 11. Motion refinement and compression module architecture.

The module takes the predicted frames and the ground truth current frame as its inputs. Thus, the

input layer must be provided with a tensor of 9 channels. After a latent representation is acquired and passed to the decoder, the decoder yields three separate tensors. These tensors are the two motion refinement tensors for the backward and forward warping as exemplified in Figure 12 and the fusion mask in order to fuse the warped frames later on.

a) b) c)

Figure 12. a) The ground truth current image. b) The motion refinement vectors in forward direction from the past reference frame to current reference frame. c) The motion refinement vectors in backward direction from the future reference frame to current reference frame. (Red color depicts motion vectors in the -x direction while blue color represents motion vectors in the +x direction.)

3.4. Gain and Inverse Gain Unit

In our motion refinement/motion compression and residual compression modules, one important component is the gain/inverse gain unit proposed by Cui et al. [6]. Although Cui et al. [6] has proposed using this component for image compression, our novel aim is to use the same component for video compression by using in both motion compression network and the residual compression network. This component allows us to train a single model to cover the complete rate-distortion curve without performing any additional trainings. Furthermore, the gain and inverse gain units allow us to form a continuous rate-distortion curve; thus, achieve arbitrary rate-distortion trade-offs without a major performance loss.

The gain and inverse gain units are simple matrices composed of learned matrices that are used to scale latent representations before the quantization step. The scaling operation is performed using the learned scale parameters of the gain and inverse gain units. The gain and inverse gain units are matrices of $N \times M$ dimensions where N stands for the compression levels that are desired, and M stands for the number of channels in the latent representation. In that case, the gain and inverse gain matrices can be thought of N scaling vectors that have M entries.

The scale parameters are paired for the gain and inverse gain units so that a scale vector, m_r in the gain unit is only matching with the scale vector, m_r' in the inverse gain unit. These scale vectors are learned per channel and are learned separately for different compression levels. In our framework, as the middle layer of both modules is composed of 128 filters, the latent representation has 128

channels and the gain and inverse gain units have $M = 128$ learned scale parameters per level.

Using the gain and inverse gain units at the inference time, we are capable of achieving a continuous rate-distortion curve by performing exponential interpolation to the quantization vectors that are present in the gain and inverse gain matrices. As the pairing of the gain units and the inverse gain vectors guarantee that the values of decoded frame and the ground truth frame remain in the same range, we can choose an arbitrary constant, C, so that the multiplication of every gain and inverse gain vector equals to C. Using this rule, the exponential interpolation operation can be described with the following mathematical operation,

$$\left(m_r \cdot m_r^{'}\right)^{l} \cdot \left(m_t \cdot m_t^{'}\right)^{1-l} = C$$

$$\left[(m_r)^{l} \cdot (m_t)^{1-l}\right] \cdot \left[\left(m_r^{'}\right)^{l} \cdot \left(m_t^{'}\right)^{1-l}\right] = C$$

$$m_v = \left[(m_r)^{l} \cdot (m_t)^{1-l}\right], m_v^{'} = \left[\left(m_r^{'}\right)^{l} \cdot \left(m_t^{'}\right)^{1-l}\right]$$

where l is the interpolation factor between 0 and 1, m_r and m_t are gain vectors of neighboring rate-distortion tradeoff values and their matching inverse gain vectors are m_r' and m_t', respectively. That way, we can come up with interpolated gain and inverse gain vectors such as m_v and m_v'.

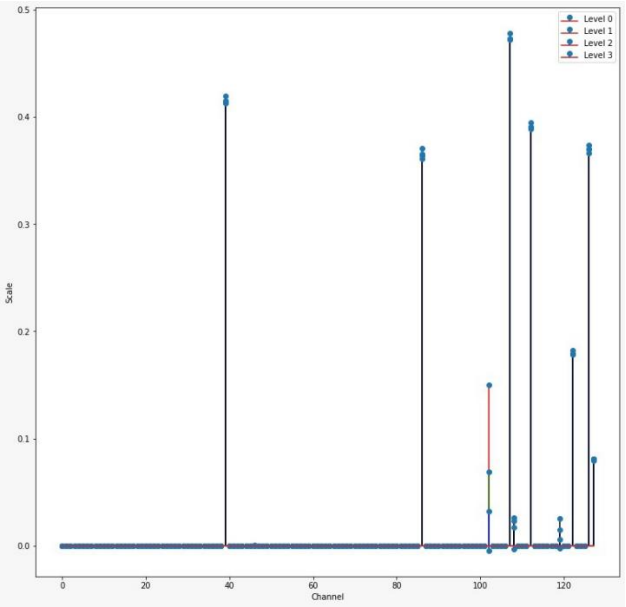

Figure 13. Demonstration of channel-wise constant vector, C which is a result of multiplication of gain and inverse gain matrices.

In Figure 13, we can visualize the multiplication of gain and inverse gain vectors for 4 separate levels with 128 channels. Following the results on the figure, we can conclude that the assumption that the multiplication of gain and inverse gain units is equal to a constant arbitrary vector, C, is valid as the

multiplication is almost same for all trade-off levels.

3.5. Learned Frame Fusion

After the motion refinement vectors and the fusion mask are collected from the motion refinement module, the motion compensation step is performed to acquire a single compensated frame using the framework depicted in Figure 14. To perform this operation, this module initially applies bilinear warping to the two previously predicted frames using the motion refinement vectors. The bilinear warping operation can be represented with the following mathematical representation,

$$\omega(\hat{x}_{t-1}, \hat{v}_t)[i,j] = \hat{x}_{t-1}\left[i + \hat{v}_t^x[i,j], j + \hat{v}_t^y[i,j]\right]$$

where $\omega(\hat{x}_{t-1}, \hat{v}_t)$ is the warping operation, $\hat{v}_t$ is the estimated motion vectors and $\hat{x}_{t-1}$ is the previously decoded frame that is sampled with bilinear interpolation. As the previously predicted frames only include the coarse motion information, the motion refinement vectors make finer touches on the frames and improve the frame quality. Subsequently, the final warped frames are fused to each other using the fusion mask that was the output of the motion refinement and compression module in Section 3.3. Fusing the two warped frames, we reduce the warping artefacts that were present after we performed warping two times. The fusion mask that is utilized in this step has the same dimensionality as our frames and can only take values between 0 and 1 since we apply a sigmoid at the final layer. That way, we force our model to take the best parts of both frames and fuse them at uncertain parts of the frame. The fusion operation can be displayed with the following operation.

$$\widehat{X_t} = K_t \times \widehat{X_{p\to t}} + (1 - K_t) \times \widehat{X_{f\to t}}$$

where K_t stands for the fusion mask, $\widehat{X_t}$ stands for the fused frame, $\widehat{X_{f\to t}}$ stands for the backward refined frame, and $\widehat{X_{p\to t}}$ stands for the forward refined frame.

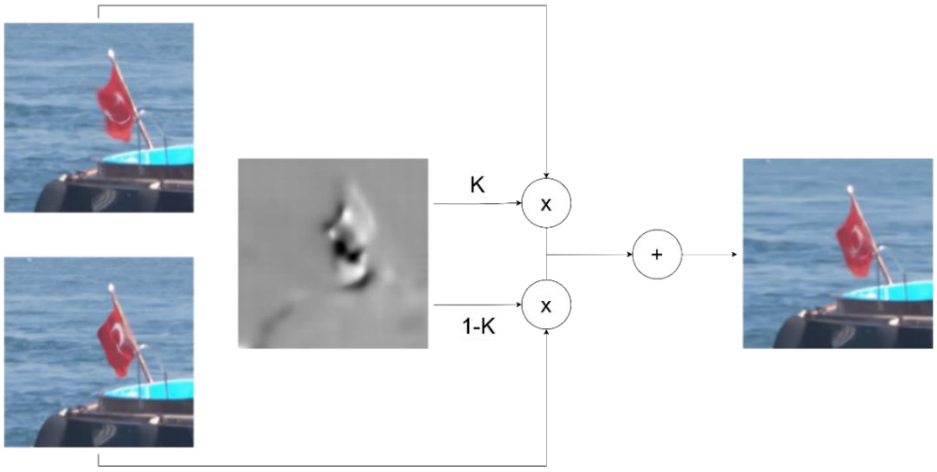

Figure 14. The diagram of frame fusion with the frame fusion mask after warping the predicted frames.

3.6. Residual Compression

Finally, the residual frame is acquired by subtracting the motion compensated frame from the ground truth frame. That way, we achieve a low entropy residual frame that is capable of correcting the motion compensated frame after it is compressed and decoded back as displayed in Figure 15.

The residual compression module has the same architecture as the motion refinement module that was presented in Section 3.3. The network again has 128 filters in the middle layers. The difference from the motion refinement module is that the residual compression network operates on an input frame which has 3 channels instead of 9 channels of motion refinement module. Furthermore, the output of the residual compression module is also a frame with 3 channels. The decoded residual frame is simply added back to the motion compensated frame in order to minimize the distortion on the output frame. That way, we acquire the desired output frame.

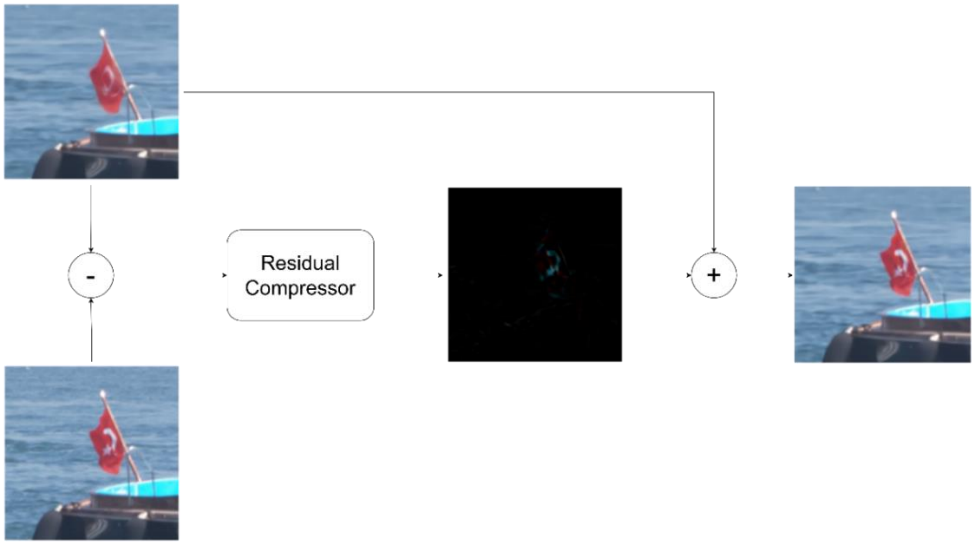

Figure 15. The diagram for the residual compression module.

