Swift: Adaptive Video Streaming with Layered Neural Codecs

Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras, *Stony Brook University*

https://www.usenix.org/conference/nsdi22/presentation/dasari

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Swift: Adaptive Video Streaming with Layered Neural Codecs

Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, Dimitris Samaras
Stony Brook University

Abstract

Layered video coding compresses video segments into layers (additional code bits). Decoding with each additional layer improves video quality incrementally. This approach has potential for very fine-grained rate adaptation. However, layered coding has not seen much success in practice because of its cross-layer compression overheads and decoding latencies. We take a fresh new approach to layered video coding by exploiting recent advances in video coding using deep learning techniques. We develop `Swift`, an adaptive video streaming system that includes i) a layered encoder that learns to encode a video frame into layered codes by purely encoding *residuals* from previous layers without introducing any cross-layer compression overheads, ii) a decoder that can *fuse* together a subset of these codes (based on availability) and decode them all in one go, and, iii) an adaptive bit rate (ABR) protocol that synergistically adapts video quality based on available network and client-side compute capacity. `Swift` can be integrated easily in the current streaming ecosystem without any change to network protocols and applications by simply replacing the current codecs with the proposed layered neural video codec when appropriate GPU or similar accelerator functionality is available on the client side. Extensive evaluations reveal `Swift`'s multi-dimensional benefits over prior video streaming systems.

1 Introduction

Internet video delivery often encounters highly variable and unpredictable network conditions. Despite various advances made, delivering the highest possible video quality continues to be a challenging problem due to this uncertainty. The problem is more acute in wireless networks as the channel conditions and mobility adds to the uncertainty [39, 46]. Interestingly, the next generation wireless networks may even make the problem more challenging (e.g., 60GHz/mmWave [10, 11, 38]).

To counter the challenges posed by such varying network capacity, current video delivery solutions predominantly practice adaptive streaming (e.g., DASH [50]), where a source video is split into segments that are encoded at the server into multiple bitrates providing different video qualities, and a client runs an adaptive bitrate (ABR) algorithm to dynamically select the highest quality that fits within the estimated network capacity for the next segment to be downloaded.

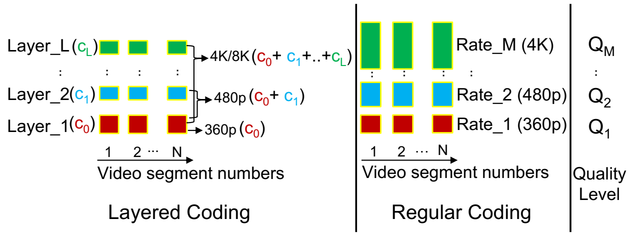


Figure 1: *Layered* vs. *Regular* coding methods. In Regular coding the video segments are coded independently at different qualities. In Layered coding a given quality can be reconstructed by combining codes for multiple layers thus facilitating incremental upgrades or downgrades.

Need for layered coding. Most of the current commercial ABR algorithms adopt a monolithic encoding practice (e.g., H.265 [53]), where the same video segment is encoded 'independently' for each quality level. The decision on fetching a segment at a certain quality is considered *final* once the ABR algorithm makes a determination based on estimating the network capacity. However, these estimations are far from accurate, resulting in either underutilizing or overshooting the network capacity. For example, the ABR algorithm may fetch at a *low quality* by underestimating the network capacity, or it may fetch at a high quality causing *video stalls* by overestimating. Consequently, even the optimal ABR algorithms fail to provide a good quality of experience (QoE), as such rigid methods that do not fit the need of the streaming conditions.

An alternate technique, called *layered coding*, has been long studied [12, 14, 36, 47, 67] that can avoid the above streaming issues. The key idea here is that, instead of *independently* encoding the segment in different qualities, the segment is now encoded into *layers*; the base layer provides a certain video quality, and additional layers improve the video quality when applied over the base layer. See Figure 1. This means that, if the network throughput improves, one can fetch additional layers to improve video quality at a much lower cost compared to a *regular* codec.[1] We use the term *regular* coding to indicate the current practice of independent encoding in multiple qualities (current standards such as H.265/HEVC [53]).

Challenges with layered coding. Layered coding, however, faces two nontrivial challenges: *compression overhead*, and *coding latency*. The compression overhead mainly comes from not having the inter-layer frame prediction to avoid reconstruction drift in quality [29, 42, 61, 67]. On the other hand, the decoding latency is a function of the number of layers as

[1] We use terms coding or codec for encoding and decoding together. Also we use the terms encoding/compression, decoding/decompression interchangeably.

each layer needs to be decoded separately. Notwithstanding these issues, some studies have indeed applied layered coding in streaming and have shown slightly better QoE compared to the regular coding methods, benefiting from its ability to do dynamic quality upgrades [31]. However, they do not address either the overhead or the latency issues directly. Industry streaming solutions continue to adopt the regular codecs, shipping these codecs in hardware to avoid computational challenges, making it harder to adopt new innovations.

Neural video codecs. A learning approach to video coding has shown tremendous improvement in compression efficiency in just a few years [43, 60, 65]. Figure 2 shows bits-per-pixel vs PSNR plots[2] for several generations of codecs of two types – *neural codecs* that use deep learning and traditional *algorithmic* codecs that use the popular H.26x standards. It took algorithmic codecs 18 years to make the same progress that neural codecs achieved in the last 4 years! One reason for this rapid development is that neural codecs can run in software that can be integrated as part of the application, support agile codec development and provide royalty-free codecs. Further, they run on data parallel platforms such as GPUs that are increasingly available and powerful.

There are several insights in using neural codecs for video coding – **1)** unlike the traditional *layered* coding methods where it is nontrivial to handcraft each layer[3] to have unique *information*, a neural network's loss function can be optimized to encode a video frame into unique layered codes by purely encoding *residuals* from previous layers without introducing a reconstruction drift; **2)** a neural network can be trained to accept a subset of the layered codes and decode all of them in a single-shot, which again was traditionally difficult to do with a handcrafted algorithm due to nonlinear relationships among the codes. Additionally, **3)** neural codecs enable software-driven coding. We note here that GPUs or similar accelerators for neural network computation are critical for success with neural codecs. Fortunately, they are increasingly common in modern devices.

Swift. Based on the above insights, we present `Swift`, a novel video streaming system using *layered* coding built on the principles of neural video codecs [32, 60, 65].[4] We show that learning can address the challenges of layered coding mentioned earlier – there is no additional compression overhead with further layering and the decoding latency is independent of the number of layers. `Swift` consists of three design components: i) server-side encoder plus decoder, ii) client-side decoder, and iii) ABR protocol adapted to layered coding and varying compute capacity (in addition to varying network capacity).

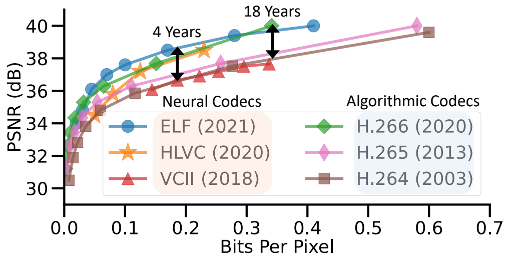


Figure 2: Evolution of neural and algorithmic video codecs showing compression efficiency plots across generations.

We evaluate `Swift` with diverse video content and FCC-released real-world network traces [8]. We compare `Swift` with state-the-art streaming algorithms that combine either regular coding [35, 51, 52] or layered coding [31] with state-of-the-art ABR algorithms. In terms of QoE, `Swift` outperforms the next-best streaming alternative by 45%. It does so using 16% less bandwidth and has a lower reaction time to changing network conditions. In terms of the neural codec, `Swift`'s layered coding outperforms the state-of-the-art layered codec (SHVC [12]) by 58% in terms of compression ratio, and by ×4 (for six layers) in terms of decoding latency. In summary, our contributions are the following:

- We show how deep learning-based coding can make layered coding both practical and high-performing, while addressing existing challenges that stymied the interest in layered coding.
- We design and build `Swift` to demonstrate a practical layered coding based video streaming system. `Swift` is an embodiment of deep learning-based encoding and decoding methods along with a purpose-built ABR protocol.
- We comprehensively evaluate and showcase the multi-dimensional benefits of `Swift` in terms of QoE, bandwidth usage, reaction times and compression efficiency.

2 Motivation

2.1 Limitations of Today's Video Streaming Due to Regular Coding

Today's video providers predominantly use source rate adaptation (e.g., MPEG-DASH [50]) where video segments are encoded at different qualities on the server and an adaptive bitrate (ABR) algorithm chooses the best quality segment to be downloaded based on the network capacity.

