Rediscovering heuristics: A litmus test for RL in networked systems

Seoyul Oh Yikai Zheng* Eric Xiang Gao Deepak Vasisht Francis Y. Yan
University of Illinois Urbana-Champaign **Tsinghua University*

Abstract

Reinforcement learning (RL) holds promise for producing policies that outperform human-designed heuristics in networked systems, yet its training process remains brittle and opaque. We present a diagnostic methodology for RL training based on a key insight: strong heuristics in networked systems are typically governed by only a handful of parameters, so a compact RL policy class that is just expressive enough to represent a heuristic baseline should reliably *rediscover* it by learning to match its behavior and performance.

We argue that heuristic rediscovery serves as a litmus test for the RL training pipeline. While passing this test does not validate a deep neural policy, failing it exposes problems in training (e.g., a poorly chosen reward function or environment distribution) before additional model complexity obscures their causes. Moreover, the compact policy space makes the training process amenable to visualization and diagnosis. It enables us to inspect three views that are normally hidden: the return landscape, the optimization path, and the correctness properties of the learned policy.

Our evaluation showcases the value of heuristic rediscovery in adaptive bitrate streaming, revealing when and why RL training succeeds or fails. Experiments on congestion control further suggest the generality of our methodology.

CCS Concepts

• **Computing methodologies** → **Reinforcement learning**; • **Networks** → **Network performance analysis**.

Keywords

Reinforcement learning, Training diagnosis, Heuristic rediscovery, Networked systems

ACM Reference Format:
Seoyul Oh, Yikai Zheng, Eric Xiang Gao, Deepak Vasisht, and Francis Y. Yan. 2026. Rediscovering heuristics: A litmus test for RL in networked systems. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16-17, 2026, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3847290.3859964

* Work done while visiting UIUC.

This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859964

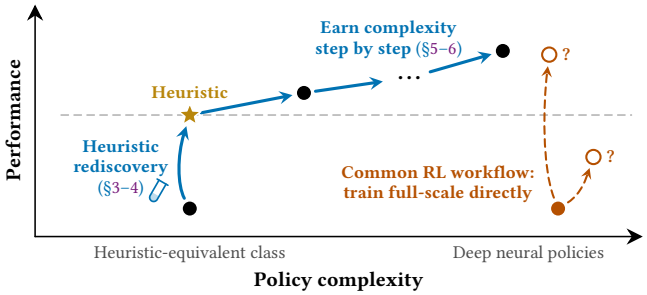


Figure 1: A common workflow commits to a full-scale policy class before training, learning only afterward whether the resulting policy outperforms a strong heuristic. Our methodology first trains a heuristic-equivalent policy as a litmus test; after rediscovering the heuristic, the pipeline incrementally introduces model complexity.

1 Introduction

Reinforcement learning (RL) has emerged as a promising approach to developing control policies for networked systems. Unlike handcrafted heuristics, RL-based policies learn through experience and reward feedback, and have demonstrated strong potential in applications such as adaptive bitrate (ABR) streaming [16], real-time video [1, 33], resource scheduling [6, 17, 30], and congestion control [14, 22].

However, RL-trained policies often perform poorly after deployment [31] and exhibit counterintuitive behavior [7], hindering real-world adoption. To address this challenge, prior work has sought to interpret [13, 19, 23], verify [7, 32], or enhance [8, 29] a policy *after* initial training. In contrast, we focus on diagnosing the end-to-end training process itself to understand *why training arrived there.*

RL training is known to be "black magic" because its outcome depends on a complex interplay of design choices, such as the reward function, state representation, and environment distribution. Deep neural policies further complicate this by obscuring how these choices influence the learned behavior. As a result, when an RL policy underperforms, practitioners would simply sweep over training configurations with little insight into the underlying causes of failure. Richer diagnostic views are generally unavailable as the policy often resides in a high-dimensional parameter space, making training difficult to visualize and analyze.

Our experience with RL training motivates a new diagnostic methodology centered on the following question: *before trusting RL to discover a better policy, can it reliably rediscover a known heuristic?* We argue that heuristic rediscovery serves as a practical litmus test for RL training (Fig. 1), providing both a minimum capability check and an instrument to visualize and diagnose the training pipeline.