4. Experiments

4.1. Setup

Our compression network is optimized in an end-to-end manner since it only contains differentiable components. To setup and optimize the model, PyTorch library [10] is used to provide a deep learning framework. Furthermore, the pretrained keyframe compression network proposed by Minnen et al. [7] is taken from the CompressAI library [11] with the name of "mbt2018_mean" and quality levels of 5, 6, 7, and 8 corresponding to trade-off values of $\lambda = \{845, 1626, 3141, 6060\}$. Further details about the environment and library versions can be found in Appendix 9.1.

4.2. Datasets

The bidirectional compression network with motion refinement is trained on the Vimeo-90K [12]

dataset. The specific septuplet dataset has 91,701 videos with seven frames per video at a resolution of 448 by 256. The dataset is also further augmented by taking different crops of 256 by 256 during the training time.

To test our model, we encode and decode the video sequences from the UVG dataset [13]. Namely, we utilized our model on the Beauty, Bosphorus, Honeybee, ShakeNDry, Jockey, ReadySetGo, and YatchRide sequences. For these video sequences, each video has 600 frames except the shake sequence which has 300 frames. The videos have a spatial resolution of 1920 by 1080 and a temporal resolution of 120 fps.

4.3. Loss Functions

Our aim for the model is to achieve the maximum frame quality with the minimum number of bits. Thus, to train our model, a rate-distortion loss function is utilized as following.

$$L = \lambda D + H(v_{mv}) + H(v_{residual})$$

where D stands for the distortion present in the decoded frames, v_{mv} stands for the latent representation of the motion vectors of the bidirectional frames and $v_{residual}$ stands for the latent representation of the residual components of the bidirectional frames. The $H(.)$ operator is for the entropy calculation of the latent representations and the result of the operator provides the bitrate for a single frame. The entropy parameters (mean and standard deviation) are calculated by the hyperprior network present in Motion Compression and Residual Compression modules. As the architecture of these modules are similar to the network presented by Minnen et al. [7], the probability distribution of individual pixels is approximated with Gaussian distribution. To achieve different bitrates and frame qualities, different trade-off values (λ) are used for each rate-distortion level. To optimize the model parameters, we use two different distortion functions. First of all, we train our model using mean squared error (MSE),

$$D(\hat{x}, x) = MSE(\hat{x}, x) = \frac{1}{h \times w} \sum_{n=1}^{h \times w} (\hat{x}_n - x_n)^2$$

where $\hat{x}_n$ is the decoded pixel, x_n is the ground truth pixel and $h \times w$ is the dimensions of the frame. Later on, the model is additionally finetuned for a second model using Multi-scale Structural Similarity Method (MS-SSIM) score [14] in order to achieve a result that is more in line with the human visual system. This score can be expressed with the following diagram in Figure 16,

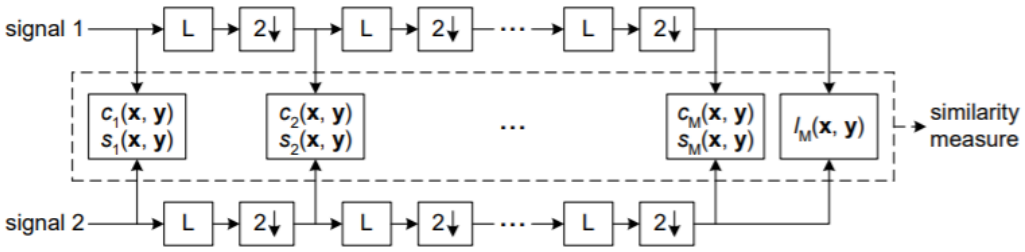

Figure 16. The diagram to calculate the MS-SSIM loss. The number of levels for our training is specified as 5.

where $c(.), s(.)$, I_M and the overall loss are expressed as following,

$$d(x,y) = [l_M(x,y)]^{\alpha_M} \prod_{j=1}^{M} [c_j(x,y)]^{\beta_j} [s_j(x,y)]^{\gamma_j}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y+(K_2L)^2}{\sigma_x^2+\sigma_y^2+(K_2L)^2}, \quad s(x,y) = \frac{\sigma_{xy}+C_2/2}{\sigma_x\sigma_y+C_2/2}, \quad l(x,y) = \frac{2\mu_x\mu_y+(K_1L)^2}{\mu_x^2+\mu_y^2+(K_1L)^2}$$

In these equation for the calculation of the MS-SSIM score, x and y are the decoded and ground truth frames while the α, β and γ are pre-determined values by Wang et al. [14], determining the relative importance of different scales and components. At the end of the training, the model is tested in terms of the PSNR and MS-SSIM scores.

4.4. Training Details

For the training, we do not use any pretrained models and optimize the complete model in an end-to-end manner. The training is performed on an NVIDIA Tesla V100 GPU for 2M iterations. The training is performed with the Vimeo-90K dataset [12] which provided crops of 256 by 256. For data augmentation, the crops are randomly selected from various parts of the 256 by 448 frames. At every 5K iterations, we perform validation on our model in order to control the performance on the non-training data and save the model if it generalizes well. The validation is performed using the first eight frames of the seven specified videos of the UVG dataset [13]. The network is optimized using Adam optimizer and initial learning rate is set to 10^{-4}. The learning rate is reduced by a half when no improvement in validation is observed for 100K iterations.

As the model needs to perform quantization to encode the latent representations, we model this effect with additive noise during training and perform hard quantization with rounding during test time. The additive noise that models the effect of rounding has a standard deviation of 0.5 with a mean of 0.

For our training, we used $\lambda = \{436, 1626, 3141, 6060\}$.which corresponds to four gain and

inverse gain vector pairs in both the motion refinement and the residual compression modules. Using these trade-off values and the gain and inverse gain vector pairs, we train a single model for each hierarchical level of each group of pictures and each quality level. Since the single model is trained to perform well on different hierarchical and quality levels, our model can successfully generalize to very different settings. During the training, we formed mini batches of 4 video sequences with 3 consequent frames. With each mini batch, we train all four levels that correspond to a trade-off value λ and its gain and inverse gain vectors.

5. Analysis and Results

To compare our bidirectional video compression with other networks proposed in the literature, we provide both quantitative and qualitative comparisons in the following sections. During the tests our model chooses a group of pictures size of 8 frames. That way we encode 7 frames as bidirectional frames between every 2 intra-coded frames that were encoded using the model proposed by Minnen et al. [7] without the context model.

5.1. Quantitative Results

We compare our network quantitatively with the networks proposed in the Related Works and additionally traditional H.265 video codec [15]. As an anchor, the performance of the SVT-HEVC codec of H.265 codec is displayed in bidirectional compression mode at very slow preset. Other than H.265, we compare our network with famous learned video compression networks. These networks are DVC [1], Scale-Space Flow [2], RLVC [3] and the model proposed by Yilmaz and Tekalp [4]. The performances of the given models are acquired from their repositories provided in GitHub.

To compare our results with the given works, we evaluate them in terms of PSNR and MS-SSIM scores and plot their rate-distortion curves. The scores are plotted against bits per pixel (bpp) over the resulting values from testing on UVG dataset [13]. The rate-distortion curves are acquired by linear interpolation as can be seen in Figure 17 and Figure 18.

Observing Figures 17 and 18, it can be seen that our model displays a superior performance over other codecs except at lower bitrates. Although our model performs better by a small margin at higher bitrates, it falls behind of SVT-HEVC codec with very slow preset and the model proposed by Yilmaz and Tekalp [4]. Other than that, our model achieves a substantial margin against other codecs. As our model is a developed version of model by Yilmaz and Tekalp [4], our main comparison should be based on their model.

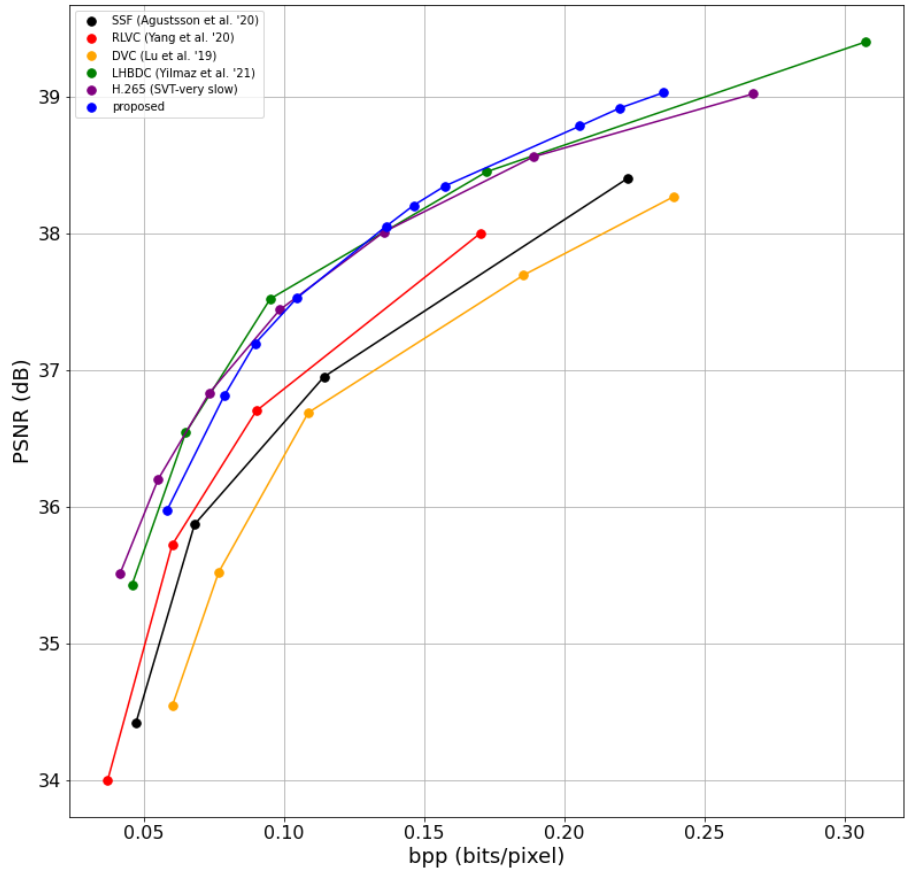

Figure 17. Rate-Distortion performance comparison with other models in terms of PSNR. The higher and to the left, the better is the performance.

