Neural Adaptive Video Streaming with Pensieve

Hongzi Mao, Ravi Netravali, Mohammad Alizadeh
MIT Computer Science and Artificial Intelligence Laboratory
{hongzi,ravinet,alizadeh}@mit.edu

ABSTRACT

Client-side video players employ adaptive bitrate (ABR) algorithms to optimize user quality of experience (QoE). Despite the abundance of recently proposed schemes, state-of-the-art ABR algorithms suffer from a key limitation: they use fixed control rules based on simplified or inaccurate models of the deployment environment. As a result, existing schemes inevitably fail to achieve optimal performance across a broad set of network conditions and QoE objectives.

We propose Pensieve, a system that generates ABR algorithms using reinforcement learning (RL). Pensieve trains a neural network model that selects bitrates for future video chunks based on observations collected by client video players. Pensieve does not rely on pre-programmed models or assumptions about the environment. Instead, it learns to make ABR decisions solely through observations of the resulting performance of past decisions. As a result, Pensieve automatically learns ABR algorithms that adapt to a wide range of environments and QoE metrics. We compare Pensieve to state-of-the-art ABR algorithms using trace-driven and real world experiments spanning a wide variety of network conditions, QoE metrics, and video properties. In all considered scenarios, Pensieve outperforms the best state-of-the-art scheme, with improvements in average QoE of 12%–25%. Pensieve also generalizes well, outperforming existing schemes even on networks for which it was not explicitly trained.

CCS Concepts: Information systems → Multimedia streaming; **Networks** → Network resources allocation; **Computing methodologies** → Reinforcement learning

Keywords: bitrate adaptation, video streaming, reinforcement learning

ACM Reference format: Hongzi Mao, Ravi Netravali, Mohammad Alizadeh MIT Computer Science and Artificial Intelligence Laboratory. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA*, 14 pags.
DOI: http://dx.doi.org/10.1145/3098822.3098843

1 INTRODUCTION

Recent years have seen a rapid increase in the volume of HTTP-based video streaming traffic [7, 39]. Concurrent with this increase has been a steady rise in user demands on video quality. Many studies have shown that users will quickly abandon video sessions if the quality is not sufficient, leading to significant losses in revenue for content providers [12, 25]. Nevertheless, content providers continue to struggle with delivering high-quality video to their viewers.

Adaptive bitrate (ABR) algorithms are the primary tool that content providers use to optimize video quality. These algorithms run on client-side video players and dynamically choose a bitrate for each video *chunk* (e.g., 4-second block). ABR algorithms make bitrate decisions based on various observations such as the estimated network throughput and playback buffer occupancy. Their goal is to maximize the user's QoE by adapting the video bitrate to the underlying network conditions. However, selecting the right bitrate can be very challenging due to (1) the variability of network throughput [18, 42, 49, 52, 53]; (2) the conflicting video QoE requirements (high bitrate, minimal rebuffering, smoothness, etc.); (3) the cascading effects of bitrate decisions (e.g., selecting a high bitrate may drain the playback buffer to a dangerous level and cause rebuffering in the future); and (4) the coarse-grained nature of ABR decisions. We elaborate on these challenges in §2.

The majority of existing ABR algorithms (§7) develop fixed control rules for making bitrate decisions based on estimated network throughput ("rate-based" algorithms [21, 42]), playback buffer size ("buffer-based" schemes [19, 41]), or a combination of the two signals [26]. These schemes require significant tuning and do not generalize to different network conditions and QoE objectives. The state-of-the-art approach, MPC [51], makes bitrate decisions by solving a QoE maximization problem over a horizon of several future chunks. By optimizing directly for the desired QoE objective, MPC can perform better than approaches that use fixed heuristics. However, MPC's performance relies on an accurate model of the system dynamics—particularly, a forecast of future network throughput. As our experiments show, this makes MPC sensitive to throughput prediction errors and the length of the optimization horizon (§3).

In this paper, we propose *Pensieve*,[1] a system that learns ABR algorithms automatically, without using any pre-programmed control rules or explicit assumptions about the operating environment. Pensieve uses modern reinforcement learning (RL) techniques [27, 30, 43] to learn a control policy for bitrate adaptation *purely through experience*. During training, Pensieve starts knowing nothing about the task at hand. It then gradually learns to make better ABR decisions through reinforcement, in the form of reward signals that reflect video QoE for past decisions.

Pensieve's learning techniques mine information about the actual performance of past choices to optimize its control policy *for the characteristics of the network*. For example, Pensieve can learn how much playback buffer is necessary to mitigate the risk of rebuffering in a specific network, based on the network's inherent throughput variability. Or it can learn how much to rely on throughput versus buffer occupancy signals, or how far into the future to plan its decisions automatically. By contrast, approaches that use fixed control

[1] A pensieve is a device used in Harry Potter [38] to review memories.

SIGCOMM '17, Los Angeles, CA, USA

978-1-4503-4653-5/17/08…$15.00
DOI: http://dx.doi.org/10.1145/3098822.3098843

rules or simplified network models are unable to optimize their bitrate decisions based on all available information about the operating environment.

Pensieve represents its control policy as a neural network that maps "raw" observations (e.g., throughput samples, playback buffer occupancy, video chunk sizes) to the bitrate decision for the next chunk. The neural network provides an expressive and scalable way to incorporate a rich variety of observations into the control policy.[2] Pensieve trains this neural network using A3C [30], a state-of-the-art actor-critic RL algorithm. We describe the basic training algorithm and present extensions that allow a single neural network model to generalize to videos with different properties, e.g., the number of encodings and their bitrates (§4).

To train its models, Pensieve uses simulations over a large corpus of network traces. Pensieve uses a fast and simple *chunk-level* simulator. While Pensieve could also train using packet-level simulations, emulations, or data collected from live video clients (§6), the chunk-level simulator is much faster and allows Pensieve to "experience" 100 hours of video downloads in only 10 minutes. We show that Pensieve's simulator faithfully models video streaming with live video players, provided that the transport stack is configured to achieve close to the true network capacity (§4.1).

We evaluate Pensieve using a full system implementation (§4.4). Our implementation deploys Pensieve's neural network model on an *ABR server*, which video clients query to get the bitrate to use for the next chunk; client requests include observations about throughput, buffer occupancy, and video properties. This design removes the burden of performing neural network computation on video clients, which may have limited computation power, e.g., TVs, mobile devices, etc. (§6).

We compare Pensieve to state-of-the-art ABR algorithms using a broad set of network conditions (both with trace-based emulation and in the wild) and QoE metrics (§5.2). We find that in all considered scenarios, Pensieve rivals or outperforms the best existing scheme, with average QoE improvements ranging from 12%–25%. Additionally, our results show Pensieve's ability to generalize to unseen network conditions and video properties. For example, on both broadband and HSDPA networks, Pensieve was able to outperform all existing ABR algorithms by training solely with a synthetic dataset. Finally, we present results which highlight Pensieve's low overhead and lack of sensitivity to system parameters, e.g., in the neural network (§5.4).

2 BACKGROUND

HTTP-based adaptive streaming (standardized as DASH [2]) is the predominant form of video delivery today. By transmitting video using HTTP, content providers are able to leverage existing CDN infrastructure and maintain simplified (stateless) backends. Further, HTTP is compatible with a multitude of client-side applications such as web browsers and mobile applications.

In DASH systems, videos are stored on servers as multiple chunks, each of which represents a few seconds of the overall video playback. Each chunk is encoded at several discrete bitrates, where a higher bitrate implies a higher quality and thus a larger chunk size. Chunks across bitrates are aligned to support seamless quality transitions, i.e., a video player can switch to a different bitrate at any chunk boundary without fetching redundant bits or skipping parts of the video.

[2] A few prior schemes [6, 8, 9, 47] have applied RL to video streaming. But these schemes use basic "tabular" RL approaches [43]. As a result, they must rely on simplified network models and perform poorly in real network conditions. We discuss these schemes further in §5.4 and §7.

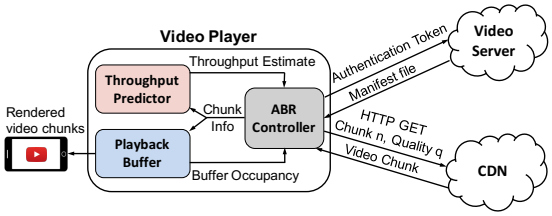


Figure 1: An overview of HTTP adaptive video streaming.

Figure 1 illustrates the end-to-end process of streaming a video over HTTP today. As shown, a player embedded in a client application first sends a token to a video service provider for authentication. The provider responds with a manifest file that directs the client to a CDN hosting the video and lists the available bitrates for the video. The client then requests video chunks one by one, using an *adaptive bitrate (ABR) algorithm*. These algorithms use a variety of different inputs (e.g., playback buffer occupancy, throughput measurements, etc.) to select the bitrate for future chunks. As chunks are downloaded, they are played back to the client; note that playback of a given chunk cannot begin until the entire chunk has been downloaded.

Challenges: The policies employed by ABR algorithms heavily influence video streaming performance. However, these algorithms face four primary practical challenges:

(1) Network conditions can fluctuate over time and can vary significantly across environments. This complicates bitrate selection as different scenarios may require different weights for input signals. For example, on time-varying cellular links, throughput prediction is often inaccurate and cannot account for sudden fluctuations in network bandwidth—inaccurate predictions can lead to underutilized networks (lower video quality) or inflated download delays (rebuffering). To overcome this, ABR algorithms must prioritize more stable input signals like buffer occupancy in these scenarios.

(2) ABR algorithms must balance a variety of QoE goals such as maximizing video quality (i.e., highest average bitrate), minimizing rebuffering events (i.e., scenarios where the client's playback buffer is empty), and maintaining video quality smoothness (i.e., avoiding constant bitrate fluctuations). However, many of these goals are inherently conflicting [3, 18, 21]. For example, on networks with limited bandwidth, consistently requesting chunks encoded at the highest possible bitrate will maximize quality, but may increase rebuffer rates. Conversely, on varying networks, choosing the highest bitrate that the network can support at any time could lead to substantial quality fluctuation, and hence degraded smoothness. To further complicate matters, preferences among these QoE factors vary significantly across users [23, 31, 32, 34].