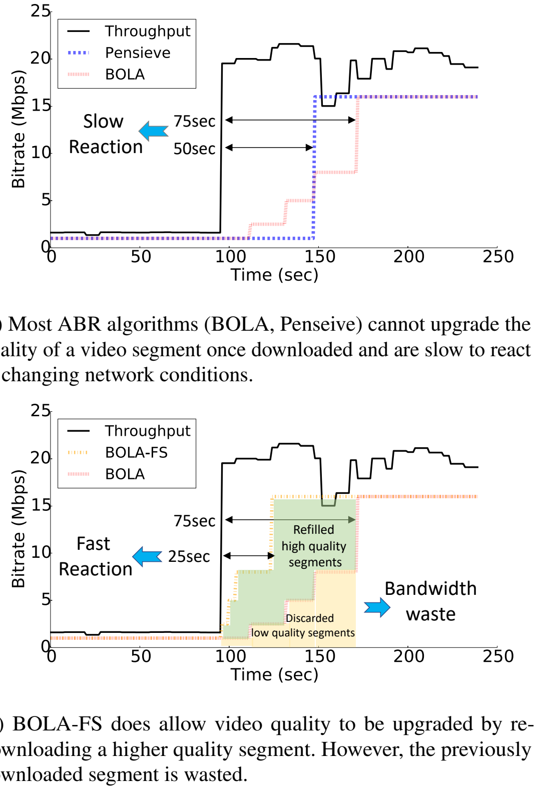
The streaming solutions that are widely deployed, use *regular*, standards-driven, algorithmic coding methods such as H.265/HEVC [53] or VP9 [4] for encoding video segments. These coding methods do not allow segments to be upgraded or downgraded based on network conditions.

Figure 3 illustrates this problem using an example experiment (more details about methodology are described in §6.1). The figure shows the quality of segments that are fetched by different state-of-the-art ABR algorithms that use regular

[2] Bits-per-pixel captures compression efficiency and PSNR (peak signal-to-noise ratio) captures image quality. Both metrics together capture codec performance.

[3] Throughout the paper, the term 'layer' refers to compressed code layers, not neural network layers.

[4] The source code of `Swift` is available at the following site: https://github.com/VideoForage/swift.



(a) Most ABR algorithms (BOLA, Penseive) cannot upgrade the quality of a video segment once downloaded and are slow to react to changing network conditions.

(b) BOLA-FS does allow video quality to be upgraded by re-downloading a higher quality segment. However, the previously downloaded segment is wasted.

Figure 3: Limitations of today's ABR algorithms because of regular coding: either slower reaction to network conditions or bandwidth wastage to achieve faster reaction time to highest quality. The reaction latency includes time to notice throughput increase as well as playing the buffered segments, and hence segment duration (5 sec here) plays a role. Penseive aggressively controls video quality fluctuations to compensate for incorrect bandwidth prediction, and hence the sudden jump in quality compared to BOLA.

coding. During the experiment, the throughput improves drastically at the 100 second mark. Two state-of-the-art streaming algorithms, Pensieve [35] and BOLA [52], cannot upgrade the quality of a segment once the segment has been downloaded. This causes a slow reaction to adjust to the improved throughput. In Figure 3(b) however, BOLA-FS [51], a version of BOLA, does allow the higher quality segment to be re-downloaded when the network conditions improve. However, the previously downloaded lower quality segment is discarded, resulting in wasted bandwidth.

2.2 Layered Coding

A more suitable coding method to address the above issues is layered coding, where a video segment is encoded into a *base layer* (providing the lowest playback quality level) and multiple *enhancement layers* as shown in Figure 1. Clearly, layered coding gives much finer control on rate adaptation compared to regular coding. For example, multiple enhancement layers for the same segment can be fetched incrementally as the estimate of the network capacity improves closer to the playback time, which is not possible in case of regular coding.

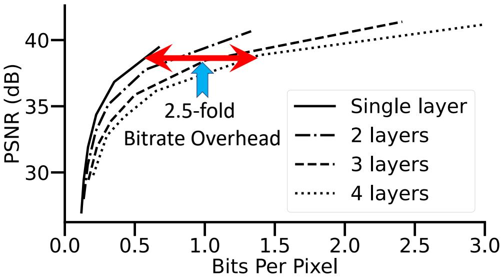


Figure 4: Compression efficiency of traditional layered coding. We use H.265 [53] and its layered extension SHVC [12] to encode the videos (described in §6.1). The single layer bitrate curve is same for both, and the additional layers are for SHVC. As shown, SHVC requires 2.5× more bits for 4 layers of SHVC compared to a single layer for the same quality.

2.3 Challenges of Adopting Traditional Layered Coding in Video Streaming

Layered coding has typically been developed and implemented as an extension to a *regular* coding technique. Published standards demonstrate this dependency: SHVC [12] has been developed as an extension of H.265 [53], similarly, older SVC [47] as an extension for H.264 [57]. Developing layered coding as an extension on top of a *regular* coding introduces multiple challenges in real-life deployments:

1) *Cross-layer compression overhead:* The key to large compression benefits in current generation video coding standards (e.g., ≈ 2000× compression ratio for H.265 [53]) is *inter-frame prediction* – the consecutive frames are similar and so it is efficient to simply encode the difference between consecutive frames. However, using the inter-layer frame prediction across enhancement layers of the current frame with respect to the previous frame makes video quality drift during decoding [29, 42, 61, 67]. To minimize or avoid the drift, most of the *layered* coding methods do not use inter-frame prediction across layers and thus lose out on its compression benefits [11, 17, 31]. In effect, to achieve the same quality, layered coding (e.g., SHVC) requires significantly more bits compared to its *regular* counterpart (e.g., H.265). In our study, we find that a 4-layer SHVC coding method needs 2.5× bits per pixel compared to its *regular* coding counterpart, H.265 (see Figure 4).

2) *High encoding and decoding latency:* The computational complexity of these algorithmic codecs mainly comes from the motion estimation process during inter-frame prediction [53, 57]. During the motion estimation, it is useful - for each pixel - to encode its motion vector, i.e., where its relative location was in the previous frame. The motion vectors are computed for each frame by dividing the frame into thousands of blocks of pixels and searching a similar block in the previous frames. In general, the codecs use a set of previous

frames to search blocks in each frame making it computationally expensive. The process becomes even more complex in case of layered coding because each layer has to be decoded one after the other because of the dependency of a layer on the previous one (to exploit the content redundancy) [11,27,30]. This serial process of layered coding makes the latency to be a function of number of layers, and therefore the latency increases progressively as we increase the number of layers.

Figure 5 shows per-frame decoding latency of the state-of-the-art layered coding (i.e., SHVC) of a 1-min video on a desktop with configuration described in §6.1. As shown, it takes more than 100ms to decode each frame for 5 layers, an order of magnitude increase in coding latency compared to its regular counterpart H.265 (an x265 [7] implementation). Despite several optimizations in the past, such range of latencies makes it infeasible to realize real-time decoding on heterogeneous platforms. Recent studies (e.g., Jigsaw [11]) tackle this challenge by proposing a lightweight *layered* coding method (using GPU implementation), but the latency is still a function of number of layers.

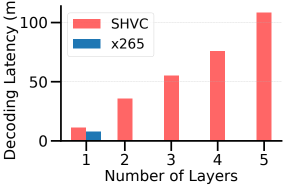


Figure 5: Latency challenges of traditional layered coding. The decoder is run on a high-end Desktop (as described in §5) using a single-threaded implementation of SHVC [3].

Because of these challenges, traditional layered coding is not used in practice today. In this work, rather than approaching this problem with yet another extension, we seek to explore layered coding via a clean-slate, learning-based approach with a goal towards efficient layered compression by embracing the opportunities of new hardware capabilities (e.g., GPUs and other data parallel accelerators).

2.4 Layered Coding using Neural Codecs

Video compression has recently experienced a paradigm shift in the computer vision community due to new advances in deep learning [32, 43, 60, 65]. The compression/decompression here is achieved using neural networks that we refer to as neural video codecs.

The basic idea is the use of an AutoEncoder (AE), a neural network architecture used to learn efficient encodings that has long been used for dimentionality reduction purposes [20]. The AE consists of an encoder and a decoder. The encoder converts an input video to a *code* vector that has a lower dimension than the input size, and the decoder reconstructs (perhaps with a small error) the original input video from the low-dimension *code* vector. The neural network weight parameters (W_i for encoder and W_i' for decoder) are trained by minimizing the *reconstruction error*, that is, minimizing the difference between the input and the output of the decoder. The smaller the code, the larger the compression factor but higher the reconstruction error.