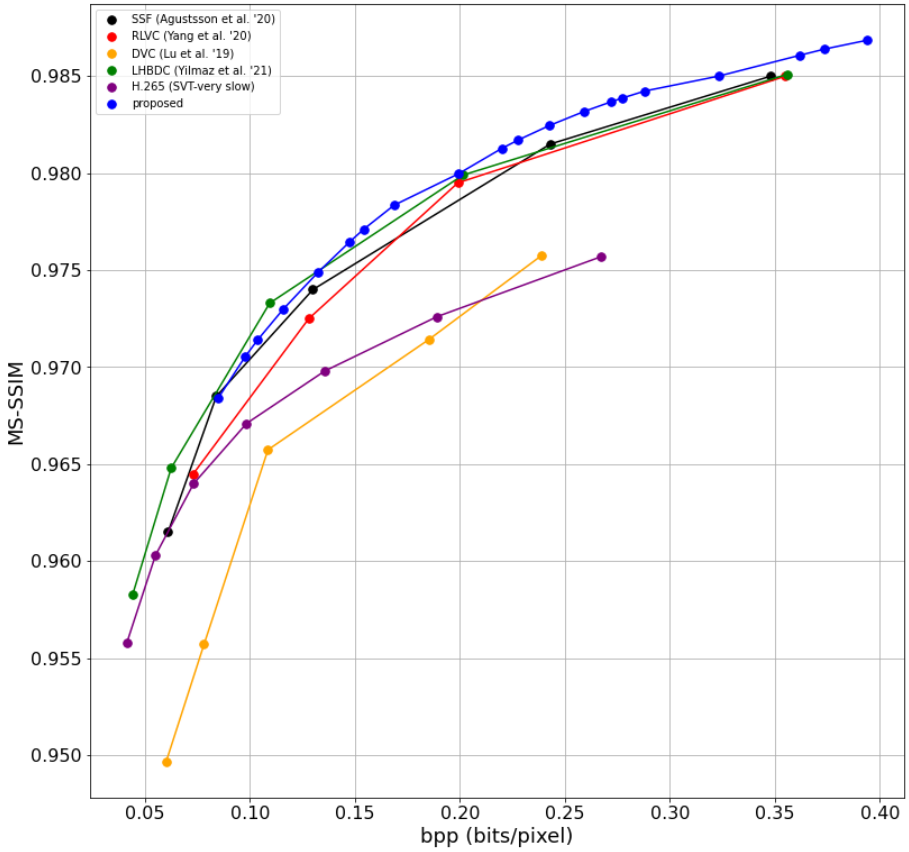

Figure 18. Rate-Distortion performance comparison with other models in terms of MS-SSIM.

These results are somewhat expected as bidirectional video compression is capable of achieving superior results compared to unidirectional video compression methods such as DVC [1] and SSF [2] due to the additional information coming from the backward motion information. However, comparing with the LHBDC model proposed by Yilmaz and Tekalp [4], the margin is relatively small. Although our model employs a more complex motion prediction module and performs motion refinement, the reason behind this fact might be due to the gain and inverse gain units that were not present in the model by Yilmaz and Tekalp [4]. As our framework is built with the training of a single model that can perform well at different bitrates, the encoders and decoders are more constrained compared to the LHBDC model [4].

Furthermore, a secondary reason for the lack of performance at lower bitrates might be the warping artifacts that occur on the predicted frames. As we apply warping with the refined motions on the predicted frames, the error from the predicted frames might be propagating and resulting in worsened frame quality. As there are multiple reasons for the inferior performance at lower bitrates, the reason for this difference will be investigated by training our model in separate instances without the gain and inverse gain units and by training a second version of our model which applies the motion refinement by adding the finer motion vector details onto the predicted motion vectors instead of applying warping to the predicted frames.

On the other hand, it is important to note that our rate-distortion curves have significantly more samples on the curve which is a result of adoption of gain and inverse gain units. This result implies another strength of our model which allows us to achieve a more continuous rate-distortion curve without training extensive numbers of models.

5.2. Qualitative Results

In addition to comparing our results in quantitative terms, a qualitative analysis is also performed by comparing the visual quality of decoded frames using our proposed model, an unofficial implementation of the SSF model [2] and the H.265/x.265 codec [15]. As the previous networks are not made publicly available, the qualitative comparison can be rather limited.

For comparison, we can observe the decoded frames from the Bosphorus video sequence of the UVG dataset in Figure 19, we can detect the higher quality of the proposed network both in terms of the quantitative measures such as the PSNR, PSNR in YCrCb channels and MS-SSIM scores. In all quantitative measures our proposed model achieves the best results while SSF model [2] comes second. Furthermore, comparing the details on the waves and the flag on Figure 19, we can detect the high frequency details more clearly with the proposed model while the results with the SSF and H.265

lack such details.

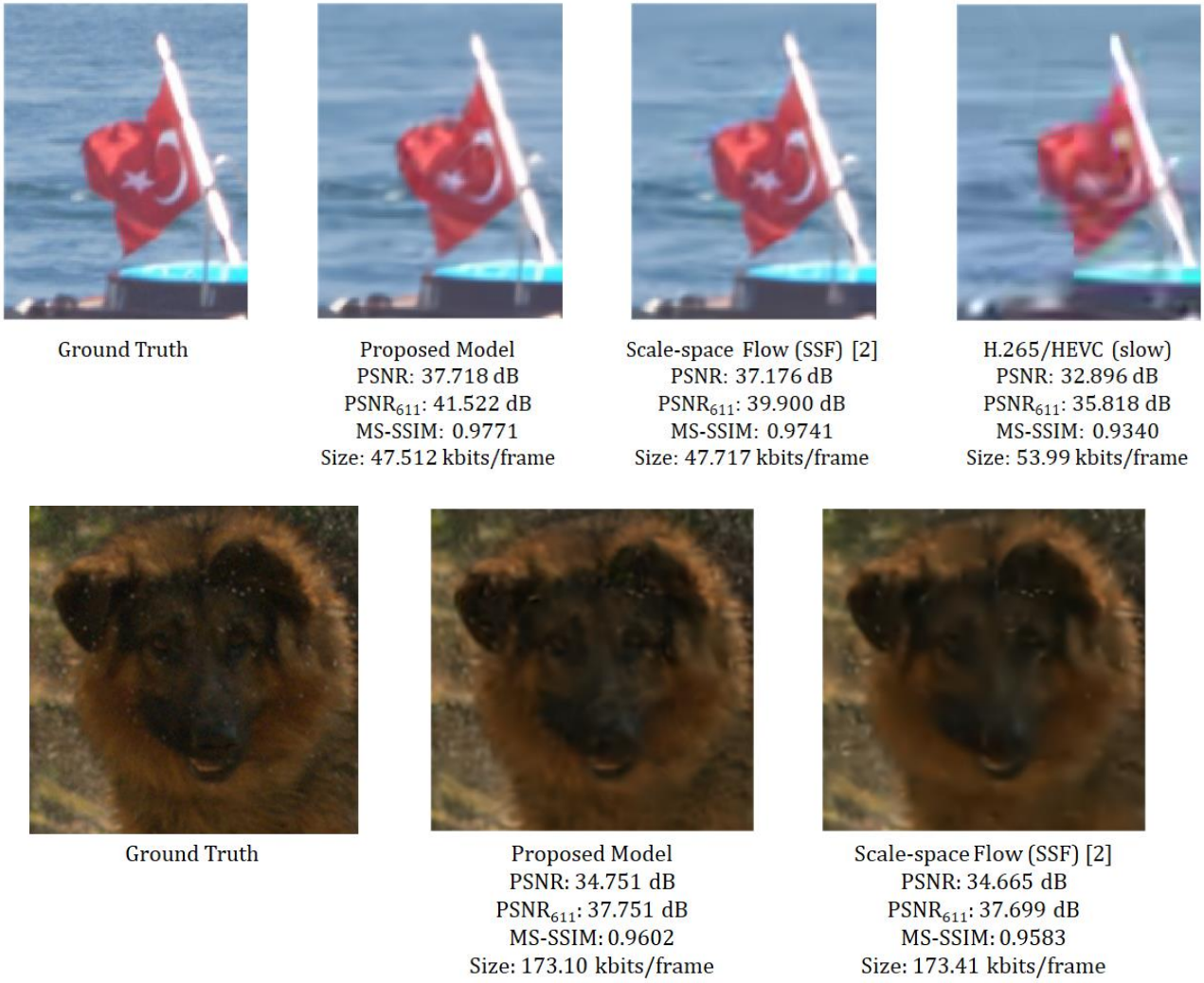

Figure 19. Qualitative comparison of the proposed model with the Scale-Space Flow model [2] and x265 codec [15] with slow preset.

6. Conclusion

Our flexible-rate bidirectional video compression network yields a competitive rate-distortion performance compared to other works in the literature of learned video compression. Although the model achieves slightly worse frame qualities at lower bitrates, it still remains competitive while achieving a better rate-distortion performance at higher bitrates when compared using the UVG dataset [13]. In addition, our model allows us to train a single model to achieve all rate-distortion trade-off values on the rate-distortion curve whereas other models are doomed to train several separate networks in order to build a rate-distortion curve and achieve bitrates at different ranges. As our aim was to build a bidirectional model that performs better in terms of rate-distortion performance at all bitrates, our design has partially met its goal.

For the future work, we aim to investigate the reasoning behind the inferior performance at the

lower bitrates and perhaps yield an improvement also in the higher bitrates. As the reason for the inferior results might be the warping artifacts that yield from the frame prediction step and secondary application of warping on the predicted frames, we aim to propose a new model which sums the predicted motion vectors and the motion refinements to apply motion compensation only once instead of twice. We believe that the warping artifacts can be reduced by making such a change.

Furthermore, an ablation study will be performed on the effect of our adoption of gain and inverse gain units. Although Cui et al. [6] claim that the units have no adversarial effect on the model performance in image compression, our integration into video compression might be different. For that reason, we plan to train our model in separate instances at all 4 levels which have been trained in single training.

7. References

[1] G. Lu, W. Ouyang, D. Xu, X. Zhang, C. Cai and Z. Gao, "DVC: An End-to-end Deep Video Compression Framework," in *Computer Vision and Pattern Recognition (CVPR) 2019*, Long Beach, California, 2019.

[2] E. Agustsson, D. Minnen, N. Johnston, J. Balle, S. J. Hwang and G. Toderici, "Scale-space flow for end-to-end optimized video compression," in *Computer Vision and Pattern Recognition (CVPR)*, Virtual, 2020.

[3] R. Yang, F. Mentzer, L. Van Gool and R. Timofte, "Learning for Video Compression With Recurrent Auto-Encoder and Recurrent Probability Model," *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 2, pp. 338-401, 2021.

[4] M. A. Yılmaz and A. M. Tekalp, "End-to-End Rate-Distortion Optimized Learned Hierarchical Bi-Directional Video Compression," *IEEE Transactions on Image Processing*, vol. 31, pp. 974-983, 2021.