(3) Bitrate selection for a given chunk can have cascading effects on the state of the video player. For example, selecting a high bitrate may deplete the playback buffer and force subsequent

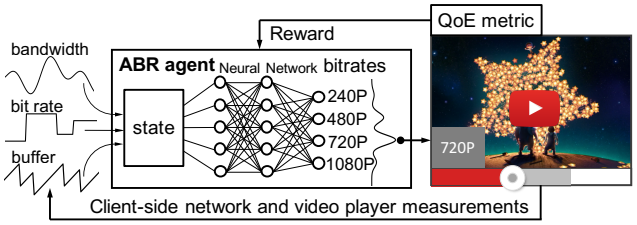


Figure 2: Applying reinforcement learning to bitrate adaptation.

chunks to be downloaded at low bitrates (to avoid rebuffering). Additionally, a given bitrate selection will directly influence the next decision when smoothness is considered—ABR algorithms will be less inclined to change bitrates.

(4) The control decisions available to ABR algorithms are coarse-grained as they are limited to the available bitrates for a given video. Thus, there exist scenarios where the estimated throughput falls just below one bitrate, but well above the next available bitrate. In these cases, the ABR algorithm must decide whether to prioritize higher quality or the risk of rebuffering.

3 LEARNING ABR ALGORITHMS

In this paper, we consider a learning-based approach to generating ABR algorithms. Unlike approaches which use preset rules in the form of fine-tuned heuristics, our techniques attempt to learn an ABR policy from observations. Specifically, our approach is based on reinforcement learning (RL). RL considers a general setting in which an *agent* interacts with an *environment*. At each time step t, the agent observes some *state* s_t, and chooses an *action* a_t. After applying the action, the state of the environment transitions to s_{t+1} and the agent receives a *reward* r_t. The goal of learning is to maximize the expected cumulative discounted reward: $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where $\gamma \in (0, 1]$ is a factor discounting future rewards.

Figure 2 summarizes how RL can be applied to bitrate adaptation. As shown, the decision policy guiding the ABR algorithm is not handcrafted. Instead, it is derived from training a neural network. The ABR agent observes a set of metrics including the client playback buffer occupancy, past bitrate decisions, and several raw network signals (e.g., throughput measurements) and feeds these values to the neural network, which outputs the action, i.e., the bitrate to use for the next chunk. The resulting QoE is then observed and passed back to the ABR agent as a reward. The agent uses the reward information to train and improve its neural network model. More details about the specific training algorithms we used are provided in §4.2.

To motivate learning-based ABR algorithms, we now provide two examples where existing techniques that rely on fixed heuristics can perform poorly. We choose these examples for illustrative purposes. We do not claim that they are indicative of the performance gains with learning in realistic network scenarios. We perform thorough quantitative evaluations comparing learning-generated ABR algorithms to existing schemes in §5.2.

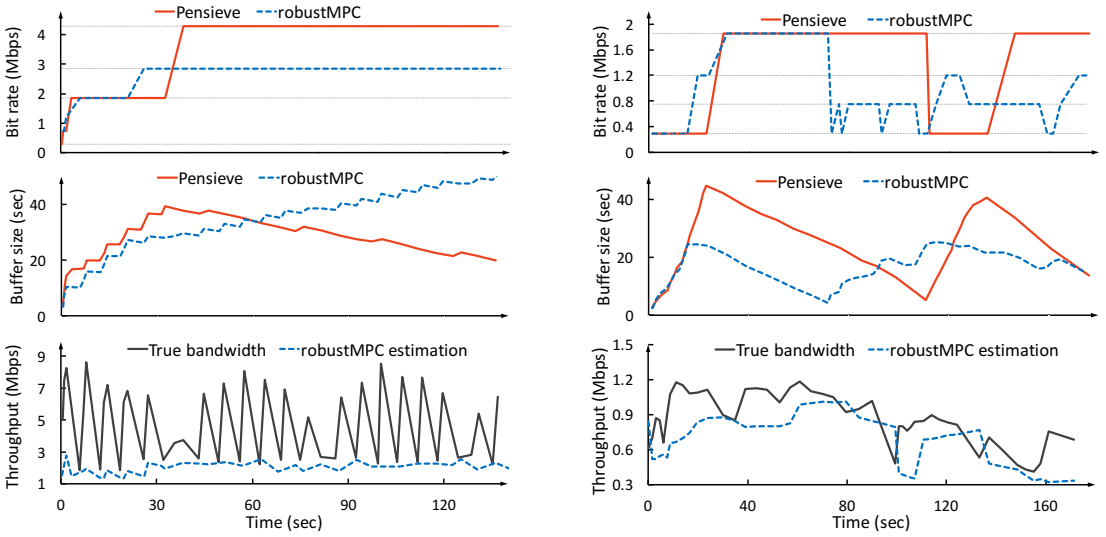
In these examples, we compare RL-generated ABR algorithms to MPC [51]. MPC uses both throughput estimates and observations about buffer occupancy to select bitrates that maximize a given QoE metric across a future chunk horizon. Here we consider robustMPC, a version of MPC that is configured to use a horizon of 5 chunks, and a conservative throughput estimate which normalizes the default throughput prediction with the max prediction error over the past 5 chunks. As the MPC paper shows, and our results validate, robustMPC's conservative throughput prediction significantly improves performance over default MPC, and achieves a high level of performance in most cases (§5.2). However, heuristics like robustMPC's throughput prediction require careful tuning and can backfire when their design assumptions are violated.

Example 1: The first example considers a scenario in which the network throughput is highly variable. Figure 3a compares the network throughput specified by the input trace with the throughput estimates used by robustMPC. As shown, robustMPC's estimates are overly cautious, hovering around 2 Mbps instead of the average network throughput of roughly 4.5 Mbps. These inaccurate throughput predictions prevent robustMPC from reaching high bitrates even though the occupancy of the playback buffer continually increases. In contrast, the RL-generated algorithm is able to properly assess the high average throughput (despite fluctuations) and switch to the highest available bitrate once it has enough cushion in the playback buffer. The RL-generated algorithm considered here was trained on a large corpus of real network traces (§5.1), not the synthetic trace in this experiment. Yet, it was able to make the appropriate decision.

Example 2: In our second example, both robustMPC and the RL-generated ABR algorithm optimize for a new QoE metric which is geared towards users who strongly prefer HD video. This metric assigns high reward to HD bitrates and low reward to all other bitrates (details in Table 1), while still favoring smoothness and penalizing rebuffering. To optimize for this metric, an ABR algorithm should attempt to build the client's playback buffer to a high enough level such that the player can switch up to and maintain an HD bitrate level. Using this approach, the video player can maximize the amount of time spent streaming HD video, while minimizing rebuffering time and bitrate transitions. However, performing well in this scenario requires long term planning since at any given instant, the penalty of selecting a higher bitrate (HD or not) may be incurred many chunks in the future when the buffer cannot support multiple HD downloads.

Figure 3b illustrates the bitrate selections made by each of these algorithms, and the effects that these decisions have on the playback buffer. Note that robustMPC and the RL-generated algorithm were both configured to optimize for this new QoE metric. As shown, robustMPC is unable to apply the aforementioned policy. Instead, robustMPC maintains a medium-sized playback buffer and requests chunks at bitrates that fall between the lowest level (300 kbps) and the lowest HD level (1850 kbps). The reason is that, despite being tuned to consider a horizon of future chunks at every step, robustMPC fails to plan far enough into the future. In contrast, the RL-generated ABR algorithm is able to actively implement the policy outlined above. It quickly grows the playback buffer by requesting chunks at 300 kbps, and then immediately jumps to the HD quality of 1850 kbps; it is able to then maintain this level for nearly 80 seconds, thereby ensuring quality smoothness.

Summary: robustMPC has difficulty (1) factoring throughput fluctuations and prediction errors into its decisions, and (2) choosing the appropriate optimization horizon. These deficiencies exist because MPC lacks an accurate model of network dynamics—thus it relies on



(a) Synthetic network. (b) HSDPA network.

Figure 3: Profiling bitrate selections, buffer occupancy, and throughput estimates with robustMPC [51] and Pensieve.

simple and sub-optimal heuristics such as conservative throughput predictions and a small optimization horizon. More generally, any ABR algorithm that relies on fixed heuristics or simplified system models suffers from these limitations. By contrast, RL-generated algorithms learn from actual performance resulting from different decisions. By incorporating this information into a flexible neural network policy, RL-generated ABR algorithms can automatically optimize for different network characteristics and QoE objectives.

4 DESIGN

In this section, we describe the design and implementation of Pensieve, a system that generates RL-based ABR algorithms and applies them to video streaming sessions. We start by explaining the training methodology (§4.1) and algorithms (§4.2) underlying Pensieve. We then describe an enhancement to the basic training algorithm, which enables Pensieve to support different videos using a single model (§4.3). Finally, we explain the implementation details of Pensieve and how it applies learned models to real streaming sessions (§4.4).

4.1 Training Methodology

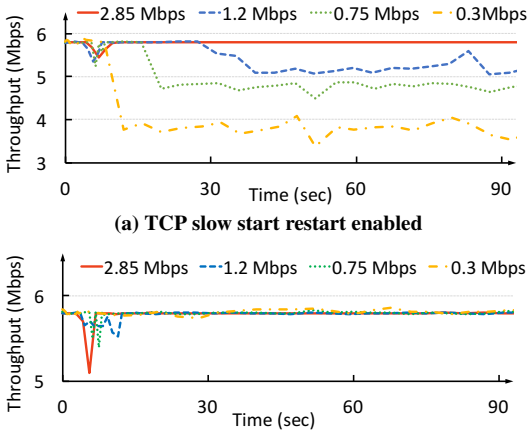
The first step of Pensieve is to generate an ABR algorithm using RL (§3). To do this, Pensieve runs a training phase in which the learning agent explores a video streaming environment. Ideally, training would occur using actual video streaming clients. However, emulating the standard video streaming environment entails using a web browser to continually download video chunks. This approach is slow, as the training algorithm must wait until all of the chunks in a video are completely downloaded before updating its model.