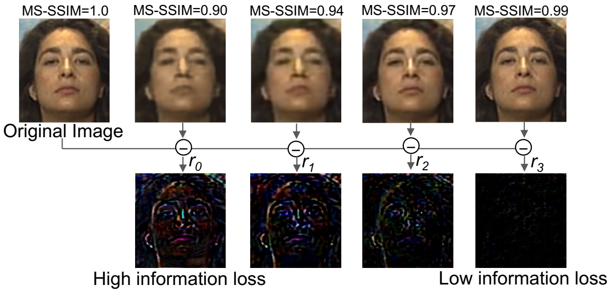


Figure 6: Illustrating the residuals ($r_0, \ldots, r_3$) from an original frame to a series of compressed-then-decoded frames. MS-SSIM [56] is a perceptual measure of image quality. A highly compressed frame (lowest MS-SSIM) has more residual information (r_0).

Our insight in using Autonencoders is that a their loss function can be optimized to encode a video frame into unique layered codes by purely encoding residuals from previous layers, unlike the traditional layered coding where it is nontrivial to handcraft each layer to have unique information.

3 Swift

3.1 Overview

Autoencoders are already shown to provide similar or better performance relative to traditional codecs [32,43,60]. Recent work such as Elf-vc [43] is also able to use Autoencoders to provide flexible-rate video coding to fit a target network capacity or achieve a target compression quality. However, current work does not provide a way to encode in the video in incrementally decodable layers. To do this, we make use of *residuals* to form layered codes. A residual is the difference between the input to an encoder and output of the corresponding decoder. Residuals has been used in the past for tasks such as recognition and compression to improve the application's efficiency (e.g., classification accuracy or compression efficiency) [19,55,60].

Swift uses residuals for designing layered codecs for video streaming. The idea is to employ a chain of Autoencoders of identical structure. Each Autoencoder in the chain encodes the residual from the previous layer, with the very first Autoencoder in the chain (implementing the base layer) encoding the input video frames. Figure 6 shows an example, where the residuals are shown from an original frame to a series of progressively compressed-then-decoded frames. The first decoded frame (marked with $MS\_SSIM = 0.9$) has a relatively high loss from the original frame. As a result, the residual r_0 has more information. When this residual information is used for the next layer's compression, the resulting decoded frame is closer to the original, and in-turn the residual has less information, and so on.

The above 'iterative' chaining implicitly represents a layered encoding mechanism. Each iteration (i.e., layer) produces a compressed version of the original video that we call 'code.' These codes encode incremental information such that with more such codes decoded, the reconstruction becomes progressively closer to the original. `Swift` essentially uses this mechanism of residuals to create the layered codes. Such iterative minimization of residual also acts as an implicit regularization to guide the reconstruction (at a given bandwidth), instead of closely-following classical compression methods as in Elf-vc [43].

Figure 7 shows the Autoencoder architecture (more details in §3.2) on the server side. The architecture jointly learns *both* the encoder and decoder in each layer. As before, the Autoencoder's weight parameters are trained to minimize the reconstruction error between the input and output of the decoder. In this process, the encoder generates a compact code in each layer which is a compressed version of the input video frames. These codes are transmitted to the client, where they can be decoded for progressively better reconstructions.

The decoder learnt at the server is then optimized further (§3.3) to be used at the client side. The client decoder initially reconstructs the base layer from the first code. Then, if more layers/codes are downloaded from the server, the decoder reconstructs the residuals from the second layer onward, and combines with the previous reconstruction(s) to generate the output video frame.

Overall, `Swift` has three main components:

1. A learning-based layered encoder-decoder pair in a single neural network to create residual-based layered codes on the server-side (§3.2).
2. A separate learning-based decoder on the client side. This decoder can decode any combination of layered codes in a single-shot for real-time decoding (§3.3).
3. Extension of an ABR algorithm that can integrate the codec into a complete end-to-end system (§4).

3.2 Layered Neural Encoder

We first describe how the encoder and the decoder are trained at the server side. Assume, I^t is the image or video frame at time t, for $t \in \{0, 1, ...\}$. The encoder ($\mathcal{E}$) takes each of these frames as input and generates a compact *code vector* (c) for each frame, i.e., $c^t = \mathcal{E}(I^t)$. This code for each frame is constructed by exploiting the redundancy across multiple previous frames in the video. Therefore, the encoder takes a set of previous frames as reference in order to encode each frame. The decoder ($\mathcal{D}$) reconstructs the frame $\hat{I}^t$ given c^t, i.e., $\hat{I}^t = \mathcal{D}(c^t)$. The optimization problem here is to train $\mathcal{E}$ and $\mathcal{D}$ pairs so as to minimize the difference between $\hat{I}^t$ and I^t. Since we add our layered coding as a generic extension to any neural codec without changing its internal logic, $\mathcal{E}$ and $\mathcal{D}$ can be assumed as blackboxes. An example of a neural codec is presented in Appendix A.

Figure 7 shows the design of our layered encoder-decoder network on the server-side. Here, each iteration (or layer) encodes a residual r_i into a code c_i, where residual r_i is the difference between the encoder input and decoder output in the previous layers. For the very first iteration, the encoder directly encodes the the original video frame. Representing this mathematically: $c_i = \mathcal{E}(r_i)$ and $r_i = r_{i-1} - \hat{r}_{i-1}$ with $\hat{r}_i = \mathcal{D}(c_i)$, for $i = 0, \ldots, L$, with the exception that for $i = 0$ (base layer), $r_0 = I$.

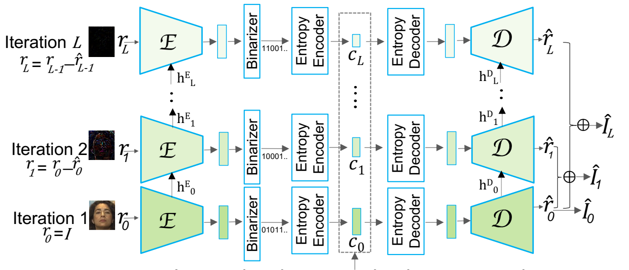


Figure 7: Deep learning based layered coding: a) iterative encoding ($\mathcal{E}$) and decoding ($\mathcal{D}$): in each iteration, $\mathcal{E}$ encodes a residual into a code (c_i) and the decoded output (from $\mathcal{D}$) is used to generate the residual for the next iteration.

At each iteration, the decoder can enhance the quality of the video frame with a plain arithmetic sum of the outputs of all previous iterations along with the base layer output. The key here is that both $\mathcal{E}$ & $\mathcal{D}$ have separate hidden states (h^E* and h^D*) that get updated iteratively, sharing information between iterations. In fact, this subset of weights shared across iterations, allows better reconstruction of residuals. The entropy (i.e., the information) is very high in the initial layers, but progressively decreases due to the presence of the hidden connections and thus the code size becomes progressively smaller. The training objective for these iterative encoder-decoder pairs is to minimize the `L1` reconstruction loss for the residuals:

$$\mathcal{L}_{\text{rec}} = \frac{1}{L} \sum_{i=0}^{L-1} \|\mathcal{D}(c_i) - r_i\|_1$$

All Autoencoders $\mathcal{E}$ & $\mathcal{D}$ in the chain share the same network and thus have identical input and output sizes. They produce the same code sizes for all iterations. `Swift` relies on a separate entropy encoding stage (Figure 7) to create the residual codes that allocate proportional number of bits to match the entropy in each iteration. The fixed length code vector from the output of the encoder $\mathcal{E}$ is binarized and passed through a traditional entropy encoder similar to CABAC [54].

Note that the learned codec can work with a variety of input video resolutions, and hence we do not need to train a separate model for each video resolution. This is mainly because the Autoencoder here takes one or more video frames as input and extracts the features through convolutions (e.g., Conv2D [41]). Each convolutional kernel (with a fixed size of $k \times k$ pixels that is much smaller than the input resolution) is applied in a sliding window fashion on $k \times k$ blocks of pixels to reduce the dimensions and form the Autoencoder's compact code vector. In our codec we use 4 downsampling convolution blocks to reduce the dimensions. This makes any input resolution ($w \times h$) to be downsampled to $(w/16) \times (h/16)$ resolution times the Autoencoder's bottleneck bits (b) after the encoding stage (see Appendix A). Therefore during the testing, the encoder's output for a 352×288 resolution would be $22 \times 18 \times b$, while it is $80 \times 45 \times b$ for 1280 × 720 resolution. Similarly the codec scales with other resolutions during testing.

3.3 Layered Neural Decoder

The above iterative coding design already includes the decoder (Figure 7) that can reconstruct the video from the layered codes. In principle, the client can use the same decoder already designed and learned on the server-side. However, the iterative method incurs decoding latency proportional to the number of iterations. This is because the residual codes are created separately in each iteration and the decoder cannot decode a code (c_i) unless the previous iteration of encoder encodes c_{i-1} and the corresponding decoder decodes it to form the residual r_i.

This latency is acceptable for video servers/CDNs that encode the videos offline and store them ready for on-demand streaming, but clients need to decode the video in real-time (≈30 fps). To address this, we develop a separate design of *single-shot* decoder to be used at the clients, that can take any combination of the codes as input and decode the corresponding frames in one shot. See Figure 8.

