Ivory: Learning Network Adaptive Streaming Codes

Salma Emara*, Fei Wang†, Isidor Kaplan‡, Baochun Li§
Department of Electrical & Computer Engineering
University of Toronto
*salma@ece.utoronto.ca, †silviafey.wang@utoronto.ca, ‡isidor.kaplan@utoronto.ca, §bli@ece.toronto.edu

***Abstract*—With the growing interest in web services during the current COVID-19 outbreak, the demand for high-quality low-latency interactive applications has never been more apparent. Yet, packet losses are inevitable over the Internet, since it is based on UDP. In this paper, we propose *Ivory*, a new real-world system framework designed to support network adaptive error control in real-time communications, such as VoIP, using a recently proposed low-latency streaming code. We design and implement our prototype over UDP that can correct or retransmit lost packets conditional on network conditions and application requirements.**

To maintain the highest quality, *Ivory* attempts to correct as many lost packets as possible on-the-fly, yet incurring the smallest footprint in terms of coding overhead over the network. To achieve such an objective, Ivory uses a deep reinforcement learning agent to estimate the best coding parameters in real-time based on observed network states and experience learned. It learns offline the best coding parameters to use based on previously observed loss patterns and takes into account the round-trip time observed to decide on the optimum decoding delay for a low-latency application. Our extensive array of experiments shows that Ivory achieves a better trade-off between recovering packets and using lower redundancy than the state-of-the-art network adaptive streaming codes algorithms.

I. Introduction

With the current COVID-19 outbreak worldwide and in a "new normal" world, the usage of low-latency interactive applications has been a daily routine. In particular, with the move of academic conferences, graduation ceremonies, corporate all-hands meetings, and online gaming to online conferencing platforms, there exists an urgent demand for a high-quality low latency interactive applications to offer an online experience that resembles in-person meetings.

Naturally, quality of a video or audio conference depends on two main factors: packet losses and latencies. Packet erasure (loss) is inevitable at the network layer for an end-to-end communication over the Internet. Two **main** implemented approaches to control end-to-end errors introduced by the network layer include: Automatic repeat request (ARQ) and forward error correction (FEC). Both methods work on recovering lost data packets due to unreliable links or congested networks. Nevertheless, using ARQ for real-time applications is not appropriate, since the time for retransmitting a packet may go beyond tolerable. Specifically, the process of retransmitting a lost packet requires 3 $\times$ the one-way delay (2 $\times$ the one-way delay to transmit and acknowledge its loss, then another one-way delay to retransmit). This 3-way delay aggregate is required to be below 150 ms to meet the requirement by ITU for interactive applications [1]. Since the minimum possible aggregate delay between two diametrically opposite points on the globe is at least 200 ms [2], it is possible to have two distant nodes with one-way delay greater than 50 ms. In this case, ARQ is impractical to use.

On the other hand, FEC schemes are applicable to low-latency communications between global users, since no re-transmissions are required. FEC adds redundant information to help recover lost packets at the receiver. For interactive applications, block and convolutional codes, such as Reed-Solomon (RS) and Random Linear Convolutional (RLC) codes, respectively can be used to recover packets quickly as in [3]–[6].

One class of FEC schemes is *streaming codes* (e.g., low-latency FEC schemes), which is a class of decoding delay-constrained codes that are able to reconstruct earlier lost packets without waiting to correct all the losses. Raptor codes [7], RaptorQ codes [8], randomized linear codes [9] and others [10], [4] are all examples of streaming codes.

However, low-latency FEC schemes suffer from high bandwidth-overhead due to the transmission of redundant information. For example, Google's first attempt with XOR-based FEC in QUIC [11], which is a fast transport layer protocol for applications requiring speedy service, demonstrates that the advantages of having retransmissions outweigh the advantages of using FEC since less redundancy is added to the link. However, for time-sensitive applications, such as audio-streaming conferences, we believe that low-latency FEC schemes is still required. The issue that recurs is the coding overhead.

To the best of our knowledge, [12], [13] represented the most recent work in the direction of deploying FEC schemes for interactive applications, such as VoIP. To get the best of both worlds, [12], [13] proposed a new low-latency streaming code that (i) corrects both arbitrary and bursty erasures subject to a "fixed" decoding delay, which is the maximum time required to recover a lost packet and (ii) adapts the coding parameters on-the-fly to increase or decrease redundant information. However, the heuristic in [12], [13] assumes that high loss rate is due to unreliable networks and may therefore inefficiently use scarce bandwidth resources by introducing more redundant packets than necessary to correct lost packets.

Given the trade-off between quickly recovering lost packets at the receiver and efficiently using the network bandwidth, in this paper, we propose *Ivory*, a real-world system design and implementation prototype designed for low-latency conferencing applications that can be used to better support interactive applications. *Ivory* uses the state-of-the-art low-latency streaming code at its core for delivering interactive audio packets over network links. It adapts the coding parameters in real-time to achieve (i) acceptable quality (ii) while minimizing the coding overhead and retransmissions incurred on the network and (iii) bounding the decoding delay.

In order to determine the choices of coding parameters and adapt to time-varying network environments, *Ivory* utilizes a deep reinforcement learning agent to choose the best coding parameters for the low-latency streaming code in [12], [13], with both offline and online learning. The deep reinforcement learning agent in *Ivory* is designed to make significant improvements over existing heuristics in [12], [13], by taking into account the history of loss patterns beyond a small observation window as well as observed round-trip time (RTT). Existing heuristics in [12], [13] ignored the observed RTT and increased the network overhead by adding redundancy, leading to decreased goodput, which is the delivery rate of useful information on a communication link.

Our proposal has three main contributions. *First*, *Ivory* is designed to outperform the heuristic in [13] using deep reinforcement learning. Its performance is evaluated using an array of performance metrics, including loss rates after packet recovery, coding overhead incurred, and Perceptual Evaluation of Speech Quality (PESQ) score, which evaluates the quality of speech after recovery. *Second*, as an online learning algorithm, *Ivory* is designed to continue exploring and learning more about the network environment it is acting upon.

Lastly, we perform an extensive set of experiments using a real-world implementation prototype we developed that transmits audio packets between two nodes to compare *Ivory* to the state-of-the-art adaptive error control algorithms in [12], [13]. Our results show that *Ivory* outperforms state-of-the-art algorithms in terms of reducing loss rate to have an acceptable speech quality using lower redundancy compared to heuristics in [12], [13]. Ivory is always capable of meeting deadlines of low-latency audio application by recovering packets quickly. While heuristics in [12], [13] show less to no resilience to changes in RTT, Ivory exhibits an improved audio speech quality by reducing percentage of low speech quality intervals from 2%, which is incurred while using heuristics, to almost 0%.

II. Preliminaries: Low-Latency Streaming Codes

In *Ivory*, we utilize the low-latency streaming code proposed in [12], [13] to protect against packet losses in audio streams while maintaining low latencies. With this streaming code, illustrated in Fig. 1, a source generates an audio frame denoted by $s[i]$ every t_a seconds, where i is the sequence number of the frame. For simplicity, we assume that every source packet $s[i]$ has the same size of k symbols. The encoder concatenates the source packet $s[i]$ with the parity-check packet $p[i]$, forming the network packet with size of n symbols. The network packet travels to the destination with a propagation delay d_p. Packets traveling over the Internet may be lost due to several reasons, which lead to different loss patterns. For example, congestion can lead to bursty loss patterns while unreliable wireless links can lead to arbitrary loss patterns. The difference between bursty and arbitrary losses is illustrated in Fig. 2.