To accelerate this process, Pensieve trains ABR algorithms in a simple simulation environment that faithfully models the dynamics of video streaming with real client applications. Pensieve's simulator maintains an internal representation of the client's playback buffer. For each chunk download, the simulator assigns a download time that is solely based on the chunk's bitrate and the input network throughput traces. The simulator then drains the playback buffer by the current chunk's download time, to represent video playback during the download, and adds the playback duration of the downloaded chunk to the buffer. The simulator carefully keeps track of rebuffering events that arise as the buffer occupancy changes, i.e., scenarios where the chunk download time exceeds the buffer occupancy at the start of the download. In scenarios where the playback buffer cannot accommodate video from an additional chunk download, Pensieve's simulator pauses requests for 500 ms before retrying.[3] After each chunk download, the simulator passes several state observations to the RL agent for processing: the current buffer occupancy, rebuffering time, chunk download time, size of the next chunk (at all bitrates), and the number of remaining chunks in the video. We describe how this input is used by the RL agent in more detail in §4.2. Using this chunk-level simulator, Pensieve can "experience" 100 hours of video downloads in only 10 minutes.

Though modeling the application layer semantics of client video players is straightforward, faithful simulation is complicated by intricacies at the transport layer. Specifically, video players may not request future chunks as soon as a chunk download completes, e.g., because the playback buffer is full. Such delays can trigger the underlying TCP connection to revert to slow start, a behavior known as *slow-start-restart* [4]. Slow start may in turn prevent the video player from fully using the available bandwidth, particularly for small chunk sizes (low bitrates). This behavior makes simulation challenging as it inherently ties network throughput to the ABR algorithm being used, e.g., schemes that fill buffers quickly will experience more slow start phases and thus less network utilization.

To verify this behavior, we loaded the test video described in §5.1 over an emulated 6 Mbps link using four ABR algorithms, each of which continually requests chunks at a single bitrate. We loaded the video with each scheme twice, both with slow-start-restart enabled

[3] This is the default request retry rate used by DASH players [2].



(a) TCP slow start restart enabled

(b) TCP slow start restart disabled

Figure 4: Profiling the throughput usage per-chunk of commodity video players with and without TCP slow start restart.

and disabled.[4] Figure 4 shows the throughput usage *during chunk downloads* for each bitrate in both scenarios. As shown, with slow-start-restart enabled, the throughput depends on the bitrate of the chunk; ABR algorithms using lower bitrates (smaller chunk sizes) achieve less throughput per chunk. However, throughput is consistent and matches the available bandwidth (6 Mbps) for different bitrates if we disable slow-start-restart.

Pensieve's simulator assumes that the throughput specified by the trace is entirely used by each chunk download. As the above results show, this can be achieved by disabling slow-start-restart on the video server. Disabling slow-start-restart could increase traffic burstiness, but recent standards efforts are tackling the same problem for video streaming more gracefully by pacing the initial burst from TCP following an idle period [13, 17].

While it is possible to use a more accurate simulator (e.g., packet-level) to train Pensieve, in the end, no simulation can capture all real world system artifacts with 100% accuracy. However, we find that Pensieve can learn very high quality ABR algorithms (§5.2) using imperfect simulations, as long as it experiences a large enough variety of network conditions during training. This is a consequence of Pensieve's strong generalization ability (§5.3).

4.2 Basic Training Algorithm

We now describe our training algorithms. As shown in Figure 5, Pensieve's training algorithm uses A3C [30], a state-of-the-art actor-critic method which involves training two neural networks. The detailed functionalities of these networks are explained below.

Inputs: After the download of each chunk t, Pensieve's learning agent takes state inputs $s_t = (\vec{x_t}, \vec{\tau_t}, \vec{n_t}, b_t, c_t, l_t)$ to its neural networks. $\vec{x_t}$ is the network throughput measurements for the past k video chunks; $\vec{\tau_t}$ is the download time of the past k video chunks, which represents the time interval of the throughput measurements; $\vec{n_t}$ is a vector of m available sizes for the next video chunk; b_t is the current buffer level; c_t is the number of chunks remaining in the video; and l_t is the bitrate at which the last chunk was downloaded.

[4] In Linux, the `net.ipv4.tcp_slow_start_after_idle` parameter can be used to set this configuration.

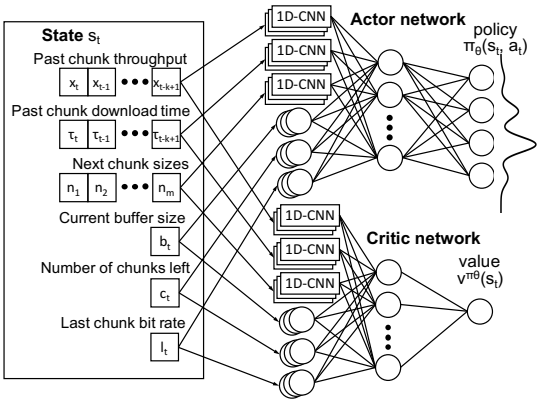


Figure 5: The Actor-Critic algorithm that Pensieve uses to generate ABR policies (described in §4.4).

Policy: Upon receiving s_t, Pensieve's RL agent needs to take an action a_t that corresponds to the bitrate for the next video chunk. The agent selects actions based on a *policy*, defined as a probability distribution over actions $\pi : \pi(s_t, a_t) \rightarrow [0, 1]$. $\pi(s_t, a_t)$ is the probability that action a_t is taken in state s_t. In practice, there are intractably many {state, action} pairs, e.g., throughput estimates and buffer occupancies are continuous real numbers. To overcome this, Pensieve uses a neural network (NN) [15] to represent the policy with a manageable number of adjustable parameters, θ, which we refer to as policy parameters. Using θ, we can represent the policy as $\pi_\theta(s_t, a_t)$. NNs have recently been applied successfully to solve large-scale RL tasks [27, 29, 40]. An advantage of NNs is that they do not need hand-crafted features and can be applied directly to "raw" observation signals. The *actor network* in Figure 5 depicts how Pensieve uses an NN to represent an ABR policy. We describe how we design the specific architecture of the NN in §5.3.

Policy gradient training: After applying each action, the simulated environment provides the learning agent with a reward r_t for that chunk. Recall from §3 that the primary goal of the RL agent is to maximize the expected cumulative (discounted) reward that it receives from the environment. Thus, the reward is set to reflect the performance of each chunk download according to the specific QoE metric we wish to optimize. See §5 for examples of QoE metrics.

The actor-critic algorithm used by Pensieve to train its policy is a *policy gradient method* [44]. We highlight the key steps of the algorithm, focusing on the intuition. The key idea in policy gradient methods is to estimate the gradient of the expected total reward by observing the trajectories of executions obtained by following the policy. The gradient of the cumulative discounted reward with respect to the policy parameters, θ, can be computed as [30]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right]. \quad (1)$$

$A^{\pi_\theta}(s,a)$ is the *advantage* function, which represents the difference in the expected total reward when we *deterministically* pick action a in state s, compared with the expected reward for actions drawn from policy π_θ. The advantage function encodes how much better a specific action is compared to the "average action" taken according to the policy.

In practice, the agent samples a trajectory of bitrate decisions and uses the empirically computed advantage $A(s_t, a_t)$, as an unbiased estimate of $A^{\pi_\theta}(s_t, a_t)$. Each update of the actor network parameter θ follows the policy gradient,

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t) A(s_t, a_t), \quad (2)$$

where α is the learning rate. The intuition behind this update rule is as follows. The direction $\nabla_\theta \log \pi_\theta(s_t, a_t)$ specifies how to change the policy parameters in order to increase $\pi_\theta(s_t, a_t)$ (i.e., the probability of action a_t at state s_t). Equation 2 takes a step in this direction. The size of the step depends on the value of the advantage for action a_t in state s_t. Thus, the net effect is to reinforce actions that empirically lead to better returns.

To compute the advantage $A(s_t, a_t)$ for a given experience, we need an estimate of the *value function*, $v^{\pi_\theta}(s)$—the expected total reward starting at state s and following the policy π_θ. The role of the *critic network* in Figure 5 is to learn an estimate of $v^{\pi_\theta}(s)$ from empirically observed rewards. We follow the standard *Temporal Difference* method [43] to train the critic network parameters θ_v,

$$\theta_v \leftarrow \theta_v - \alpha' \sum_t \nabla_{\theta_v} \left(r_t + \gamma V^{\pi_\theta}(s_{t+1}; \theta_v) - V^{\pi_\theta}(s_t; \theta_v) \right)^2, \quad (3)$$

where $V^{\pi_\theta}(\cdot; \theta_v)$ is the estimate of $v^{\pi_\theta}(\cdot)$, output by the critic network, and α' is the learning rate for the critic. For an experience (s_t, a_t, r_t, s_{t+1}) (i.e., take action a_t in state s_t, receive reward r_t, and transition to s_{t+1}), the advantage $A(s_t, a_t)$ can now be estimated as $r_t + \gamma V^{\pi_\theta}(s_{t+1}; \theta_v) - V^{\pi_\theta}(s_t; \theta_v)$. See [24] for more details.

It is important to note that the critic network merely helps to train the actor network. Post-training, only the actor network is required to execute the ABR algorithm and make bitrate decisions.

Finally, we must ensure that the RL agent explores the action space adequately during training to discover good policies. One common practice to encourage exploration is to add an entropy regularization term to the actor's update rule [30]; this can be critical in helping the learning agent converge to a good policy [50]. Concretely, we modify Equation 2 to be,

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t, a_t) A(s_t, a_t) + \beta \nabla_\theta H(\pi_\theta(\cdot|s_t)), \quad (4)$$

where $H(\cdot)$ is the entropy of the policy (the probability distribution over actions) at each time step. This term encourages exploration by pushing θ in the direction of higher entropy. The parameter β is set to a large value at the start of training (to encourage exploration) and decreases over time to emphasize improving rewards (§4.4).

