Comparing CNNs and JPEG for Real-Time Multi-view Streaming in Tele-Immersive Scenarios

Konstantinos Konstantoudakis*, Emmanouil Christakis*, Petros Drakoulis, Alexandros Doumanoglou,
Nikolaos Zioulis, Dimitrios Zarpalas, and Petros Daras
Visual Computing Lab (VCL), Information Technologies Institute (ITI)
Centre for Research and Technology - Hellas (CERTH)
Thessaloniki, Greece
Email: {k.konstantoudakis, manchr, petros.drakoulis, aldoum, nzioulis, zarpalas, daras}@iti.gr

***Abstract*—Deep learning-based codecs for lossy image compression have recently managed to surpass traditional codecs like JPEG and JPEG 2000 in terms of rate-distortion trade-off. However, they generally utilize architectures with large numbers of stacked layers, often making their inference execution prohibitively slow for time-sensitive applications.**

In this work, we assess the suitability of such compression techniques in real-time video streaming, and, more specifically, next-generation interactive tele-presence applications, which impose stringent latency requirements. To that end, we compare a recently published work on image compression based on convolutional neural networks which achieves state-of-the-art compression ratio using a relatively lightweight architecture, against a CPU and a GPU implementation of JPEG, measuring compression ratios and timings.

With these results, we run a simulation of a tele-immersion pipeline for various networking conditions and examine the performance of the compared codecs, calculating framerates and latencies for different codec/network combinations.

***Keywords*-Video, Compression, Tele-Immersion, 3D Media Streaming, Performance Evaluation**

I. Introduction

The last decade has seen a rapid growth in the amount of image and video data that is being created, stored, and transferred over the Internet. Advancements in the resolution of both capturing and display devices have increased the average amount of data per image, while the ubiquity of smartphones provides even non-expert users with the means to easily and inexpensively create image and video content. With technology providing the means, the prevalence of (largely multimedia-oriented) social media in daily life gives the incentive, thus making content creation a large-scale and everyday phenomenon.

The huge amount of images and videos that are created, stored and transferred showcases the need for efficient compression of these media. This applies even more in real-time video streaming scenarios, where low video bitrate must be balanced against low encoding and decoding time, so as to prevent significant lag in the live viewing experience. This can lead to decisions not normally applied to video coding, such as foregoing motion compensation, which leads to better compression ratios, in order to save the encoder the computational load that even fast motion estimation algorithms entail. Without motion estimation, temporal redundancy is not exploited and video frames are encoded and transferred as individual images. In such a case, efficient and fast image compression algorithms, such as JPEG, can be used to compress each frame.

Furthermore, with recent advances in the field of machine learning, alternative approaches to traditional image codecs have emerged. Convolutional neural networks (CNNs: [1]–[6]), generative adversarial networks (GANs: [7], [8]) and recurrent neural networks (RNNs: [9], [10]) have all been used to compress images. Unfortunately, most machine learning approaches suffer from slow encoding and decoding times, even when running on GPUs. However, a few report timings that are not prohibitively high for real-time applications, and could thus compete with more traditional codecs, at least in certain hardware and network environments.

The present work aims to compare different encoding strategies adapted to a scenario of live multi-view video streaming, which is one of the most exacting applications with respect to both compression and speed in encoding (and decoding). More specifically, we consider the case of tele-immersive media, where streaming multi-view video content is used to texture real-time 3D reconstructed meshes representing actual captured humans. In this setting four cameras capture a user from different angles in order to create a self-user representation [11]. Thus, four different video streams must be encoded, transferred and decoded in real time.

Towards this end, we have assessed the performance of different implementations of the JPEG algorithm as well as the CNN presented in [1], with respect to encoding and decoding time, standard image quality metric MS-SSIM, and the achieved compression ratio. The methods are compared with each other, the advantages and disadvantages of each are highlighted, and the usefulness of each in a number of possible scenarios, as regards network conditions and available hardware, are discussed.

The rest of this paper is organized as follows: Section II contains a compilation of related work; in Section III we compare three codec implementations and present the corresponding

This work has been supported by the EC Project 5G-MEDIA www.5gmedia.eu. This project has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 761699.

* indicates equal contribution.

rate-distortion curves and timings; then, in Section IV, we apply these codecs to specific use-case scenarios, relevant to real-time four-viewpoint tele-immersion; concluding, in Section V, we discuss the results and their implications.