TABLE I
Definitions of key notations.

Notation	Definition
T	decoding delay constraint
B	maximum run length of recoverable burst erasures
N	maximum number of recoverable arbitrary erasures
n	codeword size
k	raw data size
$\text{span}(e_{\mathcal{W}})$	span of the 1st and last packet lost in a sliding window
$\text{wt}(e_{\mathcal{W}})$	number of packets lost in a sliding window
$\text{C}(T, B, N)$	coding rate $\frac{k}{n}$

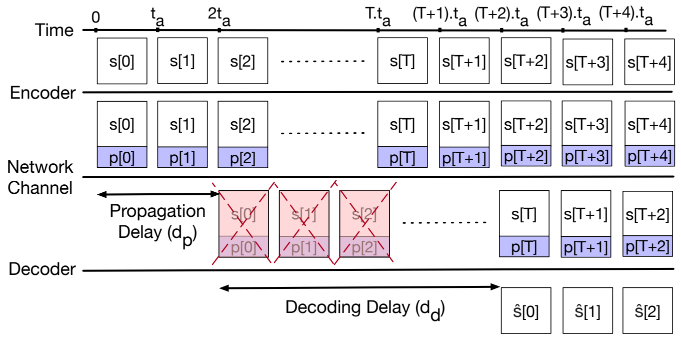


Fig. 1. The general framework of FEC streaming codes to recover lost packets.

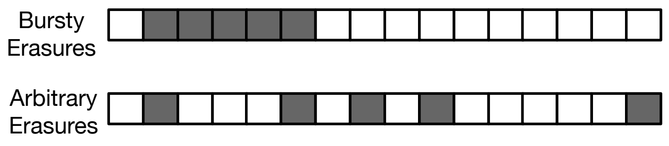


Fig. 2. An illustration of the difference between bursty and arbitrary erasures, with packet losses marked in dark color.

At the destination, the decoder aims to use the parity-check packets out of the received network packets to reconstruct lost packets. The decoder tolerates a maximum decoding delay of T packets, which is equal to $T \times t_a$ seconds, where t_a is the interarrival time between packets. In other words, the maximum decoding delay of one packet is $T \times t_a$. Therefore, T should be carefully chosen so that the resultant playback delay and propagation delay $= T \times t_a + d_p$ is less than 150 ms — as required by ITU for interactive applications [1].

In the low-latency streaming code we utilized, parity-check packets are generated in a way to recover older packets with sooner deadlines than later source packets. This is not the case with conventional FEC codes such as the Reed-Solomon code, where the recovery of all source packets occurs simultaneously [2]. For example, in Fig. 1, $s[0]$ can be recovered after

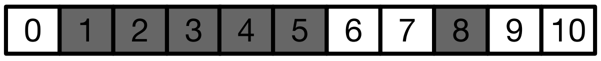


Fig. 3. An example of a window of 11 packets that can be encoded using $(T, B, N) = (11, 8, 1)$ or $(11, 6, 6)$ or $(11, 8, 6)$ to recover the losses.

receiving $s[T]+p[T]$ and $s[1]$ can be recovered after receiving $[T+1]+p[T+1]$, where both $s[0]$ and $s[1]$ were recovered within the same decoding delay $d_d = T \times t_a$.

In the low-latency streaming code we utilize, the maximum value of T is 11 packets, and coding parameters (B, N) satisfy the condition $T \geq B \geq N \geq 1$. The power of this code lies in the fact that it can correct both length-B burst packet losses and N arbitrary packet losses, with a maximum decoding delay of $T \times t_a$ seconds. Hence, (B, N) are to be carefully chosen according to network observations.

The coding rate $\frac{k}{n}$ of this low-latency streaming code is $\mathrm{C}(T, B, N)$ in Eqn. (1), and the redundancy is $1 - \frac{k}{n}$, where k is the size of raw data and n is the codeword size.

$$\mathrm{C}(T, B, N) \triangleq \frac{T - N + 1}{T - N + B + 1}. \tag{1}$$

The sliding windows are represented as $\mathcal{W} = \{j-T, j-T+1, \ldots, j\}$ of size $T+1$ such that $j \leq i$, where i is the sequence number of the packet. For each sliding window, the packet loss pattern $e_{\mathcal{W}} \triangleq (e_{j-T}, e_{j-T+1}, \ldots, e_j) \in \{0, 1\}^{T+1}$, and an element of $e_{\mathcal{W}}$ equals 1 if and only if the corresponding packet is lost. Let $\mathrm{wt}(e_{\mathcal{W}}) \triangleq \sum_{\ell \in \mathcal{W}} e_\ell$ and

$$\mathrm{span}(e_{\mathcal{W}}) \triangleq \begin{cases} 0 & \text{if } \mathrm{wt}(e_{\mathcal{W}}) = 0, \\ p_{\text{last}} - p_{\text{first}} + 1 & \text{otherwise,} \end{cases}$$

be the *weight* and *span* of $e_{\mathcal{W}}$ respectively, where p_{first} and p_{last} denote respectively the indices of the first and last non-zero elements in $e_{\mathcal{W}}$. In Table I, we summarize the definitions of key notations used in this manuscript.

To illustrate, Fig. 3 shows a window of 11 packets, where dark packets are lost. It has a packet loss pattern with $\mathrm{wt}(e_{\mathcal{W}}) = 6$ and $\mathrm{span}(e_{\mathcal{W}}) = 8$. If $T = 11$, these source packets can be encoded at the decoder using $(B, N) = (8, 1)$ or $(6, 6)$ or $(8, 6)$.

III. Why Reinforcement Learning?

In a nutshell, *Ivory* uses a deep reinforcement learning (DRL) [14] agent to make strategic decisions on the coding parameters (T, B, N) of the low-latency streaming code. In the midst of a large body of existing works on the use of DRL to solve a wide variety of engineering problems in networking, the questions that may naturally arise are: Is it yet another piece of work that uses DRL? Why are we using DRL to adapt coding parameters?

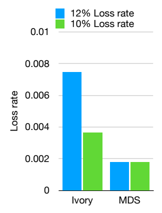
A. Issues with Existing Heuristics

Authors in [12], [13] design a heuristic algorithm that estimates the values of (B, N), which we refer to as the *heuristic*. Based on the observed loss patterns, the receiver runs the heuristic and sends recommendations to the sender on values of (B, N) used to encode source packets. The *heuristic* proposed in [12], [13] can have $B \geq N$; however, they also show similar behavior to maximum-distance separable *MDS*-adaptive streaming code, which assigns B and N to the same value of the maximum observed $\mathrm{span}(e_{\mathcal{W}})$. MDS codes are used by Skype [15].

