ABSTRACT

Title of Dissertation: An Efficient Neural Representation for Videos

Hao Chen, Doctor of Philosophy, 2023

Dissertation Directed by: Abhinav Shrivastava
Department of Computer Science

With the increasing popularity of videos, it has become crucial to find efficient and compact ways to represent them for easier storage, transmission, and downstream video tasks. Our dissertation proposes an innovative neural representation for videos called NeRV, which stores each video implicitly as a neural network. Building on NeRV, we introduce a hybrid representation for videos called HNeRV, which improves internal generalization and representation capacity. HNeRV allows for highly efficient video representation and compression, with a model size that can be up to 1000 times smaller than the original raw video.

Apart from efficiency, HNeRV's simple decoding process, which involves a feedforward operation, enables fast video loading and easy deployment. To enhance efficiency, we develope an efficient neural video dataloader called NVLoader, which is 3-6 times faster than conventional video dataloaders. We also introduce the HyperNeRV framework to address encoding speed, which utilizes a hypernetwork to directly map input videos to NeRV model weights, resulting in a 10^4 faster encoding process.

Aside from developing compact and implicit video neural representations, we explore several compelling applications, including frame interpolation, video restoration, and video editing. Furthermore, the compactness of these representations makes them an ideal output video format for video generation models, reducing the search space significantly. Additionally, they can serve as an efficient input for video understanding models.

An Efficient Neural Representation For Videos

by

Hao Chen

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:

Prof. Abhinav Shrivastava, Unviersity of Maryland (Chair/Advisor)
Prof. Furong Huang, Unviersity of Maryland
Prof. Behtash Babadi, Unviersity of Maryland
Prof. Ramani Duraiswami, Unviersity of Maryland
Prof. Saining Xie, NewYork University

Acknowledgments

I am deeply grateful to my advisor, Abhinav Shrivastava, for his exceptional guidance throughout my Ph.D. journey. His extensive knowledge and problem-solving skills have not only inspired me but also shaped my research in countless ways. I have been fortunate to work under his patient and kind mentorship, which has been invaluable in helping me navigate the challenges of research and life. Abhinav once said, "Hackers are people with the cleverest mind and always know how to solve problems," and he is a true embodiment of this definition. His wisdom and ability to solve even the most complex issues have been a constant source of motivation for me. I am thankful for everything Abhinav has taught me, and I aspire to become half as good as him through years of learning and practice. Abhinav's guidance and support have been critical to my success, and I will always be grateful for his exceptional mentorship.

I am incredibly grateful to the esteemed members of my thesis committee, Furong Huang, Behtash Babadi, Ramani Duraiswami, and Saining Xie, for the time and effort that they have invested in me. Ramani Duraiswami, who served on both my proposal and defense committee, demonstrated exceptional kindness and support, for which I am deeply appreciative. Saining Xie, a distinguished young researcher in computer vision, has been a source of inspiration for my research. I feel incredibly fortunate to have had the opportunity to collaborate with him on the efficient neural video dataloader project. Finally, I would like to express my gratitude to Christopher Metzler for attending my preliminary exam and offering valuable feedback and

constructive comments on my research.

I am grateful for my labmates, who have not only been great collaborators but also amazing friends. Yixuan Ren, Bo He, Hanyu Wang, Kamal Gupta, Max Enrich, Matthew Gwilliam, Alex Hanson, Shuaiyi Huang, Lillian Huang, Vatsal Agarwal, Sharath Girish, Chuong Huynh, Vinoj Jayasundara, Pulkit Kumar, Mara Levy, Shishira R Maiya, Soumik Mukhopadhyay, Khoi Pham, Nirat Saini, Saksham Suri, Archana Swaminathan, Gaurav Shrivastava, Matthew Walmer, and Luyu Yang have all contributed to my growth as a researcher and a person. I have learned a great deal from them, not only in the research but also in the way they approach life. In particular, I want to express my appreciation to Bo He, Hanyu Wang, Matthew Gwilliam, and Yixuan Ren. Their support and contributions have been invaluable to the success of the NeRV-series.

During my time at UMD, I had the privilege of collaborating with exceptional researchers, and I am grateful for these opportunities. Zuxuan Wu has been an outstanding mentor, providing invaluable guidance not only in research but also in career planning. Additionally, working with Xitong Yang and receiving guidance from Hengduo Li and Chen Zhu has been a great experience. I would also like to express my gratitude to the mentors I had during my internships. Ser-Nam Lim from Meta, Zhe Lin from Adobe, and Heng Wang, Binchen Liu, and Yizhe Zhu from Tiktok have all provided me with indispensable insights that have shaped my research. Their mentorship and support have been instrumental in my growth as a researcher.

I would like to express my gratitude to the exceptional individuals who ignited my passion for deep learning and computer vision. Firstly, I am indebted to Prof. Guoyou Wang, my first advisor during my Master's degree, and Prof. Xiang Bai, who provided invaluable guidance and career advice during my study abroad. Additionally, I am thankful for the wonderful collaborators and mentors I had the privilege of working with during my internship at the Shenzhen Institutes

of Advanced Technology (SIAT), including Prof. Yu Qiao, Prof. Yali Wang, Lei Zhou, Yulun Zhang, Yapeng Tian, Zhi Tian, Tong He, Kaiyang Zhou, and Xiaoxing Zeng.

Finally, I want to express my deep appreciation to my parents and all my family members, especially my adorable nephew Jiujiu, for their unwavering support throughout my journey. Without their support, none of this would have been possible.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In today's world, video has emerged as the dominant form of multimedia, and its popularity continues to increase. However, the high-dimensional and intricate visual information in videos makes it challenging to efficiently represent them for storage, transmission, and downstream video-related tasks. To tackle this issue, various attempts have been made, with the transform encoding approach being the most popular.

This approach transforms the input video into a compact embedding space that is much smaller than the original video, while maintaining high fidelity after reconstruction. We illustrate the framework in Figure 1.1. These methods can be broadly divided into two categories based on the chosen transform functions: traditional codecs with hand-crafted transforms and learning-based methods that employ deep neural networks. Traditional video compression methods like MPEG [7], H.264 [8], and H.265 [9] achieve good reconstruction results with decent decompression speeds. In contrast, learning-based methods [10, 11, 12, 13, 14, 15, 16] focus on replacing the entire compression pipeline or several components with deep learning tools, at varying levels of complexity.

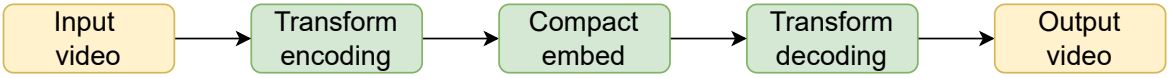


Figure 1.1: Framework of transform encoding for video compression.

Despite efforts to improve video compression, traditional codecs and learning-based methods

both have limitations. Traditional codecs often have suboptimal compression performance, while learning-based methods can be computationally expensive. As a result, a new approach is needed that can combine the strengths of both methods to enhance video compression. Recent approaches have tried to address this challenge by fine-tuning traditional codecs [17] and optimizing components of the compression pipeline [18].

This thesis aims to develop efficient implicit neural representations for videos (NeRV), where each video is represented as a deep neural network that can output the corresponding video frame given a frame index as input. Such implicit representations are appealing because they can represent a video with significantly fewer parameters and reconstruct it with high fidelity, effectively converting the video compression problem into a model compression problem. Building on NeRV, we propose a hybrid neural representation for videos (HNeRV) through a small frame embedding with a powerful decoder network, resulting in improved internal generalization and representation capacity. With evenly distributed model parameters across layers, HNeRV significantly improves convergence speed compared to NeRV.

To evaluate the efficiency of neural representation methods for videos, we consider four key perspectives, as depicted in Figure 1.2. First and foremost, the compression ratio is the most critical metric to assess the efficiency of video representations. Additionally, the encoding speed to convert the original video to efficient representations and the decoding speed to reconstruct the video from such representations should be considered. Finally, an important but often overlooked aspect is the utilization of efficient video representations in downstream video tasks. While most current approaches still rely on the original frame sequences as input, these high-dimensional sequences significantly increase the computation burden for video-related tasks such as video understanding and generation.

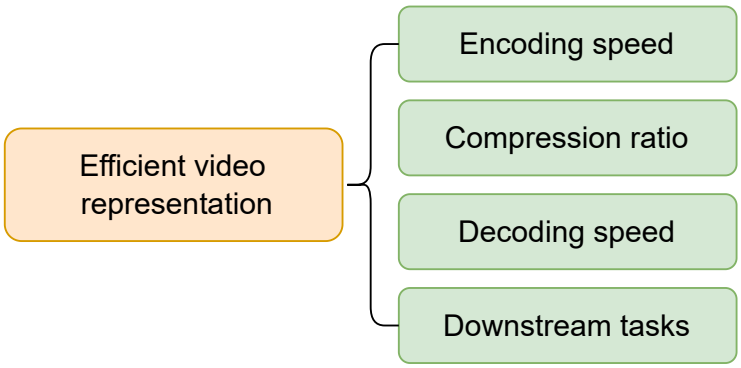


Figure 1.2: Key evaluation metrics for efficient video representations.

Firstly, the use of implicit neural representations enables us to transform the video compression problem into a model compression problem. This approach allows us to achieve comparable compression ratios to other compression methods, with our methods showing superior compression ratios for videos with still backgrounds. In addition to compression, our implicit representations also provide a decoding advantage, as only a small neural network is required to fit one video. Moreover, the simple forward pass decoding operation of HNeRV allows for easy deployment on any platform. To further enhance the efficiency of our methods, we developed an efficient neural video dataloader (NVLoader) that is approximately three times faster than conventional video dataloaders. This faster processing speed enables more efficient training and evaluation of video models.

In addition to compression and decoding speed, encoding speed remains a significant challenge for implicit neural representations due to the long and tedious training process. To address this issue, we introduce the HyperNeRV framework, which utilizes a hypernetwork to directly map input videos to NeRV model weights. This approach significantly speeds up the encoding process by approximately 10^4 times, while achieving similar reconstruction quality and generalization to unseen videos compared to training the neural network from scratch.

Besides developing efficient implicit video representations and proposing the HyperNeRV framework, we explore several downstream applications based on these representations. Due to their compactness and efficiency, we have found that they perform well for tasks such as frame interpolation, video restoration, and video editing. Furthermore, we believe that these compact and implicit video representations have even more potential to be utilized in various other applications. For instance, they can be an ideal output video format that significantly reduces the searching space or serve as an efficient input for video understanding models. These representations can also be employed in diverse other applications, such as video summarization, action recognition, and content-based video retrieval, which we believe require further investigation in future research

We present a summary of our dissertation framework in Figure 1.3. Firstly, we introduce NeRV, an implicit neural representation for video, in Figure 1.3a. We then introduce HNeRV, a content-adaptive embedding that represents video as hybrid ones, in Figure 1.3b. Next, we introduce HyperNeRV in Figure 1.3c, which enables fast learning of video neural representations. Finally, we list different downstream tasks in Figure 1.3d which use the efficient video representations directly, i.e., the model weights. The ultimate goal of these video neural representations is to introduce a new perspective on video processing, similar to the Fourier transform for signal processing. By converting video into neural space, we can greatly advance the research and utilization of video data.

The following dissertation consists of 6 chapters, each of which is explained separately. In Chapter 2, we provide a brief overview of the background knowledge relevant to this dissertation, including three main topics: the efficient video presentations, implicit neural representations, and deep neural networks. Specifically, we delve into the evolution of video compression methods

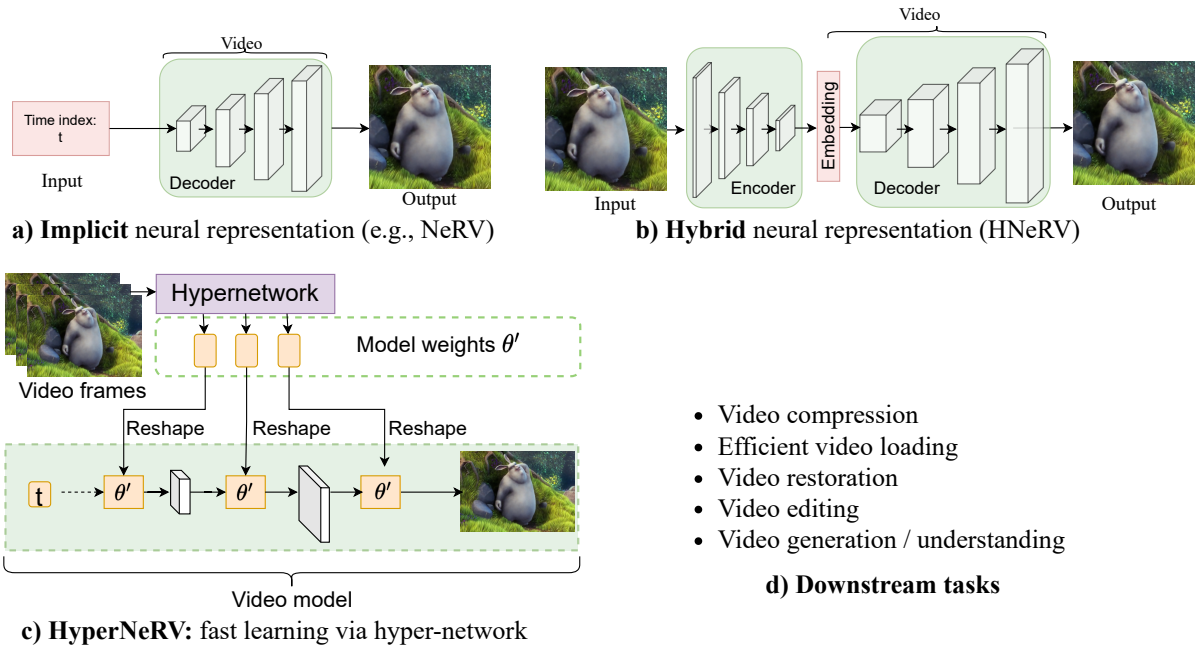


Figure 1.3: The dissertation framework. **a)** implicit neural representation NeRV. **b)** hybrid neural representation HNeRV. **c)** fast learning of NeRV weights. **d)** downstream tasks based on NeRV.

and their applications in everyday life to provide a comprehensive understanding of the current state-of-the-art techniques in the field.

In Chapter 3, we introduce an implicit neural representation for videos called NeRV. We describe a novel image-wise approach where the neural network outputs one frame given a frame index as input. Compared to previous pixel-wise representations that output one pixel at a time, NeRV significantly improves the encoding/decoding speed and the quality of reconstructed videos. In addition to the basic video reconstruction task, we also present results for video compression and video denoising.

In Chapter 4, we present a hybrid neural representation for videos (HNeRV). We replace the content-agnostic frame index input with a content-adaptive embedding generated by an encoder. This change results in video data being represented by two parts: a large video decoder network and a small frame embedding. This hybrid representation improves internal generalization, such

as video interpolation in the embedding space, and reconstruction capacity. Additionally, we propose an evenly-distributed model where the model parameters are distributed more evenly than in NeRV, leading to significant improvements in reconstruction capacity. We also explore video interpolation and video inpainting in this chapter.

In Chapter 5, we present an efficient neural video dataloader (NVLoader) that accelerates the typical data loading process for video research. Essentially, for each video in a dataset, we first fit a compact HNeRV model to it and save the model checkpoint. To load the video during training or testing, we simply load the model checkpoint and generate the video frames via a straightforward feed-forward operation. Because of the simplicity of our NVLoader, it can be easily deployed on any device, and it improves the video loading speed by 3-6 times compared to traditional data loading approaches.

In Chapter 6, we introduce the HyperNeRV framework, which uses a transformer hypernetwork to generate model weights directly. We train this hypernetwork on a large-scale video dataset to learn the mapping function between input video and model weights. With this approach, given a new video, the well-trained hypernetwork can output the model weights directly, eliminating the need for a tedious fitting process. As a result, HyperNeRV can speed up the encoding process by around 10^4 times compared to training the video model from scratch.

In Chapter 7, we provide a summary of potential downstream tasks based on neural video representations, including video compression, video restoration, efficient video loading, video understanding, and video generation. Furthermore, we summarize and conclude the dissertation by highlighting the contributions of each chapter and discussing potential future directions for research in this area.

Chapter 2: Background

We provide background knowledge in Chapter 2 for efficient video representations and their applications. Specifically, in Section 2.1, we discuss the evolution of video compression methods and their potential applications in various domains. In addition, we provide background knowledge for implicit neural representations in Section 2.2 and for deep neural networks in Section 2.3.

2.1 Video compression

We present the video compression pipeline in Figure 2.1, which is designed to decrease the video storage demand or speed up transmission. This is essential since the original video size can often be too large for storage and transmission purposes.

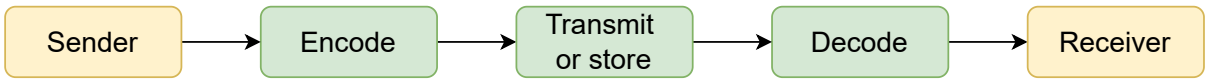


Figure 2.1: Framework of video compression.

2.1.1 Video redundancy overview

For video compression, there are three types of redundancy to remove: spatial redundancy, temporal redundancy, and perceptual redundancy. We illustrate them in Figure 2.2.

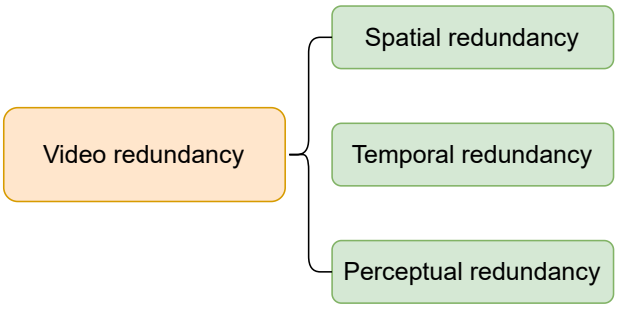


Figure 2.2: Redundancy for video compression.

The term **spatial redundancy** describes the occurrence of similar or identical information in adjacent pixels or regions within a single frame of a video. This leads to unnecessary data that can be compressed without significant loss of quality. In other words, when neighboring pixels or areas in a frame contain the same or similar information, it creates redundant data that can be removed.

In contrast, **temporal redundancy** refers to the redundancy between consecutive frames in a video sequence. Because adjacent frames in a video sequence are often very similar, much of the information in one frame can be predicted from the previous frame. Therefore, in video compression, temporal redundancy can be exploited by transmitting only the differences between frames rather than transmitting each frame entirety.

The human visual system's sensitivity to different aspects of a video signal is not uniform, which leads to **perceptual redundancy**. Thus, video compression algorithms can selectively reduce the information in less perceptually important areas. For instance, compressing a low-frequency color channel may not have a significant impact on overall image quality compared to compressing a high-frequency detail channel. By exploiting perceptual redundancy, video compression algorithms can significantly decrease the amount of data required to represent a video signal without significantly affecting perceptual quality.

While our dissertation primarily focuses on leveraging spatial and temporal redundancy to develop an efficient neural representation and a simple compression method for video data, we acknowledge that addressing perceptual redundancy based on our approach can lead to additional improvements and enhancements.

2.1.2 Image compression

Video compression is primarily based on image compression because of the spatial redundancy in video data. In order to address this issue, this section explores spatial redundancy and the background of image compression. The widely used image compression standard, JPEG [19], divides the input image into non-overlapping 8×8 blocks that are transformed into the frequency domain using block-DCT [20]. The transformed blocks' DCT coefficients are then compressed into a binary stream using quantization and entropy coding. The JPEG standard provides the essential transform and prediction modules for traditional visual compression.

Block-based image and video coding standards suffer from block-dependent compression, which limits parallelism on platforms like GPUs. Furthermore, the independent optimization strategy for each coding tool restricts performance improvement compared to end-to-end optimization compression. An alternative technological development trajectory, based on neural network techniques for image and video compression, is emerging. The resurgence of neural networks has significantly advanced traditional image and video compression by leveraging Convolutional Neural Networks (CNNs).

The first approach was proposed by Cui et al.[21], using an intra-prediction convolutional neural network (IPCNN) to refine the prediction of the current block by leveraging neighboring

reconstructed blocks as additional context. Li et al.[22] proposed a fully connected network (IPFCN) as a new intra prediction mode, which achieved obvious bitrate savings but at the cost of extremely high complexity. Li et al. also explored using CNN-based down/up-sampling techniques as a new intra prediction mode for HEVC, which achieved coding gains, particularly at low bitrates. Additionally, several attempts have been made for CNN-based chroma intra prediction, such as [23, 24], by utilizing both the reconstructed luma block and neighboring chroma blocks to improve intra chroma prediction efficiency.

The image and video compression community has taken a step forward by introducing an end-to-end optimization framework based on deep neural networks. Deep neural networks have been successful due to back-propagation and gradient descent, which require differentiability of the loss function with respect to the trainable parameters. However, directly incorporating a CNN model into end-to-end image compression is challenging due to the quantization operation. The quantization module produces zero gradients almost everywhere, preventing the parameters from updating in the CNN. Additionally, the learning loss objective must be a differentiable loss function. In 2016, Ball'e *et al.* introduced the first end-to-end optimized CNN framework for image compression under the scalar quantization assumption [25]. To handle the zero derivatives resulting from quantization, an additive i.i.d uniform noise was used to simulate quantization in the CNN training procedure, enabling gradient descent for neural network optimization. This method outperformed JPEG2000 in terms of both PSNR and MS-SSIM metrics. Later, [26] extended this end-to-end framework to a scale hyper prior, resulting in better compression performance. Many other attempts [27, 28, 29, 30, 31] have been made to advance image compression using neural networks.

Generative Adversarial Networks (GANs) are a popular deep learning technique that involves

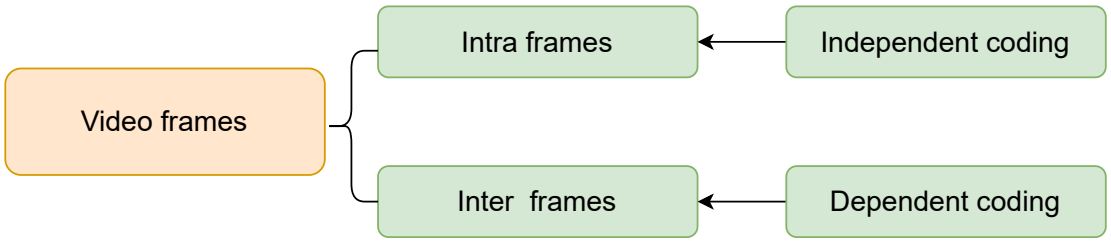


Figure 2.3: Compression for different video frames.

training a generator and a discriminator network simultaneously. In image compression, GANs have been used to improve the subjective quality of decoded images. For instance, Rippel and Bourdev [32] proposed an integrated GAN-based image compression method that achieved significant improvements in compression ratio and enhanced the subjective quality of reconstructed images. However, GAN-based compression has been successful only for narrow-domain images, such as faces, and more research is needed to establish models for general natural images.

Video coding typically involves two types of frames, illustrated in Figure 2.3: keyframes (also known as intra frames) and inter frames. A keyframe is a fully encoded frame that contains all the necessary information to reconstruct an image. It is coded independently of any other frames and serves as a reference for inter frames. Keyframes are often used as a starting point for video playback and can be decoded without relying on any other frames. Popular video coding standards such as MPEG[33], MPEG-2, H.264/AVC [8], and HEVC [34] can directly apply image compression methods to keyframes.

Inter frames, on the other hand, only contain the changes from a previously encoded frame (either a keyframe or another inter frame) and are coded based on motion estimation and compensation. Because inter frames rely on previously encoded frames, they are usually much smaller than keyframes and can achieve higher compression ratios. However, inter frames cannot be decoded independently and require reference frames for reconstruction. The combination

of keyframes and inter frames is commonly used in modern video coding standards such as H.264/AVC and HEVC to achieve efficient compression and high video quality.

2.1.3 Traditional video coding

Video coding is a fundamental process that compresses videos to enable efficient transmission and storage. It typically involves two techniques: entropy coding for lossless compression towards the Shannon limit, and lossy coding for removing redundant and less significant data in video. Although entropy coding can only achieve moderate compression ratios due to the Shannon limit, lossy compression is generally more effective because the human visual system can tolerate some loss of details.

The video coding process involves an encoder that converts video into a compressed format, and a decoder that restores the compressed video back to an uncompressed format. Together, these components form a codec (encoder/decoder), as illustrated in Figure 2.1. Video coding plays a critical role in transmitting and storing video content efficiently while minimizing the impact on image quality.

A standard video encoder typically consists of three primary components: (i) a predictive coding unit, (ii) a transform coding unit, and (iii) an entropy coding unit.

(i) Predictive coding. The predictive coding unit is a crucial component of video coding that exploits both temporal (inter-prediction) and spatial redundancies (intra-prediction) in a video sequence to reduce redundancy. This process is achieved through two methods: motion estimation (ME) and motion compensation (MC). ME involves finding a matching region in the reference frame that corresponds to a block in the current frame, while MC involves determining

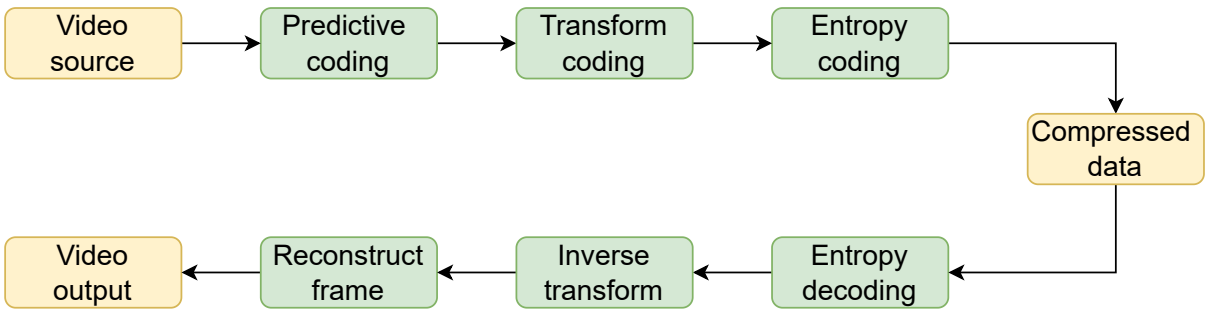


Figure 2.4: Transform coding framework for video compression.

the difference (residual) between the matching regions and the target region. This generates residuals and motion vectors that help to achieve high compression ratios while maintaining a high level of video quality.

To create residuals, the encoder subtracts the prediction from the actual current frame, while the motion vector is generated by computing the offset between the current block and the position of the candidate region. The motion vector indicates the direction of movement of the block. By using predictive coding, the encoder can reduce redundancies and transmit only the necessary information, resulting in more efficient video compression.

(ii) Transform coding. Transform coding is a crucial step in video compression that converts blocks of residual samples into a set of coefficients, each of which represents a weight for the standard basis pattern. These coefficients are then fed into a quantizer, which produces reduced precision yet bit-saving quantized coefficients. One of the most commonly used transform coding techniques is the discrete cosine transformation (DCT), which was developed in 1974.

In the H.264 video coding standard, transform coding is used to convert a block of residual samples into DCT coefficients. By reducing the dependency between sample points, transform coding enables more efficient compression of the video data. The encoder can achieve high

compression ratios by utilizing transform coding, while maintaining the video's visual quality. This reduction in data leads to improved storage and transmission efficiency, making transform coding a critical component in video compression.

(iii) Entropy coding. After predictive coding and transform coding, the video data is still not fully compressed. Entropy coding is the final stage in video coding that produces a compact and efficient bit stream for storage and transmission. It compresses the residual signals and the quantized transform coefficients generated by the previous stages.

Entropy coding techniques, such as variable length coding (VLC), arithmetic coding, and Huffman coding, assign shorter codes to more frequently occurring symbols, and longer codes to less frequent symbols. For instance, in Huffman coding, the most common symbols are assigned shorter codes, while the less common symbols are assigned longer codes. The motion vectors are also entropy coded separately using a VLC table.

By using entropy coding, the average bit rate of the encoded video stream can be further reduced, leading to higher compression ratios and improved storage and transmission efficiency.

Video decoding. The video decoding process, as shown in the bottom part of Figure 2.4, works in reverse order of the encoder. First, the entropy decoder recovers the prediction parameters and coefficients from the compressed bit stream. Then, the spatial decoder uses these parameters to reconstruct the residual frame. Finally, the prediction decoder uses the reconstructed pixels and the parameters to reconstruct the original frame, which is then displayed to the viewer. The decoding process plays a crucial role in the video playback performance since it needs to be performed in real-time. To achieve this, modern video codecs use parallel processing and specialized hardware to improve the decoding efficiency. Additionally, video decoding may also involve error resilience techniques to mitigate errors introduced during storage

or transmission, such as error concealment and error resilience coding.

2.1.4 Traditional video codec standards

As the most common video codecs, let's briefly go through the techniques used in H.264 and HEVC.

AVC H.264 encoding. The H.264 encoder operates on macroblocks, which are 16×16 pixel units. Inter-prediction is performed by utilizing a range of block sizes (from 16×16 to 4×4) to predict pixels in the current frame from similar regions in previously encoded frames. Intra-prediction uses the same range of block sizes to predict the macroblock from the previously encoded pixels within the same frame. The encoder then obtains a residual by subtracting the prediction from the current macroblock. The residual samples are transformed using a 4×4 or 8×8 integer transformation, resulting in a set of DCT coefficients. The coefficients and other information are quantized and coded into bit streams using entropy coding.

HEVC (H.265) encoding. The HEVC encoder follows a similar structure to H.264, utilizing inter/intra prediction and transform coding. Each frame of the input video sequence is divided into block-shaped regions called coding tree units (CTUs). A CTU can be of size 64×64, 32×32, or 16×16 and is organized in a quad-tree form to further partition into smaller-sized coding units (CUs). In HEVC, the first picture of the video sequence is coded using only intra-picture prediction, and all remaining pictures are coded using inter-picture predictive coding. Each CU can be predicted via intra-prediction or inter-prediction, and the prediction residual is coded using block transforms. The entropy coding module uses context-adaptive binary arithmetic coding (CABAC). The decoding process is the inverse of the encoding process.

Table 2.1: Historical development of video codecs

	Video coding standar	Year	Features
MPEG family	MPEG-1 part-2	1993	video and audio storage on CD-ROMS
	MPEG-2 part-2	1995	HDTV and video on DVDs
	MPEG-4 part-2 (visual)	1999	low bit-rate multimedia on mobile platforms
	MPEG-4 part 10 (AVS)	2003	Co-published with H.264/AVC
H.26X family	H.120	1984	The first digital video coding standard
	H.261	1988	Developed for video conferencing over ISDN
	H.262	1995	See MPEG-2 part 2
	H.263/H.263+	1996/1998	Improved quality to H.261 at lower bit rate
	H.264 AVC	2003	Significant quality improvement with lower bit rates
	H.265/HEVC	2013	50% bit-rate savings compared with H.264
	H.266/VVC	2020	50% bit-rate savings compared with H.265

Decoding. The decoding process starts by extracting the quantized, transformed coefficients and the prediction information from the bit stream. The decoder then rescales the coefficients to restore each block of the residual data. These blocks are combined together to form a residual macroblock for frame reconstruction. The decoder then adds the prediction to the decoded residual to reconstruct a decoded macroblock. Finally, the decoded macroblocks are combined to reconstruct the original video frame.

Table 2.1 summarizes the historical development of video codecs. We provide a brief summary of each codec's features below:

- **MPEG-1.** Developed for video and audio storage on CD-ROMs; Supports YUV 4:2:0 with a resolution of 352×288; Lossless motion vectors.
- **MPEG-2.** Supports HDTV and video on DVDs; Introduction of profiles and levels; Nonlinear quantization and data partitioning.
- **MPEG-4 part-2 (visual).** Supports low bit-rate multimedia applications on mobile platforms; Shares a subset with H.263; Supports object-based or content-based coding.

- **H.262.** Developed for video conferencing over ISDN. Block-based hybrid coding with integer pixel motion compensation; Supports CIF and QCIF resolutions.
- **H.263 / H.263+.** Improved quality to H.261 at a lower bit rate; Shares a subset with MPEG-4 part 2.
- **H.264 AVC.** Supports video on the Internet, computers, mobile devices, and HDTVs; Significantly improves quality with lower bit rates; Increased computational complexity; Improved motion compensation with variable block size, multiple reference frames, and weighted prediction.
- **H.265 HEVC.** Supports ultra HD video up to 8k with frame rates up to 120 fps; Greater flexibility in prediction modes and transfer block sizes; Parallel processing; 50% bit-rate savings compared with H.264 for the same video quality.
- **H.266 VVC.** Provides about 50% better compression rate for the same perceptual quality, with support for lossless and subjectively lossless compression; Supports resolutions ranging from very low resolution up to 4K and 16K, as well as 360° videos.

This revised version provides a clearer summary of each codec's features, as well as some additional details, such as the resolutions supported by each codec.

2.1.5 Learning-based video compression

Traditional video compression algorithms, such as H.265 and H.266, rely on hand-crafted motion estimation and motion compensation techniques, such as block-based motion estimation, to achieve inter-frame prediction. While these methods reduce temporal redundancy in video

data, they cannot be end-to-end optimized with other neural networks developed for various machine vision tasks, such as action recognition, using large-scale training datasets.

Recent advances in neural image compression have led to the development of neural video codecs. The pioneering work of DVC [35] follows a residual coding-based framework similar to traditional codecs. It first generates motion-compensated predictions and then encodes the residual using a hyperprior [26]. With the help of an autoregressive prior [29], DVCPro achieves even higher compression ratios.

Recent research in neural video codecs has focused on improving the motion estimation and residual coding-based framework. Some works have proposed advanced network structures to generate optimized residuals or motion. For instance, Yang *et al.* [36] adaptively scaled the residual using learned parameters, while Agustsson *et al.* [37] proposed using optical flow estimation in scale space to reduce residual energy in fast motion areas. Hu *et al.* [38] applied rate distortion optimization to improve motion coding, and Hu *et al.* [39] used deformable compensation to enhance feature space prediction. Lin *et al.* [40] proposed using multiple reference frames to reduce residual energy, and in [40, 41], motion prediction was introduced to improve motion coding efficiency.