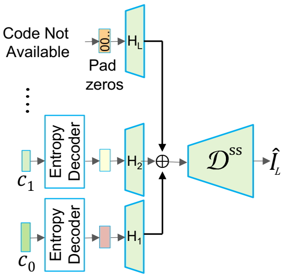


Figure 8: Single-shot decoder ($\mathcal{D}^{\text{ss}}$): a variable sized decoder that takes a subset of the codes to reconstruct a video frame in one go.

The codes available at the client are fused and padded with zeros up to a predetermined code length (corresponding to L levels) to account for unavailable codes. They are then fed into a 'multi-headed' decoder (H) as shown in the Figure 8. In each head, the padded version of each code c_i is separately processed through individual neural networks prior to combining them into a common network. When higher layers are unavailable, the corresponding heads will have no effect on reconstruction, and when available, generate a desired residual mapping. Essentially, these multiple heads are lightweight and the common network (after combining the residual codes after multiple-heads) follows the same decoder architecture $\mathcal{D}$ from §3.2, but within one model. The heads or the common decoder do not share any parameters, in contrast to the iterative decoder which shares hidden states across iterations. Note that for preparing residual codes on the server-side, we still need the iterative $\mathcal{E}$-$\mathcal{D}$ architecture as described in §3.2. To distinguish from the server-side decoder ($\mathcal{D}$), we denote this client-side single-shot decoder as $\mathcal{D}^{\text{ss}}$.

We train $\mathcal{D}^{\text{ss}}$ by extending the objective function used at the server-side. Specifically, in addition to the loss function at the server-side decoder ($\mathcal{D}$) which reconstructs a residual, we add a loss function that corresponds to the actual image reconstruction at client-side decoder ($\mathcal{D}^{\text{ss}}$) using the available code layers. Using `L1` loss function, both the objectives are as shown below.

$$\mathcal{L}_{\text{rec}} = \frac{1}{L}\sum_{i=0}^{L-1}\Big[\ \underbrace{\|\mathcal{D}(c_i) - r_i\|_1}_{\text{residual quality loss}} + \underbrace{\Big\|\mathcal{D}^{\text{ss}}(\oplus_{k=0}^{i} c_k) - I\Big\|_1}_{\text{image quality loss}}\Big]$$

Here, the server-side decoder ($\mathcal{D}$) reconstructs the residual image r_i at each iteration based on the code c_i, while the client-side $\mathcal{D}^{\text{ss}}$ reconstructs the original image I based on the subset of the codes available at the corresponding iteration, i.e., $c_0 \ldots c_i$. This function allows us to train the encoder, and both the decoders (server and clients-side) in a single training loop. During the training, all three models $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{D}^{\text{ss}}$ are jointly optimized by summing up the loss computed for $\mathcal{E}$ and $\mathcal{D}$ in §3.2 and the direct loss computed for $\mathcal{D}^{\text{ss}}$ that corresponds to original image reconstruction. This joint training with a more complex objective (i.e., multiple loss functions) does affect the performance of server-side decoder. The iterative decoder from §3.2 has simpler objective than $\mathcal{D}^{\text{ss}}$, and hence its performance is better when trained independently (in which case only the second term in the loss function is sufficient for $\mathcal{D}^{\text{ss}}$) compared to trained jointly. In our experiments we observe very little drop in quality on average with the joint training – that would be almost imperceptible to users. Moreover, training each of these models can also incur additional computation costs on servers.

4 Streaming with Layered Neural Codecs

`Swift`'s layered neural codes introduces two challenges to end-to-end streaming. The first challenge arises because `Swift`'s decoder at the client is expected to be run on the GPU or other similar data-parallel accelerators and run in software, instead of dedicated fixed hardware decoders as is the norm for regular codecs. Even though the software codecs have advantages in terms of on-demand codec upgrades and agile development, it raises the possibility of resource contention with other applications. Since GPU resource availability can vary [33, 45], the client needs to be able to adapt to the available resources.

The second challenge is in bitrate selection. Video streaming protocols encode each video segment into different qualities and uses ABR to select the next best quality video segment to stream. However, the ABR algorithm in `Swift` has a more complex choice—should one fetch the next segment at the highest possible quality or upgrade the current segment by

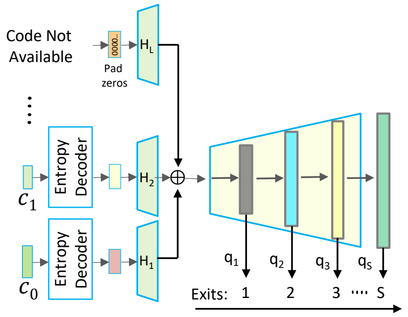


Figure 9: Scalable decoding using multiple exit heads to adapt to dynamic compute capacity. Each exit provides a different trade-off between compute capacity (and in-turn time required to decode) and quality of the video segment.

fetching additional layers? This question is made even more complex because the end-to-end streaming performance is affected both by the network and the compute variability at the client (see above). Traditional bitrate adaptation techniques are designed to only adapt to network variability.

In this section, we describe the design of a scalable decoder and neural-adapted ABR algorithm to tackle the challenges.

4.1 Scaling the Decoder based on Compute Capacity

The decoder architecture (shown in Figure 8) uses network with a certain depth. [5] As is common in Autoencoders, the more the depth, the better is the decoding accuracy, but lower the depth, lower is the compute requirement.

Swift exploits this trade-off by designing multiple, lightweight, output heads at different depths of the network. The decoder then operates at different design points in the accuracy vs compute requirement trade-off by exiting at different depths depending on the GPU capacity. We define GPU capacity as the percentage of time over the past sample period (1 sec in our case) during which one or more cores was executing on the GPU. For example, a 100% GPU utilization means all of the GPU cores are busy with other applications in the last sample period. To this end, we introduce a number of early-exit heads (hd_j, where $j = 1..S$) in the D^{ss} decoder that are corresponding to different output video qualities. See Figure 9. For example, if there are 5 exits in the network, then each exit depth outputs ×16, ×8, ×4 and ×2 smaller in resolution than the original image, with the final exit as the original reconstruction. Here, the very first early exit outputs a low quality while the final exit outputs higher quality. There has been similar early exit networks used in the literature for various tasks [28, 37].

In Swift, we define a loss function at each exit and optimize the training objective of the decoder at all exits. The decoder is trained by introducing additional L1 reconstruction losses, so that the outputs of each of the early-exit heads minimizes the difference with the original input (I). The objective function is as shown below:

$$\mathcal{L}_{\text{rec}} = \frac{1}{L}\sum_{i=0}^{L-1}\Big[\underbrace{\|\mathcal{D}(c_i) - r_i\|_1}_{\text{residual quality loss}} + \underbrace{\frac{1}{S+1}\sum_{j=0}^{S}\Big\|\mathcal{D}^{\text{ss}}_{hd_j}(\oplus_{k=0}^{i} c_k) - I\Big\|_1\Big]}_{\text{image quality loss}}$$

Here, S is the number of exits. Given a combination of these multiple exits and downloaded codes, the decoder outputs the quality corresponding to both dimensions. Figure 10 shows the heatmap of average video quality when the client decodes different number layered codes while exiting at different depths, for UVG videos described in §6.1. For example, if the client only fetches the base layer (shown as 1 in the figure) and exits at depth 1, the quality of the decoded segment is 28dB. However, if the client decodes 4 layers and decodes to completion (exit at depth 4), the quality of the decoded segment increases to 32 dB. Note here that the number of layered codes that can be fetched depends on the network capacity while the exit depth depends on the compute capacity.

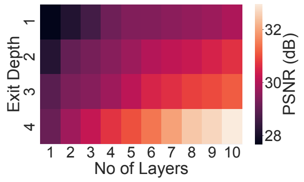


Figure 10: Quality matrix as a function of exit depth and the number of layered codes.

At runtime, Swift decoder decides on when to exit depending on the GPU capacity. The GPU capacity determines the latency in decoding a segment by computing until different depths. Given a GPU capacity, Swift chooses the maximum depth such that the segment will be decoded without incurring any stalls because the buffer is empty. In §5 we discuss how the client chooses the decoder and obtains the relationship between decode latency and exit depth.

4.2 Adapting ABR for Layered Neural Codecs

A traditional ABR algorithm [31, 35, 52] using regular codecs takes as input the available throughput, buffer levels, and details about the future video segments. The algorithm outputs the quality of the video segment to download next. Swift needs to adapt existing ABR algorithms to work with layered neural codecs. We describe this adaptation in terms of changes to ABR's output, input, and the objective function. We then discuss how we instantiate the ABR algorithm with these changes. See Figure 11 for an overview. In the discussion below, we assume that the ABR algorithm is run at the server; but it can be adapted to run at the client.