The heuristic litmus test. Many time-tested controllers in networked systems are simple rule-based heuristics. Buffer-based ABR policies map the playback buffer level to a video bitrate using a few thresholds [12, 26]; loss- or delay-based congestion control algorithms respond to loss signals [11, 21] or RTTs [3, 4] according to predefined rules.

We observe that the core behavior of these heuristics can be captured by a compact RL policy class with only a handful of parameters. One such class is logistic or softmax regression, which *linearly* maps a heuristic's observed state (input features) to logits and then normalizes them into probabilities over actions via a sigmoid or softmax function. This insight suggests a minimum capability test for an RL training pipeline: if we restrict the policy space—including both the state-action space and the parameter space—such that it is just expressive enough to represent a heuristic, RL training should reliably produce a policy that matches the heuristic's behavior and performance in principle. We refer to this test as *rediscovering the heuristic* (§3.1).

Rediscovery does not mean recovering the heuristic's exact parameters or behavior cloning (no supervision from the heuristic is used). Rather, it means that a minimal but sufficiently expressive policy class, trained solely through reward-driven RL, can learn the heuristic's essential control structure and achieve comparable performance. When the heuristic is incompatible with the chosen RL formulation, we apply the test to a compact abstraction of the heuristic (§3.1). Passing this test establishes only a minimum bar: it does not guarantee that a subsequent neural policy will be robust or perform better. However, failing the test often reveals problems to investigate in the training pipeline before increasing policy complexity. It does not imply that a larger policy will always fail.

Compact policies also facilitate diagnosis. The heuristic litmus test becomes particularly useful because reducing the policy class also makes the training process amenable to visualization and diagnosis.

Normally, the objective optimized by RL—the expected return—is intractable to evaluate over a high-dimensional parameter space (e.g., that induced by a deep neural policy). As a result, neither the return landscape nor the optimizer's trajectory through it is generally available for diagnosis. With a two-parameter policy, however, we can visualize the return landscape over a chosen parameter range and relate its regions to policy behaviors. For larger policy classes, we use projections to visualize the landscape.

We realize this idea through three diagnostic views (§3.2): the return landscape, the optimization path across that landscape, and domain-specific correctness properties of the learned policy. Together, these views reveal failures that would otherwise appear as generic RL instability. For example, the return landscape may contain a desirable heuristic-quality point, yet poor initialization, high gradient variance, or excessively large policy updates may prevent the optimizer from reaching it. Conversely, optimization may behave exactly as intended, while a misweighted reward or an unrepresentative trace set reshapes the return landscape so that it favors aggressive or conservative behavior. Distinguishing between these failure modes provides actionable guidance on how to repair the training pipeline.

Case studies and findings. We first instantiate the heuristic litmus test from the ground up in a controlled ABR setting, which provides the most intuitive visualizations (§4). We find that even vanilla policy gradient reliably learns correct, near-optimal policies across a broad hyperparameter range. We also identify two representative failure modes: *unstable optimization paths* (sound landscape, straying optimizer) and *distorted return landscapes* (inappropriate reward weight or trace distribution shifts the optimum itself). Finally, we show that our methodology generalizes beyond the controlled ABR setting, including RL-based congestion control (§5).

Despite these encouraging results, our evaluation remains preliminary. In particular, our results do not yet explain training behavior in a full neural-network parameter space, which is notoriously difficult to study. Section 6 discusses more limitations and directions for future work.

Contributions:

- We introduce *heuristic rediscovery* as a minimum capability test for RL in networked systems. When the policy class can represent a strong heuristic, training should learn a policy that matches its behavior and performance.
- We turn the compact policy class into a diagnostic instrument that combines three views that are normally hidden: the return landscape, the optimization path, and the correctness properties of the learned policy.
- We study the test primarily in ABR and reveal when and why training succeeds or fails. We further demonstrate that the methodology generalizes to congestion control.