In addition to residual coding, researchers have explored other coding frameworks for neural video codecs. One such framework is the 3D autoencoder[42, 43, 44], which encodes multiple frames simultaneously and is an extension of neural image codecs. However, this approach can introduce significant encoding delay and may not be suitable for real-time scenarios. Another emerging framework is conditional coding, which has a lower or equal entropy bound compared to residual coding [45]. For example, Ladune *et al.* [45, 46, 47] used conditional coding to code the foreground contents, while in DCVC [48], the condition is the extensible

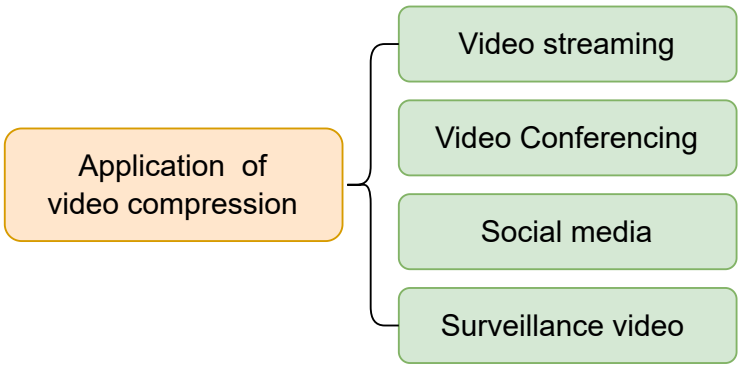


Figure 2.5: Application of video compression.

high-dimensional feature instead of the 3D predicted frame. To further boost the compression ratio, recent work has introduced feature propagation and multi-scale temporal contexts [49].

Most existing neural video codecs prioritize the optimization of the latent embedding and network design. Previous research has largely concentrated on temporal correlation. For instance, works such as [13, 48, 49, 50] employ techniques such as temporal context prior, conditional entropy coding, or recurrent entropy model to explore this area.

2.1.6 Application of video compression

Video compression is a critical need for many real-time applications, as depicted in Figure 2.5. With the advent of the internet, service providers offer cheap and high-speed bandwidth, leading to an explosion of data. As a result, a vast amount of data generated consists of videos. However, storing all this data requires significant space, making it difficult to manage. To address this challenge, efficient video compression techniques are essential. This section explores various applications where such techniques can be useful in the 21st century.

Video streaming. Video streaming has revolutionized the way users consume video content over the internet in real-time without downloading the entire video file. With the increasing

availability of high-speed internet connections and growing popularity of online video content, video streaming has become an essential part of our daily lives. However, the large size of video files poses a significant challenge in transmitting them quickly and efficiently over the internet. To overcome this challenge, video compression techniques have been developed to reduce the size of video files while preserving their quality.

Video compression algorithms can significantly reduce the amount of data that needs to be transmitted over the internet, making it possible to stream high-quality video content in real-time. As internet speeds continue to increase and video streaming grows in popularity, video compression techniques will become even more critical in delivering high-quality video content to users worldwide.

Video Conferencing. Video compression is an essential application in video conferencing, allowing individuals and businesses to connect remotely without having to worry about slow or interrupted connections. Video compression works by reducing the amount of data needed to transmit a video stream over the internet while maintaining a high-quality image. This makes video conferencing accessible to a wider audience, including those with low-bandwidth internet connections.

The use of video compression has become increasingly important as remote work and distance learning have become more prevalent. Without video compression, video conferencing would be prohibitively expensive and only accessible to those with high-speed internet connections. However, video compression algorithms allow for real-time transmission of high-quality video, making video conferencing an effective communication tool for businesses, schools, and individuals.

Social Media. Social media platforms have become an essential means of communication worldwide, and video content is increasingly becoming the most popular form of media. However,

transmitting video content over the internet can be challenging due to the large file sizes involved. As a result, video compression has emerged as a crucial application for social media platforms. It allows users to share and view videos without worrying about slow or interrupted connections. This makes it easier for social media platforms to store and transmit video content, as well as for users to upload and view videos without experiencing delays or buffering. Without video compression, social media platforms would struggle to keep up with the demand for video content and provide a seamless user experience.

Moreover, video compression has made it possible for social media platforms to incorporate live video streaming, which has become increasingly popular in recent years. Live video streaming allows users to broadcast events and experiences in real-time, connecting people from all over the world. Video compression algorithms play a critical role in making this technology accessible, allowing for real-time transmission of high-quality video over low-bandwidth internet connections. As video content continues to grow in popularity, video compression algorithms will undoubtedly play a crucial role in making it accessible to a wider audience. Overall, video compression is essential for social media platforms to meet the increasing demand for video content and provide a seamless user experience.

Surveillance Video. Video surveillance has become ubiquitous in today's world, with cameras being used for security purposes in various settings. However, the storage and transmission of the large amounts of video data generated by video surveillance systems pose a significant challenge. Video compression has emerged as a critical application in this context, allowing for more efficient storage and transmission of video data.

Video compression algorithms play a vital role in enabling real-time transmission of high-quality video over low-bandwidth internet connections, making it possible for video surveillance

to take place in remote areas with limited internet access. Additionally, the use of video compression in video surveillance has many benefits, including lower storage costs, increased efficiency, and improved accessibility. As technology continues to evolve, we can expect video compression algorithms to become even more efficient, enabling higher-quality video to be transmitted and stored at even lower costs. This development will enable businesses and individuals to improve their security measures and ensure that video surveillance is a viable and effective means of keeping people safe.

2.2 Implicit neural representation

Recent developments in deep learning have led to the emergence of implicit representations, which are compact data representations [1, 51, 52, 53] that fit a deep neural network to signals such as images, 3D shapes, and videos. One of the main branches of implicit representations is coordinate-based neural representations, which take pixel coordinates as input and output corresponding values such as density or RGB values using an MLP network. These representations have shown promising results in a range of areas including image reconstruction [54, 55], image compression [52], continuous spatial super-resolution [56, 57, 58, 59], shape regression [60, 61], and 3D view synthesis [62, 63]. To improve coordinate-based representations, several approaches have been proposed, such as using sine activation functions instead of ReLU [64] or converting input coordinates to a Fourier feature space [65].

2.3 Deep neural networks

With the interdisciplinary research of neuroscience and mathematics, the neural network (NN) was invented, which has shown strong abilities in the context of non-linear transform and classification. Intuitively, the network consists of multiple layers of simple processing units called neuron (perceptron), which interacts with each other via weighted connections. The neurons get activated through weighted connections from previously activated neurons. To achieve non-linearity, the activation functions are always applied for all the intermediate layers [66].

The learning procedure of simple perceptron has been proposed and analyzed in 1960s. During the 1970s and 1980s, backpropagation procedure [67, 68] inspired by the chain rule for derivatives of the training objectives was proposed to solve the training problem of the multi-layer perceptron (MLP). Then, the multi-layer architectures are mostly trained by stochastic gradient descent with backpropagation procedure although it is computationally intensive and suffers from bad local minima. However, the dense connections between the adjacent layers in neural networks make the amount of model parameters increase quadratically and prohibit the development of neural networks in computational efficiency. With the introduction of parameter-sharing for MLP 1990 [69], a more light-weighted version of neural network called convolutional neural network was proposed and applied in the documents recognition, which makes the large scale neural network training possible.

Over the last 10 years, several CNN architectures have been presented [70, 71]. Model architecture is a critical factor in improving the performance of different applications. Various modifications have been achieved in CNN architecture from 1989 until today. Such modifications include structural reformulation, regularization, parameter optimizations, etc. Conversely, it

should be noted that the key upgrade in CNN performance occurred largely due to the processing-unit reorganization, as well as the development of novel blocks. In particular, the most novel developments in CNN architectures were performed on the use of network depth. In this section, we review the most popular CNN architectures, beginning from the AlexNet model in 2012 and ending at the High-Resolution (HR) model in 2020. Studying these architectures features (such as input size, depth, and robustness) is the key to help researchers to choose the suitable architecture for the their target task. Table 2 presents the brief overview of CNN architectures.

AlexNet The history of deep CNNs began with the appearance of LeNet [72]. At that time, the CNNs were restricted to handwritten digit recognition tasks, which cannot be scaled to all image classes. In deep CNN architecture, AlexNet is highly respected [73], as it achieved innovative results in the fields of image recognition and classification. Krizhevesky *et al.* [73] first proposed AlexNet and consequently improved the CNN learning ability by increasing its depth and implementing several parameter optimization strategies. Figure 15 illustrates the basic design of the AlexNet architecture. The learning ability of the deep CNN was limited at this time due to hardware restrictions. To overcome these hardware limitations, two GPUs (NVIDIA GTX 580) were used in parallel to train AlexNet. Moreover, in order to enhance the applicability of the CNN to different image categories, the number of feature extraction stages was increased from five in LeNet to seven in AlexNet. Regardless of the fact that depth enhances generalization for several image resolutions, it was in fact overfitting that represented the main drawback related to the depth. Krizhevesky et al. used Hinton's idea to address this problem [74]. To ensure that the features learned by the algorithm were extra robust, Krizhevesky et al.'s algorithm randomly passes over several transformational units throughout the training stage. Moreover, by reducing the vanishing gradient problem, ReLU [75] could be utilized as

a non-saturating activation function to enhance the rate of convergence [76]. Local response normalization and overlapping subsampling were also performed to enhance the generalization by decreasing the overfitting. To improve on the performance of previous networks, other modifications were made by using large-size filters (5×5and11×11) in the earlier layers. AlexNet has considerable significance in the recent CNN generations, as well as beginning an innovative research era in CNN applications.

VGGNet After CNN was determined to be effective in the field of image recognition, an easy and efficient design principle for CNN was proposed by Simonyan and Zisserman. This innovative design was called Visual Geometry Group (VGG). A multilayer model [77], it featured nineteen more layers than AlexNet to simulate the relations of the network representational capacity in depth. This showed experimentally that the parallel assignment of these small-size filters could produce the same influence as the large-size filters. In other words, these small-size filters made the receptive field similarly efficient to the large-size filters (7×7and5×5). By decreasing the number of parameters, an extra advantage of reducing computational complication was achieved by using small-size filters. These outcomes established a novel research trend for working with small-size filters in CNN. In addition, by inserting 1×1 convolutions in the middle of the convolutional layers, VGG regulates the network complexity. It learns a linear grouping of the subsequent feature maps. In general, VGG obtained significant results for localization problems and image classification. While it did not achieve first place in the 2014-ILSVRC competition, it acquired a reputation due to its enlarged depth, homogenous topology, and simplicity. However, VGG's computational cost was excessive due to its utilization of around 140 million parameters, which represented its main shortcoming. Figure 18 shows the structure of the network.

ResNet He *et al.* [78] developed ResNet (Residual Network), which was the winner of ILSVRC 2015. Their objective was to design an ultra-deep network free of the vanishing gradient issue, as compared to the previous networks. Several types of ResNet were developed based on the number of layers (starting with 34 layers and going up to 1202 layers). The most common type was ResNet50, which comprised 49 convolutional layers plus a single FC layer. The overall number of network weights was 25.5 M, while the overall number of MACs was 3.9 M. The novel idea of ResNet is its use of the bypass pathway concept, as shown in Fig. 20, which was employed in Highway Nets to address the problem of training a deeper network in 2015. This is a conventional feedforward network plus a residual connection.

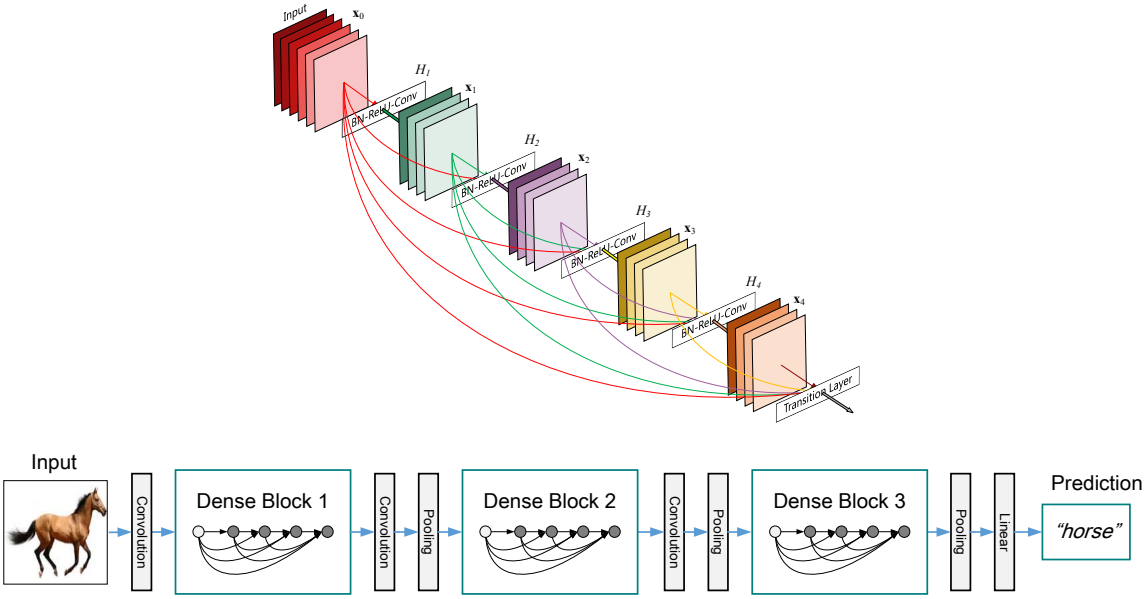


Figure 2.6: DenseNet architecture. Image obtained from [4].

DenseNet To solve the problem of the vanishing gradient, DenseNet [4] was presented, following the same direction as ResNet and the Highway network [79]. One of the drawbacks of ResNet is that it clearly conserves information by means of preservative individuality transformations, as several layers contribute extremely little or no information. In addition, ResNet has a large

number of weights, since each layer has an isolated group of weights. DenseNet employed cross-layer connectivity in an improved approach to address this problem. It connected each layer to all layers in the network using a feed-forward approach. Therefore, the feature maps of each previous layer were employed to input into all of the following layers. DenseNet demonstrates the influence of cross-layer depth wise-convolutions. Thus, the network gains the ability to discriminate clearly between the added and the preserved information, since DenseNet concatenates the features of the preceding layers rather than adding them. However, due to its narrow layer structure, DenseNet becomes parametrically high-priced in addition to the increased number of feature maps. The direct admission of all layers to the gradients via the loss function enhances the information flow all across the network. In addition, this includes a regularizing impact, which minimizes overfitting on tasks alongside minor training sets. Figure 2.6 shows the architecture of DenseNet Network.

Vision Transformer Transformer architectures are based on a self-attention mechanism that learns the relationships between elements of a sequence. As opposed to recurrent networks that pro- cess sequence elements recursively and can only attend to short-term context, Transformers can attend to complete sequences thereby learning long-range relationships. Vision Transformer (ViT) [80] (Figure 2.7) is the first work to showcase how Transformers can 'altogether' replace standard convolutions in deep neural networks on large- scale image datasets. They applied the original Transformer model [81] (with minimal changes) on a sequence of image 'patches' flattend as vectors. The model was pre-trained on a large propriety dataset (JFT dataset [82] with 300 million images) and then fine-tuned to downstream recog- nition benchmarks e.g., ImageNet classification. This is an important step since pre-training ViT on a medium-range dataset would not give competitive results, because the CNNs encode prior knowledge about the

images (inductive biases e.g., translation equivariance) that reduces the need of data as compared to Transformers which must discover such information from very large-scale data. The DeiT [83] is the first work to demonstrate that Transformers can be learned on mid-sized datasets (i.e., 1.2 million ImageNet examples compared to 300 million images of JFT used in ViT) in relatively shorter training episodes. Besides using augmentation and regularization procedures common in CNNs, the main contribution of DeiT is a novel native distillation approach for Trans- formers which uses a CNN as a teacher model (RegNetY- 16GF [84]) to train the Transformer model.

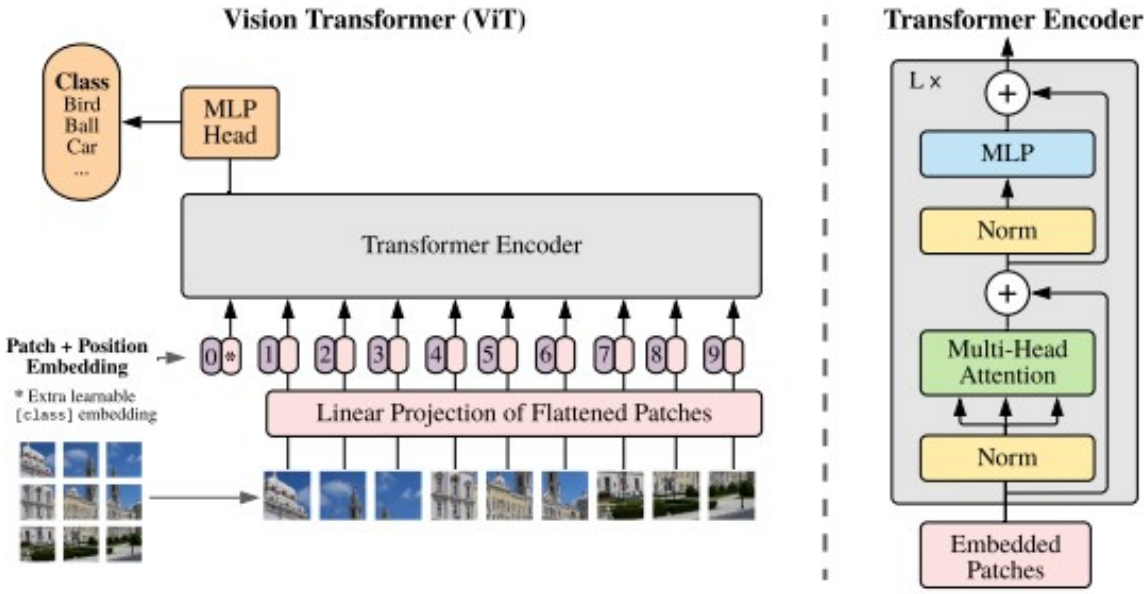


Figure 2.7: An overview of Vision Transformer (on the *left*) and the details of Transformer encoder (on the *right*). The architecture resembles Transformers used in the NLP domain and the image patches are simply fed to the model after flattening. After training, the feature obtained from the first token position is used for classification. Image obtained from [5].

Chapter 3: NeRV: Implicit Neural Representations for Videos

3.1 Introduction

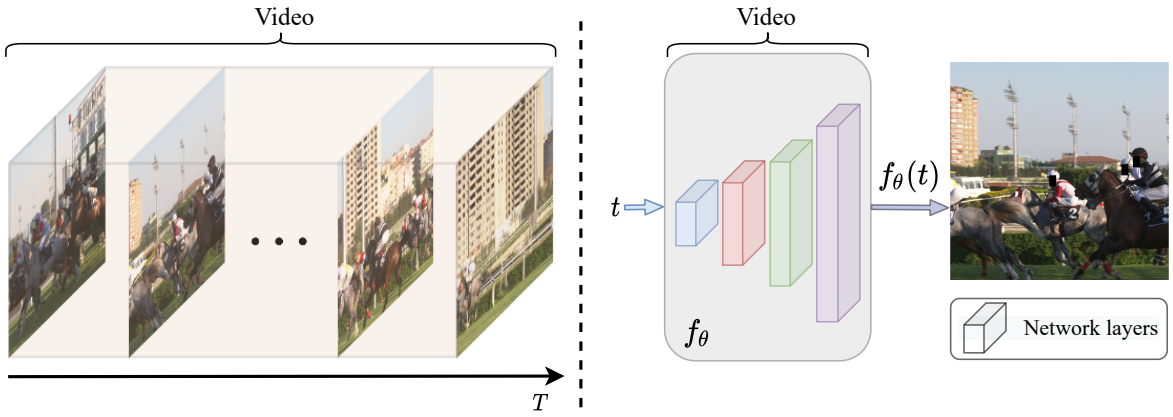
What is a video? Typically, a video captures a dynamic visual scene using a sequence of frames. A schematic interpretation of this is a curve in 2D space, where each point can be characterized with a (x, y) pair representing the spatial state. If we have a model for all (x, y) pairs, then, given any x, we can easily find the corresponding y state. Similarly, we can interpret a video as a recording of the visual world, where we can find a corresponding RGB state for every single timestamp. This leads to our main claim: *can we represent a video as a function of time?*

More formally, can we represent a video V as $V = \{v_t\}_{t=1}^{T}$, where $v_t = f_\theta(t)$, *i.e.*, , a frame at timestamp t, is represented as a function f parameterized by θ. Given their remarkable representational capacity [85], we choose deep neural networks as the function in our work. Given these intuitions, we propose NeRV, a novel representation that represents videos as implicit functions and encodes them into neural networks. Specifically, with a fairly simple deep neural network design, NeRV can reconstruct the corresponding video frames with high quality, given the frame index. Once the video is encoded into a neural network, this network can be used as a proxy for video, where we can directly extract all video information from the representation. Therefore, unlike traditional video representations which treat videos as sequences of frames, shown in Figure 6.1 (a), our proposed NeRV considers a video as a unified neural network with

Table 3.1: Comparison of different video representations. Although explicit representations outperform implicit ones in encoding speed and compression ratio now, NeRV shows great advantage in decoding speed. And NeRV outperforms pixel-wise implicit representations in all metrics.

	Explicit (frame-based)		Implicit (unified)	
	Hand-crafted (*e.g.* , HEVC [34])	Learning-based (*e.g.* , DVC [35])	Pixel-wise (*e.g.* , NeRF [62]	Image-wise (Ours)
Encoding speed	**Fast**	Medium	Very slow	Slow
Decoding speed	Medium	Slow	Very slow	**Fast**
Compression ratio	Medium	**High**	Low	Medium

all information embedded within its architecture and parameters, shown in Figure 6.1 (b).



(a) Explicit representations for videos (e.g., HEVC) (b) Neural implicit representations for videos (e.g., NeRV)

Figure 3.1: **(a)** Conventional video representation as **frame sequences**. **(b)** NeRV, representing video as **neural networks**, which consists of multiple convolutional layers, taking the normalized frame index as the input and output the corresponding RGB frame.

As an image-wise implicit representation, NeRV shares lots of similarities with pixel-wise implicit visual representations [54, 55] which takes spatial-temporal coordinates as inputs. The main differences between our work and image-wise implicit representation are the output space and architecture designs. Pixel-wise representations output the RGB value for each pixel, while NeRV outputs a whole image, demonstrated in Figure 3.2. Given a video with size of $T \times H \times W$, pixel-wise representations need to sample the video $T * H * W$ times while NeRV only need to

sample T times. Considering the huge pixel number, especially for high resolution videos, NeRV shows great advantage for both encoding time and decoding speed. Different output space also leads to different architecture designs, NeRV utilizes a MLP + ConvNets architecture to output an image while pixel-wise representation uses a simple MLP to output the RGB value of the pixel. Sampling efficiency of NeRV also simplify the optimization problem, which leads to better reconstruction quality compared to pixel-wise representations.

We also demonstrate the flexibility of NeRV by exploring several applications it affords. Most notably, we examine the suitability of NeRV for video compression. Traditional video compression frameworks are quite involved, such as specifying key frames and inter frames, estimating the residual information, block-size the video frames, applying discrete cosine transform on the resulting image blocks and so on. Such a long pipeline makes the decoding process very complex as well. In contrast, given a neural network that encodes a video in NeRV, we can simply cast the video compression task as a model compression problem, and trivially leverage any well-established or cutting edge model compression algorithm to achieve good compression ratios. Specifically, we explore a three-step model compression pipeline: model pruning, model quantization, and weight encoding, and show the contributions of each step for the compression task. We conduct extensive experiments on popular video compression datasets, such as UVG [86], and show the applicability of model compression techniques on NeRV for video compression. We briefly compare different video representations in Table 3.1 and NeRV shows great advantage in decoding speed.

Besides video compression, we also explore other applications of the NeRV representation for the video denoising task. Since NeRV is a learnt implicit function, we can demonstrate its robustness to noise and perturbations. Given a noisy video as input, NeRV generates a high-

quality denoised output, without any additional operation, and even outperforms conventional denoising methods.

The contribution of this paper can be summarized into four parts:

- We propose NeRV, a novel image-wise implicit representation for videos, representating a video as a neural network, converting video encoding to model fitting and video decoding as a simple feedforward operation.
- Compared to pixel-wise implicit representation, NeRV output the whole image and shows great efficiency, improving the encoding speed by **25**$\times$ to **70**$\times$, the decoding speed by **38**$\times$ to **132**$\times$, while achieving better video quality.
- NeRV allows us to convert the video compression problem to a model compression problem, allowing us to leverage standard model compression tools and reach comparable performance with conventional video compression methods, *e.g.* , H.264 [8], and HEVC [34].
- As a general representation for videos, NeRV also shows promising results in other tasks, *e.g.* , video denoising. Without any special denoisng design, NeRV outperforms traditional hand-crafted denoising algorithms (medium filter *etc.*) and ConvNets-based denoisng methods.

3.2 Related Work

Implicit Neural Representation. Implicit neural representation is a novel way to parameterize a variety of signals. The key idea is to represent an object as a function approximated via a neural network, which maps the coordinate to its corresponding value (*e.g.* , pixel coordinate

for an image and RGB value of the pixel). It has been widely applied in many 3D vision tasks, such as 3D shapes [87, 88], 3D scenes [89, 90, 91, 92], and appearance of the 3D structure [62, 93, 94]. Comparing to explicit 3D representations, such as voxel, point cloud, and mesh, the continuous implicit neural representation can compactly encode high-resolution signals in a memory-efficient way. Most recently, [52] demonstrated the feasibility of using implicit neural representation for image compression tasks. Although it is not yet competitive with the state-of-the-art compression methods, it shows promising and attractive proprieties. In previous methods, MLPs are often used to approximate the implicit neural representations, which take the spatial or spatio-temporal coordinate as the input and output the signals at that single point (*e.g.* , RGB value, volume density). In contrast, our NeRV representation, trains a purposefully designed neural network composed of MLPs and convolution layers, and takes the frame index as input and directly outputs all the RGB values of that frame.

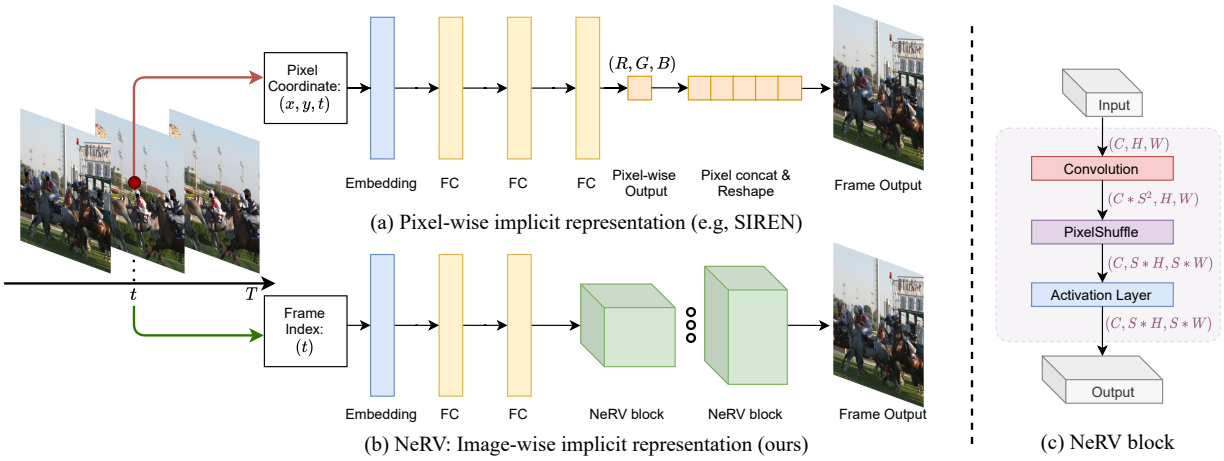
Video Compression. As a fundamental task of computer vision and image processing, visual data compression has been studied for several decades. Before the resurgence of deep networks, handcrafted image compression techniques, like JPEG [19] and JPEG2000 [95], were widely used. Building upon them, many traditional video compression algorithms, such as MPEG [33], H.264 [8], and HEVC [34], have achieved great success. These methods are generally based on transform coding like Discrete Cosine Transform (DCT) [20] or wavelet transform [96], which are well-engineered and tuned to be fast and efficient. More recently, deep learning-based visual compression approaches have been gaining popularity. For video compression, the most common practice is to utilize neural networks for certain components while using the traditional video compression pipeline. For example, [97] proposed an effective image compression approach

and generalized it into video compression by adding interpolation loop modules. Similarly, [98] converted the video compression problem into an image interpolation problem and proposed an interpolation network, resulting in competitive compression quality. Furthermore, [37] generalized optical flow to scale-space flow to better model uncertainty in compression. Later, [99] employed a temporal hierarchical structure, and trained neural networks for most components including key frame compression, motion estimation, motions compression, and residual compression. However, all of these works still follow the overall pipeline of traditional compression, arguably limiting their capabilities.

Model Compression. The goal of model compression is to simplify an original model by reducing the number of parameters while maintaining its accuracy. Current research on model compression research can be divided into four groups: parameter pruning and quantization [100, 101, 102, 103, 104, 105]; low-rank factorization [106, 107, 108]; transferred and compact convolutional filters [109, 110, 111, 112]; and knowledge distillation [113, 114, 115, 116]. Our proposed NeRV enables us to reformulate the video compression problem into model compression, and utilize standard model compression techniques. Specifically, we use model pruning and quantization to reduce the model size without significantly deteriorating the performance.

3.3 Neural Representations for Videos

We first present the NeRV representation in Section 3.3.1, including the input embedding, the network architecture, and the loss objective. Then, we present model compression techniques on NeRV in Section 3.3.2 for video compression.



(a) Pixel-wise implicit representation (e.g, SIREN)

(b) NeRV: Image-wise implicit representation (ours)

(c) NeRV block

Figure 3.2: **(a) Pixel-wise** implicit representation taking pixel coordinates as input and use a simple MLP to output pixel RGB value **(b) NeRV: Image-wise** implicit representation taking frame index as input and use a MLP + ConvNets to output the whole image. **(c) NeRV block** architecture, upscale the feature map by S here.

3.3.1 NeRV Architecture

In NeRV, each video $V = \{v_t\}_{t=1}^{T} \in \mathbb{R}^{T \times H \times W \times 3}$ is represented by a function $f_\theta : \mathbb{R} \rightarrow \mathbb{R}^{H \times W \times 3}$, where the input is a frame index t and the output is the corresponding RGB image $v_t \in \mathbb{R}^{H \times W \times 3}$. The encoding function is parameterized with a deep neural network θ, $v_t = f_\theta(t)$. Therefore, video encoding is done by fitting a neural network f_θ to a given video, such that it can map each input timestamp to the corresponding RGB frame.

Input Embedding. Although deep neural networks can be used as universal function approximators [85], directly training the network f_θ with input timestamp t results in poor results, which is also observed by [62, 117]. By mapping the inputs to a high embedding space, the neural network can better fit data with high-frequency variations. Specifically, in NeRV, we use Positional Encoding [54, 62, 81] as our embedding function

$$\Gamma(t) = \left(\sin\left(b^0 \pi t\right), \cos\left(b^0 \pi t\right), \ldots, \sin\left(b^{l-1} \pi t\right), \cos\left(b^{l-1} \pi t\right)\right) \tag{3.1}$$

where b and l are hyper-parameters of the networks. Given an input timestamp t, normalized between $(0, 1]$, the output of embedding function $\Gamma(\cdot)$ is then fed to the following neural network.

Network Architecture. NeRV architecture is illustrated in Figure 3.2 (b). NeRV takes the time embedding as input and outputs the corresponding RGB Frame. Leveraging MLPs to directly output all pixel values of the frames can lead to huge parameters, especially when the images resolutions are large. Therefore, we stack multiple NeRV blocks following the MLP layers so that pixels at different locations can share convolutional kernels, leading to an efficient and effective network. Inspired by the super-resolution networks, we design the NeRV block, illustrated in Figure 3.2 (c), adopting PixelShuffle technique [118] for upscaling method. Convolution and activation layers are also inserted to enhance the expressibilty. The detailed architecture can be found in the supplementary material.

Loss Objective. For NeRV, we adopt combination of L1 and SSIM loss as our loss function for network optimization, which calculates the loss over all pixel locations of the predicted image and the ground-truth image as following

$$L = \frac{1}{T}\sum_{t=1}^{T} \alpha \left\| f_\theta(t) - v_t \right\|_1 + (1-\alpha)(1 - \text{SSIM}(f_\theta(t), v_t)) \tag{3.2}$$

where T is the frame number, $f_\theta(t) \in \mathbb{R}^{H\times W\times 3}$ the NeRV prediction, $v_t \in \mathbb{R}^{H\times W\times 3}$ the frame ground truth, α is hyper-parameter to balance the weight for each loss component.

3.3.2 Model Compression

In this section, we briefly revisit model compression techniques used for video compression with NeRV. Our model compression composes of four standard sequential steps: video overfit,

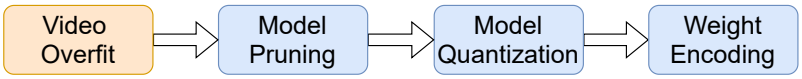


Figure 3.3: NeRV-based video compression pipeline.

model pruning, weight quantization, and weight encoding as shown in Figure 3.3.

Model Pruning. Given a neural network fit on a video, we use global unstructured pruning to reduce the model size first. Based on the magnitude of weight values, we set weights below a threshold as zero,

$$\theta_i = \begin{cases} \theta_i, & \text{if } \theta_i \geq \theta_q \\ 0, & \text{otherwise,} \end{cases} \tag{3.3}$$

where θ_q is the q percentile value for all parameters in θ. As a normal practice, we fine-tune the model to regain the representation, after the pruning operation.

Model Quantization. After model pruning, we apply model quantization to all network parameters. Note that different from many recent works [104, 119, 120, 121] that utilize quantization during training, NeRV is only quantized post-hoc (after the training process). Given a parameter tensor μ

$$\mu_i = \text{round}\left(\frac{\mu_i - \mu_{\text{min}}}{scale}\right) * \text{scale} + \mu_{\text{min}}, \quad \text{scale} = \frac{\mu_{\text{max}} - \mu_{\text{min}}}{2^{\text{bit}}} \tag{3.4}$$

where 'round' is rounding value to the closest integer, 'bit' the bit length for quantized model, μ_{max} and μ_{min} the max and min value for the parameter tensor μ, 'scale' the scaling factor. Through Equation 5.1, each parameter can be mapped to a 'bit' length value. The overhead to store 'scale' and μ_{min} can be ignored given the large parameter number of μ, e.g., they account for only 0.005% in a small 3×3 Conv with 64 input channels and 64 output channels ($37k$ parameters in total).