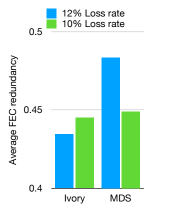
The *heuristic* and *MDS* show high recovery ratios of lost packets, which is the ratio of packets recovered to the packets lost. **However, there is always a trade-off between the packet recovery ratio, redundancy added and one-way delay observed.** An algorithm will have to add more redundant information to gain a lower loss rate. We observe that the *heuristic* and *MDS* exhibit low loss rates at the cost of high redundancy. High redundancy decreases goodput, and if the bandwidth was limited, it will backfire and reduce the quality of the interactive application. Another trade-off is between recovery packets and maximum decoding delay. The more redundant information the network adaptive streaming code can add, the higher the one-way delay will be, since $T \geq B \geq N \geq 1$. For example, to recover 6 packets in a sliding window, you need at least T to be 6.

For example, in audio conferencing applications, the acceptable packet loss rate is between 3% and 5% [16]. Hence, fixing the rules as in [12], [13] to correct the maximum number of losses may not be optimal as it will sacrifice on the delay observed and overuse bandwidth with added redundancy.

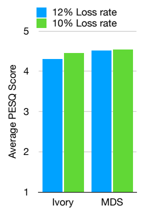
To demonstrate the importance of having a flexible trade-off between redundancy and recovery ratio, we test *Ivory* and *MDS* from [12], [13] over a loss pattern generated with 10% and 12% loss rates. We use PESQ score [17]: a metric that ranges between 1 – 5, where a higher score means a better speech quality, to observe the effect of packet losses on audio packets. In Fig. 4, we observe the loss rate, average FEC redundancy and PESQ score for an audio call that lasted 10 minutes over 10% and 12% loss rates. In Fig. 4a, Ivory shows a higher loss rate than MDS; however, in Fig. 4b, MDS exhibits 5% to 20% higher redundancy than Ivory for 1% higher PESQ score for MDS in Fig. 4c for Ivory and MDS. Average PESQ scores for both Ivory and MDS are higher than the acceptable value for speech(3.6). Indeed, the efficient bandwidth utilization of Ivory is saving bandwidth for other flows sharing the same link.



(a) The average loss rate of Ivory and MDS



(b) The average FEC redundancy of Ivory and MDS



(c) The average PESQ score of Ivory and MDS

Fig. 4. Average loss rate, FEC redundancy and PESQ score of Ivory and MDS in 10% and 12% loss rate environments

Apart from fixing rules of heuristics to correct packet erasures, the *heuristic* and *MDS* algorithms in [12], [13] do not take the RTT into consideration. Taking RTT into consideration is important to know how much time it takes for the feedback to reach the sender to update its coding parameters. This helps because if, for example, RTT was large, we know that it would take time between sending feedback to the sender and updating the coding parameters. During this time, the behavior of the network *may* change enough that the new values of (B, N) that the sender started to use are outdated. A smarter algorithm should take RTT into consideration to ensure that their estimated values of (B, N) are not outdated.

B. How DRL mitigates issues with existing heuristics?

Learning-based network adaptive algorithms have a great potential to adapt themselves to different conditions without the need to be hand-tuned or manually changed for each environment. Therefore, the fixed set of rules or assumptions that previous heuristic algorithms incorporated could be completely replaced with a model that learns from experience rather than assumptions. It would take a model a few hours to train, but many hours to hand-tune rule-based heuristics.

Out of (i) supervised, (ii) unsupervised and (iii) reinforcement learning, the former two consider instant reward, as opposed to reinforcement learning that is sequential and far-sighted considering long-term rewards. We model network adaptive error control as a sequential-decision making process that learns a policy by adjusting its actions to achieve optimal rewards over the long-term future.

Furthermore, since DRL models do not require labeled data, it can adapt online as it observes new network states. They, indeed, can change any learned assumptions in their deep neural network. Online learning will help in unlearning bad actions that were good in previously observed environments.

IV. Design Iterations

Training a model using DRL is rarely straightforward as it consists of many design parameters such as the action and state space, the reward function, the neural network model and the DRL algorithm. This paper takes a practical approach and makes our design decisions based on hands-on experiences learned from actual experiments. We share the insights we gained from extensive experimentation to achieve the best performing model and supply sufficient reasoning.

A. Training framework

Since our design iterations are empirical and require understanding of our framework, we first describe our training framework. Fig. 5 shows our training and testing framework. Our implementation of our testbed incorporates a sender that sends encoded audio frames to the receiver over User Datagram Protocol (UDP). The DRL agent resides on the receiver, and based on observations, the agent provides feedback to the sender on the recommended coding parameters. As audio is notably more loss-and delay-sensitive in interactive video-streaming applications such as video conferencing, we carry out our experiments on audio multimedia frames.

During the training process, the sender starts by sending audio packets at size of 300 bytes and interarrival time of 10 ms, which is 240 kbit/s. These numbers are practical since existing audio codecs such as Opus audio codec have a frame duration range of 2.5 – 60 ms and bit rate range of 6 – 510 kbit/s [18]. Since our DRL agent has the **action space** of MDS-adaptive streaming code, which is $(B, N) = (i, i)$ for all distinct integers i, where $0 \leq i \leq T = 10$, *only during training*, we fix the value of T to 10 packets. This is to ensure regardless of the RTT or loss patterns observed, the DRL agent can choose from a wide variety of the coding parameters (B, N) of the FEC streaming code. T will only be varied during testing and online learning phase, as they will depend on RTT measured. Recommendations of (B, N) are sent from the receiver to sender. The sender is responsible for (i) updating the coding parameters according to recommendations from the receiver and (ii) encoding packets using the MDS FEC streaming code in [12], [13].

The receiver decodes the received packets as per the (B, N) values with which the packet was encoded. The encoder at the sender and decoder at the receiver were implemented in `C++` programming language, but since there is a lack of packages for implementing sophisticated machine learning models in `C++`, we implemented our agent in `Python` using `PyTorch` on the receiver.

Communication between the decoder and the DRL agent happens at local ports over Transmission Control Protocol (TCP). The decoder measures observed loss patterns and sends these measurements to the agent. The agent runs the DRL algorithm to train a model that recommends (B, N) values to the sender.

To have all our training and testing environments emulate real-world networks, we run the sender and receiver either locally (simulating different RTT using `tc` in linux) or on two distant devices, and simulate different loss patterns between the two nodes according to pre-generated loss traces. We design a testbed that helps in running several network adaptive error control algorithms such as MDS in [12], [13], and several versions of our agent on the same network conditions for accurate and reproducible evaluations.

We generated different loss patterns. The first is by setting the bottleneck bandwidth between the two devices to 960 kbit/s. We run different calls between the two devices, which is part of the UDP cross-traffic, where each call has a throughput of 240 kbit/s. Call requests are modeled using Poisson distribution, which implies exponentially distributed call interarrival times of a mean of 3 minutes. The service time is exponentially distributed with a mean of 5 minutes. The call request rate is varied between 1 – 5 calls/s. We change the call request rate every μ seconds by $\{+0.2, -0.2\}$ to help vary the loss patterns experienced. We save the loss patterns experienced in trace files to test and compare different algorithms over the same setup. Such trace file generates a wide variety of loss

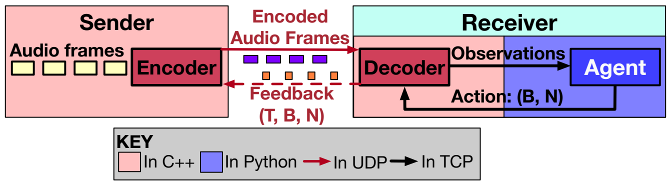


Fig. 5. Training and testing framework

ratios over windows of 1000 packets. With no coding, packet loss ratio varies between 5% to 95%. We use two different μ values $\{1, 20\}$.