II. Related work

In the recent years researchers have published works focusing on image compression using machine learning architectures, including CNNs, RNNs, and GANs. A number of them have managed to outperform traditional codecs like JPEG, JPEG 2000 and BPG, producing better rate-distortion curves. This has already proven useful in many applications where encoding and decoding times are of less importance than the amount of storage required for the encoded file. However, such is not always the case, especially in real-time video streaming, where encoding and decoding times are of importance to maintain a level of user experience quality.

In [9], Toderici et al. present an RNN-based image encoder and decoder which can encode an image in various qualities and bitrate levels depending on the number of iterations where every iteration tries to enhance the output of the previous one. This codec manages to outperform JPEG in terms of the PSNR and MS-SSIM of the decoded image, especially in the lower bitrates. However, the multiple iterations of the RNN push the encoding and decoding times orders of magnitude higher than JPEG. Minnen et al. [10] further improve upon these results by adopting a tile-based approach, enabling a finer tuning of local bitrate.

On the other hand, Theis et al. [2] propose a simpler, CNN-based, autoencoder with subpixel convolutions. Variable bitrates are achieved by fine tuning a pre-trained network for different rates, using a number of scale parameters. Cheng et al. [3] present another convolutional autoencoder architecture, in which principal component analysis is used to generate a more energy-compact representation. Li et al. [4] use a content-weighted importance map so that the bitrate used for different parts of the image is adapted according to local image content. The sum of the importance map can serve as a metric to control compression rate. In [5] Nakanishi et al. propose a model consisting of two networks, a lossy autoencoder which encodes multiscale features of the image and a parallel multiscale lossless coder. Jianrui et al. [6] report training a CNN to compress images in various bitrates without having to train a different model for each. They achieve that by including a Tucker decomposition layer, which decomposes the latent image representation into a set of projection matrices and a core tensor. The compression rate is regulated by altering the rank of the core tensor.

Mentzer et al. [1] also use a modified autoencoder architecture, controlling the trade-off between the compression ratio and distortion through the use context models and cross-entropy estimation. Different bitrates are achieved by training models with different number of channels in the bottleneck of the autoencoder and by adjusting the trade-off between the distortion and the entropy of the encoded representation in the loss function during the training phase. Mentzer et al.'s approach is of additional interest, as it combines near-state-of-the-art rate/distortion with a relatively fast and lightweight CNN architecture and publicly available code.

Finally, GANs have also been successfully used to compress images. Rippel et al.'s work [7] also features an autoencoder, including pyramidal analysis, an adaptive coding module and a GAN-based network which learns to distinguish the original image from the reconstructed by the autoencoder. The authors claim that their codec can encode or decode an image from the Kodak dataset in 10 ms using a GPU, while also achieving state-of-the-art rate distortion performance in terms of multi-scale SSIM. And in [8], Agustsson et al. study the use of GANs for lossy image compression at very low bitrates below 0.1 bits per pixel, where the texture of local regions of the image becomes impossible to preserve. The generative ability of GANs is utilized to synthesize such regions from a semantic label map extracted from the original image.

A performance comparison between three different architectures for image compression (based on CNNs, GANs, and super-resolution) is presented in [12]. However, it only takes rate-distortion into account, therefore only objectively comparing the different coding methods. In none of the aforementioned cases is the temporal dimension taken into account, therefore providing little insight as to the applicability and performance trade-offs of using such codecs in streaming applications.

III. Codec evaluation

A. Selection of codecs to compare

Tele-immersion relies on creating the illusion of co-location for all participating users. To maintain this illusion, and especially for interactive applications, it is very important that both a user's own actions, as well as the actions of other users, are near-instantaneously reproduced in the virtual environment. When the time lag between real action and 3D representation increases, the illusion cannot be maintained, as users cannot associate lag with real life. This is even more pronounced in self-user representations in VR settings, where the differences between the motions performed and the motions sensed result in cyber-sickness.

Furthermore, tele-immersion entails the transfer of much more data than regular videos. At the very least, image-based representations transfer multiple viewpoint videos of each user and interpolate intermediate positions from these. More advanced approaches entail the transmission of each user's geometry in a 3D mesh, along with a number of texture images, which capture color information from different viewpoints and which are subsequently used to texture the mesh. The present work assumes four different viewpoint videos which capture a user from different angles, accompanied by a 4D (3D+time) mesh representation of the user's geometry.

In order to keep time lag to a minimum, and also given the heavy processing required to produce this representation in real-time, the decision is often made to not employ inter-frame compression on the videos, as the process of motion estimation and compensation across four videos has a very

high computational cost. Instead, each video frame can be encoded individually with a fast and simple image encoder, such as JPEG. This increases the amount of data that needs to be transferred over the network, but decreases the computational load on the encoder, which is translated to latency as a result of the processing time required to encode 4 videos.