Entropy Encoding. Finally, we use entropy encoding to further compress the model size. By taking advantage of character frequency, entropy encoding can represent the data with a more efficient codec. Specifically, we employ Huffman Coding [122] after model quantization. Since Huffman Coding is lossless, it is guaranteed that a decent compression can be achieved without any impact on the reconstruction quality. Empirically, this further reduces the model size by around 10%.

3.4 Experiments

3.4.1 Datasets and Implementation Details

We perform experiments on "Big Buck Bunny" sequence from scikit-video to compare our NeRV with pixel-wise implicit representations, which has 132 frames of 720×1080 resolution. To compare with state-of-the-arts methods on video compression task, we do experiments on the widely used UVG [86], consisting of 7 videos and 3900 frames with 1920×1080 in total.

In our experiments, we train the network using Adam optimizer [123] with learning rate of 5e-4. For ablation study on UVG, we use cosine annealing learning rate schedule [124], batchsize of 1, training epochs of 150, and warmup epochs of 30 unless otherwise denoted. When compare with state-of-the-arts, we run the model for 1500 epochs, with batchsize of 6. For experiments on "Big Buck Bunny", we train NeRV for 1200 epochs unless otherwise denoted. For fine-tune process after pruning, we use 50 epochs for both UVG and "Big Buck Bunny".

For NeRV architecture, there are 5 NeRV blocks, with up-scale factor 5, 3, 2, 2, 2 respectively for 1080p videos, and 5, 2, 2, 2, 2 respectively for 720p videos. By changing the hidden dimension of MLP and channel dimension of NeRV blocks, we can build NeRV model with

Table 3.2: Compare with pixel-wise implicit representations. Training speed means time/epoch, while encoding time is the total training time.

Methods	Parameters	Training Speed ↑	Encoding Time ↓	PSNR ↑	Decoding FPS ↑
SIREN [55]	3.2M	1×	2.5×	31.39	1.4
NeRF [62]	3.2M	1×	2.5×	33.31	1.4
NeRV-S (ours)	3.2M	**25**×	**1**×	**34.21**	**54.5**
SIREN [55]	6.4M	1×	5×	31.37	0.8
NeRF [62]	6.4M	1×	5×	35.17	0.8
NeRV-M (ours)	6.3M	**50**×	**1**×	**38.14**	**53.8**
SIREN [55]	12.7M	1×	7×	25.06	0.4
NeRF [62]	12.7M	1×	7×	37.94	0.4
NeRV-L (ours)	12.5M	**70**×	**1**×	**41.29**	**52.9**

Table 3.3: PSNR *v.s.* epochs. Since video encoding of NeRV is an overfit process, the reconstructed video quality keeps increasing with more training epochs. NeRV-S/M/L mean models with different sizes.

Epoch	NeRV-S	NeRV-M	NeRV-L
300	32.21	36.05	39.75
600	33.56	37.47	40.84
1.2k	34.21	38.14	41.29
1.8k	34.33	38.32	41.68
2.4k	**34.86**	**38.7**	**41.99**

different sizes. For input embedding in Equation 3.1, we use $b = 1.25$ and $l = 80$ as our default setting. For loss objective in Equation 6.6, α is set to 0.7. We evaluate the video quality with two metrics: PSNR and MS-SSIM [125]. Bits-per-pixel (BPP) is adopted to indicate the compression ratio. We implement our model in PyTorch [126] and train it in full precision (FP32). All experiments are run with NVIDIA RTX2080ti. Please refer to the supplementary material for more experimental details, results, and visualizations (*e.g.* , MCL-JCV [127] results)

3.4.2 Main Results

We compare NeRV with pixel-wise implicit representations on 'Big Buck Bunny' video. We take SIREN [55] and NeRF [62] as the baseline, where SIREN [55] takes the original pixel coordinates as input and uses $sine$ activations, while NeRF [62] adds one positional embedding layer to encode the pixel coordinates and uses ReLU activations. Both SIREN and FFN use a 3-layer perceptron and we change the hidden dimension to build model of different sizes. For fair comparison, we train SIREN and FFN for 120 epochs to make encoding time comparable. And we change the filter width to build NeRV model of comparable sizes, named as NeRV-S, NeRV-M, and NeRV-L. In Table 3.2, NeRV outperforms them greatly in both encoding speed, decoding quality, and decoding speed. Note that NeRV can improve the training speed by $25\times$ to $70\times$, and speedup the decoding FPS by $38\times$ to $132\times$. We also conduct experiments with different training epochs in Table 3.3, which clearly shows that longer training time can lead to much better overfit results of the video and we notice that the final performances have not saturated as long as it trains for more epochs.

3.4.3 Video Compression

Compression ablation. We first conduct ablation study on video "Big Buck Bunny". Figure 3.4 shows the results of different pruning ratios, where model of 40% sparsity still reach comparable performance with the full model. As for model quantization step in Figure 3.5, a 8-bit model still remains the video quality compared to the original one (32-bit). Figure 3.6 shows the full compression pipeline with NeRV. The compression performance is quite robust to NeRV models of different sizes, and each step shows consistent contribution to our final results. Please

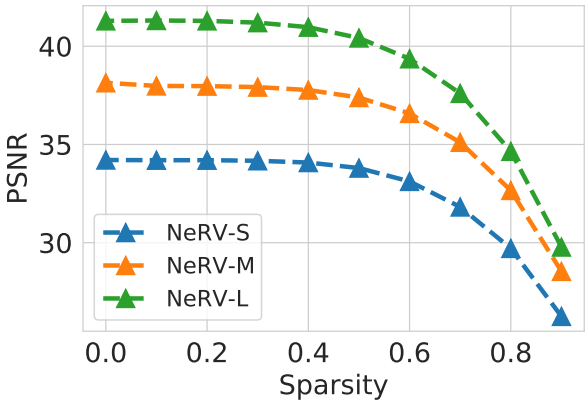


Figure 3.4: Model **pruning**. Sparsity is the ratio of parameters pruned.

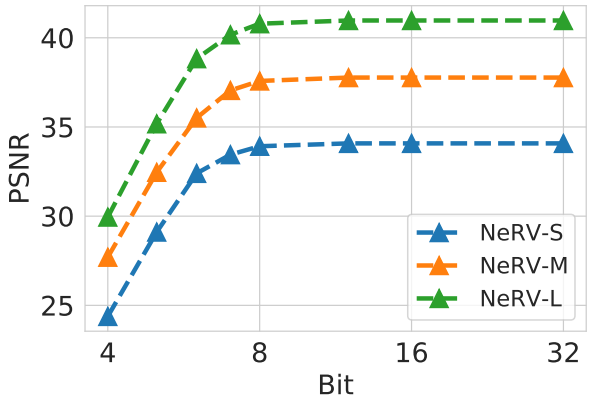


Figure 3.5: Model **quantization**. Bit is the bit length used to represent parameter value.

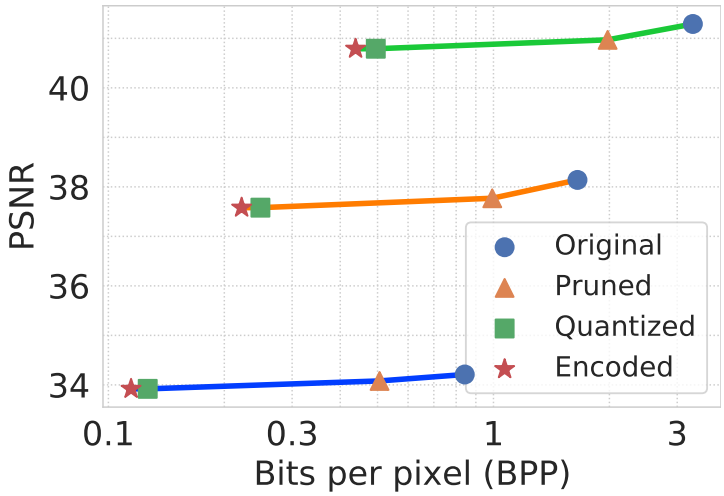


Figure 3.6: Compression **pipeline** to show how much each step contribute to compression ratio.

note that we only explore these three common compression techniques here, and we believe that other well-established and cutting edge model compression algorithm can be applied to further improve the final performances of video compression task, which is left for future research.

Compare with state-of-the-arts methods. We then compare with state-of-the-arts methods on UVG dataset. First, we concatenate 7 videos into one single video along the time dimension and train NeRV on all the frames from different videos, which we found to be more beneficial than training a single model for each video. After training the network, we apply model pruning,

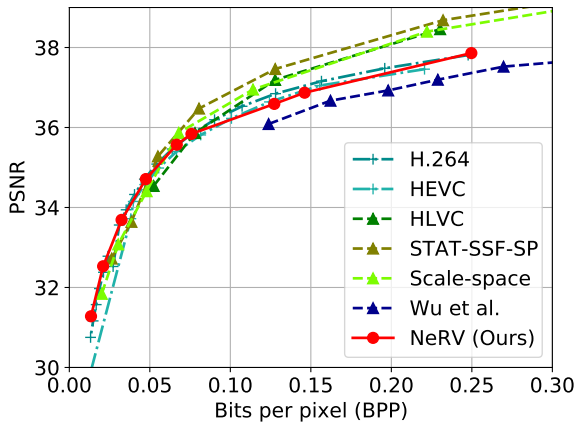


Figure 3.7: PSNR *v.s.* BPP on UVG dataset.

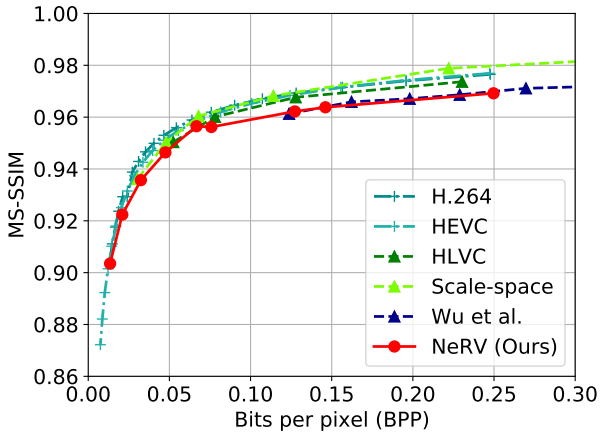


Figure 3.8: MS-SSIM *v.s.* BPP on UVG dataset.

quantization, and weight encoding as described in Section3.3.2. Figure 3.7 and Figure 3.8 show the rate-distortion curves. We compare with H.264 [8], HEVC [34], STAT-SSF-SP [36], HLVC [99], Scale-space [37], and Wu *et al*. [98]. H.264 and HEVC are performed with *medium* preset mode. As the first image-wise neural representation, NeRV generally achieves comparable performance with traditional video compression techniques and other learning-based video compression approaches. It is worth noting that when BPP is small, NeRV can match the performance of the state-of-the-art method, showing its great potential in high-rate video compression. When BPP becomes large, the performance gap is mostly because of the lack of full training due to GPU resources limitations. As shown in Table 3.3, the decoding video quality keeps increasing when the training epochs are longer. Figure 3.9 shows visualizations for decoding frames. At similar memory budget, NeRV shows image details with better quality.

Decoding time We compare with other methods for decoding time under a similar memory budget. Note that HEVC is run on CPU, while all other learning-based methods are run on a single GPU, including our NeRV. We speedup NeRV by running it in half precision (FP16). Due to the simple decoding process (feedforward operation), NeRV shows great advantage, even for

(a) Ground Truth Frame (b) NeRV Decoded Frame (0.077 BPP) (c) HEVC Decoded Frame (0.075 BPP)

Figure 3.9: Video compression visualization. At similar BPP, NeRV reconstructs videos with better details.

Table 3.4: **Decoding speed** with BPP 0.2 for 1080p videos

Methods	FPS ↑
Habibian et al. [92]	$10^{-3.7}$
Wu et al. [98]	10^{-3}
Rippel et al. [128]	1
DVC [35]	1.8
Liu et al [13]	3
H.264 [8]	9.2
NeRV (FP32)	5.6
NeRV (FP16)	**12.5**

carefully-optimized H.264. And lots of speepup can be expected by running quantizaed model on special hardware. All the other video compression methods have two types of frames: key and interval frames. Key frame can be reconstructed by its encoded feature only while the interval frame reconstruction is also based on the reconstructed key frames. Since most video frames are interval frames, their decoding needs to be done in a sequential manner after the reconstruction of the respective key frames. On the contrary, our NeRV can output frames at any random time index independently, thus making parallel decoding much simpler. This can be viewed as a distinct advantage over other methods.

3.4.4 Video Denoising

We apply several common noise patterns on the original video and train the model on the perturbed ones. During training, no masks or noise locations are provided to the model, *i.e.*, , the target of the model is the noisy frames while the model has no extra signal of whether the input is noisy or not. Surprisingly, our model tries to avoid the influence of the noise and regularizes them implicitly with little harm to the compression task simultaneously, which can serve well for most partially distorted videos in practice.

The results are compared with some standard denoising methods including Gaussian, uniform, and median filtering. These can be viewed as denoising upper bound for any additional compression process. As listed in Table 3.5, the PSNR of NeRV output is usually much higher than the noisy frames although it's trained on the noisy target in a fully supervised manner, and has reached an acceptable level for general denoising purpose. Specifically, median filtering has the best performance among the traditional denoising techniques, while NeRV outperforms it in most cases or is at least comparable without any extra denoising design in both architecture design and training strategy.

We also compare NeRV with another neural-network-based denoising method, Deep Image Prior (DIP) [6]. Although its main target is image denoising, NeRV outperforms it in both qualitative and quantitative metrics, demonstrated in Figure 3.10. The main difference between them is that denoising of DIP only comes form architecture prior, while the denoising ability of NeRV comes from both architecture prior and data prior. DIP emphasizes that its image prior is only captured by the network structure of Convolution operations because it only feeds on a single image. But the training data of NeRV contain many video frames, sharing lots of visual

Table 3.5: PSNR results for **video denoising**. "baseline" refers to the noisy frames before any denoising

noise	white ↑	black ↑	salt & pepper ↑	random ↑	Average ↑
Baseline	27.85	28.29	27.95	30.95	28.74
Gaussian	30.27	30.14	30.23	30.99	30.41
Uniform	29.11	29.06	29.10	29.63	29.22
Median	**33.89**	33.84	33.87	33.89	33.87
Minimum	20.55	16.60	18.09	18.20	18.36
Maximum	16.16	20.26	17.69	17.83	17.99
NeRV	33.31	**34.20**	**34.17**	**34.80**	**34.12**

Table 3.6: Input embedding ablation. PE means positional encoding

	PSNR	MS-SSIM
None	24.93	0.769
PE	**37.26**	**0.970**

Table 3.7: Upscale layer ablation

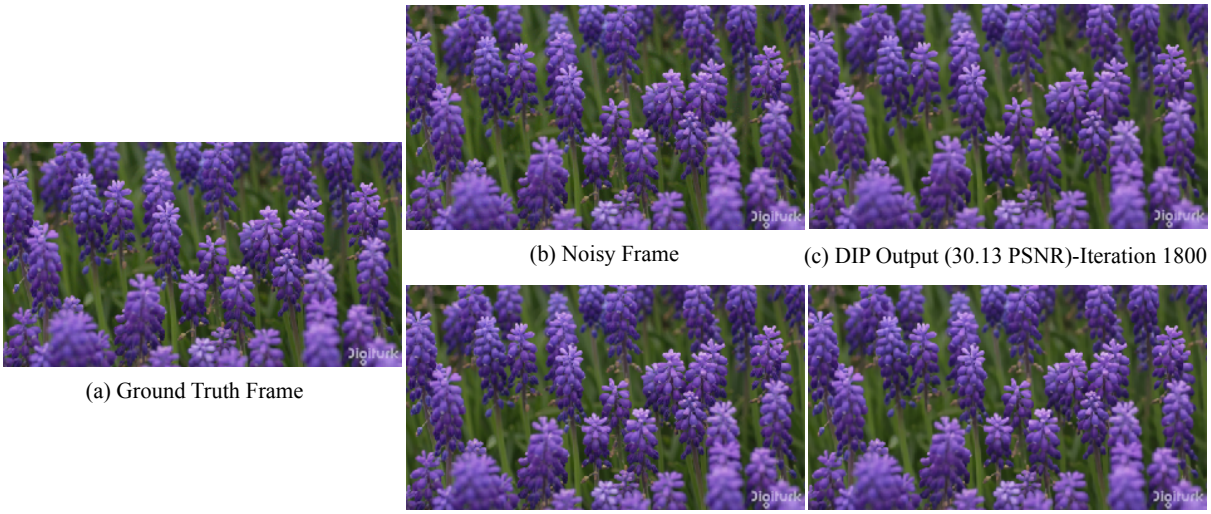
	PSNR	MS-SSIM
Bilinear Pooling	29.56	0.873
Transpose Conv	36.63	0.967
PixelShuffle	**37.26**	**0.970**

contents and consistences. As a result, image prior is captured by both the network structure and the training data statistics for NeRV. DIP relies significantly on a good early stopping strategy to prevent it from overfitting to the noise. Without the noise prior, it has to be used with fixed iterations settings, which is not easy to generalize to any random kind of noises as mentioned above. By contrast, NeRV is able to handle this naturally by keeping training because the full set of consecutive video frames provides a strong regularization on image content over noise.

3.4.5 Ablation Studies

Finally, we provide ablation studies on the UVG dataset. PSNR and MS-SSIM are adopted for evaluation of the reconstructed videos.

Input embedding. In Table 3.6, PE means positional encoding as in Equation 3.1, which greatly



(a) Ground Truth Frame

(b) Noisy Frame

(c) DIP Output (30.13 PSNR)-Iteration 1800

(d) NeRV Output (39.27 PSNR)

(e) DIP Output (32.30 PSNR)-Iteration 3600

Figure 3.10: Denoising visualization. (c) and (e) are denoising output for DIP [6]. Data generalization of NeRV leads to robust and better denoising performance since all frames share the same representation, while DIP model overfits one model to one image only.

improves the baseline, None means taking the frame index as input directly. Similar findings can be found in [62], without any input embedding, the model can not learn high-frequency information, resulting in much lower performance.

Upscale layer. In Table 3.7, we show results of three different upscale methods. *i.e.*, , Bilinear Pooling, Transpose Convolution, and PixelShuffle [118]. With similar model sizes, PixelShuffle shows best results. Please note that although Transpose convolution [129] reach comparable results, it greatly slowdown the training speed compared to the PixelShuffle.

Normalization layer. In Table 3.8, we apply common normalization layers in NeRV block. The default setup, without normalization layer, reaches the best performance and runs slightly faster. We hypothesize that the normalization layer reduces the over-fitting capability of the neural network, which is contradictory to our training objective.

Activation layer. Table 3.9 shows results for common activation layers. The GELU [130] activation function achieve the highest performances, which is adopted as our default design.

Loss objective. We show loss objective ablation in Table 3.10. We shows performance results of different combinations of L2, L1, and SSIM loss. Although adopting SSIM alone can produce the highest MS-SSIM score, but the combination of L1 loss and SSIM loss can achieve the best trade-off between the PSNR performance and MS-SSIM score.

Table 3.8: Norm layer ablation

	PSNR	MS-SSIM
BatchNorm	36.71	**0.971**
InstanceNorm	35.5	0.963
None	**37.26**	0.970

Table 3.9: Activation function ablation

	PSNR	MS-SSIM
ReLU	35.89	0.963
Leaky ReLU	36.76	0.968
Swish	37.08	0.969
GELU	**37.26**	**0.970**

Table 3.10: Loss objective ablation

L2	L1	SSIM	PSNR	MS-SSIM
✓			35.64	0.956
	✓		35.77	0.959
		✓	35.69	**0.971**
✓	✓		35.95	0.960
✓		✓	36.46	0.970
	✓	✓	**37.26**	0.970

3.5 Discussion

Conclusion. In this work, we present a novel neural representation for videos, NeRV, which encodes videos into neural networks. Our key sight is that by directly training a neural network with video frame index and output corresponding RGB image, we can use the weights of the model to represent the videos, which is totally different from conventional representations that treat videos as consecutive frame sequences. With such a representation, we show that by simply applying general model compression techniques, NeRV can match the performances of

traditional video compression approaches for the video compression task, without the need to design a long and complex pipeline. We also show that NeRV can outperform standard denoising methods. We hope that this paper can inspire further research works to design novel class of methods for video representations.

Limitations and Future Work. There are some limitations with the proposed NeRV. First, to achieve the comparable PSNR and MS-SSIM performances, the training time of our proposed approach is longer than the encoding time of traditional video compression methods. Second, the architecture design of NeRV is still not optimal yet, we believe more exploration on the neural architecture design can achieve higher performances. Finally, more advanced and cutting the edge model compression methods can be applied to NeRV and obtain higher compression ratios.

3.6 Experiment supplement

3.6.1 NeRV Architecture

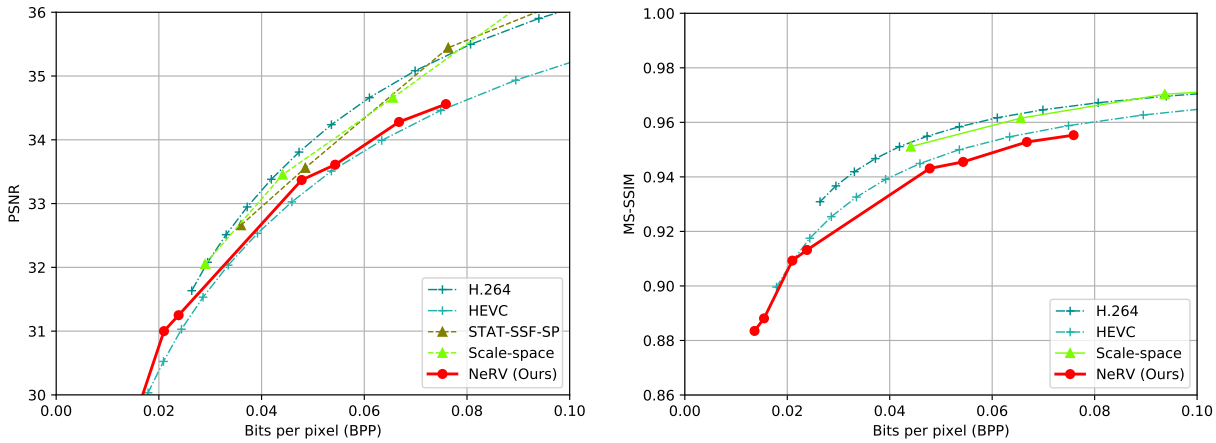
We provide the architecture details in Table 3.11. On a 1920×1080 video, given the timestamp index t, we first apply a 2-layer MLP on the output of positional encoding layer, then we stack 5 NeRV blocks with upscale factors 5, 3, 2, 2, 2 respectively. In UVG experiments on video compression task, we train models with different sizes by changing the value of C_1, C_2 to (48,384), (64,512), (128,512), (128,768), (128,1024), (192,1536), and (256,2048).

3.6.2 Results on MCL-JCL dataset

We provide the experiment results for video compression task on MCL-JCL [127]dataset in Figure 3.11a and Figure 3.11b.

Table 3.11: NeRV architecture for 1920×1080 videos. Change the value of C_1 and C_2 to get models with different sizes.

Layer	Modules	Upscale Factor	Output Size & $(C \times H \times W)$
0	Positional Encoding	-	$160 \times 1 \times 1$
1	MLP & Reshape	-	$C_1 \times 16 \times 9$
2	NeRV block	$5\times$	$C_2 \times 80 \times 45$
3	NeRV block	$3\times$	$C_2/2 \times 240 \times 135$
4	NeRV block	$2\times$	$C_2/4 \times 480 \times 270$
5	NeRV block	$2\times$	$C_2/8 \times 960 \times 540$
6	NeRV block	$2\times$	$C_2/16 \times 1920 \times 1080$
7	Head layer	-	$3 \times 1920 \times 1080$



(a) PSNR *v.s.* BPP

(b) MS-SSIM *v.s.* BPP

Figure 3.11: Rate distortion plots on the MCL-JCV dataset.

3.6.3 Implementation Details of Baselines

Following prior works, we used *ffmpeg* [131] to produce the evaluation metrics for H.264 and HEVC.

First, we use the following command to extract frames from original YUV videos, as well as compressed videos to calculate metrics:

```
ffmpeg -i FILE.y4m FILE/f%05d.png
```

Then we use the following commands to compress videos with H.264 or HEVC codec under *medium* settings:

```
ffmpeg -i FILE/f%05d.png -c:v h264 -preset medium \
    -bf 0 -crf CRF FILE.EXT

ffmpeg -i FILE/f%05d.png -c:v hevc -preset medium \
    -x265-params bframes=0 -crf CRF FILE.EXT
```

where FILE is the filename, CRF is the Constant Rate Factor value, and EXT is the video container format extension.

3.6.4 Video Temporal Interpolation

We also explore NeRV for video temporal interpolation task. Specifically, we train our model with a subset of frames sampled from one video, and then use the trained model to infer/predict unseen frames given an unseen interpolated frame index. As we show in Fig 3.12, NeRV can give quite reasonable predictions on the unseen frame, which has good and comparable visual quality compared to the adjacent seen frames.

3.6.5 More Visualizations

We provide denoising results on 'ig buck bunny' video in Figure 3.13. Given the noisy video as input, NeRV can reconstruct the original video with high fidelity. But it may also over-smooth some high-frequency details in the image and introduce blurry effect.

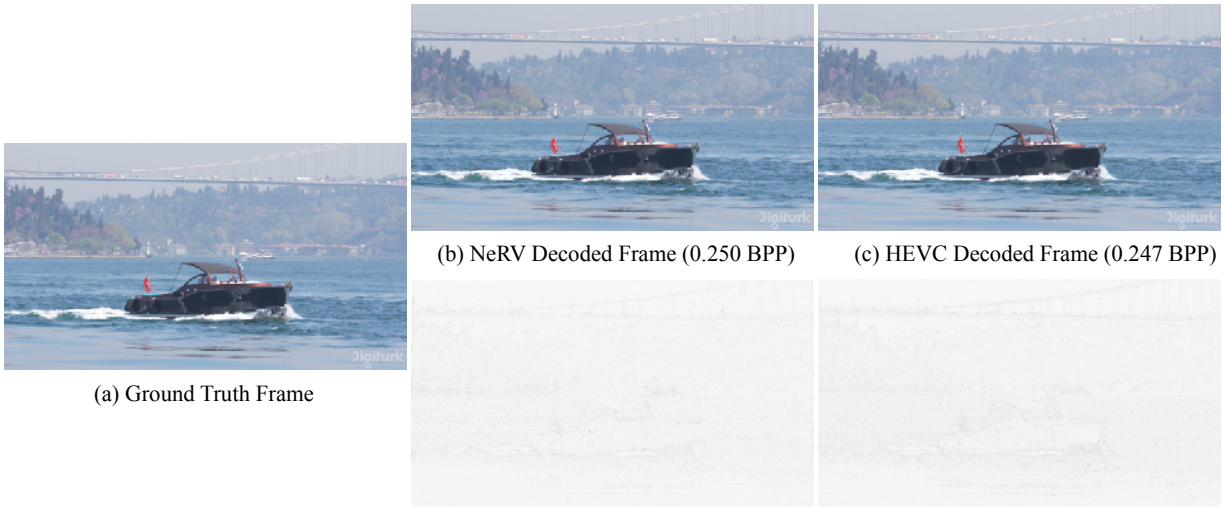
Besides, we provide more qualitative visualization results in Figure 3.14 to compare the our NeRV with H.265 for the video compression task. We test a smaller model on "Bosphorus"

(a) Seen Frame T (b) Interpolated Unseen Frame T+0.5 (c) Seen Frame T+1

Figure 3.12: Temporal interpolation results for video with small motion.

Figure 3.13: Denoising visualization. **Left**: Ground truth; **Middle**: Noisy input **Right**; NeRV output.

video, and it also has a better performance compared to H.265 codec with similar BPP. The zoomed areas show that our model produces fewer artifacts and the output is smoother.



(a) Ground Truth Frame

(b) NeRV Decoded Frame (0.250 BPP)

(c) HEVC Decoded Frame (0.247 BPP)

(d) NeRV Difference

(e) HEVC Difference

Figure 3.14: Video compression visualization. The difference is calculated by the L1 loss (absolute value, scaled by the same level for the same frame, and the darker the more different). *"Bosphorus"* video in UVG dataset, the residual visualization is much smaller for NeRV.

Chapter 4: HNeRV: A Hybrid Neural Representation for Videos

4.1 Introduction

Given the massive amount of videos generated every day, storing and transferring them efficiently is a key task in computer vision and video processing. Even for modern storage systems, the space requirements of raw video data can be overwhelming. Despite storage becoming cheaper, network speeds and I/O processing remain a bottleneck and make transferring and processing videos expensive.

Traditional video codecs, such as H.264 [132] and HEVC [9], rely on a manually-designed encoder and decoder based on discrete cosine transform [133]. With the success of deep learning, many attempts [10, 11, 12, 14, 15, 16, 18, 35, 134] have been made to replace certain components of existing compression pipelines with neural networks. Although these learning-based compression methods show high potential in terms of rate-distortion performance, they suffer from complex pipelines and expensive computation, not just to train, but also to encode and decode.

To address the complex pipelines and heavy computation, implicit neural representations [55, 60, 61, 63, 135] have become popular due to their simplicity, compactness, and efficiency. These methods show great potential for visual data compression, such as COIN [52] for image compression, and NeRV [1] for video compression. By representing videos as neural networks, video compression problems can be converted to model compression problems, which greatly simplifies the encoding

and decoding pipeline.

Implicit representation methods for video compression present a major trade-off: they embrace simplicity at the expense of generalizability. Given a frame index t as input, NeRV [1] uses a fixed position encoding function and a learnable decoder to reconstruct video frames from temporal embeddings. Another implicit representation, E-NeRV [136], takes a temporal embedding and spatial embedding to reconstruct video frames. Since the embeddings of NeRV and E-NeRV are based on spatial and/or temporal information only, without connection to the actual content of frames, they are content-agnostic. They can be computed on the fly which is necessary during training and decoding. This is quite elegant for video compression, since instead of storing many frame embeddings, one would only need to store model weights and basic metadata (*e.g.* , number of frames).

However, this comes with some major disadvantages. Firstly, since embeddings are content-agnostic, and due to how the temporal embeddings are computed, there is no way to meaningfully interpolate between frames. Secondly, and more importantly, the positional embedding used by the fully-implicit models provides no visual prior and limits the regression capacity, since all the information need to be learned by and stored in the video decoder.

In this paper, we propose this learnable encoder as a key component of hybrid neural representation for videos (HNeRV, Figure 4.1(top). Our proposed neural representation is a hybrid between implicit (network-centric) and explicit (embedding-centric) approaches since it stores videos in two parts: the tiny content-adaptive frame embeddings and a learned neural decoder. Besides the issue of content-agnostic embedding, prior work such as NeRV also suffers from an imbalance in the distribution of model parameters. In these decoders, later layers (closer to the output image) have much fewer parameters than earlier layers (closer to the embedding).

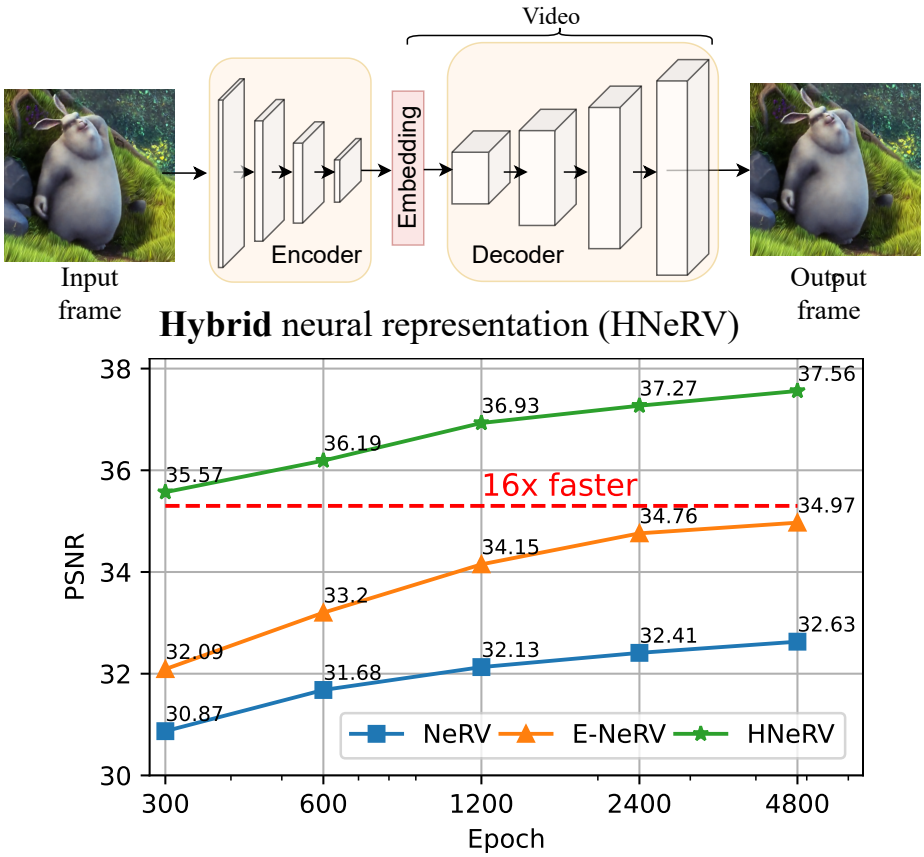


Figure 4.1: **Top) Hybrid** neural representation with learnable and content-adaptive embedding (ours). **Bottom) Video regression** for hybrid and implicit neural representations.

This hinders NeRV's ability to effectively reconstruct massive video content while preserving frame details. To rectify this, we introduce the HNeRV block, which increases kernel sizes and channel widths at later stages. With HNeRV blocks, we can build video decoders with parameters that are more evenly distributed over the entire network. As a hybrid method, HNeRV improves reconstruction quality for video regression and boosts the convergence speed by up to $16\times$ compared to implicit methods, shown in Figure 4.1(bottom). With content-adaptive embeddings, HNeRV also shows much better internal generalization (ability to encode and decode frames from the video that were not seen during training), and we verify this by frame interpolation results in Section 4.4.2.