The detailed derivation and pseudocode can be found in [30] (§4 and Algorithm S3).

Parallel training: To further enhance and speed up training, Pensieve spawns multiple learning agents in parallel, as suggested by the A3C paper [30]. By default, Pensieve uses 16 parallel agents. Each learning agent is configured to experience a different set of input parameters (e.g., network traces). However, the agents continually send their {state, action, reward} tuples to a central agent, which aggregates them to generate a single ABR algorithm model. For each sequence of tuples that it receives, the central agent uses the actor-critic algorithm to compute a gradient and perform a gradient descent step (Equations (3) and (4)). The central agent then updates the actor network and pushes out the new model to the agent which sent that tuple. Note that this can happen asynchronously among all agents, i.e., there is no locking between agents [36].

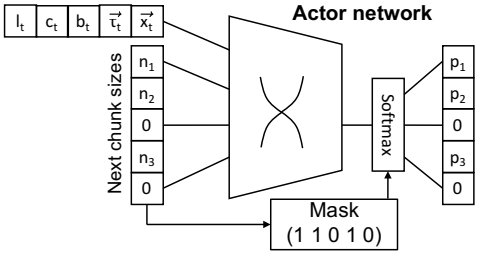


Figure 6: Modification to the state input and the softmax output to support multiple videos.

Choice of algorithm: A variety of different algorithms could be used to train the learning agent in the abstract RL framework described above (e.g., DQN [29], REINFORCE [44], etc.). In our design, we chose to use A3C [30] because (1) to the best of our knowledge, it is the state-of-art and it has been successfully applied to many other concrete learning problems [20, 48, 50]; and (2) in the video streaming application, the asynchronous parallel training framework supports online training in which many users concurrently send their experience feedback to the agent. We also compare Pensieve with previous tabular Q-learning schemes [6] in §5.4.

4.3 Enhancement for multiple videos

The basic algorithm described in §4.2 has some practical issues. The primary challenge is that videos can be encoded at different bitrate levels and may have diverse chunk sizes due to variable bitrate encoding [41], e.g., chunk sizes for 720p video are not identical across videos. Handling this variation would require each neural network to take a variable sized set of inputs and produce a variable sized set of outputs. The naive solution to supporting a broad range of videos is to train a model for each possible set of video properties. Unfortunately, this solution is not scalable. To overcome this, we describe two enhancements to the basic algorithm that enable Pensieve to generate a *single* model to handle multiple videos (Figure 6).

First, we pick canonical input and output formats that span the maximum number of bitrate levels we expect to see in practice. For example, a range of 13 levels covers the entire DASH reference client video list [11]. Then, to determine the input state for a specific video, we take the chunk sizes and map them to the index which has the closest bitrate. The remaining input states, which pertain to the bitrates that the video does not support, are zeroed out. For example, in Figure 6, chunk sizes (n_1, n_2, n_3) are mapped to the corresponding indices, while the remaining input values are filled with zeroes.

The second change pertains to how the output of the actor network is interpreted. For a given video, we apply a mask to the output of the final softmax [5] layer in the actor network, such that the output probability distribution is only over the bitrates that the video actually supports. Formally, the mask is presented by a 0-1 vector $[m_1, m_2, ..., m_k]$, and the modified softmax for the NN output $[z_1, z_2, ..., z_k]$ will be

$$p_i = \frac{m_i e^{z_i}}{\sum_j m_j e^{z_j}}, \quad (5)$$

where p_i is the normalized probability for action i. With this modification, the output probabilities are still a continuous function of

the network parameters. The reason is that the mask values $\{m_i\}$ are independent of the network parameters, and are only a function of the input video. As a result, the standard back-propagation of the gradient in the NN still holds and the training techniques established in §4.2 can be applied without modification. We evaluate the effectiveness of these modifications in more detail in §5.4.

4.4 Implementation

To generate ABR algorithms, Pensieve passes $k = 8$ past bandwidth measurements to a 1D convolution layer (CNN) with 128 filters, each of size 4 with stride 1. Next chunk sizes are passed to another 1D-CNN with the same shape. Results from these layers are then aggregated with other inputs in a hidden layer that uses 128 neurons to apply the softmax function (Figure 5). The critic network uses the same NN structure, but its final output is a linear neuron (with no activation function). During training, we use a discount factor $\gamma = 0.99$, which implies that current actions will be influenced by 100 future steps. The learning rates for the actor and critic are configured to be 10^{-4} and 10^{-3}, respectively. Additionally, the entropy factor β is controlled to decay from 1 to 0.1 over 10^5 iterations. We keep all these hyperparameters fixed throughout our experiments. While some tuning is useful, we found that Pensieve performs well for a wide range of hyperparameter values. Thus we did not use sophisticated hyperparameter tuning methods [14]. We implemented this architecture using TensorFlow [1]. For compatibility, we leveraged the TFLearn deep learning library's TensorFlow API [46] to declare the neural network during both training and testing.

Once Pensieve has generated an ABR algorithm using its simulator, it must apply the model's rules to real video streaming sessions. To do this, Pensieve runs on a standalone ABR server, implemented using the Python BaseHTTPServer. Client requests are modified to include additional information about the previous chunk download and the video being streamed (§4.2). By collecting information through client requests, Pensieve's server and ABR algorithm can remain stateless while still benefitting from observations that can solely be collected in client video players. As client requests for individual chunks arrive at the video server, Pensieve feeds the provided observations through its actor NN model and responds to the video client with the bitrate level to use for the next chunk download; the client then contacts the appropriate CDN to fetch the corresponding chunk. It is important to note that Pensieve's ABR algorithm could also operate directly inside video players. We evaluate the overhead that a server-side deployment has on video QoE in §5.4, and discuss other deployment models in more detail in §6.

5 EVALUATION

In this section, we experimentally evaluate Pensieve. Our experiments cover a broad set of network conditions (both trace-based and in the wild) and QoE metrics. Our results answer the following questions:

(1) How does Pensieve compare to state-of-the-art ABR algorithms in terms of video QoE? We find that, in all of the considered scenarios, Pensieve is able to rival or outperform the best existing scheme, with average QoE improvements ranging from 12.1%–24.6% (§5.2); Figure 7 provides a summary.

(2) Do models learned by Pensieve generalize to new network conditions and videos? We find that Pensieve's ABR algorithms are able to maintain high levels of performance both in the presence of new network conditions and new video properties (§5.3).

(3) How sensitive is Pensieve to various parameters such as the neural network architecture and the latency between the video client and ABR server? Our experiments suggest that performance is largely unaffected by these parameters (Tables 2 and 3). For example, applying 100 ms RTT values between clients and the Pensieve server reduces average QoE by only 3.5% (§5.4).

5.1 Methodology

Network traces: To evaluate Pensieve and state-of-the-art ABR algorithms on realistic network conditions, we created a corpus of network traces by combining several public datasets: a broadband dataset provided by the FCC [10] and a 3G/HSDPA mobile dataset collected in Norway [37]. The FCC dataset contains over 1 million throughput traces, each of which logs the average throughput over 2100 seconds, at a 5 second granularity. We generated 1000 traces for our corpus, each with a duration of 320 seconds, by concatenating randomly selected traces from the "Web browsing" category in the August 2016 collection. The HSDPA dataset comprises 30 minutes of throughput measurements, generated using mobile devices that were streaming video while in transit (e.g., via bus, train, etc.). To match the duration of the FCC traces included in our corpus, we generated 1000 traces (each spanning 320 seconds) using a sliding window across the HSDPA dataset. To avoid scenarios where bitrate selection is trivial, i.e., situations where picking the maximum bitrate is always the optimal solution, or where the network cannot support any available bitrate for an extended period, we only considered original traces whose average throughput is less than 6 Mbps, and whose minimum throughput is above 0.2 Mbps. We reformatted throughput traces from both datasets to be compatible with the Mahimahi [33] network emulation tool. Unless otherwise noted, we used a random sample of 80% of our corpus as a training set for Pensieve; we used the remaining 20% as a test set for all ABR algorithms. All in all, our test set comprises of over 30 hours of network traces.

Adaptation algorithms: We compare Pensieve to the following algorithms which collectively represent the state-of-the-art in bitrate adaptation:

(1) Buffer-Based (BB): mimics the buffer-based algorithm described by Huang et al. [19] which uses a reservoir of 5 seconds and a cushion of 10 seconds, i.e., it selects bitrates with the goal of keeping the buffer occupancy above 5 seconds, and automatically chooses the highest available bitrate if the buffer occupancy exceeds 15 seconds.

(2) Rate-Based (RB): predicts throughput using the harmonic mean of the experienced throughput for the past 5 chunk downloads. It then selects the highest available bitrate that is below the predicted throughput.

(3) BOLA [41]: uses Lyapunov optimization to select bitrates solely considering buffer occupancy observations. We use the BOLA implementation in dash.js [2].

(4) MPC [51]: uses buffer occupancy observations and throughput predictions (computed in the same way as RB) to select the

bitrate which maximizes a given QoE metric over a horizon of 5 future chunks.

(5) robustMPC [51]: uses the same approach as MPC, but accounts for errors seen between predicted and observed throughputs by normalizing throughput estimates by the max error seen in the past 5 chunks.

Note: MPC involves solving an optimization problem for each bitrate decision which maximizes the QoE metric over the next 5 video chunks. The MPC [51] paper describes a method, fastMPC, which precomputes the solution to this optimization problem for a quantized set of input values (e.g., buffer size, throughput prediction, etc.). Because the implementation of fastMPC is not publicly available, we implemented MPC using our ABR server as follows. For each bitrate decision, we solve the optimization problem exactly on the ABR server by enumerating all possibilities for the next 5 chunks. We found that the computation takes at most 27 ms for 6 bitrate levels and has negligible impact on QoE.