[5] Cisco, "Cisco Annual Internet Report (2018–2023) White Paper," 9 March 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. [Accessed 10 October 2021].

[6] Z. Cui, J. Wang, S. Gao, T. Guo, Y. Feng and B. Bai, "Asymmetric Gained Deep Image Compression With Continuous Rate Adaptation," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, Virtual, 2021.

[7] D. Minnen, J. Balle and G. Toderici, "Joint autoregressive and hierarchical priors for learned image compression," in *NeurIPS*, Montreal, 2018.

[8] Z. Cheng, H. Sun, Takeuchi and J. Katto, "Learned Image Compression with Discretized Gaussian Mixture Likelihoods and Attention Modules," in *IEEE Computer Vision and Pattern Recognition (CVPR)*, Seattle, 2020.

[9] A. Ranjan and M. Black, "Optical Flow Estimation using a Spatial Pyramid Network," in *CVPR 2017*, 2017.

[10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. L. Z. DeVito, A. Desmaison, L. Antiga and A. Lerer, "Automatic differentiation in PyTorch," in *NeurIPS 2017*, California, 2017.

[11] J. Begaint, F. Racape, S. Feltman and A. Pushparaja, "CompressAI: a PyTorch library and evaluation platform for end-to-end compression research," *arXiv preprint arXiv:2011.03029*, 2020.

[12] T. Xue, B. Chen, J. Wu, D. Wei and W. Freeman, "Video Enhancement with Task-Oriented Flow," *International Journal of Computer Vision*, vol. 127, no. 8, pp. 1106-1125, 2019.

[13] A. Mercat, M. Viitanen and J. Vanne, "UVG dataset: 50/120fps 4K sequences for video codec analysis and development," in *ACM Multimedia Syst. Conf.*, Istanbul, 2020.

[14] Z. Wang, E. Simoncelli and A. Bovik, "Multi-Scale Structural Similarity for Image Quality Assessment," in *IEEE Asilomar Conference on Signals, Systems and Computers*, 2003.

[15] G. J. Sullivan, J.-R. Ohm, W.-J. Han and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 22, pp. 1649-1668, 2012.

[16] L. Haopeng, Y. Yuan and W. Qi, "Video Frame Interpolation Via Residue Refinement," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Barcelona, 2020.

8. Appendices

8.1. Environment (.yml file)

```
name: icip2022
channels:
  - pytorch
  - anaconda
  - conda-forge
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - anyio=2.0.2=py38h578d9bd_4
  - argon2-cffi=20.1.0=py38h25fe258_2
  - async_generator=1.10=py_0
  - attrs=20.3.0=pyhd3deb0d_0
  - babel=2.9.0=pyhd3deb0d_0
  - backcall=0.2.0=pyh9f0ad1d_0
  - backports=1.0=py_2
  - backports.functools_lru_cache=1.6.1=py_0
  - blas=1.0=mkl
  - bleach=3.2.2=pyh44b312d_0
  - brotlipy=0.7.0=py38h8df0ef7_1001
  - ca-certificates=2020.12.5=ha878542_0
  - certifi=2020.12.5=py38h578d9bd_1
  - cffi=1.14.4=py38h261ae71_0
  - chardet=4.0.0=py38h578d9bd_1
  - cryptography=3.3.1=py38h3c74f83_0
  - cudatoolkit=11.0.221=h6bb024c_0
  - decorator=4.4.2=py_0
  - defusedxml=0.6.0=py_0
  - entrypoints=0.3=pyhd8ed1ab_1003
  - freetype=2.10.4=h5ab3b9f_0
  - idna=2.10=pyh9f0ad1d_0
  - imageio=2.9.0=py_0
  - importlib-metadata=3.4.0=py38h578d9bd_0
  - importlib_metadata=3.4.0=hd8ed1ab_0
  - intel-openmp=2020.2=254
  - ipykernel=5.4.3=py38h81c977d_0
  - ipython=7.12.0=py38h5ca1d4c_0
  - ipython_genutils=0.2.0=py_1
  - jedi=0.18.0=py38h578d9bd_2
  - jinja2=2.11.2=pyh9f0ad1d_0
  - jpeg=9b=h024ee3a_2
  - json5=0.9.5=pyh9f0ad1d_0
  - jsonschema=3.2.0=py_2
  - jupyter_client=6.1.11=pyhd8ed1ab_1
  - jupyter_core=4.7.0=py38h578d9bd_1
  - jupyter_server=1.2.2=py38h578d9bd_1
  - jupyterlab=3.0.5=pyhd8ed1ab_0
  - jupyterlab_pygments=0.1.2=pyh9f0ad1d_0
```

```
  - jupyterlab_server=2.1.2=pyhd8ed1ab_0
  - lcms2=2.11=h396b838_0
  - ld_impl_linux-64=2.33.1=h53a641e_7
  - libedit=3.1.20191231=h14c3975_1
  - libffi=3.3=he6710b0_2
  - libgcc-ng=9.1.0=hdf63c60_0
  - libgfortran-ng=7.3.0=hdf63c60_0
  - libpng=1.6.37=hbc83047_0
  - libsodium=1.0.18=h36c2ea0_1
  - libstdcxx-ng=9.1.0=hdf63c60_0
  - libtiff=4.1.0=h2733197_1
  - libuv=1.40.0=h7b6447c_0
  - lz4-c=1.9.3=h2531618_0
  - markupsafe=1.1.1=py38h8df0ef7_2
  - mistune=0.8.4=py38h25fe258_1002
  - mkl=2020.2=256
  - mkl-service=2.3.0=py38he904b0f_0
  - mkl_fft=1.2.0=py38h23d657b_0
  - mkl_random=1.1.1=py38h0573a6f_0
  - natsort=7.0.1=py_0
  - nbclassic=0.2.6=pyhd8ed1ab_0
  - nbclient=0.5.1=py_0
  - nbconvert=6.0.7=py38h578d9bd_3
  - nbformat=5.1.2=pyhd8ed1ab_1
  - ncurses=6.2=he6710b0_1
  - nest-asyncio=1.4.3=pyhd8ed1ab_0
  - ninja=1.10.2=py38hff7bd54_0
  - notebook=6.2.0=py38h578d9bd_0
  - numpy=1.19.2=py38h54aff64_0
  - numpy-base=1.19.2=py38hfa32c7d_0
  - olefile=0.46=py_0
  - openssl=1.1.1i=h27cfd23_0
  - packaging=20.8=pyhd3deb0d_0
  - pandoc=2.11.3.2=h7f98852_0
  - pandocfilters=1.4.2=py_1
  - parso=0.8.1=pyhd8ed1ab_0
  - pexpect=4.8.0=pyh9f0ad1d_2
  - pickleshare=0.7.5=py_1003
  - pillow=8.1.0=py38he98fc37_0
  - pip=20.3.3=py38h06a4308_0
  - prometheus_client=0.9.0=pyhd3deb0d_0
  - prompt-toolkit=3.0.11=pyha770c72_0
  - prompt_toolkit=3.0.11=hd8ed1ab_0
  - ptyprocess=0.7.0=pyhd3deb0d_0
  - pycparser=2.20=pyh9f0ad1d_2
  - pygments=2.7.4=pyhd8ed1ab_0
  - pyopenssl=20.0.1=pyhd8ed1ab_0
  - pyparsing=2.4.7=pyh9f0ad1d_0
  - pyrsistent=0.17.3=py38h25fe258_1
  - pysocks=1.7.1=py38h578d9bd_3
```

```
  - python=3.8.5=h7579374_1
  - python-dateutil=2.8.1=py_0
  - python_abi=3.8=1_cp38
  - pytorch=1.7.1=py3.8_cuda11.0.221_cudnn8.0.5_0
  - pytz=2020.5=pyhd8ed1ab_0
  - pyzmq=20.0.0=py38h1d1b12f_1
  - readline=8.0=h7b6447c_0
  - requests=2.25.1=pyhd3deb0d_0
  - scipy=1.5.2=py38h0b6359f_0
  - send2trash=1.5.0=py_0
  - setuptools=51.3.3=py38h06a4308_4
  - six=1.15.0=py38h06a4308_0
  - sniffio=1.2.0=py38h578d9bd_1
  - sqlite=3.33.0=h62c20be_0
  - terminado=0.9.2=py38h578d9bd_0
  - testpath=0.4.4=py_0
  - tk=8.6.10=hbc83047_0
  - torchaudio=0.7.2=py38
  - torchvision=0.8.2=py38_cu110
  - tornado=6.1=py38h25fe258_0
  - traitlets=5.0.5=py_0
  - typing_extensions=3.7.4.3=py_0
  - urllib3=1.26.2=pyhd8ed1ab_0
  - wcwidth=0.2.5=pyh9f0ad1d_2
  - webencodings=0.5.1=py_1
  - wheel=0.36.2=pyhd3eb1b0_0
  - xz=5.2.5=h7b6447c_0
  - zeromq=4.3.3=h58526e2_3
  - zipp=3.4.0=py_0
  - zlib=1.2.11=h7b6447c_3
  - zstd=1.4.5=h9ceee32_0
  - pip:
    - cupy-cuda110==8.5.0
    - cycler==0.10.0
    - fastrlock==0.5
    - kiwisolver==1.3.1
    - matplotlib==3.3.3
    - pytorch-msssim==0.2.0
prefix: /scratch/users/ecetin17/.conda/envs/icip2022
```