HNeRV only requires a network forward operation for video decoding, which offers great advantages over traditional codecs and prior deep learning approaches in terms of speed, flexibility, and ease of deployment. Additionally, most other video compression methods are auto-regressive and there is a high dependency on the sequential order of video frames. In contrast, there is no dependency on the sequential order of frames for HNeRV, which means it can randomly access frames efficiently to decode frames in parallel. Such simplicity and parallelism make HNeRV a good codec for further speedups, like a special neural processing unit (NPU) chip, or parallel decoding with huge batches.

HNeRV is still viable for video compression, while also showing promising performance for video restoration tasks. We design our encoder such that it can also be compressed; additionally, our HNeRV decoder blocks perform well in the model compression regime, such that HNeRV is competitive with state-of-the-art methods. We posit that neural representation can be robust to distortion in pixel space and therefore restore well for video distortions, and verify this observation on the video inpainting task.

In summary, we propose a hybrid neural representation for videos. With content-adaptive embedding and re-designed architecture, HNeRV shows much better video regression performance over implicit methods, in reconstruction quality ($+4.7$ PSNR), convergence speed ($16\times$ faster), and internal generalization. As an efficient video codec, HNeRV is easy to deploy, and is simple, fast, and flexible during video decoding. Finally, HNeRV shows good performance over downstream tasks like video compression and video inpainting.

4.2 Related Work

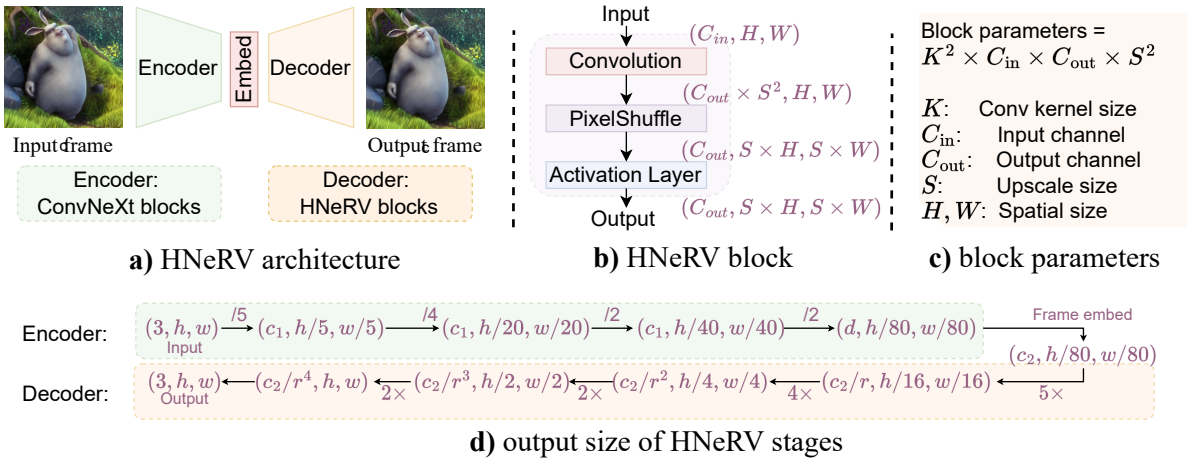
Neural Representation. Implicit representations fit to each individual test signal [53] where the model is regressed to a given image, scene, or video. Most implicit neural representations are coordinate-based. These coordinate-based implicit representations are used in image reconstruction [54, 55], shape regression [60, 61], and 3D view synthesis [62, 63]. NeRV [1] instead proposes an image-wise implicit representation, which takes frame indices as inputs and leverages neural representation for fast and accurate video compression. Relying only on index, and not coordinates, speeds up the encoding and decoding process compared to coordinate-based (pixel-wise) methods. Based on NeRV, E-NeRV [136] further boosts the video regression performance via decoupling frame index and spatial index. Traditional autoencoders would not be considered implicit representations since most information is stored in their large image-specific embeddings. Nevertheless, they are a form of neural representation, and HNeRV borrows the general concept for its encoder from standard U-shaped autoencoders [133, 137, 138, 139]. HNeRV keeps the embedding intentionally tiny and compact, so as to keep most of the representation implicit (stored in the decoder).

Video Compression. Traditional video compression methods such as MPEG [7], H.264 [8], and H.265 [9] achieve good reconstruction results with decent decompression speeds. Recently, deep learning techniques have been proposed for video compression. While these approaches focus on replacing the entire compression pipeline, they each borrow principles from the traditional handcrafted approaches. Some have framed the problem primarily as image compression and interpolation [10, 11], or attempt to solve this task with image compression via autoencoders [12], or focus purely on interpolation for the sake of compression [13]. Others essentially reformulate

traditional video compression pipelines using deep learning tools [14, 15, 16], at varying levels of complexity. Recent approaches have focused on tackling the computational inefficiencies of existing art, including by fine-tuning traditional codecs [17], and by optimizing pieces of the compression pipeline [18]. The approach which inspired much of this work, NeRV, responds to these same inefficiencies by proposing a specialized architecture for video memorization [1]. Once video is represented as a neural network, the video compression problem can be converted to a model compression problem and achieve good bit-distortion performance. With learnable embeddings and re-designed decoder blocks, HNeRV improves the video regression capacity and convergence speed, while video compression is still viable by model compression.

Model Compression. NeRV formulated video compression as model compression [1], which is a diverse area. In this paper we apply only a small subset of possible methods. We use weight pruning [140] and weight quantization [105, 141, 142]. We also use entropy encoding for lossless compression after pruning and quantization [102, 122]. Note that many other model compression methods can be leveraged to further reduce the size and video neural representation can always benefit from developments in the model compression area.

Video Inpainting & Internal Learning. Video inpainting is typically framed as some combination of object removal and attempting to recreate missing regions of images. Whereas some methods rely on priors from training on large datasets [143], ours has more in common with a recent zero-shot fully-internal approach [144]. We define "Internal learning" in terms of exploiting recurrence of information within a single domain, like within an image [145]) or within an video [146]. It can be thought of as a sort of DIP-for-video, a line of work that was started for images with DIP [6] and extended for video by double-DIP [147]. Other methods have embraced this paradigm partially, learning some priors from large external datasets, before learning video-



a) HNeRV architecture **b)** HNeRV block **c)** block parameters

d) output size of HNeRV stages

Figure 4.2: **a)** HNeRV uses ConvNeXt blocks to encode frames as tiny embeddings, which are decoded by HNeRV blocks. **b)** HNeRV blocks consist of three layers: convolution, PixelShuffle, and activation (with input/output size illustrated). **c)** We demonstrate how to compute parameters for a given HNeRV block. **d)** Output size of each stage with strides 5,4,2,2.

specific priors via internal learning [143].

4.3 Method

We first give an overview of HNeRV (section 4.3.1). We explain what makes HNeRV a "hybrid" representation, and the advantages that this offers. We provide architectural details, loss functions, and explain how we compute the size of our video representation. We then give particulars necessary for utilizing HNeRV on downstream tasks (section 4.3.2). These include vector quantization for video compression and reconstruction loss for inpainting.

4.3.1 HNeRV overview

HNeRV can be viewed as an autoencoder with tiny embeddings (Figure 4.2(a)). We choose ConvNeXt blocks [148] to build the encoder (1 block per stage), and propose novel HNeRV blocks (Figure 4.2(b)) to build the decoder.

Hybrid Neural Representation. Compact video representations can be divided into two parts: explicit methods and implicit methods. Explicit methods use an autoencoder to encode and decode all videos, and store content *explicitly* as a latent embedding. Given a *video-specific embedding* as input, the decoder can reconstruct the video. Implicit methods use only a learnable decoder to represent the video. Given fixed frame index as input, the *video-specific decoder* can reconstruct the video. With content-adaptive embedding as input, explicit representation shows better generalization and compression performance, while implicit representations have a much simpler encoding/decoding pipeline and a high potential for compression (benefits from model compression techniques) and other downstream tasks (*e.g.* efficient video dataloader, video denoising, inpainting). In this paper, we propose a hybrid neural representation to combine the advantages of both explicit and implicit methods. Similar to implicit representation, we use a learnable decoder to model video separately and store most content implicitly in the video-specific decoder. To achieve better reconstruction, we use a learnable embedding as input and store information explicitly in these frame-specific embeddings, which is similar to explicit methods. Therefore, we can use any powerful encoder to generate tiny content-adaptive embeddings to boost the performance of implicit representation. Since these embeddings are quite small (*e.g.* a 128-d vector for a 640×1280 frame), our hybrid neural representation is as compact as implicit methods, but with stronger capacity, faster convergence, and better internal generalization, while keeping the full potential for downstream tasks.

Model Architecture. Similar to a NeRV block, it consists of three layers: convolution layer, pixelshuffle layer, and activation layer. Within an HNeRV block, only the convolution layer has learnable parameters (Figure 4.2(b)). Illustrated in Table 4.1, a NeRV block uses fixed kernel sizes for all stages $K = 3$, and reduces channel width by 2, $\mathrm{C}_{\text{out}} = C_{\text{in}}/2$. Therefore, for blocks

Table 4.1: **HNeRV block NeRV block**. k is kernel size for each stage, C_{out} and C_{in} are output/input channels for each block. We decrease parameters via a small $k = 1$ for first block, and increase parameters for later layers with a larger k and wider channels.

NeRV blocks	$k = 3, \quad C_{\text{out}} = C_{\text{in}}/2$
HNeRV blocks	$k = 1, 3, ..., K_{\text{max}}, \quad C_{\text{out}} = C_{\text{in}}/r$

at later stages, the parameters are quite few and may not be strong enough to store video content at high resolution. In contrast, we increase the kernel size and channel width for later HNeRV blocks, where K increases from 1 (stage 1), 3 (stage 2), to K_{max} (5, *etc*. for later stages), and we decrease channel width by a reduction factor r (1.2, *etc*.). With kernel size 1, the first block has much fewer parameters; with larger kernel size and wider channels, HNeRV blocks at later stages are much stronger; and we therefore get a more even distribution of model parameters across layers. We list output sizes of various stages in Figure 4.2(d), with embedding dimension d and channel reduction r. Each stage has one block, and we use a 1×1 convolution layer to get low-dimension frame embeddings (channel width from c_1 to d), and a 3×3 convolution layer for final image predictions (channel width from c_2/r^4 to 3).

Loss Functions. Since HNeRV attempts to reconstruct video with high fidelity, we use the loss objective $L = \text{Loss}(x, p)$, where x is the input frame, p is the HNeRV prediction, and 'Loss' is any reconstruction loss function like L2, L1, or SSIM loss.

Total size. As a hybrid neural representation, we include both frame embedding and decoder parameters to compute the total size of our video representation: TotalSize $=$ EmbedSize $+$ DecoderSize.

4.3.2 Downstream tasks

Video Compression. We leverage both model compression and embedding quantization for video compression. Similar to NeRV, we apply global unstrutured pruning, model quantization, and weight entropy encoding for model compression (details can be found in the appendix).

For quantization of a vector μ, we linearly map every element to the closest integer,

$$\begin{aligned} \mu_i = \text{Round}\left(\frac{\mu_i - \mu_{\text{min}}}{\text{scale}}\right) * \text{scale} + \mu_{\text{min}}, \text{where} \\ \text{scale} = \frac{\mu_{\text{max}} - \mu_{\text{min}}}{2^b - 1}, \end{aligned} \tag{4.1}$$

μ_i is vector element, 'Round' is a function that rounds to the closest integer, b is the quantization bit length, μ_{max} and μ_{min} are the max and min value of vector μ, and 'scale' is the scaling factor.

Video Inpainting. For partially distorted video, we only compute loss for non-masked pixels,

$$L_{\text{inpainting}} = (1 - M) * \text{Loss}(x, p) \tag{4.2}$$

where M is the mask matrix where distorted pixels are 1 and other are 0. For inpainting output, following IIVI [144], we fill the masked region with HNeRV's output.

4.4 Experiments

We first provide information necessary for replicating our results, including datasets used and hyperparameter settings (Sec. 4.4.1). We then show our main results for video regression, decoding, and internal generalization (Sec. 4.4.2). We show the effectiveness of decoder-side parameter-redistribution for improving the appearance of high resolution frames (Sec. 4.4.3). We

Table 4.2: Video regression with different **sizes**

Size	0.35M	0.75M	1.5M	3M	avg.
NeRV	26.99	28.46	30.87	33.21	29.88
E-NeRV	27.84	30.95	32.09	36.72	31.90
HNeRV	**30.15**	**32.81**	**35.57**	**37.43**	**33.90**

Table 4.3: Video regression with different **epochs**

Epoch	300	600	1200	1800	2400	3600
NeRV	28.46	29.15	29.57	29.73	29.77	29.86
E-NeRV	30.95	32.07	32.79	33.1	33.36	33.67
HNeRV	**32.81**	**33.89**	**34.51**	**34.73**	**34.88**	**35.03**

Table 4.4: Video regression at resolution **960**×**1920**, PSNR↑ reported

Video	beauty	swan	bmx	bosph	dance	camel	bee	jockey	ready	shake	yach	avg.
NeRV	33.25	28.48	27.86	33.22	26.45	24.81	37.26	31.74	24.84	33.08	28.30	29.94
E-NeRV	33.17	29.38	28.68	33.69	27.88	25.16	37.62	31.63	25.24	34.39	28.42	30.48
HNeRV	**33.58**	**30.35**	**29.98**	**34.73**	**30.45**	**26.71**	**38.96**	**32.04**	**25.74**	**34.57**	**29.26**	**31.49**

Table 4.5: Video regression at resolution **480**×**960**, PSNR↑ reported

Video	beauty	swan	bmx	bosph	dance	camel	bee	jockey	ready	shake	yach	avg.
NeRV	36.27	29.75	28.81	35.07	29.47	26.75	40.76	32.58	25.81	35.33	30.11	31.88
E-NeRV	36.26	30.27	29.20	36.06	30.83	27.39	**43.26**	32.70	26.19	35.64	30.38	32.56
HNeRV	**36.91**	**31.92**	**31.27**	**36.95**	**33.85**	**28.85**	42.05	**33.33**	**27.07**	**36.97**	**30.96**	**33.65**

Table 4.6: **Internal generalization** results. NeRV, E-NeRV, and HNeRV use interpolated embedding as input, HNeRV† uses held-out frames as input. With content-adaptive embedding as input, HNeRV shows much better reconstruction on held-out frames

Method	beauty	swan	bmx	bosph	dance	camel	bee	jockey	ready	shake	yach	avg.
NeRV	28.05	17.94	15.55	30.04	16.99	14.83	36.99	20.00	17.02	29.15	24.50	22.82
E-NeRV	27.35	19.4	15.12	28.95	17.16	17.97	38.24	19.39	16.74	30.23	22.45	23.00
HNeRV	30.97	21.44	17.35	34.38	20.2	19.93	38.83	23.67	20.90	**32.69**	**27.30**	26.15
HNeRV†	**31.10**	**21.97**	**18.29**	**34.38**	**20.29**	**20.64**	**38.83**	**23.82**	**20.99**	32.61	27.24	**26.38**

demonstrate compelling initial results for downstream tasks including video compression and video inpainting (Sec. 4.4.4). Finally, we offer results for extensive ablation studies, both here (Sec. 4.4.5) as well as in the appendix.

4.4.1 Dataset and Implementation Details

We use the Big Buck Bunny (Bunny) [149], UVG [86] and DAVIS [127] datasets. Bunny has 132 frames with resolution 720×1280, and we center-crop 640×1280 to get tiny spatial size (*e.g.* 1×2) for embedding. UVG dataset has 7 videos[1] with size 1080×1920 at FPS 120 of 5s or 2.5s, and we center-crop 960×1920. We also take 10 videos[2] from DAVIS validation (1080×1920, 50-200 frames) and center crop the 960×1920. Unless otherwise specified, we use the Adam optimizer, with beta as (0.9, 0.999), weight decay as 0, and learning rate at 0.001 with cosine learning rate decay. We also use batch size as 2 and L2 loss as reconstruction loss function. K_{max} is set as 5, reduction r is set as 1.2 in Table 4.1. We set stride list as (5,4,4,2,2), (5,4,3,2,2), and (5,4,4,3,2) for video resolutions of 640×1280, 480×960, and 960×1920 respectively.

For evaluation metrics, we use PSNR and MS-SSIM to evaluate reconstruction quality, bits per pixel (bpp) for compression, and pixels per pixel (ppp) for model compactness. We conduct all experiments in Pytorch with RTX2080ti GPUs, where it takes around 8s per epoch to train a 130 frame video of size 640×1280. We choose HNeRV's size to ensure the PSNR lies between 30-40 for fair video reconstruction. We provide more experiment details such as architecture details, qualitative results, number of some plots, and per-video compression results, in the supplementary material..

4.4.2 Main Results

Video regression. We first compare HNeRV with implicit methods NeRV and E-NeRV on Bunny. For fair comparison, we scale channel width to make total size comparable as the

[1] Beauty, Bosphorus, HoneyBee, Jockey, ReadySetGo, ShakeNDry, YachtRide

[2] bike-packing, blackswan, bmx-trees, breakdance, camel, car-round, car-shadow, cows, dance-twirl, dog

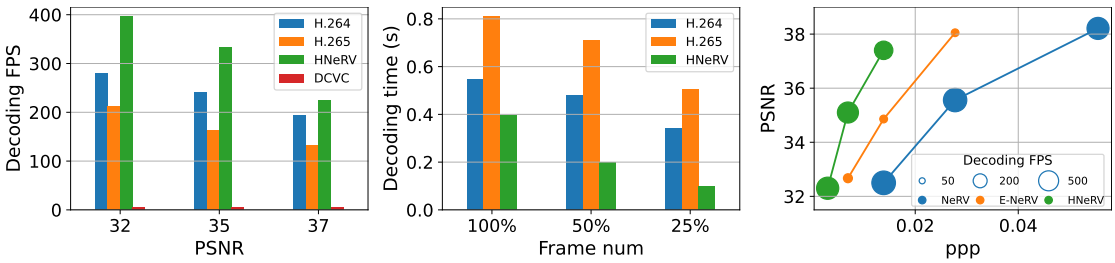


Figure 4.3: **Video decoding. Left**: HNeRV outperforms traditional video codecs H.264 and H.265, and learning-based compression method DCVC. **Middle**: HNeRV shows much better flexibility when decoding only a portion of video frames, where the decoding time decreases linearly for HNeRV while other methods still need to decode most frames. **Right**: HNeRV performs well for compactness (ppp), reconstruction quality (PSNR), and decoding speed (FPS).

Left: NeRV Middle: E-NeRV Right: HNeRV

Figure 4.4: Visualization of **Embedding interpolation**.

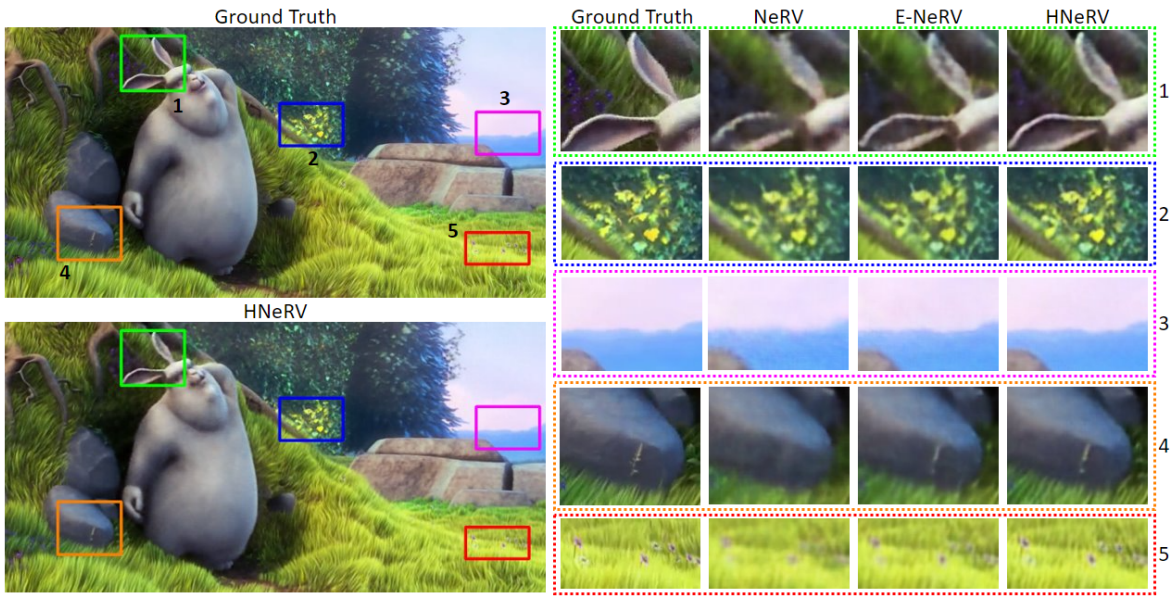


Figure 4.5: **Visualization of video neural representations** at 0.003 ppp, which means the total size is only about 0.3% of the original video size. On the **left**, we compare HNeRV to ground truth. On the **right**, we compare NeRV, E-NeRV, and HNeRV for 5 patches with discernible differences, indicated in the original frame by numbers and bounding boxes. For each patch, HNeRV preserves detail at a level of fidelity closer to the ground truth.

original paper did. In Table 4.2, with the same size and 300 epochs, HNeRV outperforms both NeRV and E-NeRV. We also show comparison of different training time in Table 4.3 with 0.75M size and in Figure 4.1(Right) with 1.5M size, where HNeRV converges much faster compared to implicit methods. We show improvements qualitatively as well in Figure 4.5. As a compact representation, HNeRV reconstructs the video well with only 0.35M parameters, at 0.003 ppp. We also evaluate it on 7 UVG videos and 4 DAVIS videos, where HNeRV shows large improvements at resolution 960×1920 in Table 4.4, and its resized 480×960 version in Table 4.5, with size 3M and 300 epochs.

Video decoding. We evaluate video decoding on Bunny with channel reduction r as 1.5, where H.264 and H.265 are tested with 4 CPUs[3], while DCVC [150] and HNeRV are tested with 1 GPU (RTX2080ti). We only measure the forward time for DCVC and HNeRV. We compare video

[3]Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz

decoding at various reconstruction qualities (PSNR at 32, 35, and 37) in Figure 4.3(Left), where HNeRV outperforms traditional codecs (H.264 and H.265) and learning-based DCVC. Note that although many prior learning-based compression methods show bit-distortion improvements, their decoding speeds *lag far behind* traditional codecs and neural representation. Besides, most compression methods encode and decode frames in an auto-regressive way and can not access frames randomly. Compared to these methods, the decoding of HNeRV is much simpler and can be deployed to any platform easily. We compare decoding time in Figure 4.3(Middle) (PSNR at 35) where $100\%, 50\%$, and 25% of frames (evenly sampled, to mimic *e.g.* a discrete reduction of FPS) are decoded. Since there is no dependency among video frames, HNeRV can decode them in parallel and decoding time decreases linearly with the number of frames decoded. In contrast, H.264 and H.265 still need to decode most frames, even though only some of them are needed. Finally, we compare with implicit methods in Figure 4.3(Right), where HNeRV is slightly slower than NeRV since the computation of later layers is more expensive due to large K and channel width. As a hybrid neural representation, HNeRV achieves much better trade-offs with respect to compactness (ppp), reconstruction quality (PNSR), and decoding speed (FPS).

Internal generalization. Since HNeRV leverages content-adaptive embeddings, we also evaluate it for the video interpolation task. Holding out every other frame as a test set, NeRV, E-NeRV, and HNeRV use interpolated embedding as input, while HNeRV† uses the test frame as input. With learnable and content-adaptive embedding, our HNeRV shows much better generalization, quantitatively in Table 4.6 and qualitatively in Figure 4.4.

Table 4.7: Analysis of parameter rebalancing.

		NeRV		HNeRV	
K	r	PSNR	MS-SSIM	PSNR	MS-SSIM
3,3	2	30.87	0.9341	29.91	0.9203
3,3	1.2	32.27	0.9496	33.09	0.9587
1,5	2	31.34	0.9399	34.32	0.9715
1,5	1.2	**33.03**	**0.9573**	**35.57**	**0.9773**

4.4.3 Parameter Distribution Analysis

As part of our novel architectural innovation, we balance the parameters in the HNeRV decoder. While NeRV-like architectures naturally have a vanishingly small number of parameters in the final layers, fig. 4.6 shows how we adjust such that the initial and final layers have a roughly equal number of parameters. Table 4.7 demonstrates how more even parameter distribution achieves optimal results not only for HNeRV, but also for NeRV[4]: setting K_{min} to 1 and K_{max} to 5, with r at 1.2, maximizes PSNR and MS-SSIM for both architectures. table 4.9 and table 4.10 further verify that these hyperparameters are optimal for HNeRV. We offer these results as strong evidence that our parameter balancing approach not only enables the success of HNeRV, but would be broadly applicable to NeRV-like networks.

4.4.4 Downstream Tasks

Video compression. With model pruning (10% pruned), embedding quantization (8 bits), model quantization (8 bits), and model entropy encoding (8% saved), we show video compression results on UVG in Figure 4.7. HNeRV outperforms the implicit method, NeRV, and traditional video codecs H.264 and H.265. *Note that HNeRV achieves this using a small model for each*

[4]We label NeRV with K_{max}=5 as (1,5) since its FC layer can be seen as a 1×1 convolution layer.

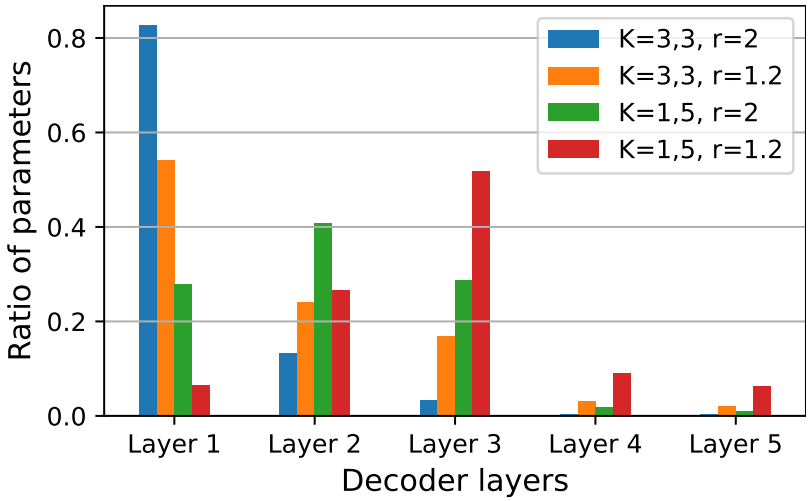


Figure 4.6: Parameter distributions for decoder blocks. See table 4.7 for PSNR and MS-SSIM results with these 4 settings.

Table 4.8: Video **inpainting** results. With 5 fixed box masks on input videos, we evaluate the output with PSNR ↑. 'Input' is the baseline of mask video and ground truth

Video	bike	b-swan	bmx	b-dance	camel	c-round	c-shadow	cows	dance-twirl	dog	avg.
Input	23.14	20.24	19.99	21.36	17.3	20.47	18.92	19.37	20.45	18.39	19.96
NeRV	30.94	33.43	32.07	27.82	31.99	29.09	31.63	30.08	30.45	33.85	31.14
IIVI	**31.87**	**36.02**	**34.36**	27.63	**35.11**	**32.61**	**33.69**	**31.26**	**31.44**	**35.7**	**32.97**
HNeRV	31.27	34.24	33.95	**27.94**	32.21	30.88	33.07	30.82	31.21	34.7	32.03

video, while NeRV fits a big model on concat videos (for better compression) which greatly slows down the encoding and decoding speed. We also show the best and worst cases of HNeRV video compression for the 'honeybee' and 'readysteadygo' videos respectively in Figure 4.8, where HNeRV achieves outstanding performance when the camera is not moving, such as with the 'honeybee' video ($10\times$ smaller than H.265 for equivalent PSNR). Given the limited performance on videos of highly dynamic scenes, we propose finding a good size and network architecture for such videos as future work.

Video inpainting. We also explore video inpainting with fixed and object masks. For fixed masks, we use 5 boxes of width 50 (Figure 4.9 top) and show quantitative results in Table 4.8

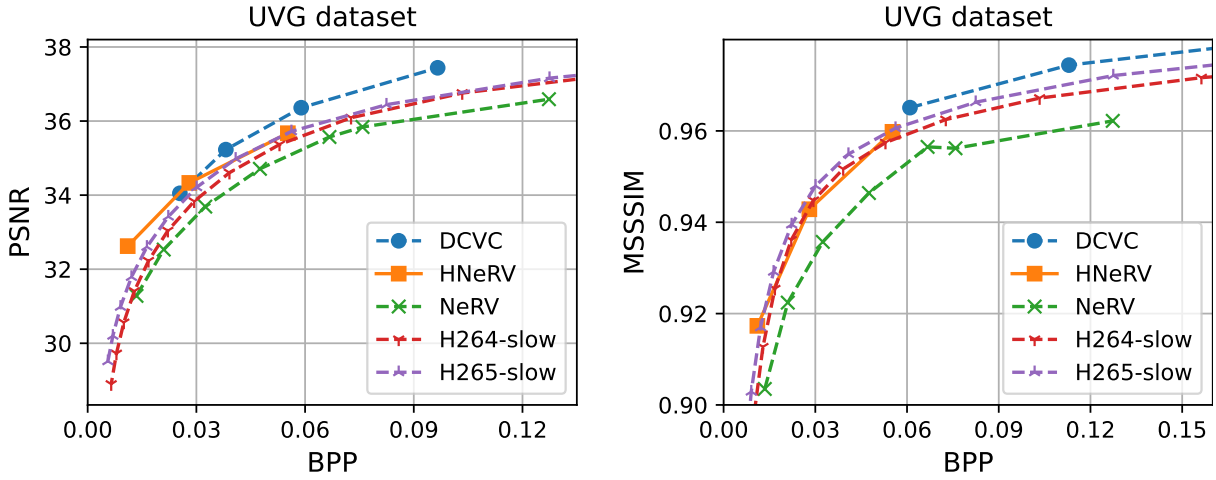


Figure 4.7: **Compression** results on UVG dataset.

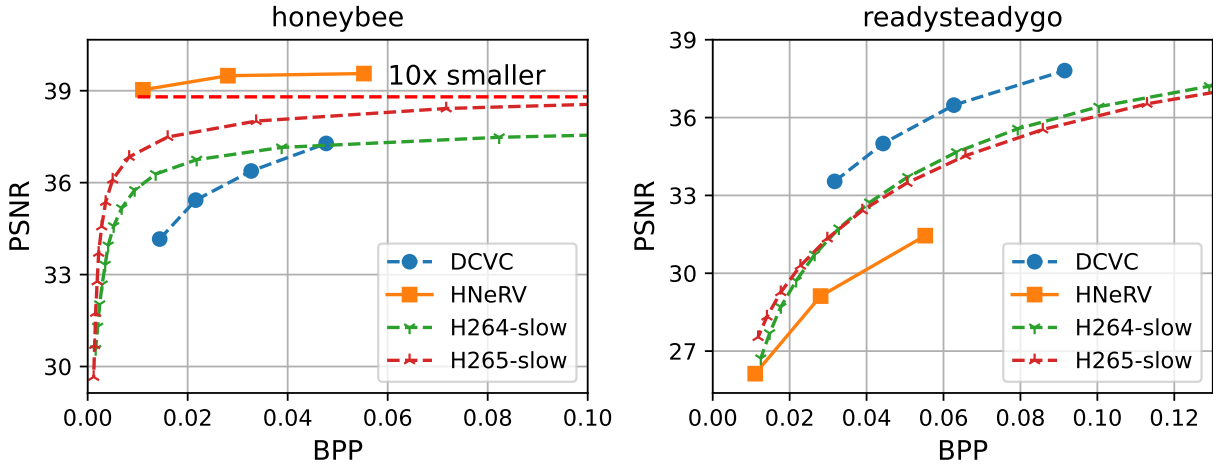


Figure 4.8: Compression results of **best/worst cases** from UVG dataset. HNeRV achieves good performance especially for videos caputured by still cameras, like 'honeybee' video.

where HNeRV improves inpainting performance over the implicit method, NeRV. Although we do not have any specific design for the inpainting task, HNeRV even achieves comparable performance with an SOTA inpainting method, IIVI [144]. We show qualitative results in Figure 4.9 and the appendix.

Table 4.9: **Kernel size** (K_{min}, K_{max}) ablation, (with r=1.2)

K	PSNR	MS-SSIM
1,3	35.02	0.9752
1,5	**35.57**	**0.9773**
1,7	35.07	0.9757
3,3	33.09	0.9587

Table 4.10: **Channel reduction** r ablation, (with K=1,5)

r	PSNR	MS-SSIM
1	34.96	0.9745
1.2	**35.57**	**0.9773**
1.5	34.98	0.9762
2	34.32	0.9715

Table 4.11: **Embedding spatial size** ablation

$h \times w$	PSNR	MS-SSIM
1×2	34.79	0.9735
2×4	**35.57**	**0.9773**
4×8	35.12	0.9761

Table 4.12: **Embedding dimension** ablation

d	PSNR	MS-SSIM
8	35.13	0.9770
16	**35.57**	**0.9773**
32	35.08	0.9758

4.4.5 Ablation study

We show the effectiveness of even-distributed parameters in Table 4.9 and Table 4.10 by increasing kernel size and channel width of later layers. For the NeRV block, it uses fixed $K = 3$, and channel reduction factor $r = 2$. We also show an embeddings ablation study, for spatial size ($h \times w$) in Table 4.11 and embedding dimensions (d) in Table 4.12.

4.5 Conclusion

In this paper, we propose a hybrid neural representation for videos (HNeRV). With content-adaptive embedding and evenly-distributed parameters, HNeRV improves video regression performance compared to implicit methods in reconstruction quality, convergence speed, and internal generalization. As a video representation, HNeRV is also simple, fast, and flexible for video decoding, and shows good performance for video compression and inpainting.