2 Background

RL and policy gradients. An RL policy $\pi_\theta(a \mid s)$ defines a probability distribution over actions a given state s, parameterized by θ: at each step t, the policy observes a state s_t, selects an action a_t, and receives a reward r_t; the resulting interaction sequence with the environment is called a trajectory. RL seeks parameters that maximize the *expected return* $J(\theta)$, i.e., the accumulated reward averaged over trajectories drawn from π_θ (denoted $\mathbb{E}_{\pi_\theta}$):

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} r_t\right]. \tag{1}$$

Policy gradient methods [27] update the parameters by gradient ascent, $\theta_{k+1} \leftarrow \theta_k + \eta \nabla_\theta J(\theta_k)$ with learning rate η, where the *policy gradient* $\nabla_\theta J(\theta)$ admits the analytical form:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \hat{R}_t\right], \qquad (2)$$

where $\hat{R}_t$ is the reward-to-go, i.e., the sum of rewards from step t onward. Intuitively, ascending this gradient pushes up the (log) probabilities of actions that yield higher return. We implement this policy gradient method for simplicity, but more generally, the reward-to-go can be replaced by an advantage function [25].

Evaluating either expectation entails enumerating *all* possible trajectories, which is generally intractable (except under specific conditions we explain later); the policy gradient is therefore typically estimated with the sample mean of a batch of trajectories collected by the current π_θ. Reducing the variance of this estimate is important for stable learning.

Primary case: adaptive bitrate (ABR) streaming. An ABR client selects among a handful of pre-encoded bitrate levels for each few-second video chunk, trading video quality against rebuffering (playback stalls) to maximize the user's quality of experience (QoE). The widely deployed buffer-based algorithm (BBA) [12] makes this choice (almost) linearly in the playback buffer level—a policy class we return to in §3.2. Cast as RL, each chunk is one step: the state is the playback buffer level, the action a bitrate index, and the reward the chunk's QoE contribution; a trajectory is one video session. Pensieve [16] uses this formulation, training a deep neural network policy over manually selected inputs with an actor-critic variant of policy gradient [20]. Later studies found its behavior erratic and hard to audit [7, 8, 19, 29], and the Puffer deployment [31] reports that it does not consistently outperform the far simpler BBA.

Second case: congestion control (CC). A congestion controller adjusts a sender's rate to keep the bottleneck link utilized without increasing queuing delay or packet loss. Here too, strong deployed algorithms follow simple rules with a few parameters: delay-based schemes such as Vegas [4] and Copa [3] back off as queuing delay rises. Aurora [14] casts rate control as RL, training a deep neural policy on network statistics. §5.2 instantiates our methodology on CC.

3 Heuristic Rediscovery as a Litmus Test

We propose heuristic rediscovery as a litmus test for diagnosing RL training. We first define the test and explain why it is a meaningful minimum capability check for an RL training pipeline (§3.1). We then introduce three diagnostic views that the test surfaces (§3.2), using Fig. 2 as a running example.

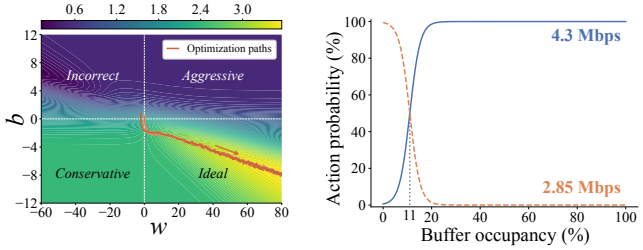


Figure 2: Diagnostic views rendered in the controlled 2-bitrate ABR setting. **(a)** Landscape of $J(\theta)$ over the plotted 2-D parameter space, overlaid with optimization paths (V1 + V2). **(b)** Bitrate selection probability of the learned policy (V3).

3.1 The Litmus Test

Constructing the test. Many heuristics in networked systems have core control behaviors that can be expressed with only a few parameters. We use these behaviors to define a rediscovery target and choose a compatible compact policy class. We distinguish two cases based on whether the heuristic can be represented directly under the chosen RL formulation.

Case 1: Direct heuristic rediscovery. The heuristic already admits a compact representation compatible with the RL input features, actions, and decision intervals. We use that rule directly as the target, as with the BBA-based buffer rules in our ABR experiments [12] (§4).