Another set of 10 different loss patterns is generated randomly, where erasures are independent and identically distributed. We vary the probability of an erasure in each pattern by having different loss rates. Loss rates vary from $2\% - 20\%$.

In another setup, we connect the source and destination over the same Wi-Fi network with an average capacity of 30 Mbit/s subject to UDP cross traffic *offered load* of 40% – constant 12 Mbit/s. Since the bandwidth of the Wi-Fi network is fluctuating greatly, we can observe losses with an *offered load* of 40%.

B. System Design: Iterations

Our exploration of designing the DRL agent began with a first cut, which may help us identify weak and strong spots and locate potential areas for improvement. Our first cut used Cross-entropy method as the DRL algorithm to train Ivory.

To be able to represent the environment, we choose the following as the **state space** of the DRL agent:

1) L_{rate}: the ratio of packets lost to packets transmitted;
2) $\text{span}(e_{\mathcal{W}})_{max}$ $(0 \leq \text{span}(e_{\mathcal{W}})_{max} \leq 10)$: maximum span of erasures in any sliding window of 11 packets observed in a step;
3) $\text{wt}(e_{\mathcal{W}})_{max}$ $(0 \leq \text{wt}(e_{\mathcal{W}})_{max} \leq 10)$: maximum number of arbitrary erasures observed in any sliding window of 11 packets observed in a step;
4) $1 - \text{C}(T, B, N)$ $(0 \leq 1 - \text{C}(T, B, N) \leq \frac{10}{11})$: average redundancy, where $\text{C}(T, B, N) = \frac{k}{n}$ is the coding rate in a step as defined earlier;
5) Q_{obs} $(0 \leq Q_{obs} \leq 1)$: ratio of packets recovered from lost packets in a step.

Given that $\text{span}(e_{\mathcal{W}})_{max}$ and $\text{wt}(e_{\mathcal{W}})_{max}$ are not in the same range as L_{rate}, $1 - \text{C}(T, B, N)$ and Q_{obs}, we normalize the input state space by dividing $\text{span}(e_{\mathcal{W}})_{max}$ and $\text{wt}(e_{\mathcal{W}})_{max}$ by 10, to ensure the initial neural network emphasizes on all elements of the state space equally.

We choose the time of the step in an episode to be double the time taken for the recommended (B, N) to be observed on an encoded packet at the receiver, *i.e.*, $2 \times$RTT. In the first half of the step, the agent observes the effect of the previous step, and in the second half, the effect of the newly used (B, N) in the step is detected.

Fundamentally, to solve an RL problem, the task has to be modeled as a Markov Decision Process (MDP) [14]. Since different network environments may not exhibit the Markov property, we need a way to retain state information over time to model our problem as MDP. The usage of a recurrent structure in our **neural network** will mitigate the issue as discussed in the literature [19]. At any point in time, the DRL agent does not exactly know the loss rate, pattern and round-trip time. Therefore, aggregating observations over past steps in a recurrent neural network will help model our problem as an MDP. Ivory uses a long short-term memory (LSTM) network with 64 hidden units and 2 hidden layers to help model the network adaptive error control problem as an MDP.

Our episode is an aggregation of decisions taken over a sequence of 1000 packets. To represent the virtue of decisions to the agent, we use the **reward function** in Eqn. 2, where $Q_{episode}$ is the ratio of recovered packets throughout the episode, $1 - \text{C}(T, B, N)$ is the average redundancy observed in the episode, and α is a constant to vary the significance of redundancy. Intermediate steps obtain 0 reward, except for the last step's reward in Eqn. 2.

$$r = Q_{episode} \times (1 - \alpha \times (1 - \text{C}(T, B, N))) \qquad (2)$$

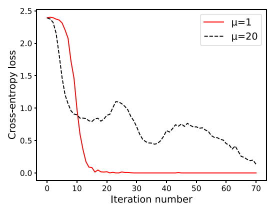
1) Effect of loss pattern in training process: Initially, we train two models using Cross-entropy method with $\alpha = 0.4$. The batch size is set to 16 episodes, and we train over the elite/top $75th$ percentile episodes according to the Cross-entropy method [20]. We train each model over different loss patterns generated from different $\mu = \{1, 20\}$. Fig. 6 shows the results of this training experiment. In Fig. 6a, we observe a quicker convergence with $\mu = 1$ compared to $\mu = 20$ in terms of cross-entropy loss. Also, in terms of reward, we observe a more stable reward per episode and convergence towards the end in Fig. 6b.

Having a small μ made the request rate of calls change quickly and hence smoothing the effect of high fluctuations of request rates on loss rates. This resulted in a much slower change in Frame Loss Rate (FLR) of no coding observed in Fig. 6c compared to the much steeper changes in FLR in Fig. 6d.

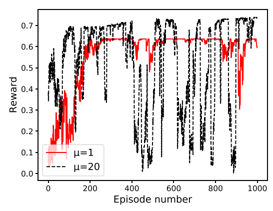
We found that whenever the loss pattern increased suddenly, the ratio of recovered packets decreased in both Fig. 6c and Fig. 6d. In other words, with higher fluctuations as in Fig. 6d, we could see slower convergence in $\mu = 20$, as compared to lower fluctuations in Fig. 6c, where we see faster convergence. In a nutshell, changes in loss patterns observed yield different convergence properties. *Hence, to have a stable performance, training on a loss pattern that changes steadily (not suddenly) is important.*

2) Effect of α: One of the most beneficial ideas in DRL is that it learns a policy according to the observed states and rewards. Whenever the reward function changes to favor one metric, the policy changes. This is the benefit of the α in Eqn. 2. With changes in α, we can emphasize or de-emphasize the importance of saving bandwidth or using less parity.

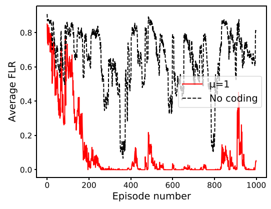
To better understand the effect of α in the reward function in Eqn. 2, we fix $\mu = 1$ and change $\alpha \in \{0.4, 0.6, 0.8, 1\}$. In Fig. 7, we show the test results of the 4 different models



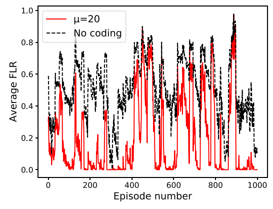
(a) Cross-entropy loss vs. iteration number of models trained over $\mu = 1$ and $\mu = 20$.