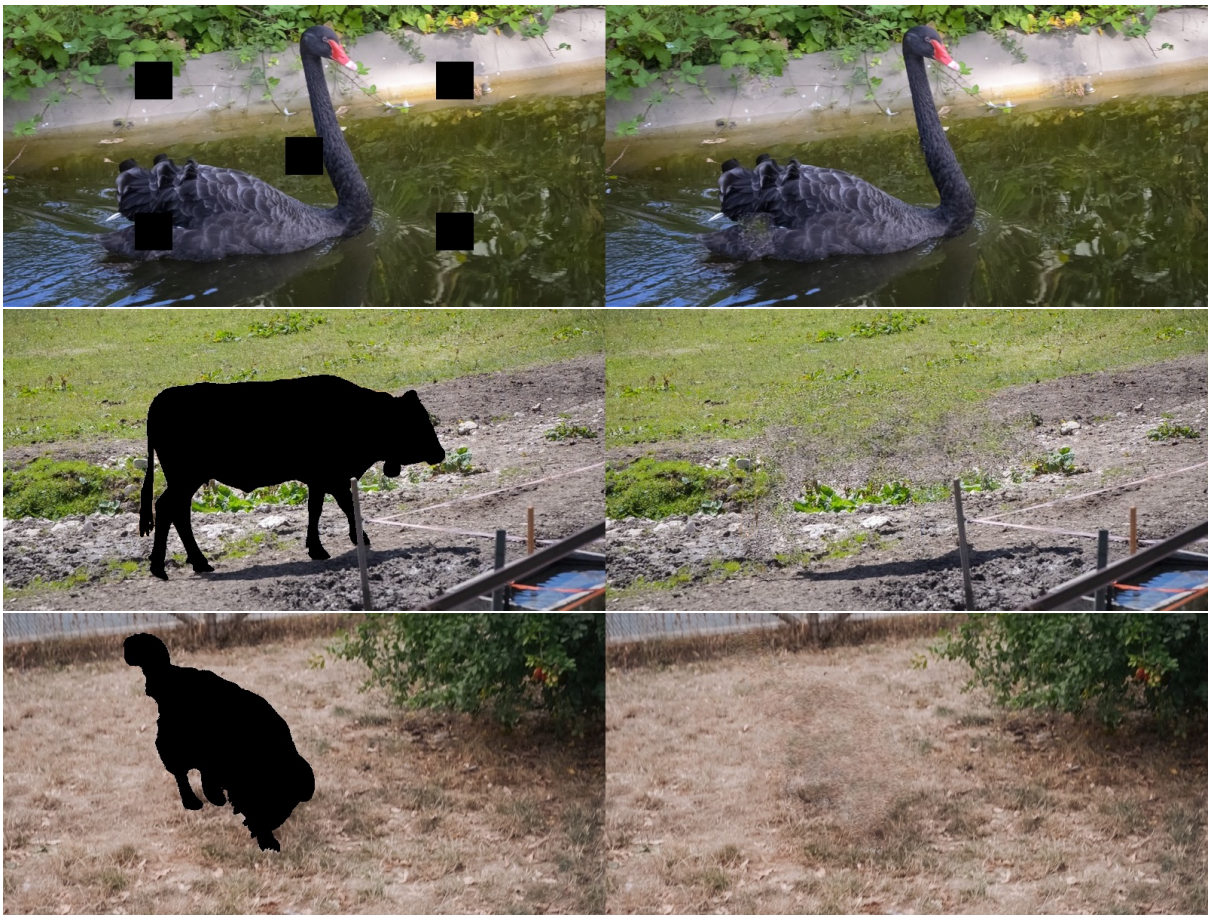


Figure 4.9: **Inpainting** results of fixed masks and object masks. **Left)** input frame; **Right)** HNeRV output.

There are many limitations of HNeRV as well. Firstly, as a neural representation, HNeRV stores each video as a neural network. Given a new video, HNeRV still needs time to train to fit the video. Secondly, although HNeRV can represent a video well, finding a best-fit embedding size, model size, and network architecture (K_{max}, r, *etc.*) remains an open problem.

4.6 Experiment supplement

4.6.1 Video decoding

We firstly show command to evaluate decoding speed of H.264 and H.265:

ffmpeg -threads ThreadsNum -i Video -preset medium -f null -benchmark -

And we also show quantitative decoding results in Table 4.13, 4.14, and Table 4.15. In Table 4.15, we can further increase video decoding speed with a smaller channel width (*i.e.*, a big reduction factor $r = 2$).

Table 4.13: Decoding FPS ↑

PSNR	32	35	37
H.264	279.7	240.9	192.7
H.265	211.9	163.2	132.5
DCVC	4.7	4.6	4.5
HNeRV	**395.9**	**332.7**	**224.8**

Table 4.14: Decoding time (s) ↓

# Frames	100%	50%	25%
H.264	0.548	0.480	0.343
H.265	0.809	0.708	0.506
DCVC	27.913	24.424	17.446
HNeRV	**0.397**	**0.198**	**0.099**

Table 4.15: HNeRV Decoding FPS

PSNR	32	35	37
r=1.5	395.9	332.7	224.8
r=1.75	397.4	373.8	320.7
r=2	**405.5**	**383.3**	**350.5**

4.6.2 Video compression

Then we show the details for downstream tasks of video compression, which can be divided into three steps: global unstructure pruning, quantization, and entropy encoding.

1) Model Pruning. Given a pre-trained model, we use global unstructured pruning to reduce

the model size, where parameters below a threshold are pruned and set as zero. For a model parameter θ_i, $\theta_i = \begin{cases} \theta_i, & \text{if } \theta_i \geq \theta_q \\ 0, & \text{otherwise,} \end{cases}$ where θ_q is the q percentile value for all model parameters θ. As a normal practice, we fine-tune the model to regain the representation after pruning.

2) Model and embedding quantization. Model quantization and embedding quantization follow the same scheme. Given an vector μ, we linearly map every element to the closest integer,

$$\begin{aligned} \mu_i &= \text{Round}\left(\frac{\mu_i - \mu_{\text{min}}}{\text{scale}}\right) * \text{scale} + \mu_{\text{min}}, \text{where} \\ \text{scale} &= \frac{\mu_{\text{max}} - \mu_{\text{min}}}{2^{\text{b}} - 1} \end{aligned} \tag{4.3}$$

μ_i is one vector element, 'Round' is a function that rounds to the closest integer, 'b' is the bit length for quantization, μ_{max} and μ_{min} are the max and min value of vector μ, and 'scale' is the scaling factor. For scaling factor and zero points at this step, we can also try other methods instead of current min-max one, like choosing 2^b evenly-distributed values to minimum the mean square error.

3) Entropy encoding. Finally, we use entropy encoding to further reduce the size. Specifically, we leverage Huffman coding [122] for quantized weights and get lossless compression.

4.6.3 Weight Pruning for Model Compression.

We appreciate this concern, which has been unresolved since the original NeRV paper. By applying entropy encoding (assigning fewer bits for frequent symbols), we can store pruned weights with limited bits, since all pruned weights share a frequent symbol: 0. We provide corrected model compression results in table 4.16, and will update the paper accordingly. We use 2 baselines (models with no pruning) – one where the model is only quantized, and another where

we apply entropy encoding after quantization. As we prune more parameters, entropy encoding enables us to use fewer bits to store the sparse model weights.

Table 4.16: Compression results. "Size ratio" compares to model with quant. only, and "Sparsity" indicates amount of weights pruned.

Compression	Quant	Prune + Quant + Entropy coding				
Sparsity	0%	0%	10%	15%	20%	25%
PSNR	37.61	37.56	37.51	37.32	37.02	36.61
Size (bits)	11.54M	10.94M	10.41M	10.09M	9.77M	9.36M
Size ratio	100%	94.8%	90.2%	87.4%	84.7%	81.1%

4.6.4 HNeRV architecture details

We also provide architecture details for HNeRV models in various tasks and datasets in Table 4.17, with total size, strides list, encoder dimension $c1$, embedding dimension d, channel width of decoder input $c2$, channel reduction r, lowest channel width Ch_{min}, min and max kernel size K_{min}, K_{max} .

Table 4.17: HNeRV architecture details

Video size	size	strides	c1	d	c2	r	Ch_{min}	K_{min}, K_{max}
640×1280	0.35	5,4,4,2,2	64	16	32	1.2	12	1,5
640×1280	0.75	5,4,4,2,2	64	16	48	1.2	12	1,5
640×1280	1.5	5,4,4,2,2	64	16	68	1.2	12	1,5
640×1280	3	5,4,4,2,2	64	16	97	1.2	12	1,5
480×960	3	5,4,3,2,2	64	16	110	1.2	12	1,5
960×1920	3	5,4,4,3,2	64	16	92	1.2	12	1,5

4.6.5 Per-video compression results

We also show video compression results for **UVG** videos in Figure 4.10.

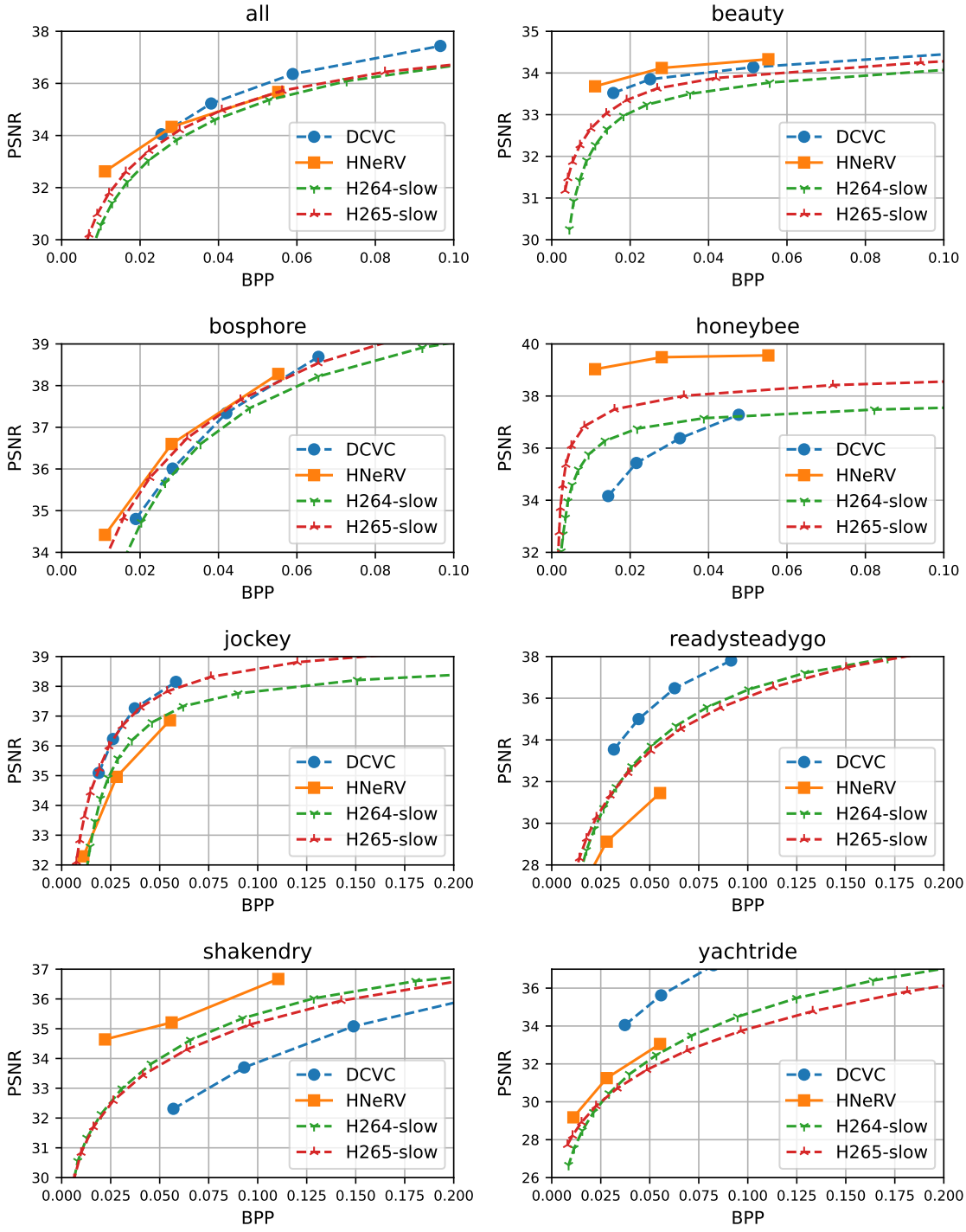


Figure 4.10: **Compression** results averaged across all **UVG** videos, and for each specific videos.

Chapter 5: NVLoader: A Neural Video Dataloader for Efficient Data Loading

5.1 Introduction

Data loading is an essential component in video research, especially for large-scale experiments. Compared to image dataloaders, video loading has a complex decoding pipeline and varied data access patterns. Computationally intensive dataloaders lead to long research cycles [151], are a severe hindrance in training models with massive datasets, and are often a bottleneck for efficient video models [152]. This is especially true for methods that require random access to patches (*e.g.* , transformers [5, 153, 154]), as opposed to full frames, since existing video dataloaders first decode the entire frame and then extract patches via post-processing. In some cases, a subset of frame patches (as few as 10%) are used as input, since masking out patches is a good augmentation strategy to improve generalization, and a common strategy to speedup training and boost performance in self-supervised learning [155, 156]. The ratio of data loading time rises from 4.3% to 31.5% in video MAE [156], which is a $7.8\times$ slow down even after using heuristics like repeat sampling and caching.

In this work, we present a drastically different video loading framework by leveraging video neural representations. By storing each video implicitly as a deep neural network, neural representations have shown a lot of promise in reconstructing the video content efficiently by inputting a frame index or pixel coordinate [1, 136, 157, 158]. Such video neural representations

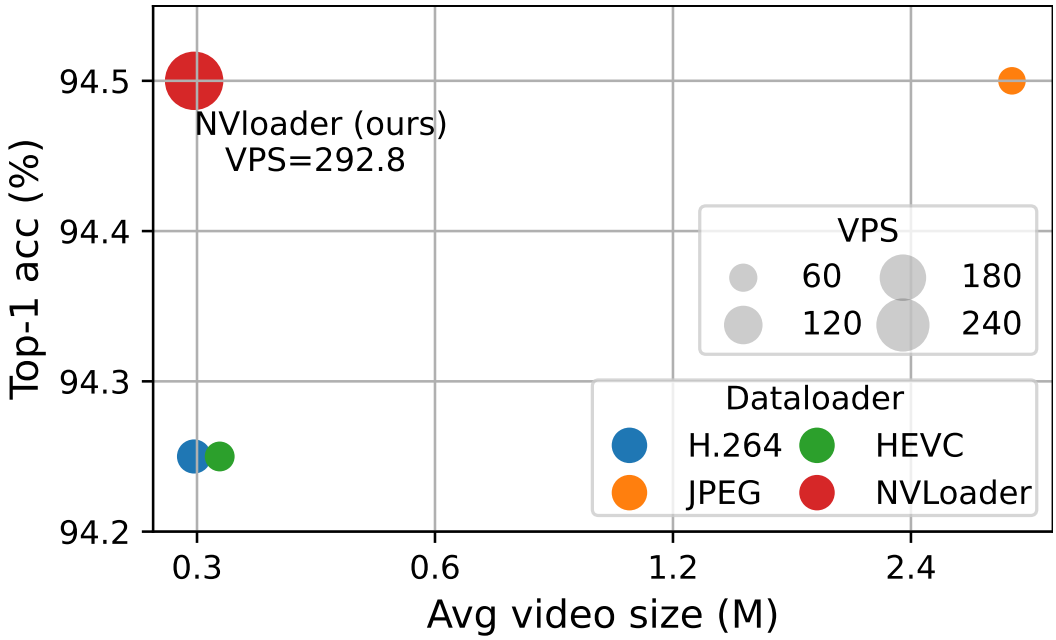


Figure 5.1: Comparison of **video dataloaders** based on H.264 videos, HEVC videos, JPEG frames, and NVLoader (ours). With similar video size, NVLoader load videos much more efficiently, measured by videos per second (VPS), without hurting accuracy for video recognition.

show promising results in many tasks, like video compression [1], spatial-temporal super-resolution [157, 158], and video inpainting [144, 157]. Inspired by its compactness, simplicity, and fast decoding, we propose a Neural Video Dataloader (**NVLoader**) for efficient data loading. Given a video dataset, we first create NVDataset by overfitting a small model for each video and storing them as checkpoints. With quantization and entropy encoding, NVDataset is of similar size to the original dataset and does not increase the storage space.

Most existing video dataloders have a complex decoding pipeline, since they are based on traditional video codecs (*e.g.* , H.264), and its speedup or deployment needs specific design and optimization on a new platform. Data loading for these dataloaders is computationally intensive and exacerbated by the sequential decoding of video frames, where the decoding of inter-frames is highly dependent on the decoding result of keyframes. This high dependency also holds for the decoding of frame patches. Therefore, these video dataloaders are unfriendly to parallelism

for further speedup. In contrast, video loading in NVLoader is quite simple and can be easily deployed to most devices that are supported by standard deep learning libraries. When loading video data, NVLoader loads the corresponding checkpoint and outputs the frame by inputting a frame embedding, via a simple forward pass. Due to its simplicity, NVLoader can be deployed on many devices (GPU, TPU, NPU, *etc*.) easily and efficiently, thus accelerating video loading due to parallel processing. Compared to existing video dataloaders, which are run on CPUs, NVLoader shows a clear loading advantage especially for deep learning research, since it can be run directly on powerful chips which are already the research platform for most.

To better support patch-based methods, like transformers, the video decoder of NVLoader is designed to decode frame patches instead of the whole frame, as illustrated in fig. 5.2. Such flexible design can further speed up the data loading when only a few patches are used, by up to $7\times$ compared to existing dataloaders. Finally, we also verify the effectiveness of NVLoader on video recognition task with various setups, such as different models, different video datasets, and various frame/patch sampling strategies. We show comparable accuracy to existing video datasets (*e.g*. , H.264, HEVC, and JPEG dataset) with much faster loading speeds and similar dataset size (fig. 6.1) demonstrating that NVLoader can serve as a good alternative.

In summary, we propose NVLoader, a neural video dataloader, for efficient data loading. It can be deployed easily on any AI chip and can serve as a good alternative for existing CPU-intensive video dataloders. Finally, without deteriorating accuracy on video recognition or increasing dataset size, NVLoader shows a significant loading speedup from $2.6\times$ to $7\times$.

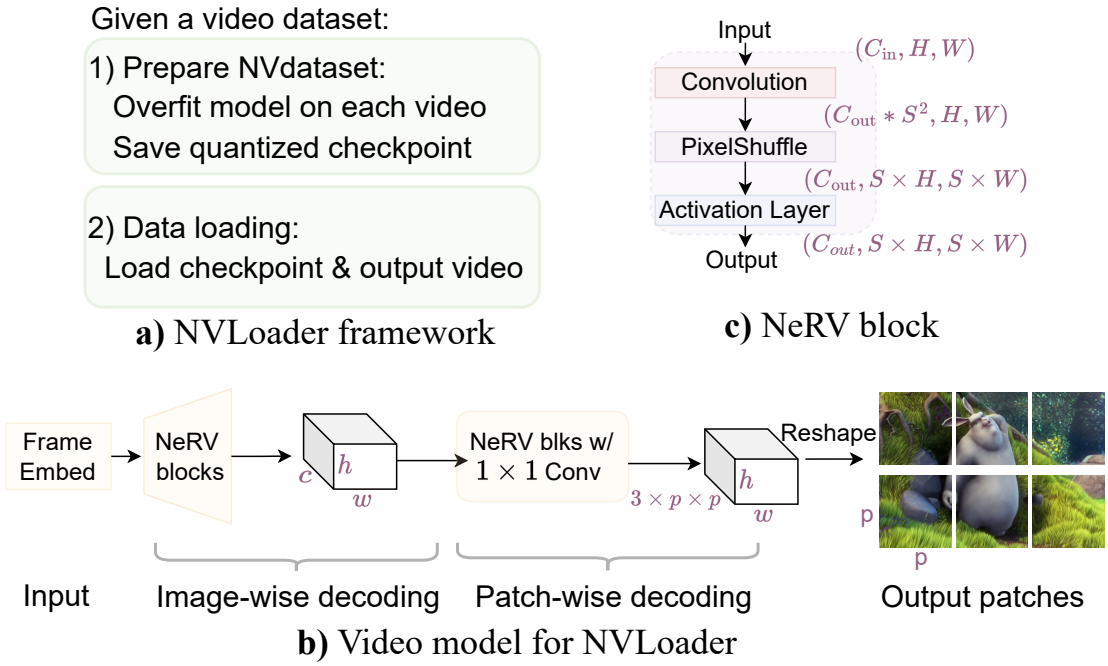


Figure 5.2: **a)** NVLoader **framework**; **b) Video model** for NVLoader, NeRV block upscale the feature map to $h \times w$, and MLP expand channels from c to $3 \times p \times p$. d is frame embedding dimension, $h \times w$ is patch number, $p \times p$ is patch size. **c)** NeRV **blocks** used in video model. S is the upscale factor.

5.2 Related Work

Implicit neural representation. Implicit representations fit a function like a deep neural network to each signal [53] like an image, a scene, or a video. Most implicit neural representations are coordinate-based which input the pixel coordinate to the neural network and output corresponding feature values at that place. These coordinate-based implicit representations are used in various tasks, like image reconstruction [54, 55], shape regression [60, 61], and 3D view synthesis [62, 63]. NeRV [1] instead proposes an image-wise implicit representation, which takes frame index as input and leverages convolution neural network, instead of the MLP used in coordinate-based methods, for accurate video reconstruction. Such image-wise neural representation speeds up the encoding and decoding process largely compared to coordinate-based (pixel-wise) methods.

Based on NeRV, E-NeRV [136] further improves the video reconstruction quality via decoupling frame index and spatial index. Similar to CNeRV [159] and HNeRV [160], we use a hybrid neural representation for videos and use a encoder from standard U-shaped autoencoders [133, 137, 139] to generate content-adaptive embedding. However, unlike all these image-wise neural representations, we leverage a more flexible decoder to output patches and enable flexible video decoding.

Efficient video codecs. Most existing video dataloaders are based on traditional video compression codecs such as MPEG [7], H.264 [8], and HEVC [9] since they can largely reduce the dataset size compared to the original lossless video or its corresponding image dataset. Although some more advanced codecs like AV1 [161] or VVC [162] provide a better compression performance it comes at the cost of longer decoding time which further slows down the speed. The complex decoding process of these hand-crafted codecs also makes the deployment difficult for many devices or achieves efficient and fast data loading.

Recently, deep learning techniques have been proposed for video compression as well. While these approaches focus on improving traditional handcrafted approaches, like via image compression and interpolation [10, 11], or via autoencoders [12], or modeling conditional entropy between frames [13], or reformulate traditional video compression pipelines with deep learning [14, 15, 16]. Recent approaches have focused on improving efficiencies of existing art, by fine-tuning traditional codecs [17], or optimizing existing compression pipeline [18]. Although these learning-based methods improve the bits-distortion performance and further reduce the video size compared to traditional codecs, they also largely increase the decoding latency and make them not a good candidate for efficient video loading.

Unlike traditional codecs or learning-based compression methods which have long decoding latency, video loading for video neural representation is simple, fast, and efficient. Ad a recent neural representation, NeRV [1] represent video as a deep convolution network and converts video compression to a model compression problem. Through model pruning, weight quantization, and entropy encoding, NeRV achieves similar compression performance with common video codecs like H.264 or HEVC. Inspired by NeRV, video model size in NVLoader can be largely reduced by quantization and entropy encoding and achieve a similar size to the common H.264 video dataset. We further improve the architecture design and make it a hybrid representation to improve the reconstruction quality, and enable flexible patch decoding.

5.3 Method

We propose NVLoader as an efficient video loading framework. As illustrated in fig. 5.2a, given a video dataset, we first generate a NVDataset (section 5.3.1) by overfitttting a video model on each video and save the checkpoint. For video loading (section 5.3.2), we load the checkpoint and output frame via a simple forward pass.

5.3.1 Prepare NVDataset

Video model. Video model is used to reconstruct the video for NVLoader. We use an auto-encoder as the video model, which takes video frame as input, and extracts a tiny frame embedding via a encoder, following by a decoder for frame reconstruction. We choose ConvNeXt [148] as the encoder. For video loading, we only need frame embedding and the decoder for its reconstruction, which are stored as the video checkpoint, as shown in fig. 5.2b. Unlike NeRV

using content-agnostic embedding as input, we leverage an encoder to generate tiny content-adaptive embedding, similar to HNeRV [160]. To achieve flexible video decoding and accessing patches easily, we design the decoder with two parts: image-wise decoding and patch-wise decoding.

For *image-wise decoding*, we use NeRV-style blocks to build the decoder for efficient reconstruction. As illustrated in fig. 5.2c, there are three layers in this block: a convolution layer with learnable parameters, a pixel shuffle layer to upscale the feature map, and an activation layer at last. Unlike NeRV to reduce the channel whideth by 2, $C_{out} = C_{in}/2$, we remain the channel width the same as the input channel, $C_{out} = C_{in}$. Also, we use a larger kernel size (5 here) for convolution layers except the first one while NeRV use kernel size 3 for all convolution layers.

For *patch-wise decoding*, we first reshape the input feature map from (c, h, w) to $(h \times w, c)$ if we have $h \times w$ patches to decode. Then a two-layer MLP is used for patch decoding, expanding the channel width from c to $3 \times p \times p$ where p is the patch size. By decoupling patch decoding in this way, the loading of video patches can be run *independently*, with enables random and fast patch access. Since we stack patches in the batch dimension, they can easily be selected by giving a patch index list and therefore we can skip the computation of un-wanted patches. As will be shown in fig. 5.3f, this flexible design can further improve the loading speed when only a part of the video patches are loaded.

Video overfitting. Given the video model with learnable encoder W_{encoder}) and decoder W_{decoder}, we firstly overfit it on the input video x with a reconstruction loss function like L2 loss. Note that the encoder W_{encoder} is only used to generate tiny frame embedding D (of shape $16 \times 1 \times 1$, *etc.*), and will be deserted in the following video storage and loading steps.

Efficient video storage. Given a list of video checkpoints, including frame embeddings D and decoder weights W_{decoder}, we use quantization and entropy decoding to further reduce model size.

We use simple uniform *quantization* with linear mapping for both frame embeddings D and decoder weights W_{decoder}. For quantization of a vector μ, we linearly map every element to the closest integer,

$$\mu_i' = \text{Round}\left(\frac{\mu_i - \mu_{\text{min}}}{\text{scale}}\right) \times \text{scale} + \mu_{\text{min}},$$
$$\text{where} \quad \text{scale} = \frac{\mu_{\text{max}} - \mu_{\text{min}}}{2^b - 1} \tag{5.1}$$

μ_i is one vector element, μ_i' is the quantized element, 'Round' is a function that rounds the value to the closest integer, b is the quantization bit length (8 in this paper), μ_{max} and μ_{min} are the max and min value of vector μ, and 'scale' is the scaling factor.

We use Huffman coding in this paper for *entropy encoding* of quantized frame embeddings $\hat{D}$ and decoder weights $\hat{W}_{\text{decoder}}$, to further reduce the video size. Concatenating $\hat{D}$ and $\hat{W}_{\text{decoder}}$ into a long flattened vector, we use Huffman coding to assign variable bits for different symbols (2^b symbols in our case) given its frequency. More details can refer to [122]. Through entropy encoding, we can achieve lossless compression at this step.

5.3.2 Data loading

We summarize the whole process in algorithm 2, where the first step is to prepare for the NVDataset as a list of checkpoints which is already explained in section 5.3.1. For data loading in the second step, we first introduce the basic operations for frame decoding, followed by a patch decoding description.

Algorithm 1 NVLoader Pseudocode (PyTorch style)

```
####### Prepare NVDataset (Sec. 3.1) #######
# Input: a video dataset (VidDataset)
# Output NVDataset: a list of video checkpoints:
# {'embed': frame_embed, 'decoder': decoder_weight}

for video in VidDataset:
   video_model = Model(cfg) # initialize video model

   # Overfit video model on given data
   video_model.fit(video)

   # Save video decoder and frame embeddings
   decoder_w = video_model.decoder.state_dict()
   frame_embed = video_model.encoder.forward(video)

   # Quantization and Huffman coding
   frame_embed = frame_embed.quant()
   decoder_w = decoder_w.quant()
   HuffmanCode([frame_embed, decoder_w])

   # Save video checkpoint
   vid = {'embed': frame_embed, 'decoder': decoder_w}
   vid.save_as(vid_ckt)


###### Data loading (Sec. 3.2) ######
# Input: NVDataset, frame_idx, patch_idx
# Output: Video frames/patches

decoder = Decoder(cfg) # initialize video decoder
for vid_ckt in NVDataset:
   # dequant checkpoint and load decoder weights
   decoder_w = vid_ckt['decoder'].dequantize()
   frame_embed = vid_ckt['embed'].dequantize()
   decoder.load(decoder_w)

   # data loading: frame selection first
   input_embed = frame_embed[frame_idx]
   if patch_idx != None:
      # output video patches with idx list
      out = decoder.forward(input_embed, patch_idx)
   else:
      # output video frames
      out = decoder.forward(input_embed)
```

`HuffmanCode`: Huffman coding;
`quantize/dequantize`: tensor quantization/dequantization.

Video loading As shown in algorithm 2, after a decoder model is initialized by the config file, we iterate over the checkpoint list for video loading. Given a quantized checkpoint, we dequantize frame embedding and decoder weights first,

$$\hat{\mu}_i = \mu'_i \times \text{scale} + \mu_{\min}, \tag{5.2}$$

where $\hat{\mu}_i$ is the dequantized vector element. Then the video decoder load weights from the dequantized checkpoint. As a normal case for video sampling, a frame index list is an input for data loading. With the input frame index, we choose the corresponding embedding as input. Then the video frames can be reconstructed by a decoder forward pass.

Patch loading In cases where only part of frame patches are needed, we also input a patch index list (w/ length N) as input. At the end of image-wise decoding, we select the patch index in the batch dimension, where the feature map is trimmed from shape $(h \times w, c)$ to (N, c). Then N patches are decoded by the MLP decoder. Since we skip the computation of un-selected patches, the decoding speed can be much faster compared to the decoding of the whole frame. Patch-wise decoding makes our NVLoader a flexible framework for many other applications as well, like adaptive patch quality where different patches can have different reconstruction qualities.

We summarize the framework of NVLoader in fig. 5.2a and provide pseudocode in algorithm 2. Due to the simplicity and efficiency of neural representation, NVLoader can be deployed to any AI chip and shows a clear speed advantage over other video-loading methods.

5.4 Experiment

5.4.1 Datasets and implementation details

We choose three common video datasets, Kinetics-400 (referred to as K400 and our default dataset)[163], Something-Something V2[164] (referred as SthV2), and UCF-101 dataset [165]. Due to limited resources and long overfitting time for a large-scale dataset, we create a subset from these datasets for our experiments to show the utility of NVLoader. For all three datasets,

we choose 20 classes with the most training videos. For k400 and SthV2, randomly pick 100 videos from the original training set for each class to construct the training set, and 20 videos from the original test set for each class as the test set. For UCF101, we use all videos from the original train/test set for these 20 classes.

For video dataloader baselines, we choose three common formats for video datasets: H.264 videos, HEVC videos, and JPEG frames. To generate these video datasets, we extract video from the original dataset with the `ffmpeg` tool. We choose H.264 video dataset as the default input for video dataloaders since it shows good performance in regards to loading speed, video size, and action recognition accuracy (shown in fig. 6.1. For a fair comparison, we resize the original video with a shorter side as S, and center crop the $S \times S$ clip. S is set to 240 unless otherwise stated. For video sampling, we choose 16×4 as the default setup, *i.e.*, , 16 frames with strides 4. As is common practice, we randomly select a starting frame as long as the following 16×4 frames exist. For patch selection, we divide one frame into 8×8 patches without overlap and do random patch selection with the given patch index.

Our implementation is based on Pytorch [126] and we choose three common video decoding libraries as the backends: Decord (default backend), PyAV, and Torchvision. Traditional video dataloaders are evaluated on CPU, where we use a 8-CPU[1] node. We set the worker number in these dataloaders as 8 unless stated otherwise. To test the data loading speed, we warm up the device with 1 epoch, and then average their loading speed over the following 3 epochs (which is the same for testing of NVLoader). We use the training dataset ($\sim$2k videos) for speed evaluation. For a fair comparison, we don't use speedup tricks which are universal for all dataloaders (including NVLoader), like caching videos, and repeat sampling at each decoding

[1]Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz

step. Since all videos are of the same size, we do not need resize or crop operation in the speed evaluation of dataloaders.

For training and testing of NVLoader, we use one RTX2080Ti GPU unless otherwise stated. Given a 240×240 video with 300 frames, it takes 2 seconds per epoch for training. We train the video for 1200 epochs on K400 and UCF101, 1800 epochs on SthV2. We use Adam optimizer with a weight decay of 0, batch size of 2, a learning rate of 0.001, and a cosine learning strategy. For model size, we keep the final checkpoint comparable with H.264 video.

To evaluate loading speed, we use video per second (VPS) as the metric. Table cells in gray are the default setups. We provide details about video model architecture and training strategies, the video subset list, `ffmpeg` command in supplementary for reproduction, as well as more quantitative tables, and qualitative results.

5.4.2 Main results

We mainly focus on the loading speed for video dataloaders in this section.

Existing video backends. We first evaluate existing video decoding backends (PyAV, Decord, and Torchvision) with two loading setups: naïve decoding and dataloader loading (default setup). For naïve decoding, we iterate the training videos one by one and decode them respectively; while for dataloader loading, we use 8 workers to load videos in parallel[2]. Results are shown in fig. 5.3a, where Decord shows the best loading performance in both cases and is chosen as the default video backend.

Different devices. Decord also supports video decoding with GPU, we compare it with NVLoader on both CPU and GPU. Since the GPU decoding of Decord does not support parallel processing,

[2]Batchsize (set as 8) in dataloaders doesn't affect loading speed much.