Experimental setup: We modified dash.js (version 2.4) [2] to support each of the aforementioned state-of-the-art ABR algorithms. For Pensieve and both variants of MPC, dash.js was configured to fetch bitrate selection decisions from an ABR server that implemented the corresponding algorithm. ABR servers ran on the same machine as the client, and requests to these servers were made using `XMLHttpRequests`. All other algorithms ran directly in dash.js. The DASH player was configured to have a playback buffer capacity of 60 seconds. Our evaluations used the "Envivio-Dash3" video from the DASH-246 JavaScript reference client [11]. This video is encoded by the H.264/MPEG-4 codec at bitrates in $\{300, 750, 1200, 1850, 2850, 4300\}$ kbps (which pertain to video modes in $\{240, 360, 480, 720, 1080, 1440\}$p). Additionally, the video was divided into 48 chunks and had a total length of 193 seconds. Thus, each chunk represented approximately 4 seconds of video playback. In our setup, the client video player was a Google Chrome browser (version 53) and the video server (Apache version 2.4.7) ran on the same machine as the client. We used Mahimahi [33] to emulate the network conditions from our corpus of network traces, along with an 80 ms RTT, between the client and server. Unless otherwise noted, all experiments were performed on Amazon EC2 t2.2xlarge instances.

QoE metrics: There exists significant variance in user preferences for video streaming QoE [23, 31, 32, 34]. Thus, we consider a variety of QoE metrics. We start with the general QoE metric used by MPC [51], which is defined as

$$QoE = \sum_{n=1}^{N} q(R_n) - \mu \sum_{n=1}^{N} T_n - \sum_{n=1}^{N-1} \left| q(R_{n+1}) - q(R_n) \right| \quad (6)$$

for a video with N chunks. R_n represents the bitrate of $chunk_n$ and $q(R_n)$ maps that bitrate to the quality perceived by a user. T_n represents the rebuffering time that results from downloading $chunk_n$ at bitrate R_n, while the final term penalizes changes in video quality to favor smoothness.

We consider three choices of $q(R_n)$:

(1) QoE_{lin}: $q(R_n) = R_n$. This metric was used by MPC [51].

(2) QoE_{log}: $q(R_n) = \log(R/R_{min})$. This metric captures the notion that, for some users, the marginal improvement in perceived quality decreases at higher bitrates and was used by BOLA [41].

Name	bitrate utility ($q(R)$)	rebuffer penalty (μ)
QoE_{lin}	R	4.3
QoE_{log}	$\log(R/R_{min})$	2.66
QoE_{hd}	0.3→1, 0.75→2, 1.2→3 1.85→12, 2.85→15, 4.3→20	8

Table 1: The QoE metrics we consider in our evaluation. Each metric is a variant of Equation 6.

(3) QoE_{hd}: This metric favors High Definition (HD) video. It assigns a low quality score to non-HD bitrates and a high quality score to HD bitrates.

The exact values of $q(R_n)$ for our baseline video are provided in Table 1. In this section, we report the average QoE per chunk, i.e., the total QoE metric divided by the number of chunks in the video.

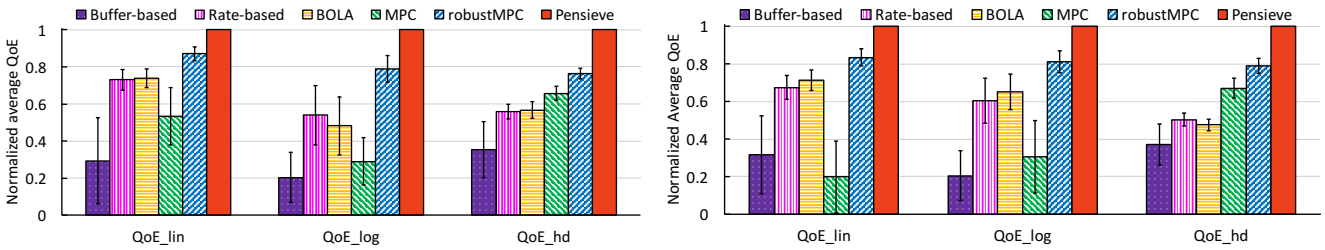
5.2 Pensieve vs. Existing ABR algorithms

To evaluate Pensieve, we compared it with state-of-the-art ABR algorithms on each QoE metric listed in Table 1. In each experiment, Pensieve's ABR algorithm was trained to optimize for the considered QoE metric, using the entire training corpus described in §5.1; both MPC variants were also modified to optimize for the considered QoE metric. For comparison, we also present results for the *offline optimal* scheme, which is computed using dynamic programing with complete future throughput information. The offline optimal serves as an (unattainable) upper bound on the QoE that an omniscient policy with complete and perfect knowledge of the future network throughput could achieve.

Figure 7 shows the average QoE that each scheme achieves on our entire test corpus. Figures 8 and 9 provide more detailed results in the form of full CDFs for each network. There are three key takeaways from these results. First, we find that Pensieve either matches or exceeds the performance of the best existing ABR algorithm on each QoE metric and network considered. The closest competing scheme is robustMPC; this shows the importance of tuning, as without robustMPC's conservative throughput estimates, MPC can become too aggressive (relying on the playback buffer) and perform worse than even a naive rate-based scheme. For QoE_{lin}, which was considered in the MPC paper [51], the average QoE for Pensieve is 15.5% higher than robustMPC on the FCC broadband network traces. The gap between Pensieve and robustMPC widens to 18.9% and 24.6% for QoE_{log} and QoE_{hd}. The results are qualitatively similar for the Norway HSDPA network traces.

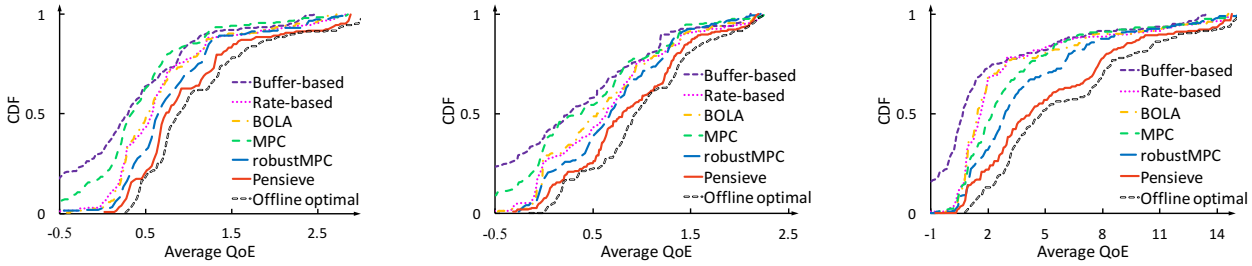
Second, we observe that the performance of existing ABR algorithms struggle to optimize for different QoE objectives. The reason is that these algorithms employ fixed control laws, even though optimizing for different QoE objectives requires inherently different ABR strategies. For example, for QoE_{log}, since the marginal improvement in user-perceived quality diminishes at higher bitrates, the optimal strategy is to avoid jumping to high bitrate levels when the risk of rebuffering is high. However, to optimize for QoE_{lin}, the ABR algorithm needs to be more aggressive. Pensieve is able to automatically learn these policies and thus, performance with Pensieve remains consistently high as conditions change.

The results for QoE_{hd} further illustrate this point. Recall that QoE_{hd} favors HD video, assigning the highest utility to the top three bitrates available for our test video (see Table 1). As discussed in §3, optimizing for QoE_{hd} requires longer term planning than the



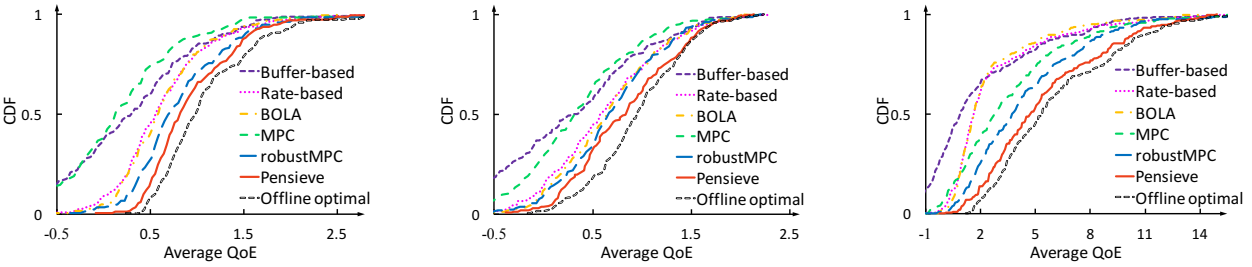
(a) FCC broadband dataset (b) Norway HSDPA dataset

Figure 7: Comparing Pensieve with existing ABR algorithms on broadband and 3G/HSDPA networks. The QoE metrics considered are presented in Table 1. Results are normalized against the performance of Pensieve. Error bars span ± one standard deviation from the average.



(a) QoE_{lin} (b) QoE_{log} (c) QoE_{hd}

Figure 8: Comparing Pensieve with existing ABR algorithms on the QoE metrics listed in Table 1. Results were collected on the FCC broadband dataset. Average QoE values are listed for each ABR algorithm.



(a) QoE_{lin} (b) QoE_{log} (c) QoE_{hd}

Figure 9: Comparing Pensieve with existing ABR algorithms on the QoE metrics listed in Table 1. Results were collected on the Norway HSDPA dataset. Average QoE values are listed for each ABR algorithm.

other two QoE metrics. When network bandwidth is inadequate, the ABR algorithm should build the playback buffer as quickly as possible using the lowest available bitrate. Once the buffer is large enough, it should then make a direct transition to the lowest HD quality (bypassing intermediate bitrates). However, building buffers to a level which circumvents rebuffering and maintains sufficient smoothness requires a lot of foresight. As illustrated by the example in Figure 3b, Pensieve is able to learn such a policy with zero tuning or designer involvement, while other schemes such as robustMPC have difficulty optimizing such long term strategies.

Finally, Pensieve's performance is within 9.6%–14.3% of the offline optimal scheme across all network traces and QoE metrics. Recall that the offline optimal performance cannot be achieved in practice as it requires complete knowledge of future throughput. This shows that there is likely to be little room for any *online* algorithm (without future knowledge) to improve over Pensieve in these scenarios. We revisit the question of Pensieve's optimality in §5.4.