(b) The rewards vs. episode number of models trained over $\mu = 1$ and $\mu = 20$

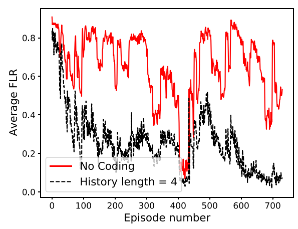


(c) The average FLR vs. episode number as we train a model over $\mu = 1$

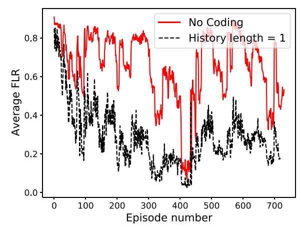


(d) The average FLR vs. episode number as we train a model over $\mu = 20$

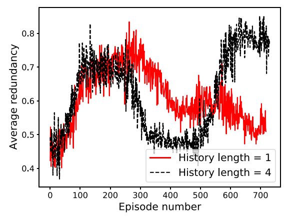
Fig. 6. Comparing two models trained using $\mu = 1$ and $\mu = 20$ using Cross-entropy method



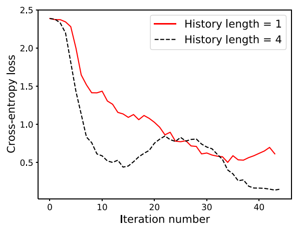
(a) The average FLR vs. episode number for history length of 4



(b) The average FLR vs. episode number for history length of 1

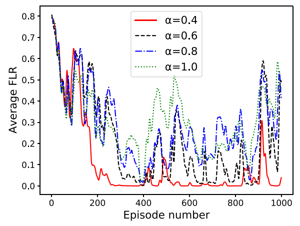


(c) The average redundancy vs. episode number

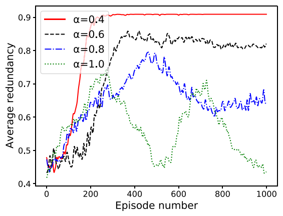


(d) The cross-entropy loss vs. iteration number

Fig. 8. Experimental results showing the effect of history length on the adaptivity and convergence of a model



(a) The average FLR vs. episode number for different α values



(b) The average redundancy vs. episode number for different α values

Fig. 7. Comparing the training process of 4 different models trained using different $\alpha \in \{0.4, 0.6, 0.8, 1\}$

trained with different values of α. Fig. 7a and 7b show the average FLR and FEC redundancy per episode as we test the models over loss pattern with $\mu = 1$. We observe that the model trained with $\alpha = 0.8$ is more adaptive to changes in loss patterns. For example, at around episode number 500 in Fig. 6c, the no coding losses start to decrease gradually, and the redundancy with $\alpha = 0.8$ starts to decrease too.

When $\alpha = 0.4$, the average redundancy over an episode did not fluctuate, which shows that this model decided that the optimum behavior is to be conservative and always chose $(B, N) = (10, 10)$ to correct all possible losses. However, a higher value of $\alpha = 1$ yields a model with lower attention to recovering losses and more regard to redundancy as seen in around episode 500, where the redundancy chosen by the model trained with $\alpha = 0.8$ is higher than the model trained with $\alpha = 1$.

3) Effect of history length: In Fig. 8, we study the effect of the history length of the state space fed to the neural network. This study focuses on convergence properties and performance of the agent when a bigger state space is fed to the neural network. In this experiment, we set $\alpha = 0.8$ to be able to observe changes in redundancy with changes in loss rates. We train the agent using Cross-entropy method over the loss pattern generated with $\mu = 1$. We observe the offline training properties of the agents with history length of 1 and 4 in Fig. 8.

In Fig. 8d, we observe the convergence of the model having history length of 4 was faster. The rewards of both the models converged to the same number, but the model with history length of 4 reaches the maximum reward faster. Fig. 8a and Fig. 8b show the changes in average loss rate as the models with history length of 4 and 1 train respectively. We observe the model with history length of 4 has a quicker increasing gap between the no coding average FLR and the agent's FLR.

In terms of adapting the redundancy with time, Fig. 8c shows that the model trained with history length of 4 adapts faster to changes in loss patterns. We observe that just before episode 300, the average packet loss rate decreased from 0.80 to 0.36 and the redundancy of that model decreased a lot to adapt to this change; however, the model with history length of 1 took almost triple the time to drop its redundancy. We see the same adaptive behavior for the model with history length of 4 at episode 450.

4) Effect of the value of T – maximum decoding delay: During the deployment stage, Ivory requires to know the RTT and based on it choose the value of T to be able to meet the requirement of 150ms one-way delay for interactive applications [1].

Ivory in the first step would choose no coding parameters until it measures the RTT. According to the RTT and the packet interarrival time, it chooses a value of T between 1 and 10. T is calculated as $T = \frac{150-RTT}{t_a}$, where t_a is the packet interarrival time. The result T is rounded down to the nearest whole number between 1 and 10.

If T is chosen to be below 10, and Ivory decides on action $(B, N) > (T, T)$, then we would clip it to be the maximum value of B and N allowed, *i.e.* $(B, N) = (T, T)$. This is because Ivory is trained on the whole action space of $(B, N) = (i, i)$ for all distinct integers i where $0 \leq i \leq T = 10$. This would indeed sacrifice loss rate for decreased decoding delay.

5) Online training phase: During the online learning phase, we continue running the Cross-entropy method algorithm on observed episodes. However, the main difference between offline and online learning is the change in the value of T. The main issue with this is that if T was chosen at the beginning of the connection to be smaller than 10, and we cap all actions of the agent to be $(B, N) \leq (T, T)$, then when continuing online training, our policy would be drifting to choose between $(0, 0) \leq (B, N) \leq (T, T)$. This will be an issue if we start another connection that handles a higher value of the previously chosen T. For this reason, online learning is considered mainly beneficial only if it starts initially from the offline trained model.

Another benefit of starting a connection from the offline-trained model is that during online training, overfitting may occur. Recall that during offline learning, the model is trained over a wide variety of loss patterns and RTTs. Overfitting the model to one environment will result in a model that is behaving well to a particular environment and not others. This is why in online learning, we start our online training from the offline-trained model.

To ensure Ivory does not behave poorly initially when it experiences a newly observed environment, we use a decaying ratio of MDS actions. These actions are considered guidance actions if Ivory is not performing well. Otherwise when Ivory observes improved rewards, dependence on MDS actions decrease. Alg. 1 summarizes our online learning algorithm.

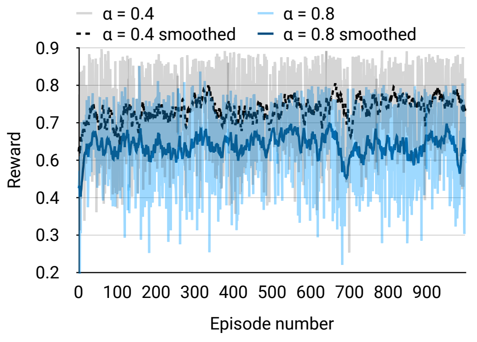
Algorithm 1 Online training of Ivory

```
1: procedure DeployIvory
2:     Use the pre-trained network parameters
3:     Initialize $\epsilon \leftarrow 0.5$
4:     for all batches do
5:         for all steps $\in$ episodes $\in$ batch do
6:             With probability $\epsilon$, take MDS action
7:             Otherwise take pre-trained network action
8:             Decay $\epsilon$
9:         Train the pre-trained model and/or the CEM ensemble model on 75% elite episodes every 16 episodes
```

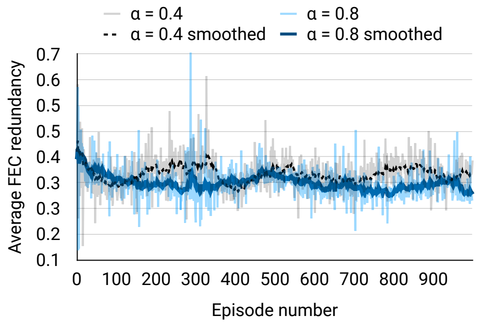
V. Performance Evaluations

1) Offline Training of Ivory: To evaluate *Ivory*'s performance, we use our testbed explained in Sec. IV-A. Ivory used in this section is trained on random loss rates bounded between 2% and 20%, similar to real world loss rates. As we train Ivory, we vary the RTT every episode between 10ms and 50ms. Each episode runs over a different RTT and a different loss rate that are changed using `tc` in Linux kernel.