8.2. U-Net Code [16]

```
import torch
from torch import nn
import torch.nn.functional as F

# Adapted from "Tunable U-Net implementation in PyTorch"
# https://github.com/jvanvugt/pytorch-unet

class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=2, depth=5, wf=5,
padding=True):
        super(UNet, self).__init__()
```

```
        self.padding = padding
        self.depth = depth
        prev_channels = in_channels
        self.down_path = nn.ModuleList()
        for i in range(depth):
            self.down_path.append(
                UNetConvBlock(prev_channels, 2 ** (wf + i), padding)
            )
            prev_channels = 2 ** (wf + i)
        self.midconv = nn.Conv2d(prev_channels, prev_channels, kernel_size=3,
padding=1)

        self.up_path = nn.ModuleList()
        for i in reversed(range(depth - 1)):
            self.up_path.append(
                UNetUpBlock(prev_channels, 2 ** (wf + i), padding)
            )
            prev_channels = 2 ** (wf + i)

        self.last = nn.Conv2d(prev_channels, out_channels,
kernel_size=3,padding=1)

    def forward(self, x):
        blocks = []
        for i, down in enumerate(self.down_path):
            x = down(x)
            if i != len(self.down_path) - 1:
                blocks.append(x)
                x = F.avg_pool2d(x, 2)
        x = F.leaky_relu(self.midconv(x), negative_slope = 0.1)
        for i, up in enumerate(self.up_path):
            x = up(x, blocks[-i - 1])

        return self.last(x)

class UNetConvBlock(nn.Module):
    def __init__(self, in_size, out_size, padding):
        super(UNetConvBlock, self).__init__()
        block = []

        block.append(nn.Conv2d(in_size, out_size, kernel_size=3,
padding=int(padding)))
        block.append(nn.LeakyReLU(0.1))

        block.append(nn.Conv2d(out_size, out_size, kernel_size=3,
padding=int(padding)))
        block.append(nn.LeakyReLU(0.1))
        self.block = nn.Sequential(*block)

    def forward(self, x):
        out = self.block(x)
        return out

class UNetUpBlock(nn.Module):
    def __init__(self, in_size, out_size, padding):
        super(UNetUpBlock, self).__init__()

        self.up = nn.Sequential(
                nn.Upsample(mode='bilinear', scale_factor=2),
                nn.Conv2d(in_size, out_size, kernel_size=3, padding=1),
            )
        self.conv_block = UNetConvBlock(in_size, out_size, padding)
```

```
    def center_crop(self, layer, target_size):
        _, _, layer_height, layer_width = layer.size()
        diff_y = (layer_height - target_size[0]) // 2
        diff_x = (layer_width - target_size[1]) // 2
        return layer[
            :, :, diff_y : (diff_y + target_size[0]), diff_x : (diff_x +
target_size[1])
        ]
    def forward(self, x, bridge):
        up = self.up(x)
        crop1 = self.center_crop(bridge, up.shape[2:])
        out = torch.cat((up, crop1), 1)
        out = self.conv_block(out)
        return out
```

8.3. Layers Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from compressai.models import MeanScaleHyperprior
from compressai.models.utils import conv, deconv
from compressai.layers import (
    GDN,
    AttentionBlock,
    ResidualBlock,
    ResidualBlockUpsample,
    ResidualBlockWithStride,
    conv3x3,
    subpel_conv3x3,
)

def conv(in_channels, out_channels, kernel_size=5, stride=2):
    return nn.Conv2d(
        in_channels,
        out_channels,
        kernel_size=kernel_size,
        stride=stride,
        padding=kernel_size // 2,
    )

def deconv(in_channels, out_channels, kernel_size=5, stride=2):
    return nn.ConvTranspose2d(
        in_channels,
        out_channels,
        kernel_size=kernel_size,
        stride=stride,
        output_padding=stride - 1,
        padding=kernel_size // 2,
    )

class Gain_Module(nn.Module):
        def __init__(self, n=6, N=128, bias=False, inv=False):
            """
            n: number of scales for quantization levels
            N: number of channels
            """
            super(Gain_Module, self).__init__()

            self.gain_matrix = nn.Parameter(torch.ones(n, N))
```

```
            self.bias = bias
            if bias:
                self.bias = nn.Parameter(torch.ones(N))

        def forward(self, x, n=None, l=1):
            B, C, H, W = x.shape

            # If we want to find a non-trained rate-distortion point
            if (l != 1):
                gain1 = self.gain_matrix[n]
                gain2 = self.gain_matrix[[n[0]+1]]
                gain = (torch.abs(gain1)**l)*(torch.abs(gain2)**(1-l))

            else:
                gain = torch.abs(self.gain_matrix[n])

            reshaped_gain = gain.unsqueeze(2).unsqueeze(3)

            rescaled_latent = reshaped_gain * x

            if self.bias:
                rescaled_latent += self.bias[n]

            return rescaled_latent

class FlowCompressor(MeanScaleHyperprior):

    def __init__(self, n=6, in_ch=9, out_ch=5, N=128, bias=False, **kwargs):
        super().__init__(N=N, M=N, **kwargs)

        self.g_a = nn.Sequential(
            ResidualBlockWithStride(in_ch, N, stride=2),
            ResidualBlock(N, N),
            ResidualBlockWithStride(N, N, stride=2),
            ResidualBlock(N, N),
            ResidualBlockWithStride(N, N, stride=2),
            ResidualBlock(N, N),
            conv3x3(N, N, stride=2),
        )

        self.h_a = nn.Sequential(
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N, stride=2),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N, stride=2),
        )

        self.h_s = nn.Sequential(
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            subpel_conv3x3(N, N, 2),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N * 3 // 2),
            nn.LeakyReLU(inplace=True),
            subpel_conv3x3(N * 3 // 2, N * 3 // 2, 2),
            nn.LeakyReLU(inplace=True),
            conv3x3(N * 3 // 2, N * 2),
        )
```

```
        self.g_s = nn.Sequential(
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            subpel_conv3x3(N, out_ch, 2),
        )
        self.g_s[-1][0].weight.data.fill_(0.0)
        self.g_s[-1][0].bias.data.fill_(0.0)

        self.gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=False)
        self.inv_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=True)

        self.hyper_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=False)
        self.hyper_inv_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=True)

    def forward(self, x, n=None, l=None, train=False):
        self.training = train

        y = self.g_a(x)
        scaled_y = self.gain_unit(y, n, l)
        z = self.h_a(scaled_y)
        scaled_z = self.hyper_gain_unit(z, n, l)
        z_hat, z_likelihoods = self.entropy_bottleneck(scaled_z)
        scaled_z_hat = self.hyper_inv_gain_unit(z_hat, n, l)
        gaussian_params = self.h_s(scaled_z_hat)
        scales_hat, means_hat = gaussian_params.chunk(2, 1)
        y_hat, y_likelihoods = self.gaussian_conditional(scaled_y, scales_hat,
means=means_hat)
        scaled_y_hat = self.inv_gain_unit(y_hat, n, l)
        x_hat = self.g_s(scaled_y_hat)

        return {
            "x_hat": x_hat,
            "likelihoods": {"y": y_likelihoods, "z": z_likelihoods},
        }

class ResidualCompressor(MeanScaleHyperprior):

    def __init__(self, n=6, in_ch=3, N=128, bias=False, **kwargs):
        super().__init__(N=N, M=N, **kwargs)

        self.g_a = nn.Sequential(
            ResidualBlockWithStride(in_ch, N, stride=2),
            ResidualBlock(N, N),
            ResidualBlockWithStride(N, N, stride=2),
            ResidualBlock(N, N),
            ResidualBlockWithStride(N, N, stride=2),
            ResidualBlock(N, N),
            conv3x3(N, N, stride=2),
        )

        self.h_a = nn.Sequential(
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N, stride=2),
            nn.LeakyReLU(inplace=True),
```

```
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N, stride=2),
        )

        self.h_s = nn.Sequential(
            conv3x3(N, N),
            nn.LeakyReLU(inplace=True),
            subpel_conv3x3(N, N, 2),
            nn.LeakyReLU(inplace=True),
            conv3x3(N, N * 3 // 2),
            nn.LeakyReLU(inplace=True),
            subpel_conv3x3(N * 3 // 2, N * 3 // 2, 2),
            nn.LeakyReLU(inplace=True),
            conv3x3(N * 3 // 2, N * 2),
        )

        self.g_s = nn.Sequential(
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            ResidualBlockUpsample(N, N, 2),
            ResidualBlock(N, N),
            subpel_conv3x3(N, in_ch, 2),
        )

        self.gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=False)
        self.inv_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=True)

        self.hyper_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=False)
        self.hyper_inv_gain_unit = Gain_Module(n=n, N=N, bias=bias, inv=True)

    def forward(self, x, n=None, l=None, train=False):
        self.training = train

        y = self.g_a(x)
        scaled_y = self.gain_unit(y, n, l)
        z = self.h_a(scaled_y)
        scaled_z = self.hyper_gain_unit(z, n, l)
        z_hat, z_likelihoods = self.entropy_bottleneck(scaled_z)
        scaled_z_hat = self.hyper_inv_gain_unit(z_hat, n, l)
        gaussian_params = self.h_s(scaled_z_hat)
        scales_hat, means_hat = gaussian_params.chunk(2, 1)
        y_hat, y_likelihoods = self.gaussian_conditional(scaled_y, scales_hat,
means=means_hat)
        scaled_y_hat = self.inv_gain_unit(y_hat, n, l)
        x_hat = self.g_s(scaled_y_hat)

        return {
            "x_hat": x_hat,
            "likelihoods": {"y": y_likelihoods, "z": z_likelihoods},
        }
```

8.4. Model Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.ops.deform_conv as df
import time
import math
```

```
from compressai.models import MeanScaleHyperprior
from compressai.models.utils import conv, deconv
from compressai.layers import GDN

from .layers import FlowCompressor, ResidualCompressor
from .unet import UNet

device = torch.device("cuda")

class BidirFlowRef(nn.Module):
    """
    Bidirectional Compression with Flow Refinement
    """
    def __init__(self, n=6, N=128):
        super(BidirFlowRef, self).__init__()

        self.flow_predictor = UNet(in_channels=6, out_channels=4, depth=5, wf=5,
padding=True)

        self.flow_compressor = FlowCompressor(n=n, in_ch=9, out_ch=5, N=N,
bias=False)
        self.residual_compressor = ResidualCompressor(n=n, in_ch=3, N=N,
bias=False)


    def forward(self, x_before, x_current, x_after, n=None, l=1, train=False):
        _, _, H, W = x_current.shape
        num_pixels = H * W

        enc_start = time.perf_counter()

        pred_input = torch.cat((x_before, x_after), dim=1)

        mv_pred = self.flow_predictor(pred_input)

        mv_before, mv_after = torch.chunk(mv_pred, 2, dim=1)

        x_before_pred = self.backwarp(x_before, mv_before)
        x_after_pred = self.backwarp(x_after, mv_after)

        x_input = torch.cat((x_current, x_before_pred, x_after_pred), dim=1)

        flow_result = self.flow_compressor(x_input, n, l, train)
        flow_hat = flow_result["x_hat"]

        dec_start = time.perf_counter()

        mv_before_refined = flow_hat[:, :2, :, :]
        mv_after_refined = flow_hat[:, 2:4, :, :]
        beta = F.sigmoid(flow_hat[:, 4:, :, :])

        x_comp = beta * self.backwarp(x_before_pred, mv_before_refined) + (1 -
beta) * self.backwarp(x_after_pred, mv_after_refined)

        dec_mid = time.perf_counter()
        dec_time = dec_mid - dec_start

        residual = x_current - x_comp

        residual_result = self.residual_compressor(residual, n, l, train)

        enc_end = time.perf_counter()
```

```
        enc_time = enc_end - enc_start

        residual_hat = residual_result["x_hat"]

        dec_mid_start = time.perf_counter()

        x_hat = x_comp + residual_hat

        dec_mid_end = time.perf_counter()
        dec_time += (dec_mid_end - dec_mid_start)

        size_flow = sum(
            (torch.log(likelihoods).sum(dim=(1, 2, 3)) / (-math.log(2)))
            for likelihoods in flow_result["likelihoods"].values()
        )
        rate_flow = size_flow / num_pixels

        size_residual = sum(
            torch.log(likelihoods).sum(dim=(1, 2, 3)) / (-math.log(2))
            for likelihoods in residual_result["likelihoods"].values()
        )
        rate_residual = size_residual / num_pixels

        return {
            "x_hat": x_hat,
            "x_before_pred": x_before_pred,
            "x_after_pred": x_after_pred,
            "mv_before": mv_before,
            "mv_after": mv_after,
            "mv_before_refined": mv_before_refined,
            "mv_after_refined": mv_after_refined,
            "beta": beta,
            "x_before_refined": self.backwarp(x_before_pred, mv_before_refined),
            "x_after_refined": self.backwarp(x_after_pred, mv_after_refined),
            "x_comp": x_comp,
            "residual": residual_hat,
            "size": size_flow + size_residual,
            "rate": rate_flow + rate_residual,
            "enc_time": enc_time,
            "dec_time": dec_time
        }

    def backwarp(self, tenInput, tenFlow):

        tenHor = torch.linspace(-1.0 + (1.0 / tenFlow.shape[3]), 1.0 - (1.0 /
tenFlow.shape[3]),
                                tenFlow.shape[3]).view(1, 1, 1, -1).expand(-1, -1,
tenFlow.shape[2], -1)
        tenVer = torch.linspace(-1.0 + (1.0 / tenFlow.shape[2]), 1.0 - (1.0 /
tenFlow.shape[2]),
                                tenFlow.shape[2]).view(1, 1, -1, 1).expand(-1, -1, -
1, tenFlow.shape[3])

        backwarp_tenGrid = torch.cat([ tenHor, tenVer ], 1).to(device)
        # end

        tenFlow = torch.cat([ tenFlow[:, 0:1, :, :] / ((tenInput.shape[3] - 1.0)
/ 2.0),
                              tenFlow[:, 1:2, :, :] / ((tenInput.shape[2] - 1.0) /
2.0) ], 1)

        return torch.nn.functional.grid_sample(input=tenInput,
                                               grid=(backwarp_tenGrid +
```

```
tenFlow).permute(0, 2, 3, 1),
                                                mode='bilinear',
padding_mode='border', align_corners=False)
```