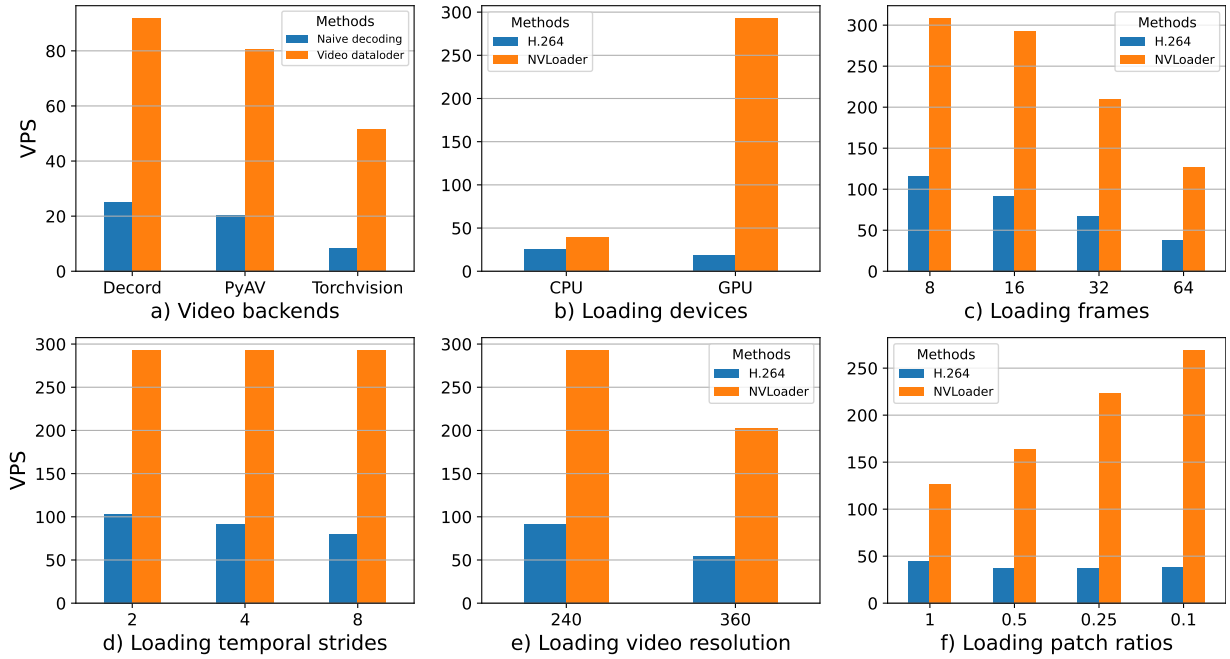


Figure 5.3: Video loading speed. **a)** common **video backends**. Naive decoding load videos one by one, video dataloader use 8 workers for parallel speedup; **b)** naive decoding on different **devices**; **c)** different **frames** for dataloader loading; **d)** different **strides** for dataloader loading; **e)** **video resolutions** for dataloader loading; **f) patch ratios** for dataloader loading (for 64 frames).

Worker number	4	8	12	16
H.264 dataloader	46.3	91.8	102.5	**117.9**

Table 5.1: Video loading speed (VPS) for video dataloader based on **H.264 videos**, with different **worker numbers**.

Frames	8	16	32	64
GTX 1080Ti	211.3	199.2	127.5	81.6
RTX 2080Ti	289.6	292.8	209.1	126.4
RTX A6000	**400.3**	**386.7**	**293.2**	**187.6**

Table 5.2: Video loading speed (VPS) for **NVLoader**, with different **GPU devices**.

we only evaluate Decord and NVLoader with naïve decoding and show results in fig. 5.3b (Decord is referred to as H.264 since it uses H.264 videos). NVLoader is much faster than Decord in both cases. Note that due to poor deployment on GPU, its decoding speed is even slower than on CPUs where Decord only has a GPU utilization around 2% while NVLoader reaches around

Frames	8*4	16*4	32*4
H.264	8.5	5.75	**3.75**
NVLoader	**7.00**	**5.50**	5.00

Table 5.3: Top-1 error (%) with different **frames**.

Frames	16*2	16*4	16*8
H.264	7.75	5.75	**4.75**
NVLoader	**6.50**	**5.50**	5.00

Table 5.4: Top-1 error (%) with different temporal **strides**.

Patches	0.1	0.25	0.5	1
H.264	5.75	5.5	6.25	5.75
NVLoader	**5.75**	**4.75**	**5.75**	**5.50**

Table 5.5: Top-1 error (%) with different **patch ratios**.

Models	MviTv2-S	MviTv2-B	MviTv2-L
H.264	5.75	**4.25**	**2.5**
NVLoader	**5.50**	4.75	3.25

Table 5.6: Top-1 error (%) with different **video models**.

100%. We believe this also holds for other most chips where the complex decoding pipelines greatly hinder its efficient deployment. Due to decoding simplicity, NVLoader can be faster than Decord even on CPUs for naïve video decoding, although Decord has better parallel support on CPUs and shows better dataloader speed.

Besides naïve decoding, we also evaluate dataloader loading speed with different workers[3] in table 5.1. Since 8 workers per GPU is the most common practice and shows good loading speed compared to dataloader with more workers, we choose it as the default setup. For NVLoader, we evaluate its loading speed with different GPUs and frame numbers in table 5.2, with strides at 4. As expected, better GPUs lead to faster loading speeds, for all cases.

Frame number. We evaluate NVLoader and H.264 video dataloader with different frames (with stride 4) in fig. 5.3c, where NVLoader shows a consistant loading advantage.

Temporal stride. We evaluate NVLoader and H.264 dataloaer with different temporal strides in fig. 5.3d. Since frame decoding is independent for NVLoader, temporal strides doesn't affect its loading speed. In contrast, the loading speed drops when increasing the temporal stride

[3]We increase node CPU number for workers 12 and 16.

since H.264 needs to decode a longer clip to load the required frames and thus introduces extra computation.

Video resolution. We evaluate NVLoader and H.264 dataloader when increasing resolutions from 240×240 to 360×360 in fig. 5.3e, where NVLoader retains the loading speed advantage and the speedup is slightly larger, from $3.2\times$ to $3.8\times$.

Patch loading. We evaluate NVLoader and H.264 dataloader when only part of video patches are loaded (using frames sampling at 64×4), which is shown in fig. 5.3f. We provide more patch loading results in the supplementary material, where we observe that when loading fewer frames (32, *etc*.), the patch loading speedup is slightly smaller (VPS from 211 to 286). Compared to loading the whole frame, NVLoader shows a clear speedup when only a subset of video patches are loaded since it can skip the computation of skipped patches. In contrast, since H.264 video dataloader uses patch loading as a post-processing step after frame decoding, it can even be slightly slower than the frame loading baseline.

5.4.3 Action recognition.

Besides loading speed, it is also important to ensure that the quality of decoded video is good for the downstream tasks. Therefore, we also evaluate the performance of video dataloaders on the downstream task of action recognition. We use top-1 error for action recognition in this section, and choose a recent transformer method MViTv2 [154] which is finetuned on K400 subset (20 classes, 120 videos per class) Similar to the last section, we also evaluate various setups, like different frames in table 5.3, different temporal strides in table 5.4, different patch ratios in table 5.5, and different models in table 5.6. NVLoader shows comparable accuracy with

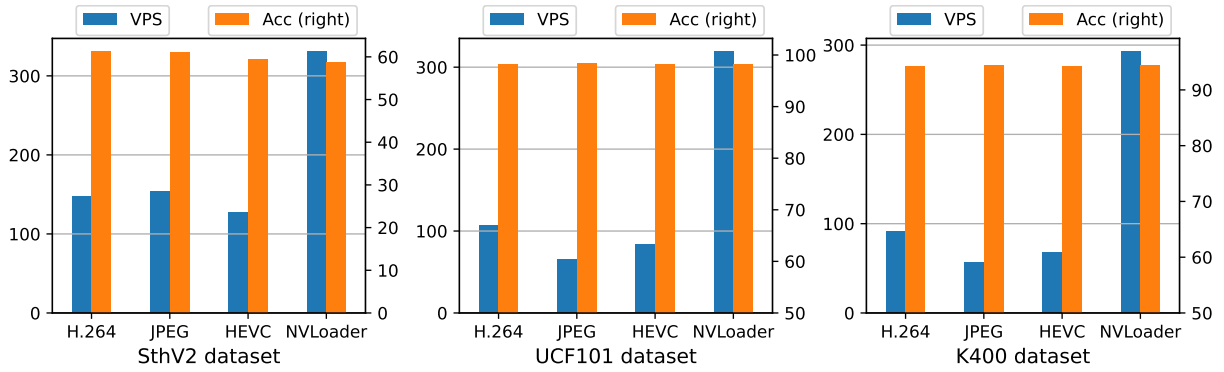


Figure 5.4: Comparison of video dataloaders based on H.264 videos, HEVC videos, HEVC videos, and NVloaders (ours) . **Left: Something V2** dataset. **Middle: UCF101** dataset. **Right: Kinetics-400** dataset. Left y-axis is loading speed, videos per second. Right y-axis is top-1 accuracy.

	K400 (240×240)			K400 (360×360)		
	Acc.	Size	VPS	Acc.	Size	VPS
H.264	5.75	**297.4k**	91.8	5.50	0.3506	53.8
NVLoader	**5.50**	297.5k	**292.8**	**5.25**	**0.3342**	**202.7**

Table 5.7: Comprehensive results on datasets of different **resolutions**, top-1 accuracy, average video size, and loading speed.

	K400 (240×240)	K400 (360×360)	SthV2	UCF101
$PSNR_{orig}$	**31.53**	**31.23**	**35.63**	**33.41**
$PSNR_{quant}$	30.86	30.92	35.18	32.75

Table 5.8: **Video quality** for NVLoaders, before ($PSNR_{orig}$) and after ($PSNR_{quant}$) quantization.

H.264 dataset in all cases, demonstrating that it can serve as a good alternative for efficient data loading.

5.4.4 Comprehensive comparison

We also compare with other video dataloaders based on JPEG folders or HEVC videos. Their performance on K400 for data loading and action recognition are shown in fig. 5.4c, where NVLoader outperforms them largely for loading speed while achieving comparable accuracy. We also compare the dataset size of these datasets and show it in fig. 6.1 where the JPEG dataset takes much more storage while at a slower loading speed. Although HEVC can save some space, it also slows down the loading speed due to a more complex decoding compared to H.264.

Besides K400, we evaluate NVLoader on SthV2 and UCF101 as well and show results in

	MViTv2-S	X3D-S		
Batch size	1	1	4	16
H.264	1.28×	1.8×	2.8×	4.8×
NVLoader	**1.08×**	**1.2×**	**1.5×**	**2.1×**

Table 5.9: **Total forward time** (model forward + data loading) at testing time. Pure computation time acts as the $1\times$ baseline. Data loading becomes a bottleneck, especially for efficient video model and large batches.

fig. 5.4a and fig. 5.4b respectively. The observation on K400 still holds for these two datasets, where NVLoader gets similar accuracy while showing much better loading speed compared to other video dataloaders, no matter whether it is based on H.264 videos, HEVC videos, or JPEG folders.

We also show a full comparison of K400 on different resolutions in table 5.7. NVLoader outperforms H.264 in both cases for data loading speed, where the speedup increase from $3.2\times$ (240×240) to $3.8\times(320\times 320)$, while remaining similar video size and video recognition accuracy,

5.4.5 Other results

Total forward time. We evaluate the total forward time at test time, including both the model computation and data loading time, and show results in table 5.9, where just the model forward time (without video loading) is $1\times$ baseline Besides model MViTv2-S, we also evaluate on another video model X3D-S [152] which is designed for efficient video recognition. Since model X3D-S is much faster than MViTv2-S [154], data loading becomes a serious bottleneck for its deployment, especially when processing videos with a large batch. It clearly shows the importance of efficient data loading for video research.

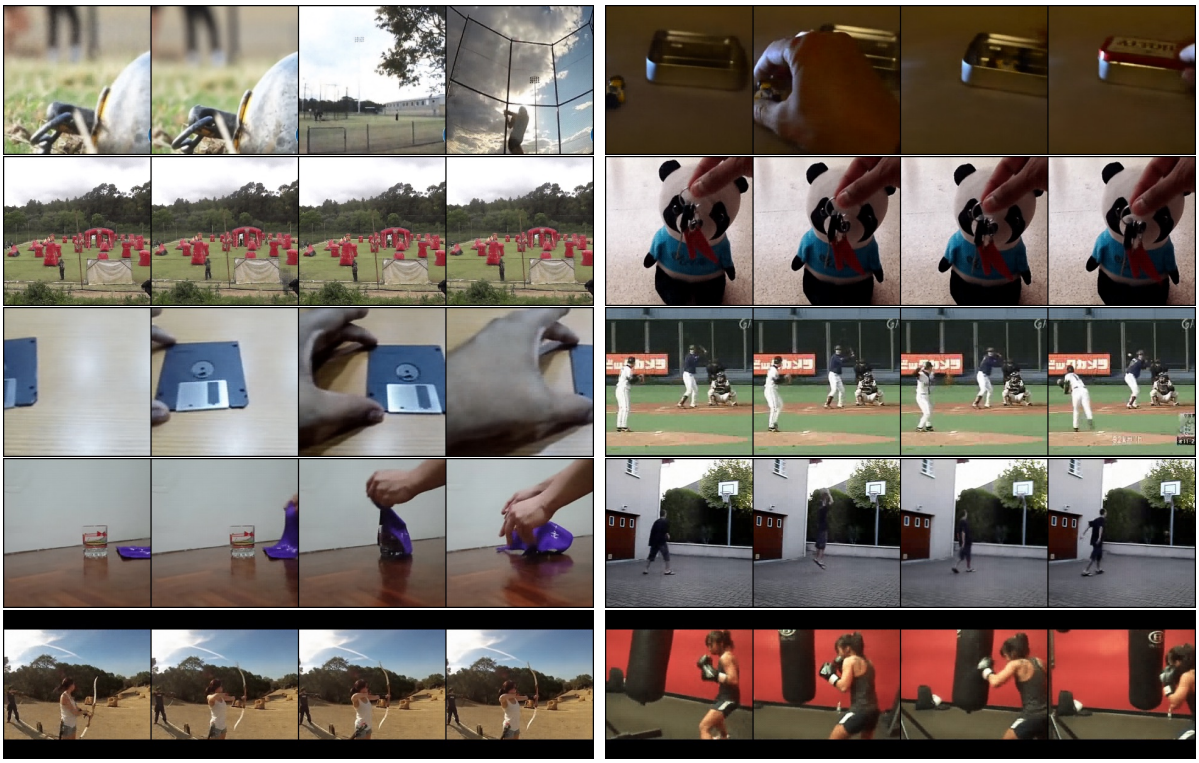


Figure 5.5: **Output video frames** for NVLoader across datasets. NVLoader can reconstruct videos well and capture details faithfully , for either ones with dynamic scenes, or with rich textures.

Video quality. We present qualitative results for NVLoader (PSNR lies between 25% to 75% over the whole dataset) in fig. 5.5 with dataset K400, SthV2, and UCF101. It can be seen that for most videos, NVLoader can reconstruct it well with high fidelity and therefore can serve as a good loading framework for downstream tasks. Finally, we show quantitative results for video quality and video size with quantization and Huffman coding. In table 5.8, we show how quantization affects video quality withf PSNR (dB).

Video sizes. In table 5.10, we show the average storage for video checkpoint files[4] by quantization and Huffman coding. To save space and evaluate NVLoader in a real scenario, we take quantization library functions from Pytorch [126] to perform quantization and dequantization, of which its

[4]Via python function os.path.getsize()

	Q	H	K400 (240 × 240)	K400 (360 × 360)	SthV2	UCF101
Parameters			301.7k	339.7k	154.8k	246.9k
Video size	✓		315.3k	353.9k	168.7k	260.4k
Video size	✓	✓	**297.5k**	**334.2k**	**158.6k**	**243.5k**

Table 5.10: **Average video size** in NVLoader. Parameters is the total parameter of video checkpoint (video decoder $W_{decoder}$ and frame embedding D), video size measures video checkpoint by MegaBytes. Q means quantization and H means Huffman coding.

Train set → Test set	16 × 2	16 × 4	16 × 8
H.264 → H.264	7.75	5.75	4.75
NVLoader → NVLoader	6.50	5.50	5.00
NVLoader → H.264	7.25	5.75	5.00

Table 5.11: **Generalization to other dataloaders**, top-1 err (%) is showed. We evaluate models with different sampling frames, which are trained and evaluated on the same dataloader or different video dataloaders.

implementation can be further improved for better reconstruction quality.

Generalization to other dataloaders. Given the long overfitting time, we also evaluate its generalization to other video loaders when a model trained with NVLoader is deployed to videos where its video checkpoint is not available. Results are shown in table 5.11. NVLoader shows good generalization to H.264 videos as well, which should be expected since their output are still RGB frames, although comes from different efficient video representations.

Limitations and future work. The largest limitations of NVLoader is the same as all implicit neural representations, the long training time for a new video. So we expect NVLoader to be used during model training first, since we can prepare an NVDataset for the whole training dataset once and all researchers can use it directly and benefit from efficient data loading. Secondly, as we discussed in section 5.3.2, the efficient storage of video checkpoints can still be improved to save more space. Finally, we need to transfer data transformation tools (augmentation, *etc.*) to GPU so that the whole dataloader process can be done on GPUs.

5.4.6 Video model architectures

We provide video model architecture details in table 5.12 used in different datasets.

	$\text{Strds}_{\text{enc}}$	$\text{Strds}_{\text{dec}}$	Size_{enc}	Size_{dec}	d
K400_240	5,4,3,2,2	5,3,2,2,4	0.28M	0.3M	16
UCF101	5,4,3,2,2	5,3,2,2,4	0.28M	0.24M	16
SthV2	5,4,3,2,2	5,3,2,2,4	0.28M	0.15M	16
K400_360	5,4,3,3,2	5,3,3,2,4	0.3M	0.34M	16

Table 5.12: **Video model architectures**. $\text{Strds}_{\text{enc}}$ and $\text{Strds}_{\text{dec}}$ are stride list used in the encoder and decoder. Size_{enc} and Size_{dec} parameter nubmers for the encoder and decoder. d is the embedding dimension.

5.5 Conclusion

In this paper, we propose a neural video dataloader for efficient data loading. Our NVLoader can be deployed easily on any AI chip and can serve as a good alternative for existing CPU-intensive video dataloders. Without deteriorating accuracy on video recognition or increasing dataset size, NVLoader shows a significant loading speedup from $2.6\times$ to $7\times$ in a wide range of setups.

Chapter 6: HyperNeRV: Towards Fast Learning of Video Neural Representation

6.1 Introduction

Recent works have shown the potential of representing high-dimensional video data implicitly as a deep neural network [1, 51, 136, 157]. With such representations, given a frame index or pixel coordinate, the neural network can reconstruct the video content with high fidelity. These representations are preferred by video compression [1, 136, 157], continuous spatial-temporal super-resolution [158], and video restoration [1, 157] due to their simplicity and compactness.

However, training a neural network to overfit to a given video using gradient-based methods can be time-consuming and hinder practical applications. To address this issue, some approaches have been proposed for pixel-wise neural representations which take pixel coordinates as input and output pixel values using an MLP. These methods generate a content-adaptive vector to modulate the model parameters [61, 64, 166, 167] or learn a good initialization with meta-learning to enable fast convergence on unseen signals [168]. Although these methods can improve encoding speed, they still require test-time optimization to achieve satisfactory performance. In contrast, our proposed HyperNeRV aims to directly generate model weights via a hypernetwork.

This paper seeks to address two fundamental questions. Firstly, is it possible to train such a hypernetwork? Can we fit the hypernetwork on a given video dataset and obtain model weights that can accurately reconstruct the videos? Secondly, can this well-trained hypernetwork

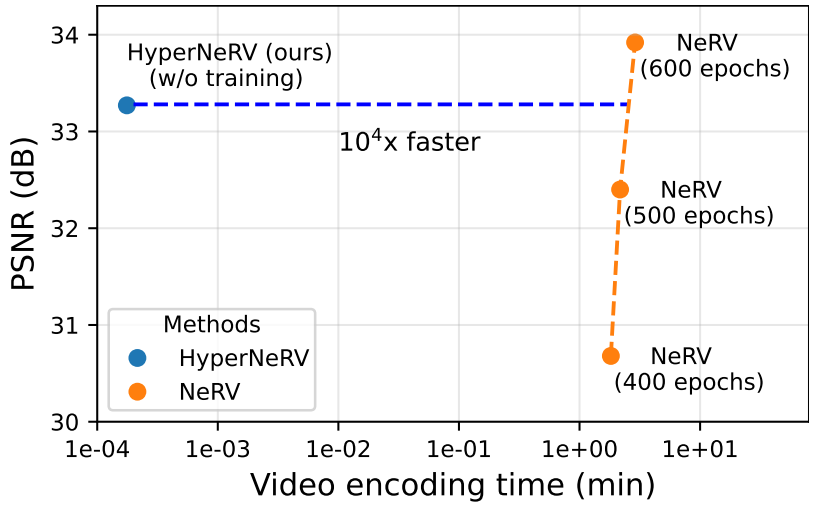


Figure 6.1: **Encoding time comparison** between HyperNeRV and training NeRV [1] from scratch. Encoding refers to generating a well-trained neural network for a given video. HyperNeRV eliminates the need for tedious fitting, enabling much faster video encoding.

generalize well to unseen videos, beyond the training dataset?

Given the exceptional performance of vision transformers in various visual tasks [5, 83, 169, 170, 171], we employ transformers as the hypernetwork to generate model weights, as illustrated in fig. 6.2. With video patches and initial weight tokens as input, HyperNeRV can produce model weights for the convolution neural network used for video representation. In this paper, we demonstrate that HyperNeRV can answer both of the questions raised above with a resounding yes. We show that it is possible to fit HyperNeRV on training videos and that it can generalize effectively to unseen videos. In comparison to training from scratch, the simple forward pass significantly reduces the encoding time, with a speed-up of $\sim 10^4\times$, as shown in fig. 6.1. Trans-INR [2], a recent work that employs hypernetworks to generate weights for implicit representations, is designed for images (rather than videos) and based on pixel-wise representations. Our proposed HyperNeRV outperforms Trans-INR in terms of reconstruction quality, generalization, and speed, as demonstrated in section 6.4.2 and 6.4.4.

In this work, we not only introduce a transformer hypernetwork for fast learning of video

neural representations, but also explore various ways to generate more efficient representations. To expand the video size, we increase the number of frames and spatial resolution, which results in more compact representations. Furthermore, we analyze the impact of architecture design, training data size and diversity, and data augmentation on HyperNeRV's performance. Our experiments demonstrate that a larger and more diverse training dataset, as well as data augmentation, can further improve the performance for unseen videos. These findings suggest that HyperNeRV has the potential for fast learning of efficient neural representations. The results presented in fig. 6.1 are obtained using 10k training videos and 400 training epochs, but better performance can be achieved by scaling up the training with more videos and epochs.

In summary, this paper introduces HyperNeRV, a transformer hypernetwork that generates model weights for efficient neural representations. We also investigate several design principles to improve video representation efficiency, including expanding video sizes, optimizing architecture design, increasing training data size and diversity, and using data augmentation. Our results demonstrate that HyperNeRV has significant potential for fast learning of video neural representations, achieving a speed-up of $\sim 10^4\times$ compared to training neural networks from scratch.

6.2 Related Work

Implicit neural representations. Recent developments in deep learning have led to the emergence of implicit neural representations, which are compact data representations [1, 51, 52, 53] that fit a deep neural network to signals such as images, 3D shapes, and videos. One of the main branches of implicit representations is coordinate-based neural representations, which take pixel coordinates as input and output corresponding values such as density or RGB values using an

MLP network. These representations have shown promising results in a range of areas including image reconstruction [54, 55], image compression [52], continuous spatial super-resolution [56, 57, 58, 59], shape regression [60, 61], and 3D view synthesis [62, 63]. To improve coordinate-based representations, several approaches have been proposed, such as using sine activation functions instead of ReLU [64] or converting input coordinates to a Fourier feature space [65].

In contrast to coordinate-based methods, NeRV [1] proposes an image-wise implicit representation, which takes the frame index as input and outputs the entire frame without iterative pixel-wise computation. NeRV employs convolutional neural networks and achieves better efficiency and regression quality compared to coordinate-based methods with a naive MLP network. By representing the given video as a neural network, NeRV converts the video compression problem to model compression and achieves comparable performance with common video codecs via model pruning, quantization, and entropy encoding. Building on the success of NeRV, E-NeRV [136] improves the architecture by incorporating both the frame index and spatial index as input; CNeRV [51] takes content-adaptive embedding as input to enhance the internal generalization. Compared to coordinate-based methods, image-wise neural representations offer higher capacity, faster encoding and decoding speed for videos. Given these advantages, our HyperNeRV is based on image-wise representations and exhibits superior performance compared to pixel-wise methods.

Hypernetworks. Hypernetworks [172] are widely used to generate model weights for another neural network based on input data or a dataset. The concept of content-adaptive weights is prevalent in the deep learning community, as seen in techniques like dynamic convolution [173] and conditional convolution [174]. Many methods have also been developed to modulate model weights in a latent space [61, 64, 166, 167] rather than generating all weights directly, as this approach can alleviate the learning difficulty. In these works, the neural network takes both

the pixel coordinate and a content-adaptive vector for modulation, where the modulate vectors function as the hypernetworks.

The most closely related work to ours is Trans-INR [2], which uses a transformer hypernetwork to facilitate fast learning of implicit neural representations. However, Trans-INR is designed for images and based on pixel-wise representations, while our HyperNeRV is tailored to videos and based on image-wise representations. In our experiments, we provide a detailed comparison between HyperNeRV and Trans-INR, highlighting the latter's difficulties in fitting video data.

Meta-learning. Meta-learning is a popular method for achieving rapid convergence for a neural network. Many prior works have employed meta-learning to address few-shot learning [175, 176, 177] or reinforcement learning [178, 179, 180], enabling a meta-learner to quickly adapt to new scenarios. Meta-learning has also been applied to learn implicit neural representations using gradient-based methods like MAML [178] and Reptile [181], as seen in MetaSDF [182] for learning signed distance functions and [168] for learning a good initialization for all signals. In contrast to these methods, our proposed HyperNeRV targets the direct generation of model weights through a simple forward pass, without any optimization required at test time.

Transformers. Transformers [81] were initially introduced for natural language processing, but they have since gained significant attention in vision research, particularly following the success of ViT [5] in image classification. Vision transformers have since been improved through advancements in training efficiency [83], multi-scale transformer structures [154], and advanced self-attention design [169]. In addition to image classification, vision transformers have also been leveraged in object detection [170] and self-supervised learning [171]. In this work, we utilize vision transformers as a meta-learner to directly generate model weights for video neural representations.

Variable	Definition
x	Input video
x_t	Video frame at time t
$\hat{x}_t$	Reconstructed video frame at time t
g_ϕ	Hypernetwork function (w/ parameter ϕ)
f_θ	Video model (w/ parameter θ)
θ_0	Hypernetwork input: initial weights
$\hat{\theta}'$	Hypernetwork output: video-specific weights
θ_1	Shared model weights
θ'	Final model weights
$D_{\text{train}}, D_{\text{test}}$	Training and test set
$C_{\text{out}}, C_{\text{in}}$	Convolution output and input channel width
K	Convolution kernel size for NeRV blocks
S	Upscale factor for NeRV blocks
M	Number of video patches
N	Number of weight tokens
N_{max}	Max number of weight tokens
d	Token dimension for hypernetwork input

Table 6.1: Variables and their definitions.

6.3 Method

6.3.1 Overview

To eliminate the tedious process of fitting a neural network for video representation, HyperNeRV aims to learn a hypernetwork function g_ϕ with parameter ϕ that maps a given video $x \in \mathbb{R}^{t\times 3\times h\times w}$ to model weights θ' of a deep neural network f_θ. The function is defined as follows:

$$\theta' = g_\phi(x). \tag{6.1}$$

With the generated model $f_{\theta=\theta'}$. we can reconstruct the video x for a given frame index t, as follows:

$$\hat{x}_t = f_{\theta=\theta'}(t). \tag{6.2}$$

Here, $\hat{x}_t$ represents the reconstructed frame for index t. Our goal is to minimize the reconstruction error between the ground truth frame x_t and its corresponding reconstructed frame $\hat{x}_t$. We define the loss function for this objective as follows:

$$\phi^* = \arg\min_\phi \sum_{x \in D_{\text{train}}} \sum_t \| f_{\theta=g_\phi(x)}(\mathbf{t}) - \mathbf{x}_t \|_2^2 \,. \tag{6.3}$$

, where D_{train} represents the training set and we use a L2 distance for illustration. After training the hypernetwork, we evaluate its generalization performance on the test set D_{test} using the following loss function:

$$L(\phi^*) = \sum_{x \in D_{\text{test}}} \sum_t \| f_{\theta=g_{\phi^*}(x)}(\mathbf{t}) - \mathbf{x}_t \|_2^2 \tag{6.4}$$

which measures the reconstruction error between the ground truth frames and reconstructed frames which are obtained using the generated model weights. Here, ϕ^* is the optimized hypernetwork obtained from the training set D_{train}, and $g_{\phi^*}(x)$ represents the generated model weights for input video x.

We aim to address two problems to find the needed mapping function g_ϕ. Firstly, can this function g_ϕ fit well on the training set, illustrated in eq. (6.3). Secondly, will the optimal mapping function g_{ϕ^*} generalize well on test set, illustrated in eq. (6.4).

6.3.2 HyperNeRV

We select a transformer neural network with L encoder layers to serve as the hypernetwork g_ϕ, which maps a given video x and initial weights θ_0 to video-specific model weights $\hat{\theta}'$. We

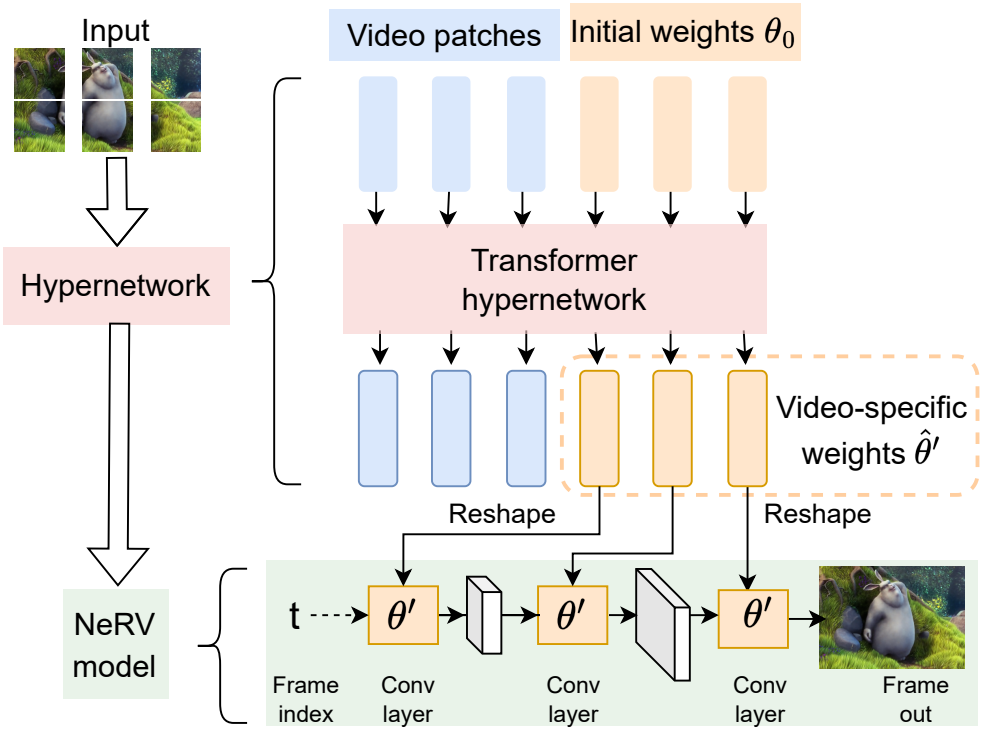


Figure 6.2: **Left:** HyperNeRV takes videos as input and outputs NeRV model weights through the hypernetwork. **Right:** Architecture details for the transformer hypernetwork (top) and the video model (bottom).

Algorithm 2 HyperNeRV Pseudocode in a PyTorch style.

```
############ 1) Obtain model weights ############
# Input: video x, initial weights θ_0, shared weights θ_1
# Transformer hypernetwork g_ϕ
# Output: final model weights θ'

# tokenize video into patches
x = FC_1(x.tokennize())                                          # d \times M

# Concat video patches and initial weights as input
x = x.concat(θ_0)                                              # d \times (M+N)

# Hypernetwork output video-specific weights θ̂'
θ̂' = g_ϕ.forward(x)[-N:].                                        # d \times N
θ̂' = FC_2(θ̂')                                                 # C_out \times N

# Element-wise product to generate final weights:
# C_out \times C_in \times K \times K
θ̂' = θ̂'.expand_as(θ_1)                                  # broadcast into needed shape
θ' = θ_1 * θ̂'                                  # C_out \times C_in \times K \times K

############## 2) Video reconstruction #############
# Initial video model with final weights
f_θ.reset_parameter(θ')

# Input: frame index t
# output: frame prediction, x̂_t
x̂_t = f_θ.forward(t)

-------- # Below is for model training only # --------
########## 3) HyperNeRV optimization (optional) ##########
# Compute loss and backward gradients
loss = MSELoss(x̂_t, x_t)
loss.backward()

# update all parameters
update([ϕ, θ_0, θ_1])
```

`FC`: fully connected layer;
`tokenize`: tokenize video into patches;
`expand_as`: broadcast into the same shape;
`MSELoss`: mean square error loss.

provide pseudocode for the process in Algorithm 2. The overall process can be divided into three steps: *a)* generate model weights θ', *b)* reconstruct the video with θ', and *c)* optimize HyperNeRV.

a) Obtain model weights. We use NeRV blocks to construct the video model. as depicted in fig. 6.3a. A NeRV block comprises three layers: a convolution layer with learnable parameters, a pixelshuffle layer to upscale the feature map, and an activation layer. We illustrate how to output model-specific weights $\hat{\theta}'$ in fig. 6.3b, while fig. 6.3c shows the process of obtaining the final

model weights θ' from $\hat{\theta}'$.

Step 1: Obtain video-specific weights. As shown in fig. 6.3b, we first divide input video x into patches and generate patch tokens via a FC layer. The patch tokens are then combined with initial weight tokens θ_0 to form the input tokens. After feeding the input tokens to the hypernetwork, we obtain the video-specific weights $\hat{\theta}'$.