QoE breakdown: To better understand the QoE gains obtained by Pensieve, we analyzed Pensieve's performance on the individual terms in our general QoE definition (Equation 6). Specifically, Figure 10 compares Pensieve to state-of-the-art ABR algorithms in terms of the utility from the average playback bitrate, the penalty from rebuffering, and the penalty from switching bitrates (i.e., the smoothness penalty). In other words, a given scheme's QoE can be computed by subtracting the rebuffering penalty and smoothness penalty from the bitrate utility. In the interest of space, Figure 10 combines the results for the FCC broadband and HSDPA traces.

As shown, a large portion of Pensieve's performance gains come from its ability to limit rebuffering across the different networks and QoE metrics considered. Pensieve reduces rebuffering by 10.6%–32.8% across the three metrics by building up sufficient buffer to handle the network's throughput fluctuations. Additionally, Figure 6 illustrates that Pensieve does not outperform all state-of-the-art schemes on *every* QoE factor. Instead, Pensieve is able to balance

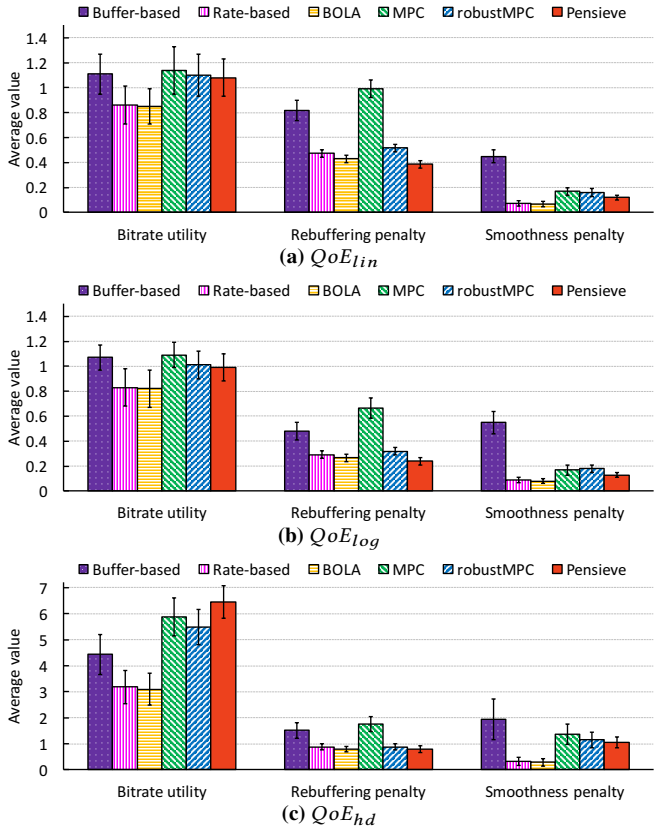


Figure 10: Comparing Pensieve with existing ABR algorithms by analyzing their performance on the individual components in the general QoE definition (Equation 6). Results consider both the broadband and HSDPA networks. Error bars span ± one standard deviation from the average.

each factor in a way that optimizes the QoE metric. For example, to optimize QoE_{hd}, Pensieve achieves the best bitrate utility by always trying to download chunks at HD bitrates, while when optimizing for QoE_{lin} or QoE_{log}, Pensieve focuses on achieving sufficiently high bitrates with the smallest amount of rebuffering and bitrate switches.

5.3 Generalization

In the experiments above, Pensieve was trained with a set of traces collected on the same networks that were used during testing; note that no test traces were directly included in the training set. However, in practice, Pensieve's ABR algorithms could encounter new networks, with different conditions (and thus, with different optimal strategies). To evaluate Pensieve's ability to generalize to new network conditions, we conduct two experiments. First, we evaluate Pensieve in the wild on two real networks. Second, we take generality to the extreme and show how Pensieve can be trained to perform well across multiple environments using a purely synthetic dataset.

Real world experiments: We evaluated Pensieve and several state-of-the-art ABR algorithms in the wild using three different networks: the Verizon LTE cellular network, a public WiFi network at a local coffee shop, and the wide area network between Shanghai and Boston. In these experiments, a client, running on a Macbook Pro

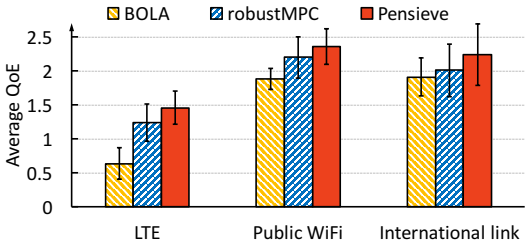


Figure 11: Comparing Pensieve with existing ABR algorithms in the wild. Results are for the QoE_{lin} metric and were collected on the Verizon LTE cellular network, a public WiFi network, and the wide area network between Shanghai and Boston. Bars list averages and error bars span ± one standard deviation from the average.

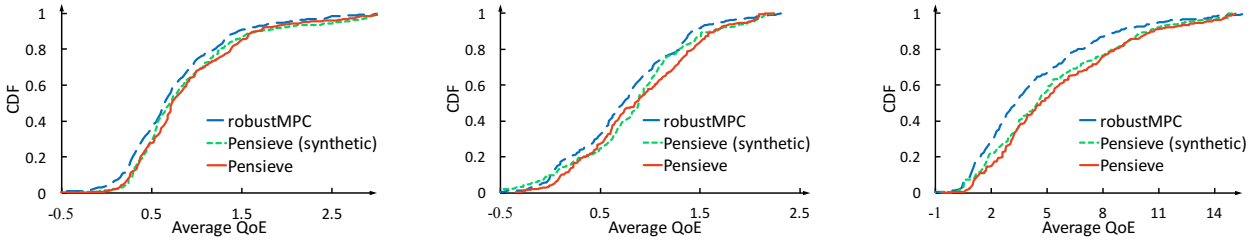
laptop, contacted a video server running on a desktop machine located in Boston. We considered a subset of the ABR algorithms listed in §5.1: BOLA, robustMPC, and Pensieve. On each network, we loaded our test video ten times with each scheme, randomly selecting the order among them. The Pensieve ABR algorithm evaluated here was solely trained using the broadband and HSDPA traces in our corpus. However, even on these new networks, Pensieve was able to outperform the other schemes on the QoE_{lin} metric (Figure 11). Experiments with the other QoE metrics show similar results.

Training with a synthetic dataset: Can we train Pensieve without *any* real network data? Learning from synthetic data alone would of course be undesirable, but we use it as a challenging test of Pensieve's ability to generalize.

We design a data set to cover a relatively broad set of network conditions, with average throughputs ranging from 0.2 Mbps to 4.3 Mbps. Specifically, the dataset was generated using a Markovian model in which each state represented an average throughput in the aforementioned range. State transitions were performed at a 1 second granularity and followed a geometric distribution (making it more likely to transition to a nearby average throughput). Each throughput value was then drawn from a Gaussian distribution centered around the average throughput for the current state, with variance uniformly distributed between 0.05 and 0.5.

We then used Pensieve to compare two ABR algorithms on the test dataset described above (*i.e.,* a combination of the HSDPA and broadband datasets): one trained solely using the synthetic dataset, and another trained explicitly on broadband and HSDPA network traces. Figure 12 illustrates our results for all three QoE metrics listed in Table 1. As shown, Pensieve's ABR algorithm that was trained on the synthetic dataset is able to generalize across these new networks, outperforming robustMPC and achieving average QoE values within 1.6%–10.8% of the ABR algorithm trained directly on the test networks. These results suggest that, in practice, Pensieve will likely be able to generalize to a broad range of network conditions encountered by its clients.

Multiple videos: As a final test of generalization, we evaluated Pensieve's ability to generalize across multiple video properties. To do this, we trained a single ABR model on 1,000 synthetic videos using the techniques described in §4.3. The number of available bitrates



(a) QoE_{lin} (b) QoE_{log} (c) QoE_{hd}

Figure 12: Comparing two ABR algorithms with Pensieve on the broadband and HSDPA networks: one algorithm was trained on synthetic network traces, while the other was trained using a set of traces directly from the broadband and HSDPA networks. Results are aggregated across the two datasets.

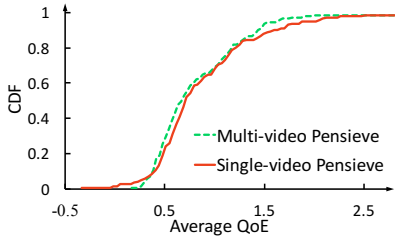


Figure 13: Comparing ABR algorithms trained across multiple videos with those trained explicitly on the test video. The measuring metric is QoE_{lin}.

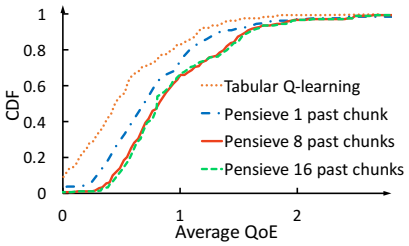


Figure 14: Comparing existing tabular RL schemes with variants of Pensieve that consider different numbers of past throughout measurements. Results are evaluated with QoE_{lin} for the HSDPA network.

for each video was randomly selected from $[3, 10]$,[5] and the value for each bitrate was then randomly chosen from $\{200, 300, 450, 750, 1200, 1850, 2350, 2850, 3500, 4300\}$ kbps. The number of video chunks for each video was randomly generated from $[20, 100]$; chunk sizes were computed by multiplying the standard 4-second chunk size with Gaussian noise $\sim \mathcal{N}(1, 0.1)$. Thus, these videos diverge on numerous properties including the bitrate options (both the number of options and value of each), number of chunks, chunk sizes and video duration. Importantly, we ensured that none of the generated training videos had the exact same bitrate options as the testing video.

We compare this newly trained model to the original model, which was trained solely on the "EnvivioDash3" video described in §5.1 (the test video). Our results measure QoE_{lin} on broadband and HSDPA network traces and are depicted in Figure 13. As shown, the generalized ABR algorithm trained across multiple videos is able to achieve average QoE_{lin} values within 3.2% of the model trained explicitly on the test video. These results suggest that in practice, Pensieve servers can be configured to use a small number of ABR algorithms to improve streaming for a diverse set of videos.