Case 2: Core-behavior rediscovery. Direct rediscovery is impractical under the chosen RL formulation. We identify a core control mechanism and its input and internal-state dependencies, then construct a compatible compact representation. The resulting target concerns the selected behavior in this reduced formulation; its usefulness must be evaluated there. Our CC instance (§5.2) follows this case by targeting delay-reactive feedback.

In both cases, training uses reward alone. Our instances use logistic or softmax regression to map input features linearly to logits and then to action probabilities. Other target behaviors may call for different input features or compact policy classes. Domain expertise guides the choice of heuristic, target behavior, and policy representation, and we discuss the implications of these choices in §6.

Requirements for passing. The policy must meet three requirements. First, *correctness*: it must satisfy the expected behavior of the heuristic. For buffer-based ABR, the bitrate must not decrease as the buffer grows. Thus, for $b_1 < b_2$,

$$\arg\max_a \pi_\theta(a \mid b_1) \le \arg\max_a \pi_\theta(a \mid b_2).$$

Second, *performance*: the policy must achieve utility within a stated margin of that heuristic. Third, *reliability*: correctness and performance must hold across a reasonable set of

seeds, initializations, and training configurations. These requirements must be set before training, and the policy must meet all three to pass. Rediscovery does not require the same parameters or the same action in every state. It requires the same core control behavior and comparable utility.

Why this is a minimum capability check. The minimum capability check requires a known, attainable heuristic. If the policy class can express the selected heuristic, a failed run cannot be blamed on insufficient model capacity. The cause must instead lie in the reward, training distribution, optimization, or implementation. Passing clears only this minimum bar; it does not show that a larger neural policy will be robust or better. Failing this test is more informative than passing it, and the views below help localize its cause.

3.2 Three Diagnostic Views

The requirements above define the test; the following views explain its outcome. A *return landscape* is the RL counterpart of a loss surface: with the reward and environment distribution fixed, it maps policy parameters to expected return $J(\theta)$, which training seeks to maximize. We compute returns exactly in the finite deterministic setting of §4. Compact classes make the following views practical, using projections when needed:

(V1) *Return landscape characterization.* V1 asks whether the reward and training environment assign high returns to the required behavior. The location and geometry of high-return regions—including ridges, basins, and plateaus—show which policies the objective favors and where gradient ascent can plausibly arrive [9]. Because V1 does not run an optimizer, it can expose a misaligned reward or trace distribution before training.

(V2) *Optimization path tracing.* V2 asks whether training follows the objective. Overlaying the parameter trajectory on V1 shows whether the optimizer climbs a useful ridge, stalls on a plateau, or is deflected by gradient noise into a poorly performing region.

(V3) *Correctness properties.* V3 asks whether the learned policy satisfies the behavioral requirement specified by the test. We evaluate the action selected by the deployed policy rather than infer correctness from return: a policy can score well on the training objective while violating a domain requirement.

► **Example:** Fig. 2 shows all three views for the controlled ABR setting in §4. The policy has two parameters, w and b, and selects the high bitrate with probability $\pi_\theta(\text{high}) = \sigma(w \cdot \text{buffer} + b)$. V1 plots the return landscape $J(w, b)$, V2 overlays the training paths, and V3 shows the resulting bitrate-selection rule. Together, these views divide the plotted parameter domain into four behavior classes: BBA-like (ideal), conservative, aggressive, and bitrate decreasing with buffer (incorrect). The last class violates the behavioral requirement even if it achieves high return.

4 A Controlled ABR Setting

In this section, we first trade realism for a training process that can be visualized and explained most effectively, by restricting the setting to two bitrates, short episodes, and constant-bandwidth traces. The resulting policy has two parameters but still represents a BBA-like buffer threshold [12], so we can plot its full return landscape and training paths without removing the target behavior. The short horizon makes every bitrate sequence enumerable, while constant traces isolate adaptation to capacity from temporal variation.

We emphasize that these are analytical conveniences, *not* requirements of the method: §5.1 restores the original large action space. Next, we show a base configuration that passes the test, then make changes to isolate two failure modes.