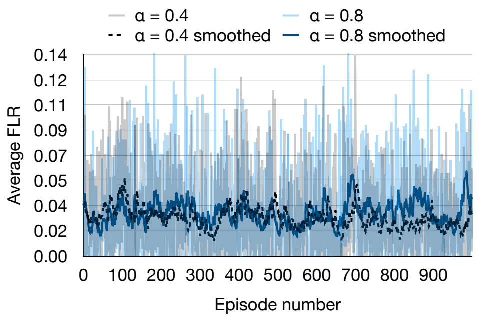
In Fig. 9, we show the training curves for Ivory. Ivory is trained over an erasure pattern that changes the loss rate every episode (i.e. 1000 packets). In Fig. 9a, we show the reward value vs. episode number. As discussed earlier, the model trained with $\alpha = 0.4$ will show higher reward, since it penalizes less the redundancy factor in the reward function compared to $\alpha = 0.8$. This also explains the low average FLR and low average FEC redundancy of the model trained using $\alpha = 0.8$ in Fig. 9cand 9b, compared to the model using $\alpha = 0.4$.



(a) The reward value vs. episode number



(b) The average redundancy vs. episode number

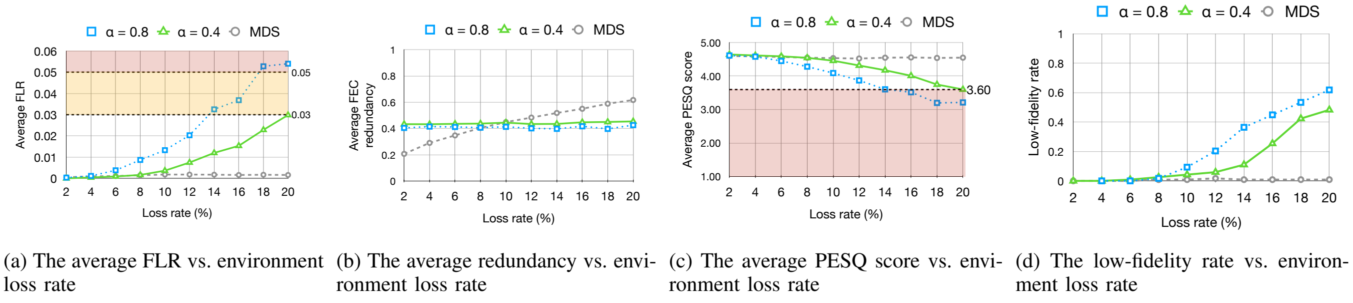


(c) The average loss rate vs. episode number

Fig. 9. Comparing *Ivory* training process over $\alpha = 0.4, 0.8$ over the bounded erasure pattern

2) Varying α: After offline training, we test Ivory on different loss patterns. We evaluate Ivory trained using $\alpha = 0.4, 0.8$ and MDS over 10 min call periods in Fig. 10. Each call experiences different loss rate varying from 2% to 20%, with a difference of 2% between each experiment. We fix the RTT to 60 ms, and the interarrival time between packets was set to 5 ms. Therefore, T was chosen to be 10 (the maximum value), to meet the one-way delay requirement.

In Fig. 10a and 10b, we plot the average FLR and FEC



(a) The average FLR vs. environment loss rate (b) The average redundancy vs. environment loss rate (c) The average PESQ score vs. environment loss rate (d) The low-fidelity rate vs. environment loss rate

Fig. 10. Comparing *Ivory* trained over $\alpha = 0.4, 0.8$ with *MDS* over the testing environment of loss rates varying from 2% to 20% in terms of average FLR, average FEC redundancy, PESQ score and low-fidelity rate.

redundancy, respectively, for each 10 min call over different loss rate. While in Fig. 10c and 10d, we plot the PESQ score and low-fidelity rate of the speech received at the receiver after decoding. To calculate the PESQ score and low-fidelity rate, we first measure the PESQ score of every 10 seconds received by comparing audio of decoded packets at the receiver with the corresponding original audio. The average of these 10 second PESQ scores is plotted as the average PESQ score in Fig. 10c, and the ratio of these 10 second PESQ scores that is below 3.6 is the low-fidelity rate as plotted in Fig. 10d.

In Fig. 10a and 10b, Ivory trained using $\alpha = 0.4$ tends to trade recovery ratio for reduced redundancy when loss rate is higher than 8%. This led to low FLR for loss rates less than 8% and high FLR rates elsewhere. However, in all of these cases, the PESQ score of Ivory trained over $\alpha = 0.4$ did not get affected with increase in FLR, and it was kept above 3.6 as shown in Fig. 10c. In summary, Ivory trained using $\alpha = 0.4$ tends to provide a better trade-off between loss rate and redundancy, by keeping the PESQ score in Fig. 10c acceptable. However, for loss rates bigger than 10%, Ivory tends to have more 10s sessions with PESQ scores lower than 3.6, as reflected in Fig. 10d. This is a result of Ivory's preference to reduced loss rates than redundancy.

For Ivory trained using $\alpha = 0.8$, the loss rate is certainly higher and redundancy is lower than Ivory trained using $\alpha = 0.4$. However, Ivory trained using $\alpha = 0.8$ seems to not fit the audio conferencing applications as loss rates crossed 5% when packet loss rates were 18% or more. This indeed degraded the performance of audio as observed in Fig. 10c with PESQ scores dropping below 3.6.

On the other hand, MDS performed well in terms of recovering more packets than Ivory trained using $\alpha = 0.4$, especially when loss rates were high. This owes to the fact that MDS was using a conservative heuristic to save more packets without much regard to redundancy. For example, at loss rate of 12%, MDS used 5% more redundant packets, and hence 5% more bandwidth than Ivory trained with $\alpha = 0.4$, to bring the quality of speech from an acceptable rate to another acceptable rate, which is unnecessary.

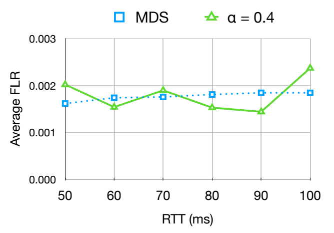
If we were to use retransmissions of lost packets, we will not be able to meet the deadline requirements of low-latency applications, as a successfully retransmitted packet would arrive after $60 \times 3 = 180ms$, after the one-way delay required by ITU, which is 150ms. However, for RTT below 50ms, we do still advocate for retransmissions, since the added redundancy of FEC would typically be higher than the added redundancy of retransmissions. For example, if RTT was 20ms and packet loss rate was 10%, if we were to retransmit a lost packet only once, the added redundancy would be just 10%, and it would still meet the delay requirements of interactive applications.