Deep learning codecs incur a much higher computational load than JPEG, which makes them quite slower, even running on GPUs. On the other hand, they promise significantly higher rate-distortion curves. Thus, a comparison between JPEG and a relatively fast deep learning image code, oriented towards the case of multiple viewpoint videos, can be important in plotting future tele-immersion strategy.

Mentzer et al. [1] have proposed a simple and relatively fast CNN codec that compares favorably, in rate-distortion terms, to the state of the art in deep learning image codecs. Furthermore, three different trained CNN models, corresponding to different points on their rate-distortion curve, are publicly offered. Other interesting machine learning methods either proved too slow for this real-time scenario ([2], [9]) or were unavailable for independent validation ([7]). For these reasons, in the following pages we will compare heavily optimized CPU [13] and GPU [14] implementations of the JPEG codec to the CNN codec described in [1]. In the following pages, all references to CNN models regard the CNN from [1].

B. Experimental setup

1) CNN architecture: The encoder in [1] downscales the input image by a factor of 8 in each dimension, with a variable channel depth, depending on the exact model used. In their work the authors alter the basic convolutional autoencoder architecture to apply an importance map as an extra channel on the encoder's output. The value of the importance map at each position defines the number of channels which will be actually encoded. The important channels for each position are quantized to 6 non-uniform quantization levels; all the rest are set to zero. In this way, the importance map regulates the number of bits to be used for each region of the encoded image.

In order to be efficiently compressed, the encoder's output must then undergo (lossless) entropy coding. Mentzer et al. propose an arithmetic encoder for this purpose, which takes into consideration the frequencies and probabilities of symbols within a specified context area for each symbol. This process produces compression very close to the theoretical limit imposed by cross-entropy, but, mainly due to this extensive context calculation, proves to be prohibitively slow for real-time applications. Hence, in our evaluation we decided to use a different, much faster entropy coder instead. Since a large number (close to half of total) of elements at the encoded output are set to zero, and most of them are grouped together, run length encoding, followed by Huffman coding, was deemed the best choice. This was implemented by the hzr open source encoder [15], which reports fast encoding and decoding times, in the range of 6-13 ms total (for all 4 textures). The hzr entropy coding scheme produces more bits per pixel than what is reported in [1], but at a fraction of the time, which is more important in the tele-immersion scenario, as will become clear in the next section. Other open source entropy coders either failed to achieve significant compression (zlib [16], lz4 [17]) or proved too slow to be considered in this real-time scenario (bzip2 [18]).

2) Metrics: In evaluating the three implementations mentioned above we measured:

- Encoding time
- Decoding time
- Bits per pixel (bpp)
- Multi-scale Structural Similarity Index Metric (MS-SSIM)

We chose to include the MS-SSIM [19] metric rather than PSNR in our evaluation, as the CNN presented at [1] is optimized to maximize MS-SSIM, and the MS-SSIM has proven to be more consistent with the subjective human evaluation of image quality. The above quantities were measured for each of the three publicly available models of the CNN, and for a number of compression levels for the JPEG implementations. The MS-SSIM metric was combined with bpp to plot a rate-distortion curve for each codec. In the case of the CNN, encoding and decoding timings include the hzr encoding and decoding time as well. Further, for the GPU implementations, the timings include all memory communication from the CPU to the GPU and vise versa, better reflecting the case of real-time streaming where the compressed memory will be delivered to the host system (i.e. CPU memory) for transmission.

3) Dataset: In order to emulate the tele-immersion scenario, all tests were conducted using actual samples from a tele-immersive pipeline [11], in which each sample consists of four color images which capture the user from a different viewpoint angle. Each image is sized 960x540 pixels, making one sample (of 4 images) the equivalent of a single full HD image, and have never undergone lossy compression. A total of 10 samples were used, each taken from a different tele-immersion video sequence.

4) Hardware and software configuration: All experiments were performed on a computer with an 4-core, 8-thread i7 7700K processor at 4.5 GHz, and 32GB of DDR4 RAM. The GPU implementation for JPEG, as well as the CNN from [1] were executed on a GeForce GTX 1080 Ti GPU with Cuda 8.0 installed [14]. The CNN requires Python and TensorFlow [20]; Python version 3.5.2 and TensorFlow 1.4.1 were used in the experiments.

C. Results

Fig. 1 shows a rate-distortion curve for each of the three codecs, plotting the mean MS-SSIM value for the whole dataset against the mean number of bits per pixel, for JPEG qualities with bitrates comparable to the CNN (0.5-1.5 bpp). The labels near each point on the figure denote the JPEG quality (Q) or the CNN model that generated it. It can be seen that the CNN outperforms both implementations of the JPEG codec. Furthermore, the CPU implementation achieves lower distortion than the GPU implementation at the same bitrate.