5.4 Pensieve Deep Dive

In this section, we describe microbenchmarks that provide a deeper understanding of Pensieve and shed light on some practical concerns with using RL-generated ABR algorithms. We begin by comparing Pensieve's RL algorithm to tabular RL schemes, which are used by some previous proposals for applying RL to video streaming. We then analyze how robust Pensieve is to varying system parameters (e.g., neural network hyperparameters, client-to-ABR server latency) and evaluate its training time. Finally, we conduct experiments to understand how close Pensieve is to the optimal scheme.

[5]This range represents the two ends of the spectrum for the number of bitrates supported by the videos provided by the DASH reference client [11].

Comparison to tabular RL schemes: A few recent schemes [6, 8, 9, 47] have applied "tabular" RL to video streaming. Tabular methods represent the model to be learned as a table, with separate entries for all states (e.g., client observations) and actions (e.g., bitrate decisions). Tabular methods do not scale to large state/action spaces. As a result, such schemes are forced to restrict the state space by making simplified (and unrealistic) assumptions about network behavior. For example, the most recent tabular RL scheme for ABR [6] assumes network throughput is Markovian, i.e., the future bandwidth depends only on the throughput observed in the last chunk download.

To compare these approaches with Pensieve, we implemented a tabular RL scheme with Q-learning [29]. Our implementation is modeled after the design in [6]. The state space is the same as described in §4.2 except that the past bandwidth measurement is restricted to only 1 sample (as in [6]). The past bandwidth measurement and buffer occupancy are quantized with 0.5 Mbps and 1 second granularity respectively. Our quantization is more fine-grained than that used in [6]; we found that this resulted in better performance in our experiments. (Note that simulation results in [6] used synthetically generated network traces with the Markov property.)

Figure 14 shows a significant performance gap (46.3%) between the tabular scheme and Pensieve. This result shows that simple network models (e.g., Markovian dynamics) fail to capture the intricacies of real networks. Unlike tabular RL methods, Pensieve can incorporate a large amount of throughput history into its state space to optimize for actual network characteristics.

To better understand the importance of throughput history, we tried to answer: how many past chunks are necessary to include in the state space? To do this, we generated three ABR algorithms with Pensieve that consider different numbers of throughput measurements:

Number of neurons and filters (each)	Average QoE_{hd}
4	3.850 ± 1.215
16	4.681 ± 1.369
32	5.106 ± 1.452
64	5.496 ± 1.411
128	5.489 ± 1.378

Table 2: Sweeping the number of CNN filters and hidden neurons in Pensieve's learning architecture.

Number of hidden layers	Average QoE_{hd}
1	5.489 ± 1.378
2	5.396 ± 1.434
5	4.253 ± 1.219

Table 3: Sweeping the number of hidden layers in Pensieve's learning architecture.

1, 8, and 16 past video chunks. As shown in Figure 14, considering only 1 past chunk does not provide enough information to infer future network characteristics and hurts performance. Considering the past 8 chunks allows Pensieve to extract more information and improve its policy. However, the benefits of additional throughput measurements eventually plateau. For example, providing Pensieve with measurements for the past 16 chunks only improves the average QoE by 1% compared to using throughput measurements for 8 chunks. This marginal improvement comes at the cost of higher burden during training.

Neural network (NN) architecture: Starting with Pensieve's default learning architecture (Figure 5), we swept a range of NN parameters to understand the impact that each has on QoE_{hd}[6]. First, using a single fixed hidden layer, we varied the number of filters in the 1D-CNN and the number of neurons in the hidden merge layer. These parameters were swept in tandem, i.e., when 4 filters were used, 4 neurons were used. Results from this sweep are presented in Table 2. As shown, performance begins to plateau once the number of filters and neurons each exceed 32. Additionally, notice that once these values reach 128 (Pensieve's default configuration), variance levels decrease while average QoE values remain stable.

Next, after fixing the number of filters and hidden neurons to 128, we varied the number of hidden layers in Pensieve's architecture. The resulting QoE_{hd} values are listed in Table 3. Interestingly, we find that the shallowest network of 1 hidden layer yields the best performance; this represents the default value in Pensieve. Performance steadily degrades as we increase the number of hidden layers. However, it is important to note that our sweep used a fixed learning rate and number of training iterations. Tuning these parameters to cater to deeper networks may improve performance, as these networks generally take longer to train.

Client-to-ABR server latency: Recall that Pensieve deploys the RL-generated ABR model on an ABR server (not the video streaming clients). Under this deployment model, clients must first query the Pensieve's ABR server to determine the bitrate to use for the next chunk, before downloading that chunk from a CDN server. To understand the overhead incurred by this additional round trip, we performed a sweep of the RTT between the client player and ABR server, considering values from 0 ms–100 ms. This experiment used the same setup described in §5.1, and measured the QoE_{hd} metric. Table 4 lists our results, highlighting that the latency from this additional RTT has minimal impact on QoE: the average QoE_{hd} with a 100 ms latency was within 3.5% of that when the latency was 0 ms. The reason is that the latency incurred from the additional round trip to Pensieve's ABR server is masked by the playback buffer occupancy and chunk download times [18, 21].

[6] QoE_{hd} is used for the parameter sweep experiments as it highlights performances differences more clearly.

RTT (ms)	Average QoE_{hd}
0	5.407 ± 1.820
20	5.356 ± 1.768
40	5.309 ± 1.768
60	5.271 ± 1.773
80	5.217 ± 1.742
100	5.219 ± 1.748

Table 4: Average QoE_{hd} values when different RTT values are imposed between the client and Pensieve's ABR server.

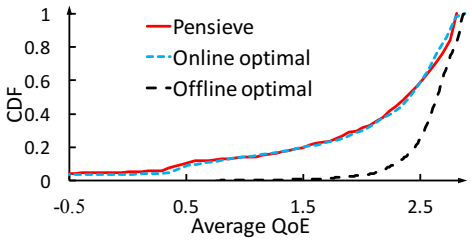


Figure 15: Comparing Pensieve with online and offline optimal. The experiment uses the QoE_{lin} metric.

Training time: To measure the overhead of generating ABR algorithms using RL, we profiled Pensieve's training process. Training a single algorithm required approximately 50,000 iterations, where each iteration took 300 ms and corresponded to 16 agents updating their parameters in parallel (using the training approach described in §4.2). Thus, in total, training took approximately 4 hours. We note that this cost is incurred offline and can be performed infrequently depending on environment stability.

Optimality: Our results illustrate that Pensieve is able to outperform existing ABR algorithms. However, Figures 8 and 9 show that there still exists a gap between Pensieve and the *offline* optimal. It is unclear to what extent this gap can be closed since the offline optimal scheme makes decisions with perfect knowledge of future bandwidth (§5.1). A practical *online* algorithm would only know the underlying distribution of future network throughput (rather than the precise throughput values). Thus Pensieve may in fact be much closer to the optimal online scheme.

Of course, we cannot compute the optimal online algorithm for real network traces, as we do not know the stochastic processes underlying these traces. Thus, to understand how Pensieve compares to the best online algorithm, we conducted a controlled experiment where the download time for each chunk is generated according to a known Markov process. Specifically, we simulate the download time T_n of chunk n as $T_n = T_{n-1}(R_n/R_{n-1}) + \epsilon$, where R_n is the bitrate of chunk n and $\epsilon \sim \mathcal{N}\left(0,\ \sigma^2\right)$. For this model, it is straightforward to compute the optimal online decisions using dynamic programming. See [28] for details.

To compare the optimal online algorithm with Pensieve, we set the video chunk length δ to be 4 seconds, mimicking the "EnvivioDash3" video described in §5.1. The initial download time T_0 was set to 4 seconds for bitrate $R_0 = 2$ kbps, and the standard deviation σ of the Gaussian noise was set to 0.5. Both buffer occupancy and download time were quantized to 0.1 second to run dynamic programming.

We used the same setup in §5.1 to train a Pensieve agent in this simulated environment, and compared Pensieve's performance with the online and offline optimal schemes. Our experiment considers the QoE_{lin} metric and the results are depicted in Figure 15. As expected, the offline optimal outperforms the online optimal by 9.1% on average. This is comparable to the performance gap between Pensieve and the offline optimal observed in §5.2. Indeed, the average QoE achieved by Pensieve is within 0.2% of the online optimal.

6 DISCUSSION

Deploying Pensieve in practice: In our current implementation, Pensieve's ABR server runs on the server-side of video streaming applications. This approach offers several advantages over deployment in client video players. First, a variety of client-side devices are used for video streaming today, ranging from multi-core desktop machines to mobile devices to TVs. By using an ABR server to simply guide client bitrate selection, Pensieve can easily support this broad range of video clients without modifications that may sacrifice performance. Additionally, ABR algorithms are traditionally deployed on clients which can quickly react to changing environments [51]. However, as noted in §4, Pensieve preserves this ability by having clients include observations about the environment in each request sent to the ABR server. Further, our results suggest that the additional latency required to contact Pensieve's ABR server has negligible impact on QoE (§5.4). If direct deployment in client video players is preferred, Pensieve could use compressed neural networks [16] or represent them in languages supported by many client applications, e.g., JavaScript [45].

Periodic and online training: In this paper, we primarily described RL-based ABR algorithm generation as an offline task. That is, with Pensieve, we assumed that the ABR algorithm was generated a priori (during a training phase) and was then unmodified after deployment. However, Pensieve can naturally support an approach in which an ABR algorithm is generated or updated *periodically* as new data arrives. This technique would enable ABR algorithms to further adapt to the exact conditions that video clients are experiencing at a given time. The extreme version of this approach is to train *online* directly on the video client. However, online training on video clients raises two challenges. First, it increases the computational overhead for the client. Second, it requires algorithms that can learn from small amounts of data and converge to a good policy quickly.