In conclusion, Ivory has an adaptive behavior as you change α. This makes Ivory flexible in terms of suiting different applications that have different requirements. Ivory using $\alpha = 0.4$ performed well in terms of loss rate, but not better than MDS, but was capable of having lower redundancy than MDS to save bandwidth. We do not claim Ivory always has a better trade-off, but we claim that Ivory has an adaptive trade-off that can be changed to well-suit an application.

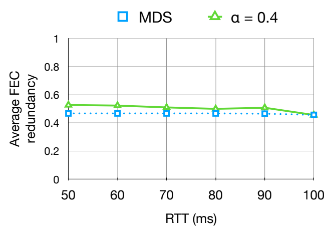
3) Varying RTT: Apart from adapting Ivory's trade-off, we are concerned with meeting the deadline of packet reception by the application to ensure quality of application is not hindered. To achieve this, Ivory considers RTT into its choice of T. In the next experiment, we use the loss rate of 10% since MDS and Ivory trained using $\alpha = 0.4$ showed a similar behavior in terms of average FLR and FEC redundancy in the previous experiments in Fig. 10. Therefore, any changes in behavior is to be attributed to change in RTT. The packet interarrival times is 10 ms and RTT varies from 50 ms to 100 ms. Audio calls run for 10 min each over different RTT. We compare the behavior of MDS and Ivory trained using $\alpha = 0.4$ in protecting packets, reducing redundancy and meeting deadlines. However, since MDS is always choosing $T = 10$, it would not meet the 150 ms deadline for 60 ms RTT and more. We could stop here and say MDS is not suitable for RTT larger than 60 ms, however, we adjust MDS's algorithm to study the effect of changing RTT on the performance.

In Fig. 11a, MDS and Ivory both show similar loss rates; however, Ivory is using slightly higher redundancy than MDS in Fig. 11b. Similar loss rates did translate to similar PESQ scores as in Fig. 11c. However, Ivory shows almost 0% low-fidelity rates in Fig. 11d compared to almost 2% low-fidelity rate for MDS.

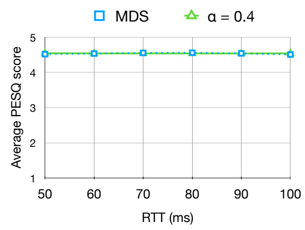
It is difficult to find out why Ivory is having lower low-



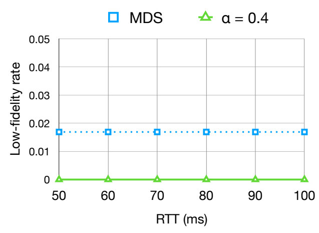
(a) The average FLR vs. environment loss rate



(b) The average redundancy vs. environment loss rate



(c) The average PESQ score vs. RTT



(d) The low-fidelity rate vs. RTT

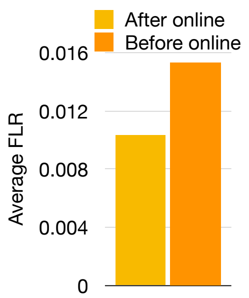
Fig. 11. Comparing average FLR, FEC redundancy and PESQ scores of *Ivory* trained over $\alpha = 0.4, 0.8$ and of *MDS* over the testing environment of RTT varying from 50 ms to 100 ms

fidelity rates, but one main reason is that Ivory was trained over different RTTs. Hence, Ivory is trained to be aware of feedback arriving late or out-dated. This is why with high RTTs, Ivory was capable of perhaps acting earlier than MDS to be able to correct future losses in packets.

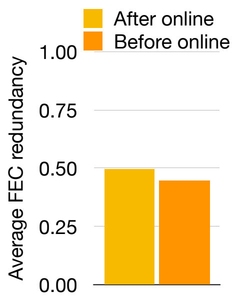
4) Online training of Ivory: Online learning is another feature of Ivory, which can be turned off or on if the receiver operating on Ivory has limited computational resources. Ivory can continue to learn online as it adapts to new loss patterns, or it can improve as it observes more samples of previously observed loss patterns.

To observe the behavior of Ivory as it learns online, we use Ivory's model trained using $\alpha = 0.4$ and make it observe the environment with 16% loss rate for 10 minutes. After online training for 10 minutes against the environment with 16% loss rate, we test Ivory for 10 minutes over the same environment. In Fig. 12, we compare the changes in average FLR, FEC redundancy, PESQ and low-fidelity rate. It is clear in Fig. 12a that Ivory is having lower FLR, yet higher than that of MDS. This is indeed a result of the slight increase in redundancy observed in Fig. 12b. However, the reduced loss rate did not significantly improve the average PESQ in Fig. 12c. This is because the improvement was targeting the huge decline in low-fidelity rate from 25% to 8%.

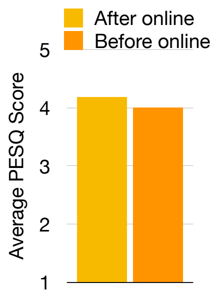
Online learning indeed benefited Ivory improvement as it trained more on 16% loss rate environment. In addition, it is noteworthy that this improvement was over the span of 10 minutes only.



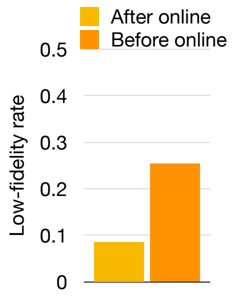
(a) The average FLR of Ivory before and after online learning



(b) The average FEC redundancy of Ivory before and after online learning



(c) The average PESQ score of Ivory before and after online learning



(d) Low-fidelity rate of Ivory before and after online learning

Fig. 12. Comparing the performance of Ivory before and after online learning over 16% loss rate environment over a 10 min call

VI. Related Work

In this field of research, there are two main areas of research investigation and effort: FEC coding schemes and adaptive FEC. There are different variations of FEC coding schemes. For non-interactive applications, Low-density parity-check (LDPC) codes [21], [22], and digital fountain codes [7], [23] are two FEC schemes that are currently used in the DVB-S2 [24] and DVB-IPTV [25] standards. Caveats of LDPC and fountain codes is that they require long wait times to receive long block lengths and computation time to encode and decode these long block lengths.

Since we give special attention to interactive audio applications, the schemes that relate most to our work are low-latency FEC streaming codes. Different variations of low-latency FEC schemes are studied in [4]–[6], [10], [15], [26] to improve interactive streaming applications. Indeed, using low-latency FEC schemes plays a major role in the success of Skype [15] and WebRTC [4]. The biggest issue with existing FEC streaming codes is that they are not optimum in correcting both bursty and arbitrary erasures, except for [12], [13], [27], [28].

Besides, there have research works on adaptive FEC algorithms for non-interactive applications on different coding streams, such as [29], [30]. Also, some works implement learning-based algorithms to learn coding parameters, such as [31]–[33]; however, they do not consider low-latency streaming codes, where decoding delay plays a major role in defining effectiveness of the solution.

To the best of our knowledge, [12], [13] is the most recent work on deploying FEC streaming codes for interactive audio conferencing. They achieve an optimum trade-off between correcting bursty and arbitrary erasures with a fixed low latency guarantee. [12], [13] also adapt the redundancy used in their coding scheme showing that they perform better than fixed coding schemes attaining lower average redundancies and higher recovery rate of dropped packets. However, they do not change their maximum decoding delay to bound it with respect to the observed RTT, as Ivory does.