Output: The crucial change to Swift's ABR is that unlike traditional ABR, our algorithm can make one of two choices: download the base layer (i.e., the code with lowest quality) of a future segment or, download an enhancement layer of one of the buffered video segments (that is not played yet) to

[5] here the depth refers to the number of layers in the neural network.

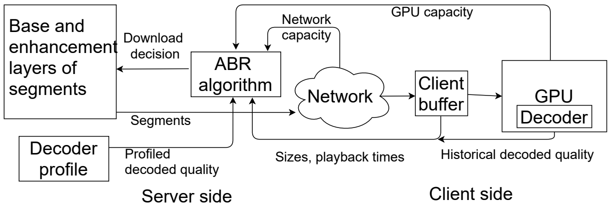


Figure 11: Swift's video streaming pipeline.

enhance quality. It makes this determination based on a (new) set of inputs and the objective function. This change to the output is needed for streaming algorithms that use layered code, including Grad [31], the state-of-the-art streaming that uses layered coding. We compare the performance of Swift to Grad in §6.

Input: Swift introduces two additional inputs to the ABR to take into account the compute variability while decoding. ABR takes as input a matrix that maps the quality of the decoded segment against compute capacity needed for this decoding (similar to Figure 10). This quality matrix is generated offline at the server for each video segment for different compute capacities i.e., for all combinations of layers and exit depths. The next input is the current GPU capacity at the client. Since we run the ABR algorithm at the server, this information is sent by the client in the segment request packet. If the ABR algorithm is run on the client, the GPU capacity is readily available there. In this case the quality matrix at the server can be sent as a part of the manifest file.

The second change is with respect to segments downloaded/buffered at the client but not yet played. Swift's ABR needs information about these segments to makes it choice. Specifically, this includes i) the size of the remaining layers for current/buffered segments, ii) the playback time of all segments in the buffer, to determine if the segment can be enhanced before playback. In addition to that, we input the history of the decoded quality of the last k displayed segments to reduce variation in quality (this is often called smoothness and is an important metric for improving QoE).

Objective function: The video Quality of Experience (QoE) is typically captured using three metrics: i) playback quality (Q), i.e., the quality of the downloaded segments and ii) rebuffering ratio (R) that measures how often the video stalls because the buffer is empty, iii) smoothness (S) that measures fluctuation in quality. Formally, $QoE = Q - \alpha R - \beta S$, where α and β are the coefficients to control the penalties of rebuffering and smoothness [31, 35, 52]. Since Swift's decoding performance is variable, it may not always provide the best quality that is possible from the downloaded segments. So instead, the objective function in Swift takes into account the quality of the *decoded segment* (Q_d) instead of the downloaded segment. Similarly, we compute the smoothness from the quality of decoded segments (S_d) instead of downloaded qualities.

5 Implementation and System Setup

Swift's end-to-end implementation includes its layered neural codecs and the ABR protocol presented in §3.1 and §4.

5.1 Layered Codec Implementation

We implement our layered coding on top of VCII [60]. VCII is a state-of-the-art neural codec that is learnt over an Autoencoder network. VCII achieves compression efficiency close to state-of-the-art regular (non-neural) video codecs. Similar to regular codecs, VCII does not produce layered codes. Instead, we implement the layered encoder over VCII as described in §3.2. For the decoder at the client, similarly, we modify the loss function to incorporate the single-shot decoding and multi-exit decoder capability. Since our layered technique can be applied as a general extension to any codec, we do not need to change the internal codec logic.

After designing the new encoder/decoder over VCII, the encoder and decoder is retrained for 100K iterations on an Nvidia RTX 2070 GPU. We use ADAM optimizer with a batch size of 16. During the training, we use multiple randomly cropped 64×64 image patches from the original images for generalization purposes. The model is trained to produce up to 10 layered codes.

For training, we use the Kinetics dataset [13]. It has 37K videos. We train on 27K, test on 10K videos. For more rigorous testing, we also test on completely different datasets (more details about testing in §6). The training takes around 6 hours. Since the training will be done offline and only once, the training time is reasonable.

5.2 Streaming Implementation

We implement our adapted ABR by modifying Pensieve [35]. For training the ABR model, we use $k = 10$ throughput and compute capacity measurements passing through a 1D-CNN with 128 filters, the quality matrix passed through a 2D-CNN, and aggregated with other inputs described in §4.2. The learning rate and discount factor for the network are 0.001 and 0.99 respectively. We run the ABR algorithm every time a segment or its layer (s) is downloaded. We train the model using simulated network and compute traces, similar to that used in Pensieve [35].

We run the ABR algorithm at the server. Similar to other video streaming servers, the Swift server processes the video segments and encodes them. The server also performs fine-grained profiling of the decoder for two bits of information. First, it creates a matrix of quality levels for different depths. Second, it creates a mapping between GPU capacities and time taken to finish decoding until different depths. Both of this are used as input to the adapted ABR algorithm.

6 Swift Evaluation

We evaluate Swift both in terms of end-to-end streaming and coding performance. We compare Swift with a suite of streaming algorithms and its layered coding with commonly used regular codec (HEVC) [68] and layered codec (SHVC) [3]. Our evaluation shows that:

- Overall QoE with Swift improves by 45% at the median compared to the second best streaming performance.
- Swift uses 16% and 22% less bandwidth compared to next best streaming algorithms that use regular and layered codecs, respectively.
- Swift's neural layered codec improves compression efficiency by 58% over state-of-the-art layered codec SHVC.

6.1 Evaluation Methodology

In this section, we describe the methodology to evaluate Swift's end-to-end streaming performance.

Experimental set up. We conduct all experiments on a desktop with Nvidia 2070 RTX GPU as the client. Our evaluation uses FullHD videos from UVG [5] dataset consisting of 7 videos for streaming.[6] Each video is of 5 mins and is divided into 5 second segments. Each experiment runs for all segments in the video emulated over network capacity and compute capacity traces (described below). The performance is reported as an average across all the segments in the video.

Network and compute conditions. Most of our evaluation is over real traces collected by FCC [8], similar to recent video streaming works [35, 69]. We use 500 traces and filter the traces to have a minimum bandwidth of 1 Mbps. After filtering, FCC dataset has an average bandwidth of 8.2 Mbps with a standard deviation of 3.6 Mbps. These traces capture real world network throughput variations.

Unlike other video streaming approaches, Swift is affected by compute capacity. To stress test our system, we evaluate Swift by synthetically varying the client's GPU capacity. We modify the GPU capacity by choosing a random number of the processes to be active in each time slot; the number of processes active is modeled as a Poisson distribution with $\lambda = 5$. Each process shares the GPU equally and we constrain the maximum number of processes to 5.

An ideal scenario for Swift is when the GPU capacity is fixed and 100% of the GPU is available. For completeness, we run experiments under this condition. We refer to this as Swift-C in the graphs. For a fair comparison, we compare Swift with existing methods assuming they have hardware accelerated decoding, while varying GPU resources for Swift.

Metrics. We measure streaming performance using the following metrics: 1) video QoE (as defined in §4.2) normalized against maximum QoE possible and averaged across all segments for all traces, 2) bandwidth usage, 3) reaction time (as defined in §6.2.3).

Baselines. We compare the performance of Swift with multiple state-of-the-art streaming algorithms that use different combinations of video coding and ABR algorithms:

- **Grad [31]:** Grad is the state-of-the-art algorithm using layered coding technique (SHVC [12]) combined with ABR. This is the closest system to Swift. Grad employs a hybrid coding mechanism with SHVC to minimize the cross layer compression overhead and uses a reinforcement learning-based ABR adapted from Penseive [35].
- **BOLA [52]:** BOLA and the two alternatives below use regular (not layered) codec H.265. BOLA uses an ABR algorithm that maximizes the quality of the video segment based on the buffer levels at the client. BOLA is commonly used in the industry [2].
- **Pensieve [35]:** Pensieve is also built over H.265 [53] but uses a reinforcement learning-based ABR algorithm.
- **BOLA-FS [51]:** BOLA-FS builds over H.265 [53] and uses buffer levels at the client to choose the next video segment, similar to BOLA. However, different from BOLA, BOLA-FS allows video quality upgrades, where low quality segments in the buffer are replaced with higher quality by re-downloading them, when network conditions improve. The problem is that the previously downloaded segments are not used, resulting in wasted bandwidth.

In all of these cases, when using H.265 [53], we encode each segment into six bitrates:{1Mbps, 5Mbps, 8Mbps, 12Mbps, 16Mbps}. For Grad, which uses scalable coding, we encode the video in six layers to achieve similar quality levels. We note that in both cases, encoding the videos into 6 quality levels provided the best results. In case of H.265, it does not support fine-grained adaptation to work well with more quality levels. In case of Grad/SHVC, the compression overhead is too high when using more quality levels. For Swift, we encode up to 10 layers for more flexible adaptation as there is no compression overhead.