Retraining frequency depends on how quickly new network behaviors emerge to which existing models do not generalize. While our generalization results (§5.3) suggest that retraining frequently may not be necessary, techniques to determine when to retrain and investigating the tradeoffs with online training are interesting areas for future work.

7 RELATED WORK

The earliest ABR algorithms can be primarily grouped into two classes: rate-based and buffer-based. Rate-based algorithms [21, 42] first estimate the available network bandwidth using past chunk downloads, and then request chunks at the highest bitrate that the network is predicted to support. For example, Festive [21] predicts throughput to be the harmonic mean of the experienced throughput for the past 5 chunk downloads. However, these methods are hindered by the biases present when estimating available bandwidth on top of HTTP [22, 26]. Several systems aim to correct these throughput estimates using smoothing heuristics and data aggregation techniques [42], but accurate throughput prediction remains a challenge in practice [53].

In contrast, buffer-based approaches [19, 41] solely consider the client's playback buffer occupancy when deciding the bitrates for future chunks. The goal of these algorithms is to keep the buffer occupancy at a pre-configured level which balances rebuffering and video quality. The most recent buffer-based approach, BOLA [41], optimizes for a specified QoE metric using a Lyapunov optimization formulation. BOLA also supports chunk download abandonment, whereby a video player can restart a chunk download at a lower bitrate level if it suspects that rebuffering is imminent.

Each of these approaches performs well in certain settings but not in others. Specifically, rate-based approaches are best at startup time and when link rates are stable, while buffer-based approaches are sufficient and more robust in steady state and in the presence of time-varying networks [19]. Consequently, recently proposed ABR algorithms have also investigated combining these two techniques. The state-of-the-art approach is MPC [51], which employs model predictive control algorithms that use both throughput estimates and buffer occupancy information to select bitrates that are expected to maximize QoE over a horizon of several future chunks. However, MPC still relies heavily on accurate throughput estimates which are not always available. When throughput predictions are incorrect, MPC's performance can degrade significantly. Addressing this issue requires heuristics that make throughput predictions more conservative. However, tuning such heuristics to perform well in different environments is challenging. Further, as we observed in §3, MPC is often unable to plan far enough into the future to apply the policies that would maximize performance in given settings.

A separate line of work has proposed applying RL to adaptive video streaming [6, 8, 9, 47]. All of these schemes apply RL in a "tabular form," which stores and learns the value function for all states and actions explicitly, rather than using function approximators (e.g., neural networks). As a result, these schemes do not scale to the large state spaces necessary for good performance in real networks, and their evaluation has been limited to simulations with synthetic network models. For example, the most recent tabular scheme [6] relies on the fundamental assumption that network bandwidth is Markovian, i.e., the future bandwidth depends only on the throughput observed in the last chunk download. This assumption confines the state space to consider only one past bandwidth measurement, making the tabular approach feasible to implement. As we saw in §5.4, the information contained in one past chunk is not sufficient to accurately infer the distribution of future bandwidth. Nevertheless, some of the techniques used in the existing RL video

streaming schemes (e.g., Post-Decision States [6, 35]) could be used to accelerate learning in Pensieve as well.

8 CONCLUSION

We presented Pensieve, a system which generates ABR algorithms using reinforcement learning. Unlike ABR algorithms that use fixed heuristics or inaccurate system models, Pensieve's ABR algorithms are generated using observations of the resulting performance of past decisions across a large number of video streaming experiments. This allows Pensieve to optimize its policy for different network characteristics and QoE metrics directly from experience. Over a broad set of network conditions and QoE metrics, we found that Pensieve outperformed existing ABR algorithms by 12%–25%.

Acknowledgments. We thank our shepherd, John Byers, and the anonymous SIGCOMM reviewers for their valuable feedback. We also thank Te-Yuan Huang for her guidance regarding video streaming in practice, and Jiaming Luo for fruitful discussions regarding the learning aspects of the design. This work was funded in part by NSF grants CNS-1617702, CNS-1563826, and CNS-1407470, the MIT Center for Wireless Networks and Mobile Computing, and a Qualcomm Innovation Fellowship.

REFERENCES

[1] M. Abadi et al. 2016. TensorFlow: A System for Large-scale Machine Learning. In *OSDI*. USENIX Association.
[2] Akamai. 2016. dash.js. https://github.com/Dash-Industry-Forum/dash.js/. (2016).
[3] S. Akhshabi, A. C. Begen, and C. Dovrolis. 2011. An Experimental Evaluation of Rate-adaptation Algorithms in Adaptive Streaming over HTTP. In *MMSys*.
[4] M. Allman, V. Paxson, and E. Blanton. 2009. *TCP congestion control*. RFC 5681.
[5] C. M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.
[6] F. Chiariotti et al. 2016. Online learning adaptation strategy for DASH clients. In *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 8.
[7] Cisco. 2016. Cisco Visual Networking Index: Forecast and Methodology, 2015-2020.
[8] M. Claeys et al. 2013. Design of a Q-learning-based client quality selection algorithm for HTTP adaptive video streaming. In *Adaptive and Learning Agents Workshop*.
[9] M. Claeys et al. 2014. Design and optimisation of a (FA) Q-learning-based HTTP adaptive streaming client. *Connection Science* (2014).
[10] Federal Communications Commission. 2016. Raw Data - Measuring Broadband America. (2016). https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016
[11] DASH Industry Form. 2016. Reference Client 2.4.0. http://mediapm.edgesuite.net/dash/public/nightly/samples/dash-if-reference-player/index.html. (2016).
[12] F. Dobrian et al. 2011. Understanding the Impact of Video Quality on User Engagement. In *SIGCOMM*. ACM.
[13] G. Fairhurst et al. 2015. Updating TCP to Support Rate-Limited Traffic. *RFC 7661* (2015).
[14] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks.. In *Aistats*, Vol. 9. 249–256.
[15] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús. 1996. *Neural network design*. PWS publishing company Boston.
[16] S. Han, H. Mao, and W. J. Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR, abs/1510.00149* 2 (2015).
[17] M. Handley, J. Padhye, and S. Floyd. 2000. TCP Congestion Window Validation. *RFC 2861* (2000).
[18] T.Y. Huang et al. 2012. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference (IMC)*. ACM.
[19] T.Y. Huang et al. 2014. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *SIGCOMM*. ACM.
[20] M. Jaderberg et al. 2017. Reinforcement learning with unsupervised auxiliary tasks. In *ICLR*.
[21] J. Jiang, V. Sekar, and H. Zhang. 2012. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *CoNEXT*.
[22] J. Jiang et al. 2016. CFA: A Practical Prediction System for Video QoE Optimization. In *NSDI*. USENIX Association.
[23] I. Ketykó et al. 2010. QoE Measurement of Mobile YouTube Video Streaming. In *Proceedings of the 3rd Workshop on Mobile Video Delivery (MoViD)*. ACM.
[24] V. R Konda and J. N. Tsitsiklis. 2000. Actor-critic algorithms. In *Advances in neural information processing systems*. 1008–1014.
[25] S. S. Krishnan and R. K. Sitaraman. 2012. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference (IMC)*. ACM.
[26] Z. Li et al. 2014. Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale. *IEEE Journal on Selected Areas in Communications* (2014).
[27] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *HotNets*. ACM.
[28] H. Mao, R. Netravali, and M. Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. (2017). http://web.mit.edu/pensieve/content/pensieve-tech-report.pdf
[29] V. Mnih et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.
[30] V. Mnih et al. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.
[31] R. K.P. Mok, E. W. W. Chan, X. Luo, and R. K.C. Chang. 2011. Inferring the QoE of HTTP Video Streaming from User-viewing Activities. In *Proceedings of the First ACM SIGCOMM Workshop on Measurements Up the Stack (W-MUST)*.
[32] R. K. P. Mok, E. W. W. Chan, and R. K. C. Chang. 2011. Measuring the quality of experience of HTTP video streaming. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*.
[33] R. Netravali et al. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*.
[34] K. Piamrat, C. Viho, J. M. Bonnin, and A. Ksentini. 2009. Quality of Experience Measurements for Video Streaming over Wireless Networks. In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations (ITNG)*. IEEE Computer Society.
[35] W. B. Powell. 2007. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons.
[36] B. Recht, C. Re, S. Wright, and F. Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 693–701.
[37] H. Riiser et al. 2013. Commute Path Bandwidth Traces from 3G Networks: Analysis and Applications. In *Proceedings of the 4th ACM Multimedia Systems Conference (MMSys)*. ACM.
[38] J. K. Rowling. 2000. *Harry Potter and the Goblet of Fire*. London: Bloomsbury.
[39] Sandvine. 2015. Global Internet Phenomena-Latin American & North America.
[40] D. Silver et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016), 484–503.
[41] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. 2016. BOLA: Near-Optimal Bitrate Adaptation for Online Videos. *CoRR* abs/1601.06748 (2016).
[42] Y. Sun et al. 2016. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *SIGCOMM*. ACM.
[43] R. S. Sutton and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. MIT Press.
[44] R. S. Sutton et al. 1999. Policy gradient methods for reinforcement learning with function approximation.. In *NIPS*, Vol. 99. 1057–1063.
[45] Synaptic. 2016. synaptic.js – The javascript architecture-free neural network library for node.js and the browser. https://synaptic.juancazala.com/. (2016).
[46] TFLearn. 2017. TFLearn: Deep learning library featuring a higher-level API for TensorFlow. http://tflearn.org/. (2017).
[47] J. van der Hooft et al. A learning-based algorithm for improved bandwidth-awareness of adaptive streaming clients. In *2015 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE.
[48] A. S. Vezhnevets et al. 2017. FeUdal Networks for Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1703.01161* (2017).
[49] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*.
[50] Y. Wu and Y. Tian. 2017. Training agent for first-person shooter game with actor-critic curriculum learning. In *ICLR*.
[51] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. 2015. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *SIGCOMM*. ACM.
[52] Y. Zaki et al. 2015. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM Computer Communication Review*. ACM.
[53] X. K. Zou. 2015. Can Accurate Predictions Improve Video Streaming in Cellular Networks?. In *HotMobile*. ACM.