Given a convolution layer with parameters $\theta_1 \in R^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$, we use $C_{\text{in}} \times K \times K$ as the token number for initial weights $\theta_0 \in R^{d \times N}$ where d is the token dimension and N is the the token number. Sometimes the token number can be too large, so we trim it to address the memory issues,

$$N = min(N_{\text{max}}, C_{\text{in}} \times K \times K) \tag{6.5}$$

where N_{max} is the max token number for each convolution layer. We concatenate the initial weights θ_0 and the video patches and input them to the transformer hypernetwork. We then select the output of the last N tokens, which correspond to the output of the initial weights. The transformer output is then converted to video-specific weights $\hat{\theta}' \in R^{C_{\text{out}} \times N}$ via an FC layer.

Step 2: Obtain final model weights. As shown in fig. 6.3c, after we get the video-specific weights $\hat{\theta}'$, we generate the final model weights by an element-wise multiply between $\hat{\theta}'$ and shared weights θ_1. $\hat{\theta}'$ is broadcast into the same shape as $\theta_1 \in R^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$ firstly if needed, since N can be set as N_{max} instead of $C_{\text{in}} \times K \times K$.

b) Video reconstruction. After we get weights θ' for all convolution layers, we can output the video frame when given a time index t. Similar to other implicit neural representations [62], t is firstly normalized into [-1, 1] and then goes through a positional encoding function and outputs a time embedding vector. The video model f_θ takes the time embedding as input and output

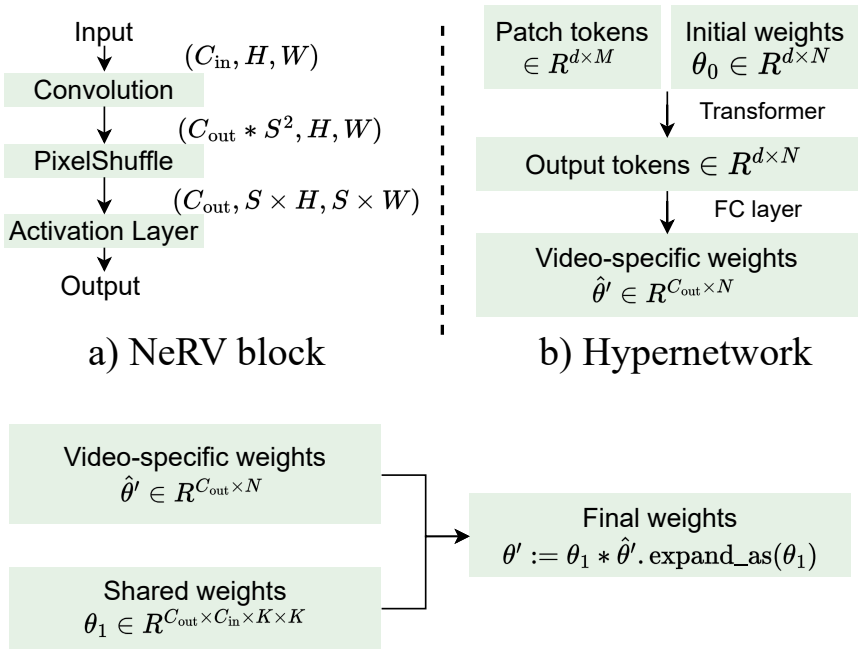


Figure 6.3: **a) NeRV block**, only the convolution layer has learnable parameters $\theta \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$. **b) Transformer hypernetwork**, takes video patches x and initial weights θ_0 as input tokens, and outputs video-specific weights $\hat{\theta}'$. **c)** Obtain **final model weights** θ' by an element-wise multiply of shared weights θ_1 and video-specific weights $\hat{\theta}'$.

video frame $\hat{x}_t$ The output of the final layer is sumed with an out bias 0.5 since the images are normalized between [0,1].

c) HyperNeRV optimization. HyperNerv has three parts of learnble parameters: hypernetwork parameter ϕ, initial weights θ_0, and shared model weights θ_1. For the optimization, we use a reconstruction loss between the output frame and ground truth frames as the training objective,

$$\text{loss} = \text{MSELoss}(\hat{x}_t, x_t), \tag{6.6}$$

where MSELoss is the mean square error loss.

6.3.3 Efficient video neural representation

Video neural representations are preferred due to their compactness and efficiency, which means they should be much smaller compared to original RGB videos. Therefore, we explore using HyperNeRV to generate efficient video neural representations in two ways. Firstly, we scale HyperNeRV to larger videos, either in terms of video length or resolution. This allows us to store more video content within the same neural representation size. Secondly, we scale HyperNeRV to stronger training setups, either by fitting on a larger or more diverse training dataset, extending the training epochs, or introducing stronger data augmentation to improve the generalization performance. For a given video, a more faithful reconstruction also means a more efficient neural representation for such representation.

6.4 Experiment

6.4.1 Datasets and implementation details

In our experiments, we use three commonly used video datasets: Kinetics-400 (referred to as K400 and our training dataset) [163], Something-Something V2 [164] (referred to as SthV2), and UCF-101 [165]. K400 and SthV2 focus on different visual cues for action recognition. K400 contains around 240k training videos and 20k test videos of 10 seconds from 400 classes. As the full dataset is too large and may take too long for HyperNeRV training, we randomly select a subset of K400 as our default training set, with 25 videos per class. We use test set of K400, SthV2 (around 20k motion-centric videos), and UCF101 (around 3.5k human-centric videos) for evaluation. The quality of the reconstructed videos is assessed using PSNR and SSIM.

The default video size is 128×128 with 4 frames. To preprocess the data, we first resize the input video so that its shorter side is 128, and then perform a center crop to obtain a 128×128 clip. We then uniformly sample 4 frames from the clip and input them to the model. For data tokenization, we divide the videos into 16×16 patches. The video model consists of four NeRV blocks, each with upscale factors of 8, 4, 2, and 2, respectively [1]. For larger spatial size videos, the patch and stride sizes are increased accordingly. The convolution layers in these blocks have a fixed channel width of 32, except for the input channel of the first block (time embedding dimension) and the output channel of the last block (video channels). The kernel size is 1 for the first convolution layer and 3 for the others. N_{max} is 128 for weight tokens unless stated otherwise.

The transformer hypernetwork consists of 6 encoder layers with a hidden dimension of 720 and a forward dimension of 2880. We use the Adam optimizer with a batch size of 16, an initial learning rate of 1e-4, 400 epochs, and step-wise learning rate decay with a decay factor of 0.1 at epoch 360. Our implementation is based on Pytorch [126]. We provide additional implementation details and visualization results in the supplementary material.

6.4.2 Main results

As a method to reduce the encoding time for video neural representation, we compare HyperNeRV with two baseline methods: NeRV [1], which trains the video model from scratch, and Trans-INR [2], which leverages hypernetwork to generate pixel-wise neural representation. For three test datasets, we randomly select 100 test videos from each for evaluation as training the entire test set (around 44k videos) would be time-consuming. We provide PSNR results in table 6.2a and SSIM results in table 6.2b, respectively. Compared to NeRV, HyperNeRV

Methods	Epochs	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑	Encoding time ↓
NeRV [1]	300	27.91	27.86	30.15	28.64	$\sim 7.5k\times$
NeRV [1]	400	29.94	29.96	32.14	30.68	$\sim 10k\times$
NeRV [1]	500	31.7	31.65	33.85	32.4	$\sim 12.5k\times$
NeRV [1]	600	**33.2**	33.2	**33.92**	**35.35**	$\sim 15k\times$
Trans-INR [2]	-	24.19	24.32	24.65	24.38	$2.7\times$
HyperNeRV	-	32.98	**33.27**	33.55	33.27	**1**×

(a) **PSNR** results.

Methods	Epochs	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑	Encoding time ↓
NeRV [1]	500	0.9078	0.8964	0.9102	0.9048	$\sim 12.5k\times$
NeRV [1]	600	0.9280	0.9176	0.9299	0.9252	$\sim 15k\times$
NeRV [1]	800	0.9522	0.9427	0.9538	0.9496	$\sim 20k\times$
NeRV [1]	1000	**0.9655**	**0.9605**	**0.9675**	**0.9645**	$\sim 25k\times$
Trans-INR [2]	-	0.6996	0.7229	0.7561	0.7262	$2.7\times$
HyperNeRV	-	0.9555	0.9589	0.9596	0.9580	**1**×

(b) **SSIM** results

Table 6.2: **Encoding comparison** for methods: NeRV [1] (train from scrach), Trans-INR [2], and HyperNeRV (ours).

significantly reduces the encoding time required to generate video neural representations. It can generate model weights efficiently with a simple forward pass, resulting in encoding speedups of up to $\sim 10^4\times$ while achieving comparable reconstruction quality. On an RTX2080ti GPU, HyperNeRV can encode videos at a real-time processing speed of about 100 videos per second. Compared to Trans-INR, HyperNeRV can output efficient neural representations with a faster encoding speed ($2.7\times$ faster), while also resulting in higher video quality according to both PSNR and SSIM metrics.

In fig. 6.4, we present qualitative results where HyperNeRV demonstrates the ability to reconstruct unseen videos with high fidelity and *generalizes well across different test datasets* (also shown in table 6.2). The reconstructions capture most video details, including dynamic

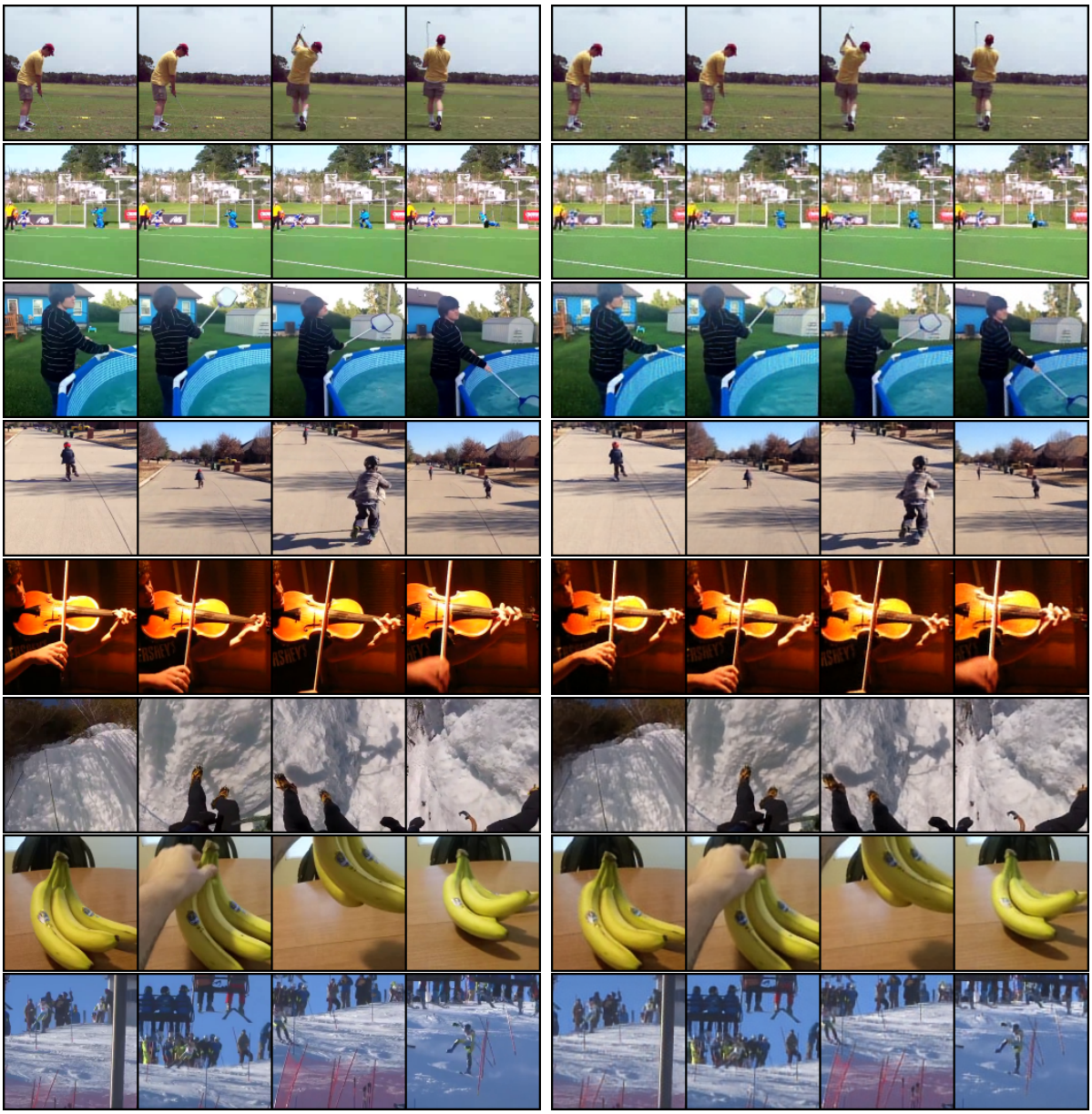


Figure 6.4: **Left:** video ground truth. **Right:** HyperNeRV output. HyperNeRV can reconstruct various videos across different datasets, and capture video details with high fidelity, for either dynamic scenes, complex textures, or moving objects.

scenes, complex textures, and moving objects, with faithful reconstruction. These results, along with the quantitative evaluations, demonstrate the effectiveness of HyperNeRV towards fast learning of video neural representations. We also provide visualization comparisons with Trans-INR in fig. 6.5, where HyperNeRV outputs videos of significantly better quality.

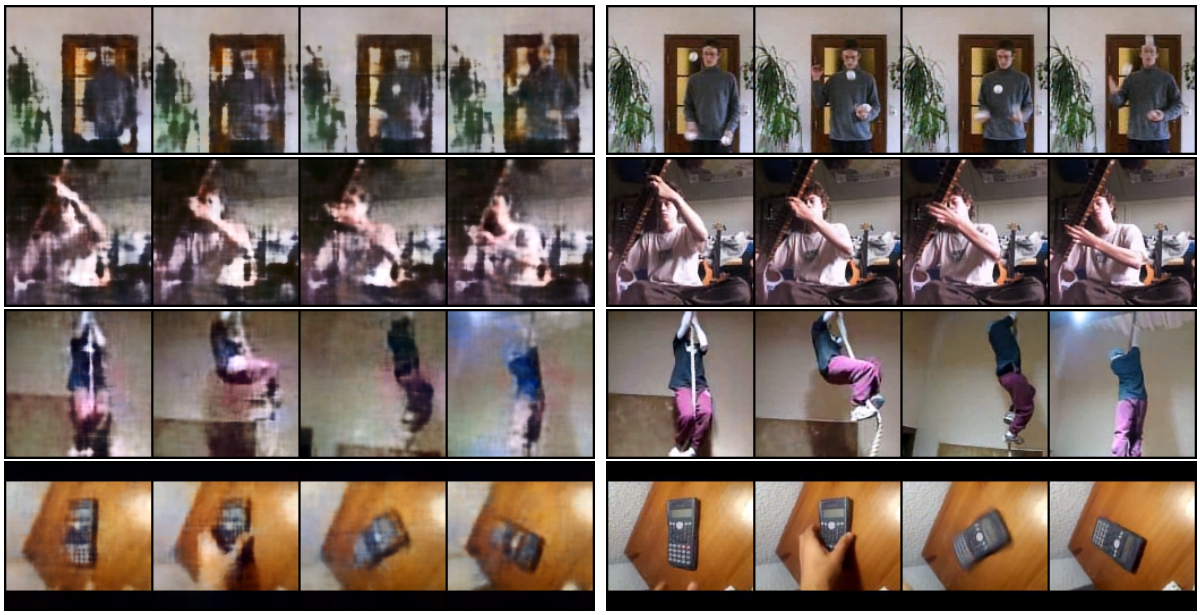


Figure 6.5: **Left**: Trans-INR [2] output: **Right**: HyperNeRV output (ours). HyperNeRV shows much better reconstruction quality than Trans-INR [2], with more faithful details, sharper textures, and better visual preference.

6.4.3 Efficient neural representation.

In this section, we extend the capabilities of HyperNeRV to generate more efficient neural representations that can accommodate larger videos or achieve better video reconstruction quality with stronger training setups.

Compact video representation. We evaluate HyperNeRV with larger videos by increasing spatial sizes or frame numbers and report the results in table 6.3a. We observe that wile the performance can drop gradually when increasing video frames, it remains robust to bigger spatial sizes and maintains the reconstruction quality. Specifically, the test PSNR drops by $0.6-0.8$ when decreasing bits per pixel (bpp) by $3-4\times$ via increasing spatial size, while the test PSNR drops by around 2 when decreasing bpp by $\sim 2\times$ via incrasing video frames. We also compare HyperNeRV with NeRV to generate compact representations in Tables 6.3b and 6.3c. Compared

Video size $T \times H \times W$	bpp ↓	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
$4 \times 128 \times 128$	24.9	32.98	33.27	33.55	33.27
$4 \times 256 \times 256$	9.37	32.21	32.67	32.94	32.61
$4 \times 512 \times 512$	2.41	31.24	31.69	32.53	31.82
$8 \times 128 \times 128$	12.5	30.98	31.19	31.56	31.24
$16 \times 128 \times 128$	6.29	29.07	29.18	29.43	29.23
$32 \times 128 \times 128$	3.16	26.85	27.25	27.55	27.22
$8 \times 512 \times 512$	1.25	28.63	29.08	29.37	29.03
$16 \times 512 \times 512$	0.61	26.56	26.96	27.27	26.93

(a) HyperNeRV results for different **video sizes**. bpp (bits per pixel) is the video checkpoint size divided by pixel number.

Methods	Epochs	bpp ↓	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑	Encoding time ↓
NeRV [1]	100	1.25	23.50	22.91	25.60	22.91	$\sim 3.2k\times$
NeRV [1]	200	1.25	26.63	26.06	28.74	26.06	$\sim 6.4k\times$
NeRV [1]	300	1.25	28.74	28.28	30.70	28.28	$\sim 9.6k\times$
NeRV [1]	400	1.25	**30.27**	**29.85**	**32.14**	**29.85**	$\sim 12.8k\times$
HyperNeRV	-	1.25	28.63	29.08	29.37	29.03	**1**×

(b) Encoding comparison for video size $8 \times 512 \times 512$.

Methods	Epochs	bpp ↓	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑	Encoding time ↓
NeRV [1]	100	0.61	23.41	22.12	25.25	22.12	$\sim 2.4k\times$
NeRV [1]	200	0.61	26.17	24.76	27.88	24.76	$\sim 4.8k\times$
NeRV [1]	300	0.61	28.31	26.86	29.90	26.86	$\sim 7.2k\times$
NeRV [1]	400	0.61	**29.56**	**28.10**	**30.90**	**28.10**	$\sim 9.6k\times$
HyperNeRV	-	0.61	26.56	26.96	27.27	26.93	**1**×

(c) Encoding comparison for video size $16 \times 512 \times 512$.

Table 6.3: PSNR results for **compact video representations**.

to training NeRV from scratch, HyperNeRV can significantly reduce the encoding time (more than 7×10^3 speedup) while achieving similar reconstruction quality.

Stronger training. *a) Data augmentation.* We investigate the use of augmentation techniques as outlined in RandAugment [3] to boost the generalization performance. Results in table 6.4a show that data augmentation during training can further improve performance on the test set (+1.73

Aug	Train ↑	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
no	32.9	31.03	31.53	31.81	31.46
yes	**33.05**	**32.61**	**33.06**	**33.61**	**33.09**

(a) Results with **data augmentation**.

C, V	N	Train ↑	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
400, 25	10k	32.9	31.03	31.53	31.81	31.46
100, 100	10k	30.49	28.39	28.85	29.14	28.79
400, 50	20k	**34.16**	31.97	32.5	32.78	32.42
400, 100	40k	34.12	32.65	33.42	33.65	33.24
400, 200	80k	34.03	**33.35**	**34.21**	**34.52**	**34.03**

(b) **Training dataset size and diversity** for HyperNeRV. C is class numbers, V is the video number per class. N is the number of total training videos.

Epochs	Train ↑	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
200	31.25	30.33	30.76	31.03	30.71
400	32.9	31.03	31.53	31.81	31.46
600	33.73	31.33	31.89	32.18	31.80
800	**35.19**	**31.78**	**32.36**	**32.65**	**32.26**
1000	**35.19**	31.44	31.93	32.23	31.87
1200	35.09	31.04	31.58	31.87	31.50

(c) Results for **longer training epochs**.

Table 6.4: **Stronger training setups** to obtain efficient video representations. PSNR is reported for training and test videos (UCF101, K400, SthV2, and Avg.).

PSNR), even though training PSNR does not increase much (+0.13 PSNR). We also examine various common augmentation strategies in this study, including random scaling, flipping, and changing the aspect ratio of the video, with results provided in the supplementary material.

b) Training data size and diversity. Results of different training datasets are presented in table 6.4b, where we investigate how the size and diversity of the training dataset affect the performance of HyperNeRV. We compare the default training dataset of 400 classes with 25 videos per class (400,25) to other datasets. We find that training videos with higher diversity

Methods	Total size	Video model	Img-wise	Act.	N_{max}	Train ↑	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑	$\text{Speed}_{\text{train}}$ ↓ (min/epoch)	$\text{Speed}_{\text{test}}$ ↑ (VPS)
Trans-INR [2]	44.9M	296.7K	no	ReLU	64	27.16	24.19	24.32	24.65	24.38	1.9	36.5
HyperNeRV (ours)	39.9M	223.2K	yes	ReLU	64	28.27	26.38	26.83	27.14	26.78	1.1 (**1.7**×)	99.5 (**2.7**×)
	39.9M	223.2K	yes	GELU	64	31.26	28.87	29.32	29.61	29.27		
	40.1M	223.2K	yes	GELU	128	**32.9**	**31.03**	**31.53**	**31.81**	**31.46**		

Table 6.5: **Component analysis** for HyperNeRV. 'Total size' is the number of all learnable parameters, 'Video model' is the parameters for the video model, 'Img-wise' is based on image-wise neural representation while Trans-INR [2] is based on pixel-wise neural representation. 'Act.' is the activation layer in the video model, N_{max} is the maximum number for weight tokens, 'Train' and 'Avg.' are the average PSNR on the training and test set. 'VPS' is videos per second.

leads to much better performance on the test set (+2.67 PSNR) when comparing (400,25) to (100,100), even though both datasets have 10k total videos. In addition to video diversity, adding more videos to the training set also improved the reconstruction quality. For example, (400,50), (400,100), and (400,200) outperformed (400,25). Although (400,50) does not improve the training PSNR, it improves the test PSNR by +0.96, indicating better generalization due to the increased number of videos. Further increasing the video number to (400,100) and (400,200) resulted in even greater improvements of +1.78 and +2.57, respectively.

c) Longer training. In table 6.4c, we provide results on how training time affects the final performance of HyperNeRV. We find that longer training leads to better fitting on training videos and improved performance on the test set, up to a certain point (800 epochs in this case). Increasing the training epochs from 200 to 400 results in a training PSNR increase of $+1.65$ and a test PSNR increase of $+0.75$. This trend continues when increasing the epochs from 400 to 800, resulting in a training PSNR increase of $+2.29$ and a test PSNR increase of $+0.8$.

6.4.4 Component analysis.

We compare HyperNeRV with Trans-INR [2] which also uses a transformer hypernetwork to generate model weights for implicit neural representations. The video model in HyperNeRV takes frame index as input and uses a convolutional neural network for reconstruction, while that of Trans-INR takes pixel coordinates as input and uses an MLP to output pixel RGB values. All other setups follow the original paper. To ensure a fair comparison, the total parameters and video model size are slightly smaller than those of Trans-INR. Results are presented in table 6.5.

Image-wise video model. Changing the video model from a pixel-wise to image-wise leads to better generalization and a significant speedup. While HyperNeRV only improves training PSNR by $+1.1$, it improves test PSNR by $+2.4$, indicating that it generalizes better to unseen videos. In addition to its stronger capacity and better generalization, HyperNeRV also has a speed advantage, since pixel-wise methods need to forward the video model $H \times W$ times to output all pixels which results in a significant slowdown. Compared to Trans-INR [2], HyperNeRV is $1.7\times$ faster for training and a $2.7\times$ faster for testing. An image-wise neural representation only requires one forward pass to output an entire frame, while pixel-wise methods need to forward the video model $H \times W$ times to output all pixels, leading to a significant slowdown. The qualitative results in fig. 6.5 show that HyperNeRV reconstructs unseen videos with a sharper appearance and more faithful details across various videos.

Activation layer. We replace the ReLU activation layer in the naive HyperNeRV baseline with GELU, which has been shown to be a better activation function for both pixel-wise and image-wise methods. The improved activation layer significantly increases the video model's capacity and boosts the final video quality by an additional $+2.5$ PSNR.

Number of weights tokens. Increasing maximum number of weight tokens (N_{max}) can further improve the performance, +1.64 for training PSNR and +2.19 for test PSNR. We can interpret the increase in N_{max} as increasing the capacity of the hypernetwork to generate more complex and expressive video models that can better capture the underlying structure of the input videos.

6.4.5 Discussion and limitations

In summary, we have presented two ways to enhance the reconstruction quality of test videos. Firstly, by designing better architectures for the hypernetwork or video model, we can improve the fitting on training videos. Secondly, exploring data augmentation techniques and increasing the size and diversity of the training dataset can help achieve better generalization for unseen videos. We believe that there is still much potential to explore for HyperNeRV and hope that our work can contribute to further advancements for fast learning of video neural representation.

As shown in table 6.3, there is still room for improvement when using HyperNeRV to generate compact video representations, especially for videos with long sequences. Additionally, HyperNeRV currently only works with a fixed number of frames and video resolutions. An interesting future direction would be to extend HyperNeRV to support videos of varying lengths and resolutions, allowing it to work with any videos in our daily lives and potentially further popularize video neural representations.

6.5 Conclusion

In this paper, we present HyperNeRV, a transformer hypernetwork designed to generate model weights for video neural representations. We also explore several design principles, including bigger videos, architecture design, training dataset size and diversity, and data augmentation, to improve representation efficiency. Our experiments demonstrate that HyperNeRV can achieve fast learning of video neural representations, with a speed-up of $\sim 10^4\times$ compared to training the video model from scratch.

6.6 Experiment supplement

6.6.1 Ablation study

We first provide ablation results for data augmentation, weight token number, and learning rate schedule.

6.6.1.1 Data augmentation

We first do an ablation study for common data augmentations, like cropping the video with random aspect ratios, randomly resize the video before cropping, randomly flipping, and random augmentation [3]. Results are shown in table 6.6. For random cropping, we choose a aspect ratio between [0.67, 1.5]; for random scaling, we choose one scaling factor from [0.8, 1.25]; for random augmentation, we set the augmentation step as 1, the augmentation magnitude as 2, and the augmentation magnitude bins as 5. Since random cropping and scaling of the video may introduce a domain gap between training videos and test videos, they do not improve

performance on the test set. In contrast, random flipping boosts the reconstruction of test videos with a small margin at $+0.2$. Finally, we found random augmentation can significantly improve the generalization of HyperNeRV, by $+1.3$.

Aug.	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
no	31.03	31.53	31.81	31.46
rand ratios	27.59	28.11	28.43	28.04
rand scale	30.75	31.31	31.58	31.21
rand flip	31.21	31.76	32.05	31.67
rand aug	**32.61**	**33.06**	**33.61**	**33.09**

Table 6.6: **Data augmentations.** 'rand ratios' is cropping the video with random aspect ratios between $[0.67, 1.5]$. 'rand size' is randomly scaling, between $[0.8\times, 1.25\times]$, the video before cropping. 'rand aug' is random augmentation [3].

6.6.1.2 Weight token number.

We also provide ablation study on N_{max} since it leads to significant improvements in table 6.5 when increasing N_{max} from 64 to 128. But HyperNeRV with a larger N_{max} (128) does not lead to further improvements, which is shown in table 6.7.

N_{max}	Train ↑	UCF101 ↑	K400 ↑	SthV2 ↑	Avg. ↑
64	31.26	28.87	29.32	29.61	29.27
128	**32.9**	**31.03**	**31.53**	**31.81**	**31.46**
256	31.44	29.59	30.07	30.37	30.01

Table 6.7: Ablation study for $\boldsymbol{N}_{\text{max}}$. Increasing $\boldsymbol{N}_{\text{max}}$ from 128 to 256 does not improve the performance further.

6.6.1.3 Learning rate schedules.

Finally, we conduct ablation study for learning rate schedules in table 6.8, where we compare step-wise learning rate decay with the common cosine decay schedule. Step-wise

learning rate schedule gives better results for both training and test videos.

	Train ↑	UCF101↑	K400↑	SthV2↑	Avg. ↑
Cosine	31.46	29.85	30.36	30.65	30.29
Step-wise	**32.9**	**31.03**	**31.53**	**31.81**	**31.46**

Table 6.8: Ablation for **learning rate schedules**.

6.6.2 More implementation details

We provide more implementation details in table 6.9 for the default experiment setup, for input frames, NeRV architecture, Transformer architecture, and Adam optimizer respectively.

Input	Frame num	4
	Frame size	128
	Patch size	16
	Batch size	16
NeRV details	Strides	8,4,2,2
	Kernel size	1,3,3,3
	Channle width	32
	Output bias	0.5
Transformer	Model dim	720
	Feedforward dim	2880
	Attention heads	12
	Encoder layers	6
Optimizer	Learning rate	0.0001
	Max epoch	400
	Decay epoch	360
	Lr decy	0.1

Table 6.9: Implementation **details** for HyperNeRV.

Chapter 7: Conclusion

7.1 Efficient video representation

The dissertation's main objective is to develop efficient implicit neural representations, specifically for videos, which we will call NeRV. These representations aim to use a deep neural network to generate the corresponding video frame when given a frame index, requiring fewer parameters and achieving high accuracy. In essence, this approach converts the video compression problem to a model compression problem.

To further enhance NeRV's performance, we propose a hybrid neural representation for videos (HNeRV). HNeRV combines a small frame embedding with a powerful decoder network, resulting in better internal generalization and representation capacity. Additionally, the distribution of model parameters across layers in HNeRV results in faster convergence compared to NeRV.

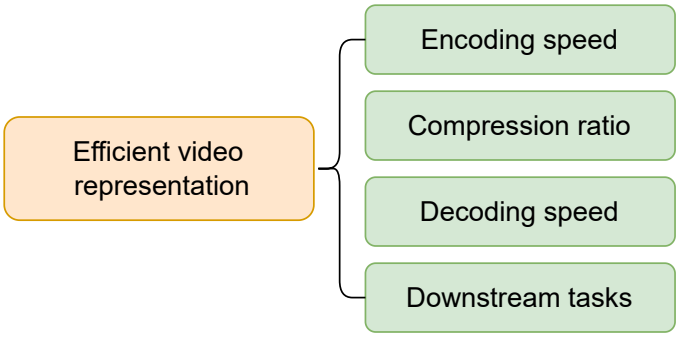


Figure 7.1: Framework of efficient video representation.

To provide a comprehensive evaluation of neural representation methods for videos, we

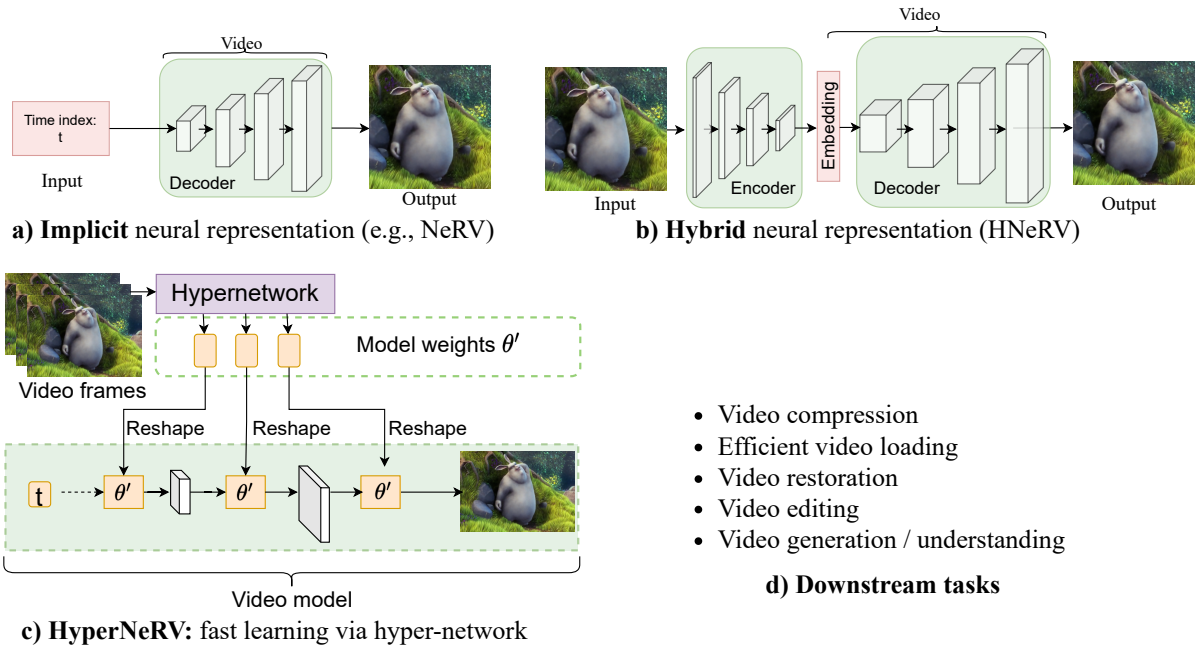


Figure 7.2: Dissertation framework overview.

have identified four key perspectives, as illustrated in Figure 7.1. The compression ratio is the primary metric for evaluating the efficiency of video representations, as it determines the amount of data required to store and transmit the video. Additionally, the encoding and decoding speeds required to convert the original video into efficient representations and reconstruct the video from these representations should also be considered.

Moreover, the use of efficient video representations is a critical application that has not been explored thoroughly in previous studies. Most downstream video tasks still rely on the original frame sequences as input, which have high dimensions and significantly increase the computational burden for video-related tasks such as video understanding and video generation. However, utilizing compact and efficient video representations has the potential to alleviate this issue and improve the efficiency of video-related tasks.