Setup. We use the policy from Fig. 2, with bitrates of 2.85 and 4.3 Mbps. Each episode contains eight 4-second chunks and runs on one of seven constant-bandwidth traces from 3.0 to 6.0 Mbps in 0.5 Mbps increments. The reward follows Pensieve [16]: $\text{bitrate}/\text{max_bitrate} - \lambda \cdot \text{stall_time}/\text{chunk_length}$ with $\lambda = 4.3$, and omits the smoothness term to keep the reward interpretable. The finite number of possible sequences lets us compute the expected return and gradient exactly; training itself uses sampled policy-gradient updates.

4.1 When Training Succeeds

Our base configuration initializes the policy near the origin and uses policy gradient with batch size 16 and Adam [15] at learning rate 0.01.

The pipeline passes. We check the three requirements of §3.1 directly. *Correctness* (V3): the learned policy increases the probability of the high bitrate as the buffer grows (Fig. 2b), satisfying the monotonicity requirement and recovering BBA's threshold structure from reward alone. *Performance*: we compare against BBA*, an oracle over deterministic one-threshold policies. Such a policy selects 2.85 Mbps below a buffer threshold and 4.3 Mbps above it; BBA* picks the threshold that maximizes average QoE over the seven traces. It is therefore optimal within the BBA-like behavior that the litmus test asks RL to rediscover. The learned policy's performance nearly matches BBA* using the base configuration (the red bars in Fig. 3). *Reliability*: correctness and performance hold across repeated runs with near-origin initializations, batch sizes 1–64, and learning rates 0.005–0.1; QoE only degrades at more extreme learning rates and under SGD (Fig. 3). These experiments use the training traces for

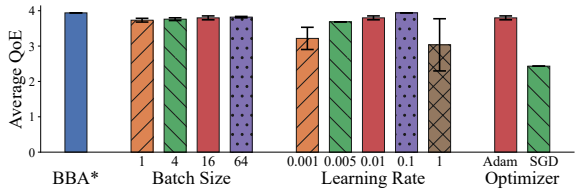


Figure 3: Average QoE across different configurations. Red bars represent the base configuration. Error bars show 95% CIs.

evaluation; they test the training pipeline, not generalization to unseen or time-varying traces.

Why it passes (V1 and V2). The verdict above is behavioral; the remaining two views explain it. The return landscape (V1) in Fig. 2a has a high-return ridge in the ideal quadrant and lower-return plateaus elsewhere. The ridge flattens as return increases, but gradients also vanish on the inferior plateaus, so a small gradient—or a flat reward curve—cannot certify success; V1 separates the desired ridge from places where training stops prematurely. The optimization paths (V2) overlaid in the same figure enter the ideal quadrant from near-origin initializations and follow the ridge toward higher return. The objective contains the target and training reaches it, explaining why the test is passed.

We note that this result does not extend to every initialization, optimizer, or larger neural policy. Those boundaries are the subject of the failures below.

4.2 When Training Fails

Outside the range tested above, two failure modes emerge, and the three diagnostic views (§3.2) cleanly separate them:

F1: unstable optimization paths. V1 still shows a high-return ridge in the ideal quadrant, but V2 misses it. For example, an initialization far from the origin makes the policy nearly deterministic, and the local gradient moves it farther into the flat, incorrect region instead of toward the ideal ridge (Fig. 4a). A large learning rate can instead overshoot the ridge and stop on a lower-return plateau; Fig. 3 shows the resulting QoE loss. In both cases, the target remains available without changing the reward or traces.

F2: distorted return landscapes. Changing the reward or traces changes V1. With $\lambda = 1.3$, the reward underweights stalls and moves the high-return region toward aggressive policies (Fig. 4b). Restricting training to 3.0–4.0 Mbps traces instead favors conservative policies because the high bitrate is rarely useful. The optimizer follows its objective in both cases. Here, "failure" concerns recovering adaptive behavior for the original 3.0–6.0 Mbps operating range: the altered objective no longer favors that target. A conservative policy can be appropriate if the restricted range is the intended

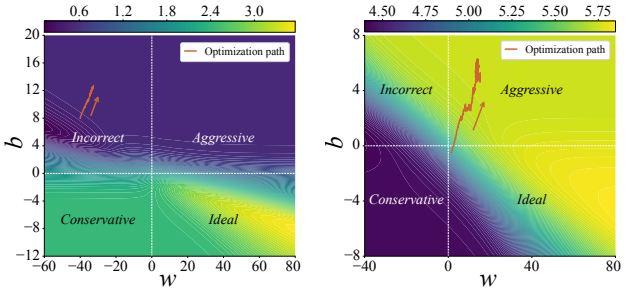


Figure 4: Controlled ABR exposes two failure modes. **(a)** F1: initialization at $(-40, 8)$ sends optimization away from the ideal ridge. **(b)** F2: $\lambda = 1.3$ shifts the high-return region toward aggressive policies.

task. Restoring the original diagnostic target requires revisiting the reward or training distribution, rather than only changing the optimizer.