VII. Concluding Remarks

In this paper, we present *Ivory*, a new real-world system design powered by deep reinforcement learning, which aims at recovering as many lost packets as possible without over utilizing the bandwidth of the network link. Ivory is tested over audio packets, since audio quality is more sensitive to delays than video packets. We design *Ivory* to train offline to have a pre-trained model for deployment, and as *Ivory* gets deployed, it continues online learning to adapt to newly seen scenarios. Ivory's main goal is to achieve a better trade-off between recovering all packets to achieve high-quality interactive experience and minimizing redundancy overhead to better utilize network bandwidth while maintaining low-latencies.

Our evaluation results show that *Ivory* is competitive with the state-of-the-art in terms of adaptability and flexibility of trade-offs between packet recovery rate, redundancy and meeting latency deadlines. In some cases, it outperforms the state-of-the-art, and when it comes to achieving a trade-off between redundancy, FLR and meeting latency deadlines in stress testing scenarios, Ivory performs better than the state-of-the-art. Ivory's performance is also resilient to changes in RTT. We believe *Ivory* represents a step forward towards audio conferencing with adaptive streaming codes over UDP.

References

[1] International Telecommunication Union, "One-way transmission time," Recommendation G.114, May 2003.

[2] A. Badr, A. Khisti, W.-T. Tan, and J. Apostolopoulos, "Perfecting protection for interactive multimedia: A survey of forward error correction for low-delay interactive applications," *IEEE Signal Process. Mag.*, vol. 34, pp. 95 – 113, 2017.

[3] A. Hameed, R. Dai, and B. Balas, "A decision-tree-based perceptual video quality prediction model and its application in FEC for wireless multimedia communications," *IEEE Trans. Multimedia*, vol. 18, no. 4, pp. 764–774, April 2016.

[4] S. Holmer, M. Shemer, and M. Paniconi, "Handling packet loss in WebRTC," in *Proc. IEEE Intl. Conference on Image Process.*, Sept 2013.

[5] J. Korhonena and P. Frossard, "Flexible forward error correction codes with application to partial media data recovery," *Signal Processing: Image Communication*, vol. 24, no. 3, pp. 229 – 242, 2009.

[6] M. Nagy, V. Singh, J. Ott, and L. Eggert, "Congestion control using FEC for conversational multimedia communication," in *Proc. the 5th ACM Multimedia Systems Conference*, 2014.

[7] A. Shokrollahi, "Raptor codes," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[8] M. Watson, T. Stockhammer, and M. Luby. (2012, August) Raptor forward error correction (FEC) schemes for FECFRAME. [Online]. Available: https://tools.ietf.org/html/rfc6681

[9] V. Roca and B. Teibi. (2020, January) Sliding window random linear code (RLC) forward erasure correction (FEC) schemes for FECFRAME. [Online]. Available: https://tools.ietf.org/html/rfc8681

[10] J. Wang and D. Katabi, "ChitChat: Making video chat robust to packet loss," Master's thesis, Massachusetts Institute of Technology, 2010.

[11] A. Langley, A. Riddoch, A. Wilk, and et al., "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. Conference of the ACM Special Interest Group Data Communication. SIGCOMM*, August 2017.

[12] S. Emara, S. Fong, B. Li, A. Khisti, W.-T. Tan, X. Zhu, and J. Apostolopoulos, "Low-latency network-adaptive error control for interactive streaming," *IEEE Transactions on Multimedia*, pp. 1–1, 2021.

[13] S. L. Fong, S. Emara, B. Li, A. Khisti, W.-T. Tan, X. Zhu, and J. Apostolopoulos, "Low-latency network-adaptive error control for interactive streaming," in *Proc. 27th ACM International Conference on Multimedia*, Oct 2019.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[15] T. Huang, P. Huang, K. Chen, and P. Wang, "Could Skype be more satisfying? A QoE-centric study of the FEC mechanism in an Internet-scale VoIP system," *IEEE Network*, vol. 24, no. 2, pp. 42 –48, 2010.

[16] December 2018. [Online]. Available: https://www.vyopta.com/blog/video-conferencing/understanding-packet-loss/

[17] I. T. Union, "Wideband extension to recommendation p.862 for the assessment of wideband telephone networks and speech codecs," International Telecommunication Union, Recommendation P.862.2, Nov 2007.

[18] J. Valin, K. Vos, and T. Terriberry. (2012, Sept) Definition of the opus audio codec. [Online]. Available: https://tools.ietf.org/html/rfc6716

[19] S. D.Whitehead and L.-J. Lin, "Reinforcement learning of non-markov decision processes," *Artificial Intelligence*, vol. 73, no. 1, pp. 271–306, 1995.

[20] R. Rubinstein and D. Kroese, *The Cross-Entropy Method*. Springer, 2004.

[21] R. G. Gallagher, "Low density parity check codes," *IRE Transactions on Inf. Theory*, vol. 8, pp. 21 – 28, 1962.

[22] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, pp. 457 – 458, 1997.

[23] M. Luby, "LT codes," in *Proc. 43rd Annual IEEE Symposium on Foundations Computer Science*, 2002.

[24] E. T. S. Institute, "Digital video broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for broadcasting, interactiv services, news gathering and other broadband satellite applications; Part 1: DVB-S2," European Telecommunications Standards Institute, Standard ETSI EN 302 307-1, Nov 2014.

[25] ——, "Digital video broadcasting (DVB); Transport of MPEG-2 TS based DVB services over IP based networks," European Telecommunications Standards Institute, Standard ETSI TS 102 034, April 2016.

[26] A. Badr, A. Khisti, W. Tan, X. Zhu, and J. Apostolopoulos, "FEC for VoIP using dual-delay streaming codes," in *Proc. IEEE INFOCOM*, 2017.

[27] E. Domanovitz, S. L. Fong, and A. Khisti, "An explicit rate-optimal streaming code for channels with burst and arbitrary erasures," in *Proc. IEEE Information Theory Workshop (ITW)*, 2019.

[28] D. Dudzicz, S. L. Fong, and A. Khisti, "An explicit construction of optimal streaming codes for channels with burst and arbitrary erasures," *IEEE Transactions on Communications*, vol. 68, no. 1, pp. 12–25, 2020.

[29] A. Nguyen, B. Li, and F. Eliassen, "Chameleon: Adaptive peer-to-peer streaming with network coding," in *Proc. IEEE INFOCOM*, 2010.

[30] W. Dong, J. Yu, and X. Liu, "CARE: corruption-aware retransmission with adaptive coding for the low-power wireless," in *IEEE 23rd International Conference on Network Protocols (ICNP)*, 2015.

[31] Q. Wang, J. Liu, K. Jaffres-Runser, Y. Wang, C. He, C. Liu, and Y. Xu, "INCdeep: Intelligent network coding with deep reinforcement learning," in *Proc. IEEE INFOCOM*, 2021.

[32] S. Cheng, H. Hu, X. Zhang, and Z. Guo, "DeepRS: Deep-learning based network-adaptive FEC for real-time video communications," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.

[33] H. Hu, S. Cheng, X. Zhang, and Z. Guo, "LightFEC: Network adaptive FEC with a lightweight deep-learning approach," in *Proc. 29th ACM International Conference on Multimedia*, 2021.