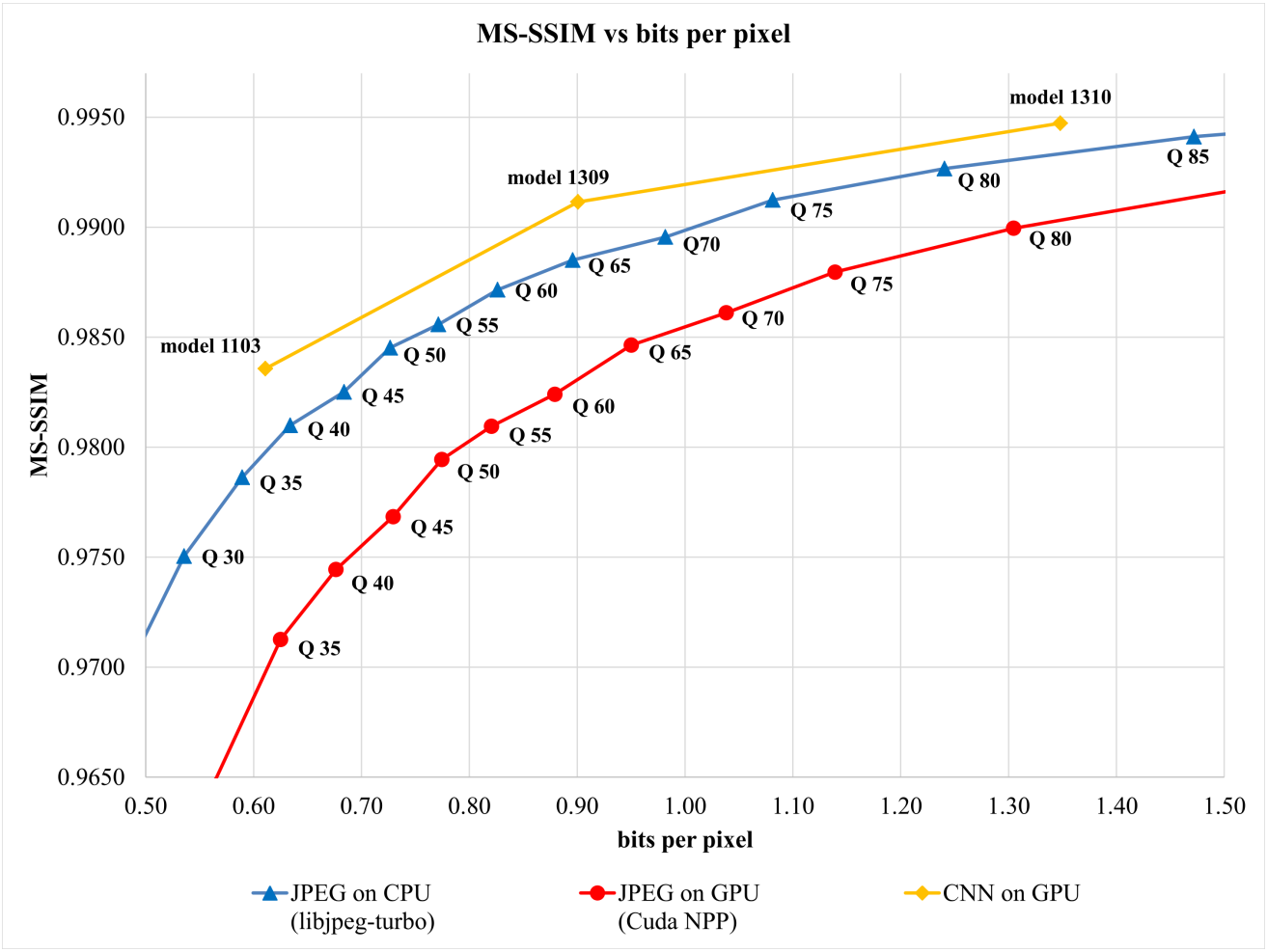

Fig. 1: Rate/distortion curve for the CPU and GPU JPEG implementations, and the CNN from [1].

In order to achieve a fair comparison in the following experiments, both in this and in the next section, each of the three CNN models was matched to that quality levels of the two JPEG implementations that produced roughly the same MS-SSIM. Table I shows the correspondence between equivalent, in terms of MS-SSIM, modes of the three codecs.

In the following pages each model of the CNN will be compared to the corresponding qualities of the JPEG implementations.