5 Beyond the Controlled Setting

The controlled ABR setting introduced earlier uses several simplifications to make the training process most amenable to visualization and diagnosis. We now relax them along two axes and ask whether the litmus test and its diagnostic views remain useful: restoring Pensieve's full six-bitrate action space (§5.1), and applying the methodology to a different task, congestion control (§5.2).

5.1 ABR with Six Bitrates

We restore Pensieve's six-rate ladder, which spans 0.3–4.3 Mbps [16]. The policy assigns each bitrate a linear score, giving 12 parameters (w_i, b_i). Fixed-bandwidth traces extend to 0.5–6.0 Mbps so that the full ladder is usable (the lowest bitrate is 0.3 Mbps). The 12-D landscape cannot be exactly enumerated. We approximate V1 with 2-D projections and random-direction samples [2], and inspect V3 through action logits. These views expose the displayed failure, but they cannot rule out structure outside the sampled directions.

Success persists. We retain the controlled training configuration and add entropy regularization for early exploration in the larger action space [20]. The learned policies remain buffer-monotone. Because we cannot compute the optimal 12-parameter policy, we compare against BBA and BBA*. BBA uses Pensieve's default reservoir and cushion; BBA* generalizes the threshold oracle from §4 by tuning both parameters to maximize average QoE over the evaluation traces. In Fig. 5, RL Base uses the base configuration, whereas RL Tuned selects training parameters while holding the reward, traces, and policy class fixed. RL Base outperforms BBA, and RL Tuned approaches BBA*, across repeated runs and multiple training configurations.

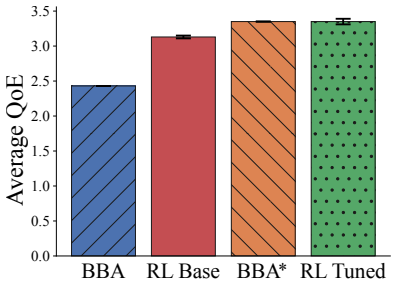
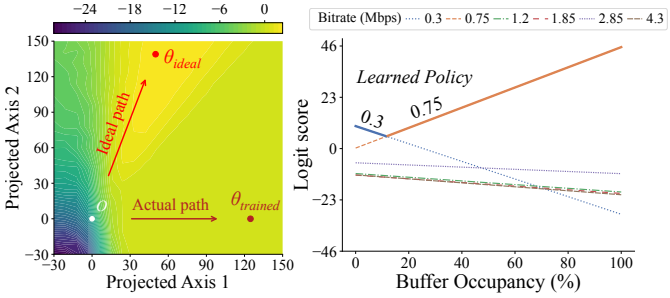


Figure 5: The base learned policy outperforms BBA; the tuned learned policy approaches BBA*.



(a) (b)

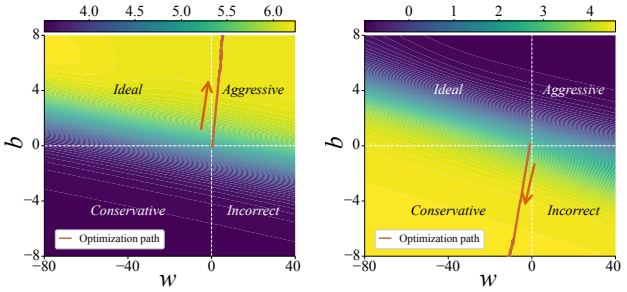
Figure 6: ABR six-bitrate failure mode. **(a)** On the plane with ideal and trained policies, the actual path does not reach the highest-return region. **(b)** The learned policy favors low-bitrate actions.