8.5. Utility Code

```
import torch
from torch import optim
import numpy as np
from natsort import natsorted
import glob
import random
import sys
import imageio
import math
import torch.nn as nn
import logging

def normalize(tensor):
    norm = (tensor) / 255.
    return norm

def float_to_uint8(image):
    clip = torch.clamp(image, 0., 1.) * 255.
    im_uint8 = torch.round(clip).type(torch.uint8)
    return im_uint8

def MSE(gt, pred):
    mse = torch.mean((gt - pred) ** 2)
    return mse

def PSNR(mse, data_range):
    psnr = 10 * torch.log10((data_range ** 2) / mse)
    return psnr

def calculate_distortion_loss(out, real, dim):
    """Mean Squared Error"""
    distortion_loss = torch.mean((out - real) ** 2, dim=dim)
    return distortion_loss

def pad(im):
    """Padding to fix size at validation"""
    (b, c, w, h) = im.size()

    p1 = (64 - (w % 64)) % 64
    p2 = (64 - (h % 64)) % 64

    pad = nn.ReflectionPad2d(padding=(0, p2, 0, p1))
    return pad(im).squeeze(0)

# ### Training & Test Video & Image Datasets

from torch.utils.data import Dataset

def tensor_crop(frames, patch_size, rng):
    """
    Crop frames according to the patch size
    Output is a numpy array
    """
    X_train = []
    sample_im = imageio.imread(frames[0])

    x = rng.randint(0, sample_im.shape[1] - patch_size)
```

```
    y = rng.randint(0, sample_im.shape[0] - patch_size)

    for k in range(len(frames)):

        img = imageio.imread(frames[k])
        img_cropped = img[y:y + patch_size, x:x + patch_size]
        img_cropped = img_cropped.transpose(2, 0, 1)

        if k == 0:
            img_concat = np.array(img_cropped)
        else:
            img_concat = np.concatenate((img_concat, img_cropped), axis=0)

    return img_concat

class VimeoTrainDataset(Dataset):
    """Dataset for custom vimeo"""

    def __init__(self, data_path, patch_size, gop_size, skip_frames, num_frames,
rng, dtype=".png"):
        """
        data_path: path to folders of videos,
        patch_size: size to crop for training,
        gop_size: GoP size,
        skip_frames: do we skip frames (int),
        num_frames: whether we limit the number of frames in the GoP,
        rng: random number generator,
        dype: png or jpeg
        """

        self.data_path = data_path

        # Pick the videos with sufficient resolution
        videos = []
        folders = natsorted(glob.glob(data_path + "*"))
        for folder in folders:
            videos += natsorted(glob.glob(folder + "/*"))

        self.videos = videos

        self.patch_size = patch_size
        self.gop_size = gop_size
        self.skip_frames = skip_frames
        self.dtype = dtype

        # Random number generator for reproducability
        self.rng = rng

        # How many frames to take
        if num_frames:
            self.num_frames = num_frames
        else:
            self.num_frames = (self.gop_size // self.skip_frames) + 1

        self.dataset_size = len(self.videos)

    def __len__(self):
        return self.dataset_size

    def __getitem__(self, item):
        video = self.videos[item]
        video_im_list = natsorted(glob.glob(video + "/*." + self.dtype))
```

```
        length = len(video_im_list)

        s = self.rng.randint(0, length - 1 - (self.num_frames - 1) *
self.skip_frames)
        video_split = video_im_list[s:s + self.skip_frames *
self.num_frames:self.skip_frames]

        video_split = tensor_crop(video_split, self.patch_size, self.rng)
        video_split = normalize(video_split)

        return video_split

class UVGTestDataset(Dataset):
    """Dataset for UVG"""

    def __init__(self, data_path, video_names, gop_size, skip_frames,
test_size=2):
        """
        data_path: path to folders of videos,
        video_name: video name (e.g. beauty),
        skip_frames: do we skip frames (int, e.g. 1),
        """
        # Get the frame paths for each frame
        self.data_path = data_path
        self.skip_frames = skip_frames
        self.gop_size = gop_size
        self.test_size = test_size
        self.frames = []

        for video_name in video_names:
            video = data_path + video_name
            frames = natsorted(glob.glob(video +
"/*.png"))[:test_size*gop_size+1]

            for idx, frame in enumerate(frames):
                self.frames.append(frame)
                if (idx % gop_size == 0) and (idx != 0) and (idx // gop_size !=
test_size):
                    self.frames.append(frame)

        self.dataset_size = len(self.frames)
        self.orig_img_size = imageio.imread(self.frames[0]).shape

    def __len__(self):
        return self.dataset_size

    def __getitem__(self, item):
        frame = self.frames[item]

        im = imageio.imread(frame).transpose(2, 0, 1)
        im = normalize(torch.from_numpy(im)).unsqueeze(0)
        im = pad(im)

        return im

class KodakTestDataset(Dataset):
    """Dataset for Kodak"""

    def __init__(self, data_path):
        """
        data_path: path to folders of videos,
        video_name: video name (e.g. beauty),
        skip_frames: do we skip frames (int, e.g. 1),
```

```
        """
        # Get the frame paths for each frame
        self.data_path = data_path
        self.images = natsorted(glob.glob(self.data_path + "*.png"))

    def __len__(self):
        return len(self.images)

    def __getitem__(self, item):
        im = self.images[item]

        im = imageio.imread(im).transpose(2, 0, 1)
        im = normalize(torch.from_numpy(im))

        return im

# ### I-Frame image compressor

def image_compress(im, compressor):
    out = compressor(im)
    dec = out["x_hat"]
    size_image = sum(
        (torch.log(likelihoods).sum() / (-math.log(2)))
        for likelihoods in out["likelihoods"].values()
    )

    return dec, size_image

# ### Save and load model

def save_model(model, optimizer, aux_optimizer, scheduler, num_iter, exceptions,
save_name="checkpoint.pth"):
    """
    Save a model with its optimizer, aux_optimizer, scheduler and # of iteration
info.
    If some of them are not desired, give None as input instead of it
    """
    save_dict = {}
    if optimizer:
        save_dict["optimizer"] = optimizer.state_dict()
    if aux_optimizer:
        save_dict["aux_optimizer"] = aux_optimizer.state_dict()

    if scheduler:
        save_dict["scheduler"] = scheduler.state_dict()

    if num_iter:
        save_dict["iter"] = num_iter

    for child, module in model.named_children():
        # If we don't want to save a child, we skip it
        if child in exceptions:
            continue
        save_dict[child] = module.state_dict()
        logging.info("Saved " + child + " at " + save_name)

    torch.save(save_dict, save_name)

def load_model(model, pretrained_dict, exceptions):
    """
    Load the model parameters from a dictionary. The dictionary must have key
names same
    as the model attributes (which are submodules). The save_model() function is
```

```
designed
    to be matching with this function.
    """
    model_child_names = [name for name, _ in model.named_children()]

    for name, submodule in pretrained_dict.items():
        # If we don't want to load a module, we skip it
        if name in exceptions:
            continue

        if name in model_child_names:
            message = getattr(model, name).load_state_dict(submodule)
            logging.info(name + ": " + str(message))
    return model

def configure_seeds(random_seed=None, torch_seed=None):
    if random_seed is None:
        random_seed = random.randrange(sys.maxsize)
    if torch_seed is None:
        torch_seed = torch.seed()
    else:
        torch.manual_seed(torch_seed)

    rng = random.Random(random_seed)
    logging.info("Random library seed: " + str(random_seed))
    logging.info("PyTorch library seed: " + str(torch_seed))

    return rng

def configure_optimizers(model, args):
    """Separate parameters for the main optimizer and the auxiliary optimizer.
    Return two optimizers"""
    # Use list of tuples instead of dict to be able to later check the elements
are unique and there is no intersection
    parameters = []
    aux_parameters = []
    parameter_dict = {}
    for name, param in model.named_parameters():
        parameter_dict[name] = param
        if not name.endswith(".quantiles"):
            parameters.append((name, param))
        else:
            aux_parameters.append((name, param))

    aux_param_set = set(p for n, p in aux_parameters)
    num_aux_params = sum([np.prod(p.size()) for p in aux_param_set])

    logging.info("There are " + str(num_aux_params) + " aux_parameters")

    # Make sure we don't have an intersection of parameters
    parameters_name_set = set(n for n,p in parameters)
    aux_parameters_name_set = set(n for n, p in aux_parameters)
    assert len(parameters) == len(parameters_name_set)
    assert len(aux_parameters) == len(aux_parameters_name_set)

    inter_params = parameters_name_set & aux_parameters_name_set
    union_params = parameters_name_set | aux_parameters_name_set
    assert len(inter_params) == 0
    assert len(union_params) - len(parameter_dict.keys()) == 0

    optimizer = optim.Adam((p for (n, p) in parameters if p.requires_grad),
                           lr=args.learning_rate)
    aux_optimizer = optim.Adam((p for (n, p) in aux_parameters if
```

```
p.requires_grad),
                                lr=args.aux_learning_rate)

    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
factor=0.5,
                                                     patience=args.patience,
min_lr=args.min_lr)

    return optimizer, aux_optimizer, scheduler

def load_optimizer(pretrained_dict, device, optimizer, aux_optimizer=None):
    """
    Load the optimizer parameters from a dictionary that was saved using the
save_model()
    function.
    """
    message = optimizer.load_state_dict(pretrained_dict["optimizer"])

    logging.info("Optimizer: " + str(message))

    if aux_optimizer:
        aux_optimizer.load_state_dict(pretrained_dict["aux_optimizer"])

        logging.info("Aux Optimizer: " + str(message))

    return optimizer, aux_optimizer

# ### Info passing during training and validation

class Infographic():
    """
    Build a logging class to save & load the training results
    """
    def __init__(self):
        self.step_train_dist_loss = 0
        self.step_train_rate_loss = 0
        self.step_train_loss = 0
        self.avg_psnr_dec = 0
        self.avg_bpp = 0
        self.avg_val_loss = 0

        self.best_val_loss = 10**10
        self.psnr_dec_at_best_loss = -1
        self.bpp_at_best_loss = -1

    def update_train_info(self, distortion_loss, rate_loss, loss):
        self.step_train_dist_loss += distortion_loss
        self.step_train_rate_loss += rate_loss
        self.step_train_loss += loss

    def zero_train_info(self):
        self.step_train_dist_loss = 0
        self.step_train_rate_loss = 0
        self.step_train_loss = 0

    def update_val_info(self, avg_val_loss, avg_psnr, avg_bpp):
        self.avg_val_loss = avg_val_loss
        self.avg_psnr_dec = avg_psnr
        self.avg_bpp = avg_bpp

    def update_best_val_info(self):
        self.best_val_loss = self.avg_val_loss
```

```
        self.psnr_dec_at_best_loss = self.avg_psnr_dec
        self.bpp_at_best_loss = self.avg_bpp
```