Fig. 2 and 3 show the timings of the encoding and decoding processes, respectively, for the three CNN models and the corresponding JPEG qualities, in logarithmic scale. As expected, the CNN is much slower than the two JPEG implementations during both encoding and decoding, with no significant variations between models. While the GPU implementation of JPEG in much faster in the encoding phase compared to the CPU one, decoding is actually faster in the CPU implementation. We found that this interesting fact holds only for the higher qualities of JPEG, while on the lower qualities the GPU implementation is faster during both encoding and decoding.

With the JPEG timings well below the 10 ms mark, neither JPEG implementation is expected to impact negatively on the frame rate or latency of a tele-immersion system. The CNN, on the other hand, showing an average encoding time of 124 ms, is bound to have a drastic impact on both frame rate and latency. In the next section we will validate these expectations, testing the above results on three different network conditions.

TABLE I: CNN models and equivalent JPEG qualities

CNN model [1]	CPU JPEG quality	GPU JPEG quality	MS-SSIM
1103	50	65	≈ 0.9836
1309	75	85	≈ 0.9912
1310	85	90	≈ 0.9947

IV. Use-case scenario

A. *Tele-immersion pipeline*

Having acquired the benchmarking measurements from the experiments detailed in Section III, we can now use them in a theoretical model of a tele-immersion pipeline, which considers conditions on the encoder's side, on the decoder's side, and on the network.

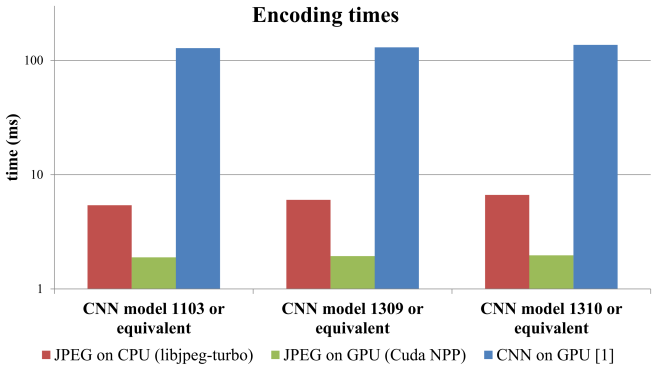

Fig. 2: Encoding times for the three CNN models and equivalent qualities of CPU and GPU JPEG.

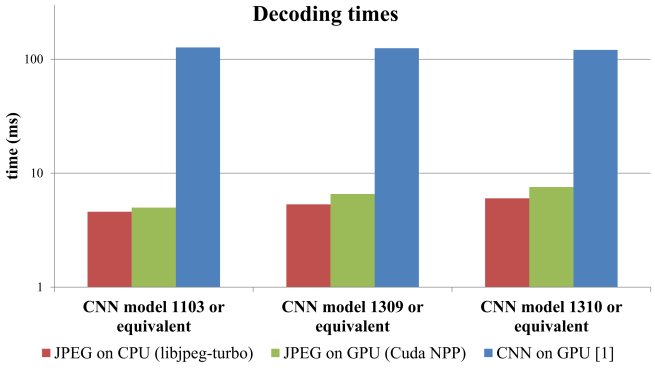

Fig. 3: Decoding times for the three CNN models and equivalent qualities of CPU and GPU JPEG.

Fig. 4 illustrates a typical video streaming pipeline, consisting of the encoder, the decoder, and the network that connects the two. This is a simplified version of a tele-immersion pipeline as presented in [21], which would also include 3D mesh transmission, alongside the image frames. Each frame of input video, on the sender's side, must go through the following stages before being ready for consumption by the remote user:

1) compression by the encoder
2) queue to enter the network
3) transfer via the network to the remote computer
4) queue to be processed by the decoder
5) decompression by the decoder

Each of the above stages incurs a possible time cost; after each stage the frame rate at that point can be calculated, with each stage along the way having a frame rate either equal to or lower than the rate of the previous stage. The compression and decompression times have been directly measured. The other timings, as well as the frame rates at each stage, can be inferred by: the average compressed frame size, which has also been directly measured and comprises the amount of data to be transferred over the network; and the network's own conditions. The latter can be described by the network's effective bandwidth (EBW), as calculated using the Mathis equation [22], and its round-trip time (RTT) between sender and receiver. Thus, by setting the EBW and RTT we can define a number of different network scenarios, and calculate the frame rate at the remote user as well as the total end-to-end latency.

Accordingly, we have constructed a theoretical model that:

1) considers the codec specifications (i.e. timings and compressed data size), and the network conditions
2) emulates the tele-immersion pipeline described above
3) calculates the projected frame rate, and the minimum and maximum end-to-end latency.