6.2 End-to-end Streaming Results

6.2.1 End-to-end QoE Results

Figure 12 shows the overall QoE of Swift compared with the four alternatives, along with Swift-C. Swift-C represents the best possible performance of Swift, when compute capacity does not vary and GPU availability is 100%.

We first compare Swift with Grad and BOLA-FS which can both upgrade quality of the buffered video segments when network conditions improve. Swift improves QoE by 43% and 48% compared to Grad and BOLA-FS respectively. In the case of Grad, the problem is the high compression overhead incurred in implementing layered coding (§2.3). In case of BOLA-FS, there is a significant bandwidth wastage. When

[6] Note that the compression performance is evaluated using a more diverse and standard set of video sequences (see §6.3.1).

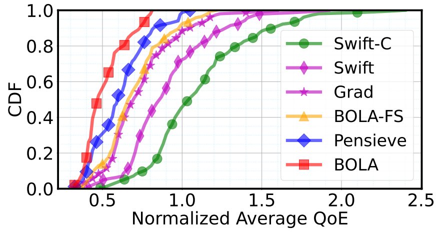


Figure 12: End-to-end QoE. Swift improves QoE by 45% at the median compared to the second best performing algorithm.

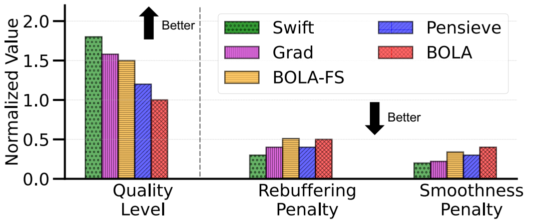


Figure 13: Breakdown of QoE. Overall, Swift has higher quality level while having less rebuffering and smoothness penalty.

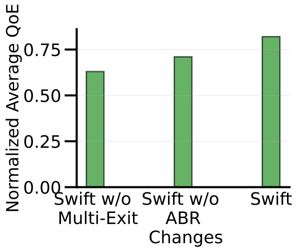


Figure 14: Breakdown Swift's performance with its individual components.

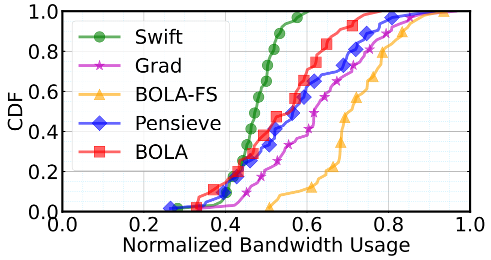


Figure 15: Bandwidth usage of Swift compared to state-of-the-art streaming systems. Swift improves bandwidth usage especially with respect to systems that upgrade quality when the network conditions improve, namely, BOLA-FS and Grad.

compared to Pensieve and BOLA, Swift outperforms by 67% and 74% respectively. Both Pensieve and BOLA do not upgrade video segment quality when the network improves, resulting in poorer quality.

Swift-C (Figure 12) shows the QoE achieved when GPU is not fluctuating and 100% of the GPU is used. As expected, Swift-C outperforms Swift under varying GPU. We also evaluated Swift under WiFi (802.11ac) network (client-server RTT: 20ms) without throttling the bandwidth and Nvidia 2070 GPU at 100%. We find that, Swift still outperforms the next-best-performing algorithm by 28%.

QoE breakdown Figure 13 shows the performance of the five streaming algorithms in terms of each QoE component: average quality of the video segments, rebuffering, and smoothness penalty. Swift improves average quality by 19% compared to the next-best streaming alternative. Swift also decreases rebuffering and smoothness penalty by 8% and 11%, respectively, compared to the next best streaming alternative.

Ablation study Figure 14 shows the impact of Swift's components: 1) Swift without multi-exit, 2) Swift without adapted ABR. The figure shows that both components are critical to the performance of Swift. Swift without multi-exit performs poorly compared to the full Swift because the decoder runs through the entire network even when GPU capacity is low rather than exit early. This results in high decoding latency and in-turn high video stalls. In case of Swift without adapting ABR, the system performs poorly because it only adapts to network variations and not compute variations.

6.2.2 Bandwidth Benefits

Figure 15 shows the bandwidth benefits of Swift over existing streaming alternatives. Swift uses 16% and 22% less bandwidth compared to Grad and BOLA-FS, incurred due to compression overhead and wasted bandwidth respectively. Pensieve and BOLA results in comparatively less bandwidth waste, but cannot upgrade quality when the network improves resulting in poorer video quality (Figure 12).

6.2.3 Reaction to Bandwidth Fluctuations

One key advantage of Swift, or layered coding in general, is that it can adapt to bandwidth fluctuation without wasting bandwidth (see Figure 3). To illustrate this, we use an example network trace that starts with an average low bandwidth of 1 Mbps for 100 seconds and increases to average 18 Mbps for the rest of the trace (250 secs).

To compare the performance of these different streaming techniques, we measure the reaction time in two ways: 1) reaction time to any quality (RTA), which is the elapsed time between when the bandwidth increases to when the user experiences any higher quality video, 2) reaction time to highest quality (RTH), which is the elapsed time between when the bandwidth increases to when the user experiences the highest sustainable video quality.

Figure 16 shows one scenario how the different streaming algorithms adapt to changing network condition for a 250 second sample trace. The black line shows the change in throughput. Swift is the first to react to the change in throughput of all the other alternatives. Figure 17 shows qualitatively that Swift reacts faster, both in terms of RTA and RTH, compared to the alternatives.

Overall, the normalized average video segment quality of Swift throughout the trace was 1.8 compared to the next best alternative, which was 1.6. The reaction time is low even when the throughput decreases instead of increasing (not shown).

6.3 Compression Results

We compare Swift's codec with:

- **HEVC [53]:** This is the most commonly used video codec for video streaming today. We use the libx265 library

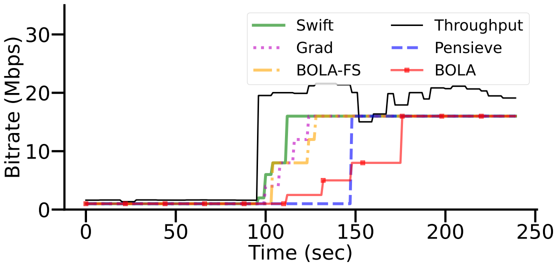
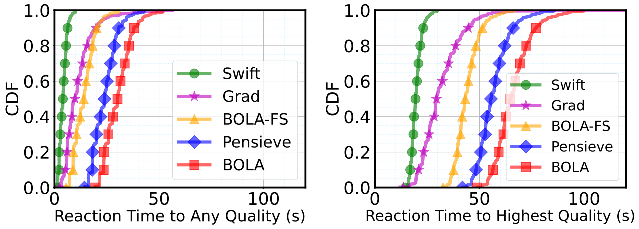


Figure 16: Swift reacts faster compared to all other alternatives. Throughput changes at the 100 second mark.



(a) Reaction time to any quality (b) Reaction time to highest quality

Figure 17: Reaction time. Swift reacts 2× and 2.5× faster than Grad and BOLA-FS to reach highest quality, and 25% and 40% faster to reach any high quality.

from FFMPEG [7]. We did not investigate the latest coding standard VVC [59] as it is still in its early stage. We report H.265 results with commonly used codec configuration.[7]

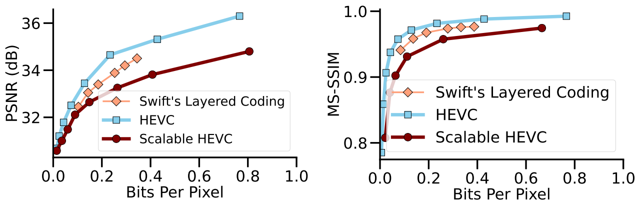
- **Scalable HEVC (SHVC) [12]:** This is a state-of-the-art layered coding method, built as a scalable extension of HEVC also known as SHVC [12]. We evaluate SHVC using a reference implementation from [1].

We present the result averaged over three datasets. One dataset is the test set from the Kinetics dataset (§5). The other two datasets are VTL [6] and UVG [5] that are not used in training. All VTL test videos are in 352 × 288 and UVG videos are in 1920 × 1080 resolution.