We have developed a comprehensive approach for addressing various challenges associated with neural video representations. Our implicit neural representations enable us to transform the

video compression problem into a model compression problem, achieving comparable compression ratios to state-of-the-art methods. Additionally, our approach outperforms other compression methods in videos with still backgrounds, making it an effective solution for different types of videos.

Moreover, our implicit representations offer decoding advantages, allowing for easy deployment on any platform. We have also developed an efficient neural video dataloader (NVLoader) to enable faster training and evaluation of video models, which is approximately three times faster than conventional video dataloaders.

To address the challenge of encoding speed, we introduce the HyperNeRV framework, which utilizes a hypernetwork to directly map input videos to NeRV model weights. This approach significantly accelerates the encoding process by approximately 10^4 times, while maintaining similar reconstruction quality and generalizing well to unseen videos, compared to training the neural network from scratch.

In addition to our contributions of developing efficient implicit neural representations and introducing the HyperNeRV framework, we have also explored the potential of these representations in downstream applications. Our findings demonstrate that the compactness and efficiency of these video representations make them well-suited for frame interpolation, video restoration, and video editing tasks.

Furthermore, we believe that these implicit video representations have a broad range of applications beyond the ones we have explored. They can be used as an output video format that significantly reduces the search space, or as an efficient input for video understanding models. Additionally, these representations have the potential to enhance video summarization, action recognition, and content-based video retrieval, among other applications. We recommend that

future research further investigate these potential applications.

To provide an overview of our dissertation, we have created Figure 7.2, which illustrates the key contributions of our research. The NeRV framework, an implicit neural representation for videos, is introduced in Figure 7.2a and Chapter 3. In Figure 7.2b and Chapter 4, we upgrade the representation by incorporating a content-adaptive embedding, which leads to the development of the Hybrid Neural Representation for Videos (HNeRV). Additionally, we introduce the HyperNeRV framework in Figure 7.2c and Chapter 6, which enables fast learning of video neural representations. Finally, we demonstrate the versatility of the efficient video representations by showcasing various downstream tasks that utilize the learned model weights directly in Figure 7.2d, such as efficient video loading in Chapter 5.

7.2 Downstream tasks based on neural representations

The use of implicit representations allows for the representation of video data in a compact and efficient manner. We explore several downstream tasks that directly utilize these representations, either as input or output. This offers significant advantages over other video-related methods that rely on high-dimensional frame sequences, as our video data requires much fewer computation resources to process.

Our ultimate goal is to introduce a novel approach to video processing that is analogous to the Fourier transform in signal processing. By transforming videos into neural space, we aim to advance the research and application of video data.

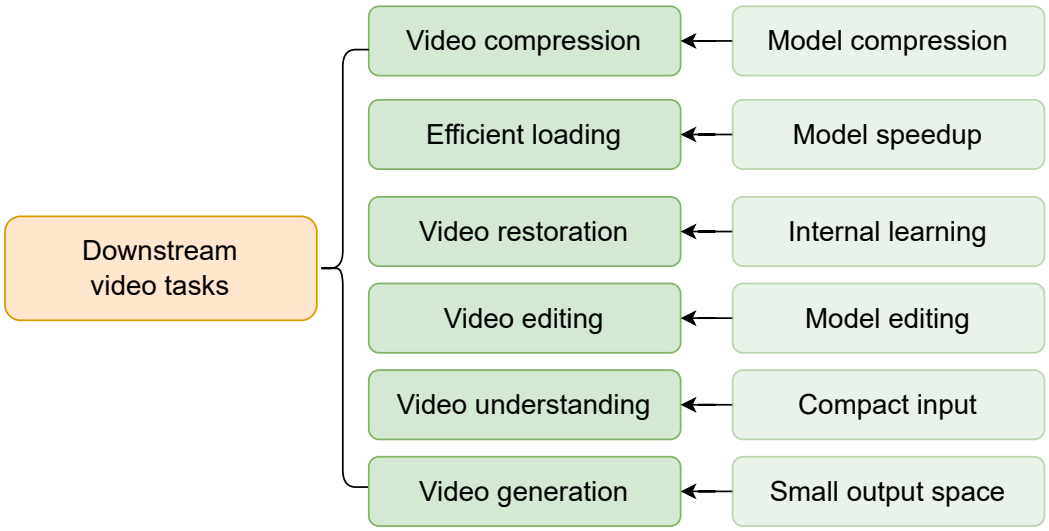


Figure 7.3: Downstream video tasks based on implicit neural representations.

7.2.1 Video compression

Our first task is to explore video compression using implicit neural representations. Our approach is particularly attractive because the small size of the representation is inherently more robust to parameter precision, allowing for pruning or quantization operations that can greatly reduce storage space without sacrificing video quality. In this dissertation, we explore three common compression techniques: model pruning, model quantization, and entropy coding. However, we believe there is potential for even greater improvements through the use of other compression techniques such as quantization-aware training, sparsity regularization, or stronger encoding algorithms.

7.2.2 Efficient video loading

In addition to video compression, our neural representations also enable efficient video loading. Since decoding a video is simply a feedforward operation of the neural network, our

approach enables 3-6 times faster loading compared to conventional video dataloaders. Furthermore, the simplicity of our decoding process allows for easy deployment on any device or platform, without requiring device-specific optimization.

However, we believe that there is still potential for further speed improvements through additional model optimization techniques. For example, we could use neural architecture search to find a more efficient and faster neural network, or employ post-training quantization to speed up the model decoding process.

7.2.3 Video restoration

In addition to compression and efficient data loading, our research shows that implicit neural representations can also be robust to video degradation in the RGB space, making them useful for video restoration tasks. We found that our approach can effectively remove noisy pixels and inpaint missing regions in videos, without the need for specific designs or preprocessing steps. This is because noisy pixels are difficult to optimize during the learning process, as they lack clear patterns, but our implicit representation can easily remove them in the final output. Similarly, for distorted videos, we simply ignore the masked areas during training, and our neural network can automatically inpaint these regions during decoding. This feature is particularly appealing because noise and distortion are common in real-world video scenarios.

7.2.4 Video editing

Implicit neural representations can also be used for video editing, allowing us to directly edit the model weights and apply the changes to the entire video. Because the entire video is

represented with one model, all frames share the same representation. We hypothesize that if we can obtain the target model using a few frames, we can transfer the editing results to all other frames. We have demonstrated promising preliminary results for various editing tasks such as video colorization, background blur, and object colorization. With only 10% of the frames edited, we were able to effectively edit the entire video.

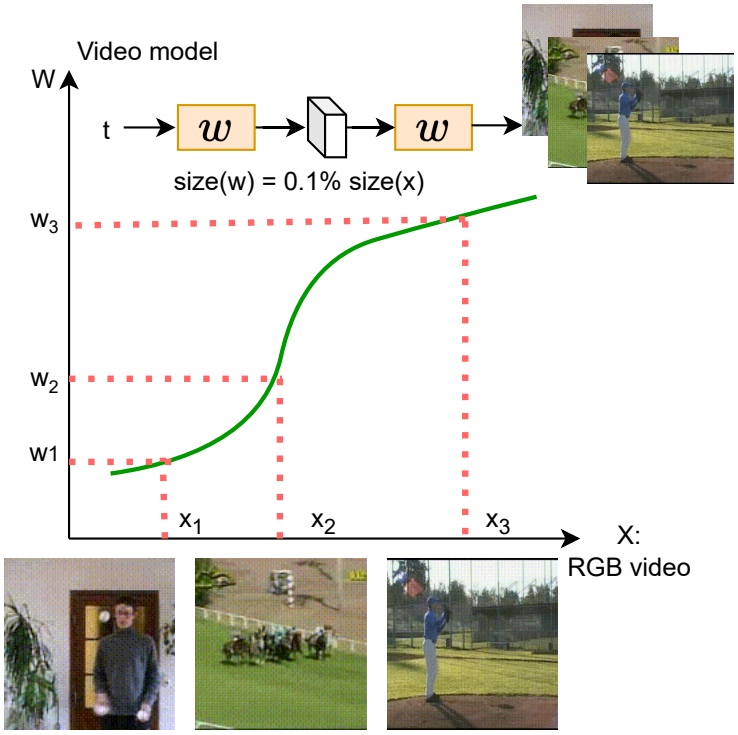


Figure 7.4: Implicit neural representation is a compact input for video understanding and perfect for video generation due to its smaller size compared to original video data.

7.2.5 Video understanding

The compactness of implicit neural representations, as demonstrated in Figure 7.4, makes them a promising input for video understanding models. Given that computation resources such as processing latency and memory consumption increase significantly with input data dimension, the ability of implicit neural representations to reconstruct videos despite their small size enables

downstream recognition tasks while dramatically reducing computational demands.

7.2.6 Video generation

Implicit neural representations can also serve as an ideal output for video generation. The compactness of these representations significantly reduces the search space, making the generation process more efficient. Since the entire video is represented as a single entity, implicit representations can also provide better temporal consistency in generated videos.

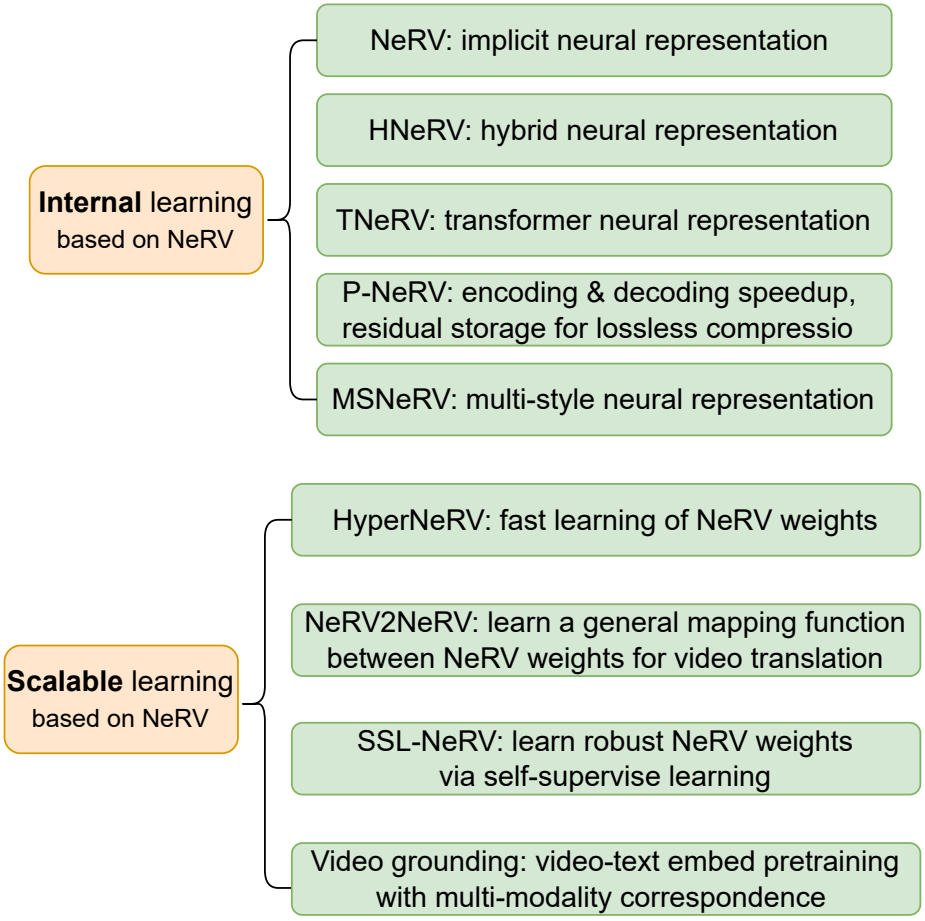


Figure 7.5: Some ongoing or potential projects based on NeRV.

7.3 Future work and limitations

We categorize our existing and ongoing projects into two types: internal learning based on NeRV and scalable learning based on NeRV, as depicted in Figure 7.5. In the case of internal learning, the NeRV model is trained using data from a single video, while scalable learning involves training the model on a large-scale dataset to obtain model weights.

7.3.1 Internal learning

Besides implicit neural representations for videos (NeRV) and hybrid neural representations for videos (HNeRV), we are also working on three projects to build better video representaions.

T-NeRV. As demonstrated in NVLoader, patch-wise decoding has been shown to offer improved flexibility and efficiency in video data loading. Therefore, our current research focuses on developing a Transformer Neural Representation (T-NeRV) that utilizes a transformer neural network for patch-wise decoding. We believe that T-NeRV can serve as an efficient hybrid neural representation, and patch-wise modeling presents a novel approach for capturing patch correspondence and geometry information, offering new perspectives in video analysis.

P-NeRV. We are currently engaged in research aimed at developing practical neural representations for videos (P-NeRV), with the goal of addressing four key bottlenecks associated with video compression methods: encoding and decoding speed, compression ratios, and lossless compression. To accelerate the encoding speed, we are exploring the use of HyperNeRV to generate neural representations directly, thereby avoiding the need to train models from scratch and significantly increasing encoding efficiency. Additionally, we are employing post-quantization techniques

to reduce the feed-forward latency and improve decoding speed. Furthermore, we are actively working on improving compression ratios for videos featuring dynamic scenes. Lastly, we plan to efficiently store residual videos for lossless compression using low-rank decomposition techniques.

MS-NeRV. We are also actively developing a multi-scale neural representation for videos, known as MS-NeRV. This novel representation enables a single model to store videos with various styles, demonstrating superior efficiency compared to other existing video representations. One notable advantage of hybrid neural representations is their ability to exhibit good internal generalization, meaning that similar embeddings produce similar video frames. As a result, video styles can propagate seamlessly among frames, enabling efficient style propagation within the representation. Since multiple styles are stored in one model, conditioned on a style embedding, we further investigate style interpolation in the style embedding space. This approach allows us to smoothly interpolate between different styles, enhancing the versatility and expressiveness of the representation. Furthermore, we investigate video decomposition for different object areas, where instead of using a global style for the entire video, we can assign different styles to different objects. This approach enables us to achieve a wide range of video combinations, adding to the versatility of MS-NeRV.

7.3.2 Scalable learning

In addition to our efforts in developing more efficient video neural representations based on NeRV, we are also actively working on scalable video neural representations based on HyperNeRV. These representations are designed to handle larger-scale video data and offer enhanced scalability

for applications that require processing and analyzing videos at a larger scale.

SSL-NeRV. Our initial focus is on generating general video neural representations that can be utilized in various downstream video processing tasks. For instance, if we aim to generate videos by sampling from the NeRV parameter space, it is desirable for the NeRV model parameters to follow a distribution, such as a Gaussian distribution. Additionally, we strive to ensure that the NeRV parameters are robust to variations in visual context, spatial/temporal cropping, and action category, to enable their effective application in diverse video processing scenarios.

NeRV2NeRV. In addition to learning a distribution of video neural representations, our research also focuses on learning a general mapping function that can convert one NeRV model into another, enabling video-to-video translation tasks such as video colorization, video super-resolution, video segmentation, and video stylization. To achieve this, we will optimize the hyper-network (to generate source NeRV model) and mapping function (to translate source NeRV to target NeRV) jointly. Our aim is to ensure that the mapping function can adapt effectively to the NeRV parameter space, allowing for accurate and versatile video-to-video translation capabilities.

Video-text embed pretraining Drawing inspiration from contrastive language-image pretraining (CLIP), we are also exploring the computation of similarity between videos and text. Our approach involves converting videos into the NeRV space, which offers a more efficient and effective representation for videos, enabling us to leverage multi-modality data for learning tasks that involve both videos and text.

7.3.3 Limitations

Despite the advantages of implicit video neural representations, there are limitations as well. One of the challenges is extracting spatial or temporal information from NeRV parameters without decoding video frames. Since implicit video representations lack explicit spatial or temporal dimensions, extracting such information remains a challenge at present.

Bibliography

[1] Hao Chen, Bo He, Hanyu Wang, Yixuan Ren, Ser-Nam Lim, and Abhinav Shrivastava. Nerv: Neural representations for videos, 2021.

[2] Yinbo Chen and Xiaolong Wang. Transformers as meta-learners for implicit neural representations. In *European Conference on Computer Vision*, 2022.

[3] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *CVPR workshops*, pages 702–703, 2020.

[4] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

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

[6] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9446–9454, 2018.

[7] Didier Le Gall. Mpeg: A video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, April 1991.

[8] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[9] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

[10] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[11] Abdelaziz Djelouah, Joaquim Campos, Simone Schaub-Meyer, and Christopher Schroers. Neural inter-frame compression for video coding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[12] Amirhossein Habibian, Ties van Rozendaal, Jakub M. Tomczak, and Taco S. Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[13] Jerry Liu, Shenlong Wang, Wei-Chiu Ma, Meet Shah, Rui Hu, Pranaab Dhawan, and Raquel Urtasun. Conditional entropy coding for efficient video compression. In *ECCV*, 2020.

[14] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G. Anderson, and Lubomir Bourdev. Learned video compression. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[15] Haojie Liu, Tong Chen, Ming Lu, Qiu Shen, and Zhan Ma. Neural video compression using spatio-temporal priors, 2019.

[16] Eirikur Agustsson, David Minnen, Nick Johnston, Johannes Balle, Sung Jin Hwang, and George Toderici. Scale-space flow for end-to-end optimized video compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[17] Mehrdad Khani, Vibhaalakshmi Sivaraman, and Mohammad Alizadeh. Efficient video compression via content-adaptive super-resolution. *arXiv preprint arXiv:2104.02322*, 2021.

[18] Oren Rippel, Alexander G. Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elf-vc: Efficient learned flexible-rate video coding, 2021.

[19] Gregory K Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

[20] Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao. Discrete cosine transform. *IEEE transactions on Computers*, 100(1):90–93, 1974.

[21] Wenxue Cui, Tao Zhang, Shengping Zhang, Feng Jiang, Wangmeng Zuo, and Debin Zhao. Convolutional neural networks based intra prediction for hevc. *arXiv preprint arXiv:1808.05734*, 2018.

[22] Jiahao Li, Bin Li, Jizheng Xu, Ruiqin Xiong, and Wen Gao. Fully connected network-based intra prediction for image coding. *IEEE Transactions on Image Processing*, 27(7):3236–3247, 2018.

[23] Marc Gorriz Blanch, Saverio Blasi, Alan Smeaton, Noel E O'Connor, and Marta Mrak. Chroma intra prediction with attention-based cnn architectures. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 783–787. IEEE, 2020.

[24] Yue Li, Li Li, Zhu Li, Jianchao Yang, Ning Xu, Dong Liu, and Houqiang Li. A hybrid neural network for chroma intra prediction. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1797–1801. IEEE, 2018.

[25] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[26] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*, 2018.

[27] Lei Zhou, Zhenhong Sun, Xiangji Wu, and Junmin Wu. End-to-end optimized image compression with attention mechanism. In *CVPR workshops*, page 0, 2019.

[28] Yueyu Hu, Wenhan Yang, and Jiaying Liu. Coarse-to-fine hyper-prior modeling for learned image compression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 11013–11020, 2020.

[29] David Minnen, Johannes Ballé, and George D Toderici. Joint autoregressive and hierarchical priors for learned image compression. *Advances in neural information processing systems*, 31, 2018.

[30] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned image compression with discretized gaussian mixture likelihoods and attention modules. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7939–7948, 2020.

[31] Jun-Hyuk Kim, Byeongho Heo, and Jong-Seok Lee. Joint global and local hierarchical priors for learned image compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5992–6001, 2022.

[32] Oren Rippel and Lubomir Bourdev. Real-time adaptive image compression. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2922–2930. PMLR, 06–11 Aug 2017.

[33] Didier Le Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991.

[34] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[35] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11006–11015, 2019.

[36] Ruihan Yang, Yibo Yang, Joseph Marino, and Stephan Mandt. Hierarchical autoregressive modeling for neural video compression. *arXiv preprint arXiv:2010.10258*, 2020.

[37] Eirikur Agustsson, David Minnen, Nick Johnston, Johannes Balle, Sung Jin Hwang, and George Toderici. Scale-space flow for end-to-end optimized video compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8503–8512, 2020.

[38] Zhihao Hu, Zhenghao Chen, Dong Xu, Guo Lu, Wanli Ouyang, and Shuhang Gu. Improving deep video compression by resolution-adaptive flow coding. In *European Conference on Computer Vision*, pages 193–209. Springer, 2020.

[39] Zhihao Hu, Guo Lu, and Dong Xu. FVC: A new framework towards deep video compression in feature space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1502–1511, 2021.

[40] Jianping Lin, Dong Liu, Houqiang Li, and Feng Wu. M-LVC: multiple frames prediction for learned video compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[41] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. ELF-VC: Efficient learned flexible-rate video coding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 14479–14488, October 2021.

[42] Jorge Pessoa, Helena Aidos, Pedro Tomás, and Mário AT Figueiredo. End-to-end learning of video compression using spatio-temporal autoencoders. In *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2020.

[43] Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *ICCV*, 2019.

[44] Wenyu Sun, Chen Tang, Weigui Li, Zhuqing Yuan, Huazhong Yang, and Yongpan Liu. High-quality single-model deep video compression with frame-conv3d and multi-frame differential modulation. In *European Conference on Computer Vision (ECCV)*, pages 239–254. Springer, 2020.

[45] Théo Ladune, Pierrick Philippe, Wassim Hamidouche, Lu Zhang, and Olivier Déforges. Optical flow and mode selection for learning-based video coding. In *22nd IEEE International Workshop on Multimedia Signal Processing*, 2020.

[46] Théo Ladune, Pierrick Philippe, Wassim Hamidouche, Lu Zhang, and Olivier Déforges. Conditional coding for flexible learned video compression. In *Neural Compression: From Information Theory to Applications – Workshop @ ICLR*, 2021.

[47] Théo Ladune, Pierrick Philippe, Wassim Hamidouche, Lu Zhang, and Olivier Déforges. Conditional coding and variable bitrate for practical learned video coding. *CLIC workshop, CVPR*, 2021.

[48] Jiahao Li, Bin Li, and Yan Lu. Deep contextual video compression. *Advances in Neural Information Processing Systems*, 34, 2021.

[49] Xihua Sheng, Jiahao Li, Bin Li, Li Li, Dong Liu, and Yan Lu. Temporal context mining for learned video compression. *arXiv preprint arXiv:2111.13850*, 2021.

[50] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with recurrent auto-encoder and recurrent probability model. *IEEE Journal of Selected Topics in Signal Processing*, 15(2):388–401, 2021.

[51] Hao Chen, A Gwilliam Matthew, Bo He, Ser-Nam Lim, and Abhinav Shrivastava. CNeRV: Content-adaptive neural representation for visual data. In *BMVC*, 2022.

[52] Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. Coin: Compression with implicit neural representations. *arXiv preprint arXiv:2103.03123*, 2021.

[53] Ishit Mehta, Michaël Gharbi, Connelly Barnes, Eli Shechtman, Ravi Ramamoorthi, and Manmohan Chandraker. Modulated periodic activations for generalizable local functional representations, 2021.

[54] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.

[55] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions, 2020.

[56] Yinbo Chen, Sifei Liu, and Xiaolong Wang. Learning continuous image representation with local implicit image function. In *CVPR*, pages 8628–8638, 2021.

[57] Ivan Skorokhodov, Savva Ignatyev, and Mohamed Elhoseiny. Adversarial generation of continuous images. In *CVPR*, pages 10753–10764, 2021.

[58] Ivan Anokhin, Kirill Demochkin, Taras Khakhulin, Gleb Sterkin, Victor Lempitsky, and Denis Korzhenkov. Image generators with conditionally-independent pixel synthesis. In *CVPR*, pages 14278–14287, 2021.

[59] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks. *arXiv preprint arXiv:2106.12423*, 2021.

[60] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[61] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[62] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

[63] Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. Graf: Generative radiance fields for 3d-aware image synthesis, 2021.

[64] Vincent Sitzmann, Julien N.P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *NeurIPS*, 2020.

[65] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *NeurIPS*, 2020.

[66] Geoffrey E Hinton. Learning translation invariant recognition in a massively parallel networks. In *PARLE Parallel Architectures and Languages Europe: Volume I: Parallel Architectures Eindhoven, The Netherlands, June 15–19, 1987 Proceedings 1*, pages 1–13. Springer, 1987.

[67] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA*, 1974.

[68] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[69] Yann Le Cun, Ofer Matan, Bernhard Boser, John S Denker, Don Henderson, Richard E Howard, Wayne Hubbard, LD Jacket, and Henry S Baird. Handwritten zip code recognition with multilayer networks. In *[1990] Proceedings. 10th International Conference on Pattern Recognition*, volume 2, pages 35–40. IEEE, 1990.

[70] Ajay Shrestha and Ausif Mahmood. Review of deep learning algorithms and architectures. *IEEE access*, 7:53040–53065, 2019.

[71] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial intelligence review*, 53:5455–5516, 2020.

[72] Yann LeCun, Lawrence D Jackel, Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, Urs A Muller, Eduard Sackinger, Patrice Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261(276):2, 1995.

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[74] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[75] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[76] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[79] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[80] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[82] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pages 843–852, 2017.

[83] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *ICML*, pages 10347–10357. PMLR, 2021.

[84] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10428–10436, 2020.

[85] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[86] Alexandre Mercat, Marko Viitanen, and Jarno Vanne. Uvg dataset: 50/120fps 4k sequences for video codec analysis and development. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 297–302, 2020.

[87] Kyle Genova, Forrester Cole, Daniel Vlasic, Aaron Sarna, William T Freeman, and Thomas Funkhouser. Learning shape templates with structured implicit functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7154–7164, 2019.

[88] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas A Funkhouser. Deep structured implicit functions. 2019.

[89] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. *arXiv preprint arXiv:1906.01618*, 2019.

[90] Chiyu Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Nießner, Thomas Funkhouser, et al. Local implicit grid representations for 3d scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6001–6010, 2020.

[91] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. *arXiv preprint arXiv:2003.04618*, 2, 2020.

[92] Rohan Chabra, Jan E Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard Newcombe. Deep local shapes: Learning local sdf priors for detailed 3d reconstruction. In *European Conference on Computer Vision*, pages 608–625. Springer, 2020.

[93] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3504–3515, 2020.

[94] Michael Oechsle, Lars Mescheder, Michael Niemeyer, Thilo Strauss, and Andreas Geiger. Texture fields: Learning texture representations in function space. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4531–4540, 2019.

[95] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The jpeg 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58, 2001.

[96] Marc Antonini, Michel Barlaud, Pierre Mathieu, and Ingrid Daubechies. Image coding using wavelet transform. *IEEE Transactions on image processing*, 1(2):205–220, 1992.

[97] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learning image and video compression through spatial-temporal energy compaction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10071–10080, 2019.

[98] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 416–431, 2018.

[99] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with hierarchical quality and recurrent enhancement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6628–6637, 2020.

[100] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. 2011.

[101] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[102] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*, 2016.

[103] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *arXiv preprint arXiv:1608.03665*, 2016.

[104] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[105] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[106] Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2754–2761, 2013.

[107] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *arXiv preprint arXiv:1404.0736*, 2014.

[108] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[109] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999. PMLR, 2016.

[110] Shuangfei Zhai, Yu Cheng, Weining Lu, and Zhongfei Zhang. Doubly convolutional neural networks. *arXiv preprint arXiv:1610.09716*, 2016.

[111] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pages 2217–2225. PMLR, 2016.

[112] Sander Dieleman, Jeffrey De Fauw, and Koray Kavukcuoglu. Exploiting cyclic symmetry in convolutional neural networks. In *International conference on machine learning*, pages 1889–1898. PMLR, 2016.

[113] Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013.

[114] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[115] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 742–751, 2017.

[116] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[117] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310. PMLR, 2019.

[118] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1874–1883, 2016.

[119] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046*, 2018.

[120] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel Roy, and Ali Ramezani-Kebrya. Adaptive gradient quantization for data-parallel sgd. *arXiv preprint arXiv:2010.12460*, 2020.

[121] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *arXiv preprint arXiv:1812.08011*, 2018.

[122] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[123] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[124] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[125] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, volume 2, pages 1398–1402. Ieee, 2003.

[126] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[127] Haiqiang Wang, Weihao Gan, Sudeng Hu, Joe Yuchieh Lin, Lina Jin, Longguang Song, Ping Wang, Ioannis Katsavounidis, Anne Aaron, and C-C Jay Kuo. Mcl-jcv: a jnd-based h. 264/avc video quality assessment dataset. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1509–1513. IEEE, 2016.

[128] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. In *ICCV*, 2019.

[129] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[130] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[131] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[132] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[133] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[134] Haojie Liu, Ming Lu, Zhan Ma, Fan Wang, Zhihuang Xie, Xun Cao, and Yao Wang. Neural video coding using multiscale motion compensation and spatiotemporal context model. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(8):3182–3196, 2021.

[135] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5301–5310. PMLR, 09–15 Jun 2019.

[136] Zizhang Li, Mengmeng Wang, Huaijin Pi, Kechun Xu, Jianbiao Mei, and Yong Liu. E-nerv: Expedite neural video representation with disentangled spatial-temporal context. *ECCV*, 2022.

[137] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. ICML '08, New York, NY, USA, 2008. Association for Computing Machinery.

[138] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[139] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. Variational autoencoder for deep learning of images, labels and captions. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[140] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, pages 1135–1143, 2015.

[141] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[142] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, June 2018.

[143] Tengfei Wang, Hao Ouyang, and Qifeng Chen. Image inpainting with external-internal learning and monochromic bottleneck. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5120–5129, 2021.

[144] Hao Ouyang, Tengfei Wang, and Qifeng Chen. Internal video inpainting by implicit long-range propagation. In *International Conference on Computer Vision (ICCV)*, 2021.

[145] Assaf Shocher, Nadav Cohen, and Michal Irani. "zero-shot" super-resolution using deep internal learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[146] Haotian Zhang, Long Mai, Ning Xu, Zhaowen Wang, John Collomosse, and Hailin Jin. An internal learning approach to video inpainting. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2720–2729, 2019.

[147] Yosef Gandelsman, Assaf Shocher, and Michal Irani. "double-dip": Unsupervised image decomposition via coupled deep-image-priors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[148] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *CVPR*, 2022.

[149] Big buck bunny, sunflower version. `http://bbb3d.renderfarming.net/download.html`. Accessed: 2010-09-30.

[150] Jiahao Li, Bin Li, and Yan Lu. Deep contextual video compression. *NeurIPS*, 34, 2021.

[151] Chao-Yuan Wu, Ross Girshick, Kaiming He, Christoph Feichtenhofer, and Philipp Krahenbuhl. A multigrid method for efficiently training video models. In *CVPR*, pages 153–162, 2020.

[152] Christoph Feichtenhofer. X3d: Expanding architectures for efficient video recognition. In *CVPR*, pages 203–213, 2020.

[153] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[154] Yanghao Li, Chao-Yuan Wu, Haoqi Fan, Karttikeya Mangalam, Bo Xiong, Jitendra Malik, and Christoph Feichtenhofer. Mvitv2: Improved multiscale vision transformers for classification and detection. In *CVPR*, 2022.

[155] Zhan Tong, Yibing Song, Jue Wang, and Limin Wang. VideoMAE: Masked autoencoders are data-efficient learners for self-supervised video pre-training. In *NeurIPS*, 2022.

[156] Christoph Feichtenhofer, Haoqi Fan, Yanghao Li, and Kaiming He. Masked autoencoders as spatiotemporal learners. In *NeurIPS*, 2022.

[157] Subin Kim, Sihyun Yu, Jaeho Lee, and Jinwoo Shin. Scalable neural video representations with learnable positional features. *NeurIPS*, 2022.

[158] Zeyuan Chen, Yinbo Chen, Jingwen Liu, Xingqian Xu, Vidit Goel, Zhangyang Wang, Humphrey Shi, and Xiaolong Wang. VideoINR: Learning video implicit neural representation for continuous space-time super-resolution. In *CVPR*, 2022.

[159] Hao Chen, A Gwilliam Matthew, Bo He, Ser-Nam Lim, and Abhinav Shrivastava. CNeRV: Content-adaptive neural representation for visual data. In *BMVC*, 2022.

[160] Anonymous. HNeRV: A hybrid neural representation for videos. In *ICLR submission*, 2023.

[161] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. An overview of core coding tools in the av1 video codec. In *2018 Picture Coding Symposium (PCS)*, pages 41–45. IEEE, 2018.

[162] Benjamin Bross, Ye-Kui Wang, Yan Ye, Shan Liu, Jianle Chen, Gary J Sullivan, and Jens-Rainer Ohm. Overview of the versatile video coding (vvc) standard and its applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3736–3764, 2021.

[163] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, et al. The kinetics human action video dataset. *arXiv preprint arXiv:1705.06950*, 2017.

[164] Raghav Goyal, Samira Ebrahimi Kahou, Vincent Michalski, Joanna Materzynska, Susanne Westphal, Heuna Kim, Valentin Haenel, Ingo Fründ, Peter Yianilos, Moritz Mueller-Freitag, Florian Hoppe, Christian Thurau, Ingo Bax, and Roland Memisevic. The

"something something" video database for learning and evaluating visual common sense. In *ICCV*, 2017.

[165] Khurram Soomro, Amir Roshan Za4mir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[166] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *CVPR*, June 2019.

[167] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *NeurIPS*, 2019.

[168] Matthew Tancik, Ben Mildenhall, Terrance Wang, Divi Schmidt, Pratul P. Srinivasan, Jonathan T. Barron, and Ren Ng. Learned initializations for optimizing coordinate-based neural representations. In *CVPR*, 2021.

[169] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *ICCV*, 2021.

[170] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *ECCV*. Springer, 2020.

[171] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. *arXiv preprint arXiv:2111.06377*, 2021.

[172] David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. In *ICLR*, 2017.

[173] Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *CVPR*, pages 11030–11039, 2020.

[174] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. *NeurIPS*, 32, 2019.

[175] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. *NeurIPS*, 29:3630–3638, 2016.

[176] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. Learning to compare: Relation network for few-shot learning. In *Proceedings of the CVPR*, pages 1199–1208, 2018.

[177] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *ICLR*, 2018.

[178] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, pages 1126–1135. PMLR, 2017.

[179] Chrisantha Fernando, Jakub Sygnowski, Simon Osindero, Jane Wang, Tom Schaul, Denis Teplyashin, Pablo Sprechmann, Alexander Pritzel, and Andrei Rusu. Meta-learning by the baldwin effect. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1313–1320, 2018.

[180] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[181] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[182] Vincent Sitzmann, Eric R. Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein. Metasdf: Meta-learning signed distance functions. In *NeurIPS*, 2020.