Since the objective of this paper is to ascertain the suitability of each codec in compressing textures in a realistic tele-immersion scenario, our calculations include encoding time and bandwidth for compressed mesh data, as well as a 0.05% probability for packet loss, which is considered typical [23]. Each frame is considered to be a group of four texture images of dimension 960x540 pixels, as well as the 3D data (mesh and attributes required for texturing/rendering). For simplicity, the queues at the encoder and the decoder are assumed to have a length of 1, i.e. if a frame arrives at the queue while the previous frame has not started being processed, the newest arrival replaces the older. Thus, we calculate two values for latency, minimum and maximum: minimum latency occurs when a frame waits no time at all at both queues (i.e. the queues are empty when the frame arrives, and so its processing begins immediately - first queue on Fig. 4), while maximum latency denotes maximal additional time at both queues. Since

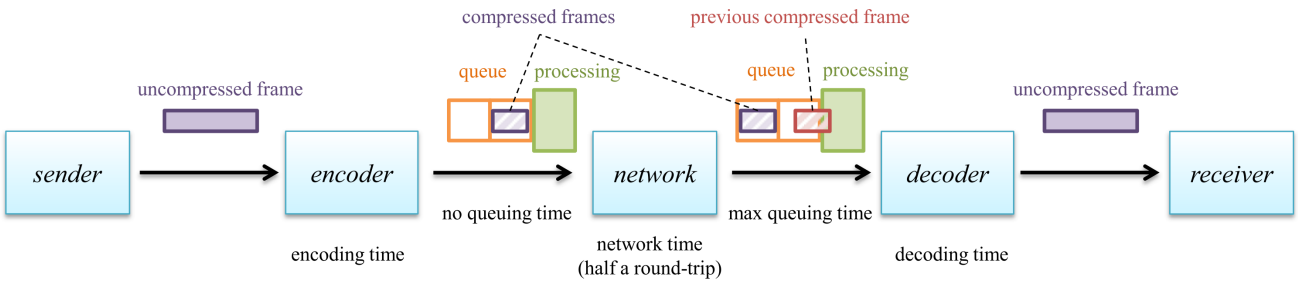

Fig. 4: Video streaming pipeline, with the flow of image data from sender to receiver (above), and the latencies incurred at each step (below). The first and second queue show examples of minimum and maximum latencies respectively.

we assume a queue length of 1, maximum waiting time at a queue corresponds to the case where the previous frame has just begun processing, and is the time required for one whole frame to be processed (second queue on Fig. 4).

B. Experimental results

We evaluate the codecs in three common network scenarios:

1) A standard network within the same country (100 Mbps bandwidth, 10 ms ping time)
2) A high speed and low latency network (1 Gbps bandwidth, 1 ms ping time)
3) A low bandwidth and high latency network (10 Mbps bandwidth, 50 ms ping time)

For each network scenario, we evaluate each of the three models of the CNN codec, and compare it with its corresponding CPU and GPU JPEG implementations, i.e. those that produce the same MS-SSIM (see Table I). Thus, with a given visual quality and within a given network, we compare the three codecs regarding frame rate and latency.

Fig. 5 presents the results of these experiments. Each sub-plot focuses on a given network scenario (columns) and a given distortion level (rows). In each, one of the three available CNN models is compared with the equivalent CPU and GPU JPEG implementations from Table I, and thus corresponds to a particular visual quality. The different sub-plots present the resultant frame rate (fps) at the decoder, and the minimum and maximum latencies involved. Each quantity follows its own scale, which is constant across all nine sub-plots.

Clearly, in the first two network scenarios, the CNN models produce both much lower frame rates and much higher latencies than either JPEG implementation. In the low-bandwidth, high-latency network scenario, where network conditions cap the minimum latency and higher compression is important because of the limited bandwidth, the CNN produces comparable, and slightly higher, frame rates compared to JPEG. However, in this scenario all three codecs produce frame rates prohibitive to real-time video.