8.6. Training Code

```
import torch
import numpy as np
import os
import sys
import warnings

warnings.filterwarnings('ignore')
import imageio

from compressai.zoo import mbt2018_mean

try:
    import wandb
    wandb_exist = True
except ImportError:
    wandb_exist = False

import argparse
import logging
import time

from torch.utils.data import DataLoader, RandomSampler

model_path = os.path.abspath('..')
sys.path.insert(1, model_path)

from model import b_model
from utils import float_to_uint8, MSE, PSNR, calculate_distortion_loss
from utils import VimeoTrainDataset, UVGTestDataset
from utils import image_compress, save_model, load_optimizer
from utils import configure_seeds, configure_optimizers
from utils import load_model, Infographic

# Argument parser
parser = argparse.ArgumentParser()

# Hyperparameters, paths and settings are given
# prior the training and validation
parser.add_argument("--project_name", type=str, default="ICIP2022")   # Project
name
parser.add_argument("--model_name", type=str,
default="BidirRefinement_finetune_logloss")       # Model name
parser.add_argument("--random_seed", type=int, default=None) # Get the seeds if
available
parser.add_argument("--torch_seed", type=int, default=None)
parser.add_argument("--train_path", type=str,
default="/datasets/vimeo_septuplet/sequences/")   # Dataset paths
parser.add_argument("--val_path", type=str,
default="/scratch/users/ecetin17/UVG/full_test/")
parser.add_argument("--total_train_step", type=int, default=200000) # # of total
iterations
parser.add_argument("--train_step", type=int, default=5000) # # of iterations
for recording
parser.add_argument("--learning_rate", type=float, default=1.e-5)# learning rate
parser.add_argument("--aux_learning_rate", type=float, default=1.e-3)
parser.add_argument("--min_lr", type=float, default=1.e-7)       # min. learning
rate
parser.add_argument("--patience", type=int, default=20)          # scheduler
patience
```

```
parser.add_argument("--batch_size", type=int, default=4)          # Batch size
parser.add_argument("--patch_size", type=int, default=256)        # Train patch
sizes
parser.add_argument("--train_gop_size", type=int, default=8)      # Train gop
sizes
parser.add_argument("--train_num_frames", type=int, default=3)    # Train number
of frames
parser.add_argument("--train_skip_frames", type=int, default=2)   # Train number
of frames skipped

parser.add_argument("--val_gop_size", type=int, default=8)        # Val gop sizes
parser.add_argument("--val_numbers", type=int, default=1)         # How many
times to validate on a video
parser.add_argument("--val_skip_frames", type=int, default=1)     # Val number of
frames skipped

parser.add_argument("--N_b", type=int, default=128)               # Number of
channels for B-comp, N
parser.add_argument("--b_save_dir", type=str, default="../BidirRef.pth")  # Save
file for the bidir. compressor

parser.add_argument("--device", type=str, default="cuda")         # device "cuda"
or "cpu"
parser.add_argument("--workers", type=int, default=4)             # number of
workers

# We don't take any pretrained models for initial trainings (except I-frame
compressor and optical flow, which are loaded in the code)
parser.add_argument("--pretrained", type=str, default="../BidirRef.pth")
# Load model from this file

parser.add_argument("--cont_train", action='store_true', default=False)
# load optimizer
parser.add_argument("--wandb", action='store_true', default=False)
# Store results in wandb
parser.add_argument("--log_results", action='store_false', default=True)
# Store results in log

args = parser.parse_args()
args.save_name = args.model_name

logging.basicConfig(filename= args.save_name + ".log", level=logging.INFO)

rng = configure_seeds(args.random_seed, args.torch_seed)
device = torch.device(args.device)

# CompressAI trade-off values (For each trade-off, we pick one above I-
compressor quality mbt2018_mean)
args.betas_mse = torch.tensor([0.0067*(255**2), 0.0250*(255**2),
0.0483*(255**2),
                               0.0932*(255**2)]).to(device)
args.num_i = (5, 6, 7, 8)                          # beta for rate-distortion
trade-off
args.levels = args.betas_mse.shape[0]              # Number of points on rate-
distortion curve

coding_order = [0, 8, 4, 2, 1, 3, 6, 5, 7]         # Frame order for decoding
# prev_frame, future_frame, frame_level
decoding_info = {4: [0, 8], 2: [0, 4], 1: [0, 2], 3: [2, 4], 6: [4, 8], 5: [4,
6], 7: [6, 8]}

# ### Training Function
```

```
def train_one_step(im_batch, model, im_models, optimizer, aux_optimizer, betas,
device):
    """
    im_batch: video frames of shape (b, c * gop_size, h, w)
    model: B-frame compressor model
    im_models: Image compressor models
    optimizer: Optimizer of the model
    aux_optimizer: Auxiliary optimizer for the entropy model
    betas: Rate-distortion tradeoff (distortion coeff.)
    device: cuda or cpu
    """

    x0 = im_batch[:, 0:3]
    x1 = im_batch[:, 3:6]
    x2 = im_batch[:, 6:9]

    # Losses for each layer
    dist_loss = 0
    rate_loss = 0

    dec0, dec2 = {}, {}
    streams = {}

    for i in range(args.levels):
        streams[i] = torch.cuda.Stream(device=device)

    level_b = torch.arange(0, args.levels).to(device)

    level_i = torch.clamp(
        level_b + torch.clamp(
            torch.abs(torch.round(torch.randn(args.levels).to(device))), max=2
        ), min=0, max=args.levels-1
    ).long()

    with torch.no_grad():
        torch.cuda.synchronize()

        with torch.cuda.stream(streams[0]):
            dec0[0] = im_models[level_i[0]](x0[0].unsqueeze(0))["x_hat"]
            dec2[0] = im_models[level_i[0]](x2[0].unsqueeze(0))["x_hat"]

        with torch.cuda.stream(streams[1]):
            dec0[1] = im_models[level_i[1]](x0[1].unsqueeze(0))["x_hat"]
            dec2[1] = im_models[level_i[1]](x2[1].unsqueeze(0))["x_hat"]

        with torch.cuda.stream(streams[2]):
            dec0[2] = im_models[level_i[2]](x0[2].unsqueeze(0))["x_hat"]
            dec2[2] = im_models[level_i[2]](x2[2].unsqueeze(0))["x_hat"]

        with torch.cuda.stream(streams[3]):
            dec0[3] = im_models[level_i[3]](x0[3].unsqueeze(0))["x_hat"]
            dec2[3] = im_models[level_i[3]](x2[3].unsqueeze(0))["x_hat"]

        torch.cuda.synchronize(device=device)

        dec0 = torch.cat(tuple(dec0.values()), dim=0)
        dec2 = torch.cat(tuple(dec2.values()), dim=0)

    output = model(
        x_before=dec0,
        x_current=x1,
        x_after=dec2,
        n=level_b,
```

```
        l=1,
        train=True
    )

    dist_loss = betas * calculate_distortion_loss(output["x_hat"], x1, dim=(1,
2, 3))
    rate_loss = output["rate"]
    loss = dist_loss + rate_loss

    loss = torch.exp(torch.mean(torch.log(loss)))
    dist_loss = torch.exp(torch.mean(torch.log(dist_loss)))
    rate_loss = torch.exp(torch.mean(torch.log(rate_loss)))

    # AUXILIARY LOSS
    aux_loss = (model.flow_compressor.aux_loss() +
model.residual_compressor.aux_loss())

    optimizer.zero_grad()
    aux_optimizer.zero_grad()

    loss.backward()
    aux_loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    optimizer.step()
    aux_optimizer.step()

    return dist_loss.item(), rate_loss.item(), loss.item()

# ### Validation Function

def validate(model, im_models, betas, device, args):
    """
    test_loader: Test loader for UVG
    model: B-frame compressor model
    im_models: Image compressor model
    alpha: Rate-distortion tradeoff (distortion coeff.)
    device: cuda or cpu
    """
    with torch.no_grad():
        coding_order_eff = coding_order[2:]

        rate_loss = 0
        dist_loss = 0
        total_loss = 0

        folder_names = ["beauty", "bosphorus", "honeybee", "jockey", "ready",
"shake", "yatch"]

        psnr_dict = {k: 0 for k in range(args.levels)}
        size_dict = {k: 0 for k in range(args.levels)}
        frame_num_dict = {k: 0 for k in range(args.levels)}
        pixel_num_dict = {k: 0 for k in range(args.levels)}

        test_dataset = UVGTestDataset(
            args.val_path,
            folder_names,
            gop_size=args.val_gop_size,
            skip_frames=args.val_skip_frames,
            test_size=args.val_numbers
        )
```

```
        h, w, _  = test_dataset.orig_img_size

        # To adjust the bidirectional scheme, we increase the batch size by 1
        test_loader = DataLoader(test_dataset, batch_size=args.val_gop_size+1,
shuffle=False, num_workers=args.workers)

        decoded = {}

        # Loading videos in batches of form I-B-B-B-B-B-B-B-I
        for idx, gop in enumerate(test_loader):
            gop = gop.unsqueeze(1).to(device)
            # _, _, h, w = gop[0].shape

            for level in range(args.levels):

                # If first batch of video, we compress the first frame with I-
compressor
                if idx % args.val_numbers == 0:
                    decoded[level] = {}
                    dec0 = im_models[level](gop[0])
                    decoded[level][0] = dec0["x_hat"]

                dec_last = im_models[level](gop[-1])
                decoded[level][coding_order[1]] = dec_last["x_hat"]

                for order in coding_order_eff:
                    output = model(
                        x_before=decoded[level][decoding_info[order][0]],
                        x_current=gop[order],
                        x_after=decoded[level][decoding_info[order][1]],
                        n=[level],
                        l=1,
                        train=False
                    )
                    decoded[level][order] = output["x_hat"]

                    cur_dist_loss = betas[level] *
calculate_distortion_loss(output["x_hat"], gop[order], dim=(0, 1, 2, 3))
                    cur_rate_loss = output["rate"].squeeze(0)
                    cur_loss = torch.exp(torch.mean(torch.log(cur_dist_loss +
cur_rate_loss)))
                    total_loss += cur_loss