6.3.1 Compression Efficiency

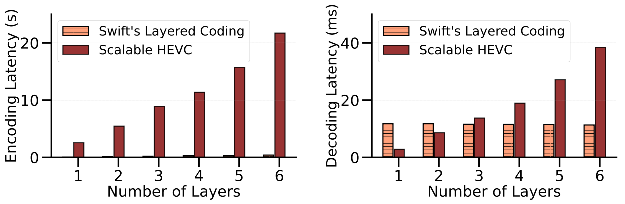
Figure 18 shows the video quality vs. video size in terms of bits per pixel (BPP) after compression. The metrics we use are: 1) PSNR – this computes the peak signal to noise ratio between two images (higher PSNR indicates better quality of reconstruction), and 2) MS-SSIM (multi-scale structural similarity index method) – a perceptual quality metric taking into account the structural information to weigh more on the spatially close pixels with strong inter-dependencies [56]. For SHVC and Swift's layered coding a total of 6 layers are used to produce the plots – each point refers to the joint performance of all 6 layers. For HEVC or H.265, each quality point is encoded independently with a different bitrate.

Swift's layered coding achieves 58% better compression on average compared to traditional layered coding (SHVC).



(a) PSNR (b) MS-SSIM

Figure 18: Compression efficiency. Swift's Layered coding outperforms the traditional layered coding SHVC and performs close to HEVC.



(a) Encoding Latency (b) Decoding Latency

Figure 19: Encoding and Decoding latency. Both the encoding and decoding latency Swift's layered coding is significantly less than traditional layered coding. More importantly, the decoding latency of Swift's layered coding is independent of number of layered codes unlike traditional layered coding.

The compression difference is mainly due to the cross-layer overhead incurred by SHVC. Swift's compression is close to that of HEVC with Swift performing poorer by 0.02 MS-SSIM and 0.8 dB in the median case. However, the QoE when using Swift is still better than the QoE than HEVC because of the fine-grained rate adaptation benefits.

6.3.2 Encoding and Decoding Latency

In this set of experiments, we compare the encoding and decoding latency of Swift's codec and the state-of-the-art layered codec (we omit regular HEVC codec here because it has negligible latencies). The evaluation here benchmarks the latency on a Desktop machine (Intel 12 core CPU with Nvidia RTX 2070 GPU).[8] Figure 19 shows average per-frame encoding and decoding latency as a function of number of layers. There is 15× increase in encoding latency from layer one to six in case of traditional layared coding (SHVC). Swift's layered coding has an encoding latency of 20ms for one layer and increases proportionately by 6× for the sixth layer. While improvements are still needed to get close to the encoding latency of regular (non-layered) codec, the encoding latency in Swift is significantly less than SHVC. More importantly, the decoding latency of Swift's layered coding is independent of the number of layers, while SHVC increases proportionately similar to encoding latency. This is due to the use of the single-shot decoder of Swift (§3.3).

[7] We use *fast* preset with group of pictures value 30.

[8] Of note, HEVC and SHVC are run on the CPU and our layered coding runs on GPU as it is the most efficient on the GPU.

7 Discussion

Swift's layered coding addresses compression overhead challenge highlighted in §2. Below we discuss some of the additional benefits as well as limitations of layered neural codecs.

7.1 Additional Opportunities of Neural Codecs

Flexible data-driven approach: In a learning-based approach to video coding the learning can be made video specific, for example, customized to video types [9], likely providing an opportunity to better learn video type-specific features. This ultimately leads to streaming quality improvements relative to the one-size-fits-all solution that exists today.

Software-defined coding: Unlike existing codecs, neural codecs do not need to be baked into a fixed hardware. They are more easily upgradable. Common ML tools (e.g., Pytorch with CUDA support) ensure that running neural codecs on data parallel co-processors (e.g., GPUs) requires significantly less development cycle compared to porting traditional codecs. The softwarization of video coding gives the content providers flexibility to integrate codec features on demand, support agile codec development, provide royalty-free codecs, and eliminate compatibility issues.

Design of application-specific codecs: Various video analytics solutions [21, 24, 62] often apply DNN-based analytics (e.g., object detection and classification) on video streams that are coded using traditional video codecs. However, this results in suboptimal performance because they are originally designed for human perceptual quality. Instead, neural codecs are amenable to training with loss functions more tuned towards appropriate analytics. Similarly, specialized codecs can be designed for conferencing or surveillance that may have constant backgrounds or other commonly appearing features that, once learnt, can be compressed very efficiently.

7.2 Limitations of Neural Codecs

One key assumption of Swift is that the client devices need to be equipped with GPUs or other similar accelerators to run neural networks. Otherwise, the decoding latency could become a bottleneck. While such accelerators are expected to be commonplace, they do add to the device cost and energy consumption. Also, the current design of Swift targets on-demand video streaming because the iterative layered coding does not offer real-time encoding. More work is needed on the encoding side for applying Swift to live video applications (such as conferencing or live analytics) to overcome the encoding latency challenges.

Finally, the QoE evaluations for Swift are done using a learning based ABR algorithm based on Pensieve [35] and Grad [31]. It may be challenging to generalize such algorithms for unknown environments that can still occur in practice [34, 63]. However, given our characterization of the input and output along with the objective function, we expect that other algorithmic ABR approaches (such as BOLA [52] or FUGU [63]) are equally applicable for Swift.

8 Related Work

Video streaming: There has been an extensive prior work on improving QoE for regular video streaming. Much of the previous work focuses on improving the adaptive bitrate algorithms by better predicting the available throughput. Festive [25] predicts throughput using a harmonic mean. BBA [23] and BOLA [52] take into account the buffer capacity to determine video bitrate. Fugu [63] and MPC [66] use learning-based throughput prediction. There is a recent interest in using reinforcement learning for adaptive bitrate selection (e.g., Pensieve [35] and other follow-up work). Recent solutions such as SENSEI [69] improves QoE by introducing user sensitivity into ABR algorithms. Swift is able to extend existing ABR algorithms for use with neural codecs and can synergistically optimize network and compute resources to improve QoE.

Video compression: Traditional compression methods such as H.264/265 [53, 57] employ many algorithms that include frame prediction [64, 70], transform coding and quantization [18, 40, 48, 58], and entropy coding [54]. In the past decade or two, there have been several studies on improving both the compression efficiency and coding latency for these algorithms on an individual basis [15, 16, 26, 49]. Similarly, there have been extensive studies on improving the traditional layered coding, while still facing challenges of compression overhead and high latency [11, 14, 67]. Unlike all these algorithmic codecs, there is a recent shift in codec design using deep learning [32, 43, 44, 60]. Swift belongs to this second category and develops layered coding on top of neural codecs.

9 Conclusions

We have described Swift, an adaptive video streaming system using layered neural codecs that use deep learning. Swift's neural codec achieves efficient layered compression without introducing cross layer compression overheads and eliminates the dependency of decoding latency on the number of layers. Swift extends existing ABR frameworks to accommodate layered neural codecs and demonstrates significant performance benefits compared to state-of-the-art adaptive video streaming systems.

Acknowledgements

We thank our shepherd Junchen Jiang and the anonymous reviewers for their feedback, which greatly improved the paper. This work was partially supported by the Partner University Fund, the SUNY2020 ITSC, and a gift from Adobe.

References

[1] A Reference Implementation of SHVC (Scalable extentino to HEVC). https://hevc.hhi.fraunhofer.de/shvc.

[2] Akamai players. https://players.akamai.com/players/dashjs.

[3] HEVC scalability extension. https://hevc.hhi.fraunhofer.de/shvc.

[4] libvpx-vp9. https://trac.ffmpeg.org/wiki/Encode/VP9.

[5] Ultra video group. http://ultravideo.fi/.

[6] Video trace library. http://trace.eas.asu.edu/index.html.

[7] x265. https://trac.ffmpeg.org/wiki/Encode/H.265.

[8] Measuring broadband America, FCC. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-eighth, 2018.

[9] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *preprint arXiv:1609.08675*, 2016.

[10] Shivang Aggarwal, Urjit Satish Sardesai, Viral Sinha, Deen Dayal Mohan, Moinak Ghoshal, and Dimitrios Koutsonikolas. LiBRA: learning-based link adaptation leveraging PHY layer information in 60 GHz WLANs. In *ACM Confernece on Emerging Networking Experiments and Technologies*, pages 245–260, 2020.

[11] Ghufran Baig, Jian He, Mubashir Adnan Qureshi, Lili Qiu, Guohai Chen, Peng Chen, and Yinliang Hu. Jigsaw: Robust live 4K video streaming. In *MobiCom*, pages 1–16, 2019.

[12] Jill M Boyce, Yan Ye, Jianle Chen, and Adarsh K Ramasubramonian. Overview of shvc: Scalable extensions of the high efficiency video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):20–34, 2015.

[13] Joao Carreira and Andrew Zisserman. Quo vadis, Action recognition? A new model and the kinetics dataset. In *CVPR*. IEEE, 2017.

[14] Jacob Chakareski, Sangeun Han, and Bernd Girod. Layered coding vs. multiple descriptions for video streaming over multiple paths. *Multimedia Systems*, 10(4):275–285, 2005.