In terms of latency, the CNN also produces the highest,

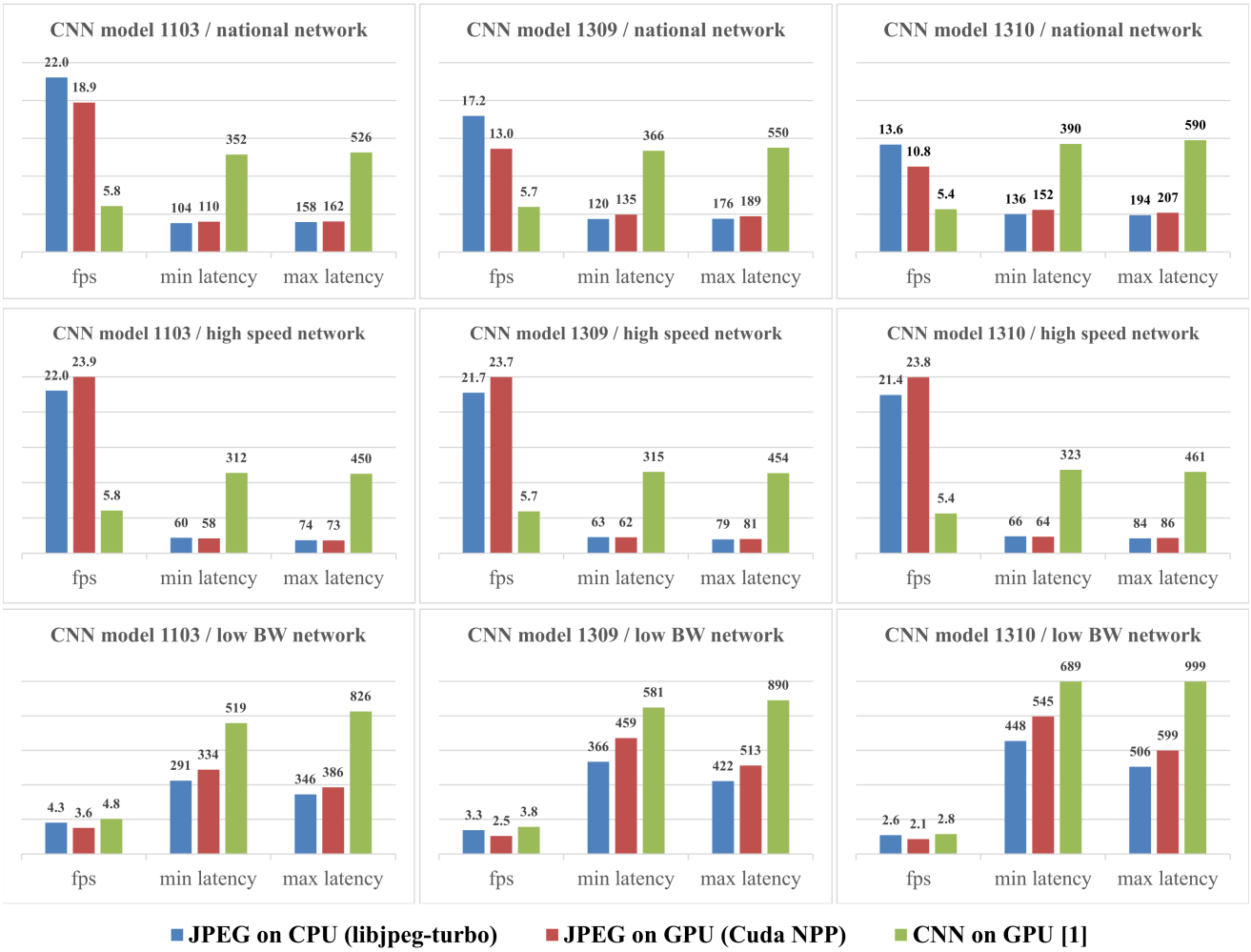

Fig. 5: Frames per second, minimum and maximum latency for different network scenarios (rows) and distortion levels (columns).

though this difference becomes less pronounced on slower networks, where the network's intrinsic latency is larger. The latencies observed for the CNN, at all three network scenarios, would impact negatively both the illusion of co-location and any interaction, though they are acceptable for (non-interactive) spectating of live events.

Comparing the two JPEG implementations to each other, it may be noted that running on a GPU produces better results on the high speed network, where the somewhat larger bitrates produced have no significant impact. However, on an average national network the CPU implementation may be preferable, due to its higher compression ratio for a given distortion level.

V. Conclusion

In the previous sections a relatively fast image compression CNN was compared to two implementations of the popular JPEG codec, in order to determine the suitability of deep learning codecs to tele-immersion. The codecs were first compared by their rate-distortion curves, where it was shown that the CNN can produce lower distortion for the same bitrate, or, equivalently, lower bitrate for the same distortion level.

The main drawback of the CNN lies in its larger computational load. Compared to the JPEG codecs, which have been well optimized since that codec's inception, the CNN exhibits much slower timings during both encoding and decoding.