                    uint8_real = float_to_uint8(gop[order][0, :, :h, :w])
                    uint8_dec_out = float_to_uint8(output["x_hat"][0, :, :h,
:w])

                    cur_psnr = PSNR(
                        MSE(uint8_dec_out.type(torch.float),
uint8_real.type(torch.float)),
                        data_range=255
                    )

                    psnr_dict[level] += cur_psnr
                    size_dict[level] += output["size"].squeeze(0)
                    frame_num_dict[level] += 1
                    pixel_num_dict[level] += uint8_real.shape[1] *
uint8_real.shape[2]

                decoded[level] = {0: dec_last}

    total_frames = sum(frame_num_dict.values(), 0.0)
    total_pixels = sum(pixel_num_dict.values(), 0.0)
```

```
    total_size = sum(size_dict.values(), 0.0)
    total_psnr = sum(psnr_dict.values(), 0.0)

    average_bpp_dict = {k: (v / pixel_num_dict[k]).item() for k, v in
size_dict.items()}
    average_psnr_dict = {k: (v / frame_num_dict[k]).item() for k, v in
psnr_dict.items()}

    average_psnr = total_psnr / total_frames
    average_loss = total_loss / total_frames
    average_bpp = total_size / total_pixels

    return average_loss.item(), average_psnr.item(), average_bpp.item(),
average_psnr_dict, average_bpp_dict

# ### Main Function
# We just train the b-coding model

def main(args):

    if args.wandb and wandb_exist:
        wandb.init(
            project=args.project_name,
            name=args.model_name,
            config=vars(args)
        )

    image_compressors = [mbt2018_mean(q, "mse",
pretrained=True).to(device).float()
                         for q in args.num_i]

    for idx, image_compressor in enumerate(image_compressors):
        for param in image_compressor.parameters():
            param.requires_grad = False

        image_compressors[idx] = image_compressor.eval()

    # Build the model
    model = b_model.BidirFlowRef(n=args.levels, N=args.N_b).to(device).float()

    infographic = Infographic()

    if args.pretrained:
        checkpoint = torch.load(args.pretrained, map_location=device)
        model = load_model(model, checkpoint, exceptions=[])

        model = model.eval()

        avg_val_loss, avg_psnr, avg_bpp, avg_psnr_dict, avg_bpp_dict = validate(
            model=model,
            im_models=image_compressors,
            betas=args.betas_mse,
            device=device,
            args=args
        )

        infographic.update_val_info(avg_val_loss, avg_psnr, avg_bpp)
        infographic.update_best_val_info()
        logging.info(f"Initial validation loss: {avg_val_loss}")
        logging.info(f"Initial PSNR: {avg_psnr}")
        logging.info(f"Initial bpp: {avg_bpp}")
        logging.info("Initial Best Validation loss: " +
str(infographic.best_val_loss))
```

```
        logging.info("Initial PSNR at best Validation loss: " +
str(infographic.psnr_dec_at_best_loss))
        logging.info("Initial bpp at best Validation loss: " +
str(infographic.bpp_at_best_loss))

        infographic = Infographic()

    optimizer, aux_optimizer, scheduler = configure_optimizers(model, args)

    # If we want to continue training using a checkpoint we load the optimizers
& scheduler
    if args.cont_train:
        optimizer, aux_optimizer = load_optimizer(
                                        checkpoint=checkpoint,
                                        device=device,
                                        optimizer=optimizer,
                                        aux_optimizer=aux_optimizer
                                    )
        scheduler.load_state_dict(checkpoint["scheduler"])

    model_parameters = filter(lambda p: p.requires_grad, model.parameters())
    params = sum([np.prod(p.size()) for p in model_parameters])

    if args.wandb and wandb_exist:
        wandb.config.update({"Num. params": params})

    if args.log_results:
        logging.info("Num. params: " + str(params))

    train_dataset = VimeoTrainDataset(
                        args.train_path,
                        patch_size=args.patch_size,
                        gop_size=args.train_gop_size,
                        skip_frames=args.train_skip_frames,
                        num_frames=args.train_num_frames,
                        rng=rng,
                        dtype="png"
                    )
    train_sampler = RandomSampler(train_dataset, replacement=True)

    time_start = time.perf_counter()

    # If we want to continue training using a checkpoint we load the number of
iterations
    if args.cont_train:
        iteration = checkpoint["iter"]
    else:
        iteration = 0

    model = model.train()

    while iteration <= args.total_train_step:
        train_loader = DataLoader(train_dataset, batch_size=args.batch_size,
sampler=train_sampler, num_workers=args.workers, drop_last=True)

        for gop_im_batch in train_loader:

            dist_loss, rate_loss, loss = train_one_step(
                im_batch=gop_im_batch.to(args.device).float(),
                model=model,
                im_models=image_compressors,
                optimizer=optimizer,
                aux_optimizer=aux_optimizer,
```

```
                betas=args.betas_mse,
                device=device
            )

            infographic.update_train_info(dist_loss, rate_loss, loss)
            iteration += 1

            if iteration % args.train_step == 0:

                model = model.eval()

                avg_val_loss, avg_psnr, avg_bpp, avg_psnr_dict, avg_bpp_dict =
validate(
                    model=model,
                    im_models=image_compressors,
                    betas=args.betas_mse,
                    device=device,
                    args=args
                )

                infographic.update_val_info(avg_val_loss, avg_psnr, avg_bpp)

                scheduler.step(avg_val_loss)
                learning_rate = optimizer.param_groups[0]["lr"]

                time_end = time.perf_counter()
                duration = time_end - time_start

                if avg_val_loss < infographic.best_val_loss:
                    # Save every submodule of the model separately
                    save_model(
                        model=model,
                        optimizer=optimizer,
                        aux_optimizer=aux_optimizer,
                        scheduler=scheduler,
                        num_iter=iteration,
                        exceptions=[],
                        save_name= args.b_save_dir
                    )

                    infographic.update_best_val_info()

                    if args.log_results:
                        logging.info("********** NEW BEST! **********")

                # Log to wandb if supported
                if args.wandb and wandb_exist:
                    wandb_dict = {
                        "Time": duration,
                        "Learning rate": learning_rate,
                        "Distortion loss": infographic.step_train_dist_loss /
args.train_step,
                        "Rate loss": infographic.step_train_rate_loss /
args.train_step,
                        "Train loss": infographic.step_train_loss /
args.train_step,
                        "Validation PSNR": infographic.avg_psnr_dec,
                        "Validation bpp": infographic.avg_bpp,
                        "Validation loss": infographic.avg_val_loss,
                        "Best Validation loss": infographic.best_val_loss,
                        "PSNR at best Validation loss":
infographic.psnr_dec_at_best_loss,
                        "bpp at best Validation loss":
```

```
infographic.bpp_at_best_loss,
                    }
                    for k in avg_psnr_dict.keys():
                        wandb_dict[f"Level {k} PSNR"] = avg_psnr_dict[k]
                        wandb_dict[f"Level {k} bpp"] = avg_bpp_dict[k]

                    wandb.log(wandb_dict, step=iteration)

                # Log to logfile if wanted
                if args.log_results:
                    logging.info("Iteration: " + str(iteration))
                    logging.info("Time: " + str(duration))
                    logging.info("Learning rate: " + str(learning_rate))
                    logging.info("Distortion loss: " +
str(infographic.step_train_dist_loss / args.train_step))
                    logging.info("Rate loss: " +
str(infographic.step_train_rate_loss / args.train_step))
                    logging.info("Train loss: " +
str(infographic.step_train_loss / args.train_step))
                    logging.info("Validation PSNR: " +
str(infographic.avg_psnr_dec))
                    logging.info("Validation bpp: " + str(infographic.avg_bpp))
                    logging.info("Validation loss: " +
str(infographic.avg_val_loss))
                    logging.info("Best Validation loss: " +
str(infographic.best_val_loss))
                    logging.info("PSNR at best Validation loss: " +
str(infographic.psnr_dec_at_best_loss))
                    logging.info("bpp at best Validation loss: " +
str(infographic.bpp_at_best_loss))
                    logging.info("---------------------------------")
                    logging.info("-- PSNR per level --")
                    for k, v in avg_psnr_dict.items():
                        logging.info("Level " + str(k) + " PSNR: " + str(v))
                    logging.info("-- bpp per level --")
                    for k, v in avg_bpp_dict.items():
                        logging.info("Level " + str(k) + " bpp: " + str(v))
                    logging.info("*********************************")

                infographic.zero_train_info()

                # Take the model back into training mode
                model = model.train()
                time_start = time.perf_counter()

            if iteration >= args.total_train_step:
                break

if __name__ == '__main__':
    main(args)
```