[15] Mei-Juan Chen, Yu-De Wu, Chia-Hung Yeh, Kao-Min Lin, and Shinfeng D Lin. Efficient CU and PU decision based on motion information for interprediction of HEVC. *IEEE Transactions on Industrial Informatics*, 14(11):4735–4745, 2018.

[16] Santiago De-Luxán-Hernández, Valeri George, Jackie Ma, Tung Nguyen, Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. An intra subpartition coding mode for vvc. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1203–1207. IEEE, 2019.

[17] Anis Elgabli, Vaneet Aggarwal, Shuai Hao, Feng Qian, and Subhabrata Sen. LBP: robust rate adaptation algorithm for SVC video streaming. *IEEE/ACM Transactions on Networking*, 26(4):1633–1645, 2018.

[18] Vivek K Goyal. Theoretical foundations of transform coding. *IEEE Signal Processing Magazine*, 18(5):9–21, 2001.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[21] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *Usenix OSDI*, pages 269–286, 2018.

[22] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*, volume 33, pages 3812–3821, 2019.

[23] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[24] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*, pages 253–266, 2018.

[25] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN)*, 22(1):326–340, 2014.

[26] Y-H Kim, J-W Yoo, S-W Lee, J Shin, J Paik, and H-K Jung. Adaptive mode decision for H.264 encoder. *Electronics letters*, 40(19):1172–1173, 2004.

[27] PoLin Lai, Shan Liu, and Shawmin Lei. Low latency directional filtering for inter-layer prediction in scalable video coding using hevc. In *2013 Picture Coding Symposium (PCS)*, pages 269–272. IEEE, 2013.

[28] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. Adaptive inference through early-exit networks: Design, challenges and directions. *arXiv preprint arXiv:2106.05022*, 2021.

[29] Athanasios Leontaris and Pamela C Cosman. Drift-resistant snr scalable video coding. *IEEE transactions on image processing*, 15(8):2191–2197, 2006.

[30] Weiyao Lin, Krit Panusopone, David M Baylon, and Ming-Ting Sun. A computation control motion estimation method for complexity-scalable video coding. *IEEE transactions on circuits and systems for video technology*, 20(11):1533–1543, 2010.

[31] Yunzhuo Liu, Bo Jiang, Tian Guo, Ramesh K Sitaraman, Don Towsley, and Xinbing Wang. Grad: Learning for overhead-aware adaptive video streaming with scalable video coding. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 349–357, 2020.

[32] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. DVC: An end-to-end deep video compression framework. In *CVPR*, pages 11006–11015, 2019.

[33] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[34] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. *arXiv preprint arXiv:2008.12858*, 2020.

[35] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*, pages 197–210. ACM, 2017.

[36] Steven McCanne, Martin Vetterli, and Van Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE journal on selected areas in communications*, 15(6):983–1001, 1997.

[37] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. ePerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 382–394, 2020.

[38] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.

[39] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[40] Tung Nguyen, Philipp Helle, Martin Winken, Benjamin Bross, Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. Transform coding techniques in hevc. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):978–989, 2013.

[41] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[42] Amy R Reibman, Leon Bottou, and Andrea Basso. Scalable video coding with managed drift. *IEEE transactions on circuits and systems for video technology*, 13(2):131–140, 2003.

[43] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elf-vc: Efficient learned flexible-rate video coding. *arXiv preprint arXiv:2104.14335*, 2021.

[44] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. *preprint arXiv:1811.06981*, 2018.

[45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 Usenix ATC 21*, pages 397–411, 2021.

[46] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11 ad/ac Wireless LANs. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[47] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[48] Heiko Schwarz, Tung Nguyen, Detlev Marpe, and Thomas Wiegand. Hybrid video coding with trellis-coded quantization. In *2019 Data Compression Conference (DCC)*, pages 182–191. IEEE, 2019.

[49] Mahmut E Sinangil, Vivienne Sze, Minhua Zhou, and Anantha P Chandrakasan. Cost and coding efficient

motion estimation design considerations for high efficiency video coding (HEVC) standard. *IEEE Journal of selected topics in signal processing*, 7(6):1017–1028, 2013.

[50] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4), 2011.

[51] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2s):1–29, 2019.

[52] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[53] 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.

[54] Vivienne Sze and Madhukar Budagavi. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, 2012.

[55] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *CVPR*, pages 5306–5314, 2017.

[56] 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.

[57] 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.

[58] Mathias Wien. Variable block-size transforms for H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):604–613, 2003.

[59] Mathias Wien and Benjamin Bross. Versatile video coding–algorithms and specification. In *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP)*, pages 1–3. IEEE, 2020.

[60] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *ECCV*, pages 416–431, 2018.

[61] Feng Wu, Shipeng Li, and Ya-Qin Zhang. A framework for efficient progressive fine granularity scalable video coding. *IEEE transactions on Circuits and Systems for Video Technology*, 11(3):332–344, 2001.

[62] Xiufeng Xie and Kyu-Han Kim. Source compression with bounded dnn perception loss for iot edge computer vision. In *ACM MobiCom*, pages 1–16, 2019.

[63] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.

[64] Jiheng Yang, Baocai Yin, Yanfeng Sun, and Nan Zhang. A block-matching based intra frame prediction for H.264/AVC. In *2006 IEEE International Conference on Multimedia and Expo*, pages 705–708. IEEE, 2006.

[65] 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.

[66] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 325–338. ACM, 2015.

[67] Sangki Yun, Daehyeok Kim, Xiaofan Lu, and Lili Qiu. Optimized layered integrated video encoding. In *INFOCOM*, pages 19–27. IEEE, 2015.

[68] Alireza Zare, Alireza Aminlou, Miska M Hannuksela, and Moncef Gabbouj. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 601–605. ACM, 2016.

[69] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. Sensei: Aligning video streaming quality with dynamic user sensitivity. In *NSDI*, pages 303–320, 2021.

[70] Shiping Zhu, Shupei Zhang, and Chenhao Ran. An improved inter-frame prediction algorithm for video coding based on fractal and H.264. *IEEE Access*, 5:18715–18724, 2017.

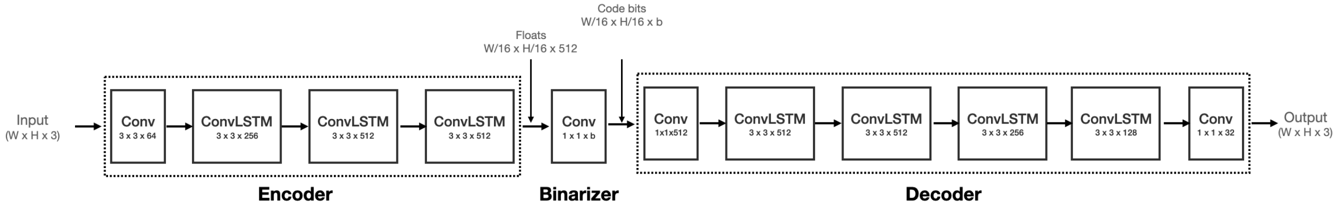


Figure 20: An example of a base neural codec and its internal logic to encode and decode a video frame in a single iteration.

A Appendix. Example Neural Codec

Swift's layered neural coding is designed as a generic extension that can be implemented on top of any neural codec, and hence throughout the paper, we considered neural codec as a blackbox without discussing the internal details of codec logic. In this section, we present an example codec for a better understanding of neural coding principles. This example follows the design of VCII [60], the same design we also used in our implementation.

In general, most of the existing neural codecs follow traditional concepts of I, P, and B frames when compressing a video [22, 43, 60]. An I frame is compressed much like an image with no reference, and P/B frames reference other frames for reconstruction as they encode motion and residual information relative to the reference frames. Swift adopts a similar approach of compressing I frames and P/B frames separately by using i) a neural image codec [55] for compressing I frames, and ii) a neural video codec [60] for compressing P/B frames. The output for each of these frames after the encoding stage from the Autoencoder, is a neural representation, i.e., code bits with floating point values. The code bits for I frames represent directly the frame data, however, the code bits P/B frames represent motion and residual information with respect to the reference frames.

Figure 20 shows an example codec structure followed by [55, 60] as well as Swift. It contains three key parts: encoder, binarizer, and decoder. The encoder takes the original video frame as input and applies convolutions (along with an LSTM block) to downscale the frame into a low dimensional vector. In our example, we have four such blocks, each downscaling the frame resolution by half. For example, when we encode a 1280×720 frame, the output of encoding stage contains $80 \times 45 \times 512$ resolution with floating point representations. After the encoding, a binarizer converts the floats to a binary bitstream with the same resolution but packs each float in b bits. Optionally, these bits can be further passed through an entropy encoder [54] to compress the bitstream efficiently. During the decoding process, a reverse process is learned by upsampling the frame resolution at each stage in the network, achieving the original resolution at the final stage.