Applying these results to three typical simulated networks, it is apparent that the CNN's slow timings render it unsuitable for live video streaming at high resolutions in general, and tele-immersion in particular. The two implementations of JPEG are roughly equivalent to each other, with each being best suited to particular network conditions.

The combination of high compression ratio and slow execution should make the CNN approach more suitable on scenarios characterized by very powerful encoder and decoder hardware and a low-bandwidth network. At present, this does not seem likely, as cloud and edge computing push computer systems to rely more on high speed networks and less on local hardware. However, with new, more powerful GPUs in the near future, CNNs could prove to be useful in specific cases.

In any event, it is clear that CNNs need to be made significantly faster if they are to compete with the more traditional image encoding algorithms in real-time streaming scenarios. With machine learning applied to image compression a relatively young scientific field, it is likely that significant progress may be achieved in this regard in the near future. Besides boosting their encoding/decoding performance, CNNs provide a multitude of additional benefits that can be exploited such as simultaneous multi-scale predictions for adaptive streaming scenarios, or simultaneous multi-view coding. Future work should aim in fully exploiting the potential that CNNs offer to align them with real-time interactive streaming scenarios.

References

[1] F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. Van Gool, "Conditional probability models for deep image compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[2] L. Theis, W. Shi, A. Cunningham, and F. Huszr, "Lossy image compression with compressive autoencoders," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/pdf?id=rJiNwv9gg

[3] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, "Deep convolutional autoencoder-based lossy image compression," *CoRR*, vol. abs/1804.09535, 2018.

[4] M. Li, W. Zuo, S. Gu, D. Zhao, and D. Zhang, "Learning convolutional networks for content-weighted image compression," *arXiv preprint arXiv:1703.10553*, 2017.

[5] K. Nakanishi, S. ichi Maeda, T. Miyato, and D. Okanohara, "Neural multi-scale image compression," *CoRR*, vol. abs/1805.06386, 2018.

[6] J. Cai, Z. Cao, and L. Zhang, "Learning a single tucker decomposition network for lossy image compression with multiple bits-per-pixel rates," *CoRR*, vol. abs/1807.03470, 2018.

[7] O. Rippel and L. Bourdev, "Real-time adaptive image compression," in *International Conference on Machine Learning*, 2017.

[8] E. Agustsson, M. Tschannen, F. Mentzer, R. Timofte, and L. Van Gool, "Generative adversarial networks for extreme learned image compression," *arXiv preprint arXiv:1804.02958*, 2018.

[9] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell, "Full resolution image compression with recurrent neural networks," *CoRR*, vol. abs/1608.05148, 2016. [Online]. Available: http://arxiv.org/abs/1608.05148

[10] D. Minnen, G. Toderici, M. Covell, T. Chinen, N. Johnston, J. Shor, S. J. Hwang, D. Vincent, and S. Singh, "Spatially adaptive image compression using a tiled deep network," in *Image Processing (ICIP), 2017 IEEE International Conference on*. IEEE, 2017, pp. 2796–2800.

[11] D. S. Alexiadis, A. Chatzitofis, N. Zioulis, O. Zoidi, G. Louizis, D. Zarpalas, and P. Daras, "An integrated platform for live 3d human reconstruction and motion capturing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 4, pp. 798–813, April 2017.

[12] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, "Performance comparison of convolutional autoencoders, generative adversarial networks and super-resolution for image compression," 07 2018.

[13] "libjpeg-turbo," https://libjpeg-turbo.org/, accessed: 2018-08-29.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 classes*. ACM, 2008, p. 16.

[15] "hzr," https://github.com/mbitsnbites/hz, accessed: 2018-08-29.

[16] "zlib," https://zlib.net/, accessed: 2018-08-29.

[17] "lz4 2.1.0," https://pypi.org/project/lz4/, accessed: 2018-08-29.

[18] "The bzip2 and libbzip2 official home page," https://www.sourceware.org/bzip2/, accessed: 2018-08-29.

[19] Z. Wang, E. P. Simoncelli, and A. C. Bovik, "Multiscale structural similarity for image quality assessment," in *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2. Ieee, 2003, pp. 1398–1402.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[21] N. Zioulis, D. Alexiadis, A. Doumanoglou, G. Louizis, K. Apostolakis, D. Zarpalas, and P. Daras, "3d tele-immersion platform for interactive immersive experiences between remote users," in *2016 IEEE International Conference on Image Processing (ICIP)*, Sept 2016, pp. 365–369.

[22] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the tcp congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.

[23] D. Zhang and D. Ionescu, "Reactive estimation of packet loss probability for ip-based video services," *IEEE Transactions on Broadcasting*, vol. 55, no. 2, pp. 375–385, 2009.