Failure persists. Without entropy regularization, training converges to a conservative policy dominated by the lowest bitrates (Fig. 6b). Fig. 6a projects the return landscape onto the plane spanned by that policy and a BBA-like ideal policy. The actual path diverges from the ideal direction and reaches a different high-return region. The projected training path misses the BBA-like target and converges to a conservative policy. These views expose a rediscovery failure in the larger action space.

5.2 Congestion Control

We next test whether the same diagnostic signatures appear in RL congestion control, which has a different state, action, and reward. The goal is to test whether the methodology transfers, not to design a new congestion-control algorithm.

Analyzable CC environment. We instantiate Case 2 using a core feedback principle of delay-based CC, exemplified by Vegas [4] and Copa [3]: reduce sending aggressiveness as queuing delay rises. Our training setting uses constant-bandwidth traces (1–4 Mbps in 0.5 Mbps steps), a fixed $\mathrm{RTT}_{\min} = 30$ ms across traces, and 8 monitor intervals per episode. We make the compact RL policy take the preceding interval's RTT in seconds as its state, $s_t = \mathrm{RTT}_{t-1}$, and output a binary multiplicative rate action $a_t \in \{0.9, 1.1\}$. We use the two-parameter logistic rule $\pi_\theta(1.1 \mid s_t) = \sigma(ws_t + b)$. Since $\mathrm{RTT}_{\min}$ is fixed, changes in RTT reflect changes in



(a) (b)

Figure 7: F2 recurs in congestion control. **(a)** A low delay penalty moves the ridge into the aggressive quadrant. **(b)** A narrow trace set favors a conservative policy.

queueing delay. The policy's rate changes fill or drain the queue, keeping RTT informative even on constant-bandwidth traces. This gives a 2-D parameter space (w, b) where $J(\theta)$ is cheap to evaluate. We use an Aurora-inspired reward [14], $r = T/C - \alpha\ell - \beta d$, where T is throughput, C capacity, $\ell \in [0, 1]$ loss fraction, and $d = \mathrm{RTT}/\mathrm{RTT}_{\min} - 1 \in [0, 3]$ normalized queueing delay; defaults are $(\alpha, \beta) = (0.2, 0.6)$. We use delay-based heuristics because their back-off rule fits an interval-based RL formulation. Event-driven TCP does not share that formulation, and we further discuss how to expand our method under such mismatches in §6.

Correctness property (V3). A correct policy must not send faster as the RTT rises: for $s_1 < s_2$, $\arg\max_a \pi_\theta(a \mid s_1) \geq \arg\max_a \pi_\theta(a \mid s_2)$. This reverses the ABR monotonicity check because high RTT is undesirable. The Ideal quadrant is therefore $w < 0, b > 0$; the Incorrect quadrant is $w > 0, b < 0$, where the policy sends faster as the queue grows.

The same signatures recur. Under the default configuration, V1 places the ridge in the Ideal quadrant and V2 reaches it. F1 recurs when starting in the Incorrect quadrant traps training. F2 recurs in two ways: reducing the delay penalty to $\beta = 0.03$ moves the ridge into the Aggressive quadrant (Fig. 7a), while training only on 1–2 Mbps bottlenecks moves training into a conservative, low-throughput region (Fig. 7b). As in §4.2, we interpret this shift relative to the original operating range. These match the ABR signatures, showing the diagnostic views' value in a second controlled setting.

6 Discussion and Future Work

How we simplify CC heuristics for rediscovery. Our CC experiment (§5.2) targets a core behavior of delay-based control: sending should become less aggressive as queuing delay rises. We choose this behavior because it admits a compact representation using one delay signal, making the training process easy to inspect.

Our setup differs from the original Vegas controller, which makes congestion-control decisions once per RTT based on

ACK feedback [4], choosing to increase, decrease, or maintain its congestion window during congestion avoidance. In contrast, our policy adjusts the sending rate at each monitor interval using only increase and decrease actions. This simplification allows us to represent the policy with just two logistic parameters and visualize its entire return landscape. Thus, our experiment serves as a Case 2 test of the selected feedback behavior rather than an attempt to reproduce the full Vegas controller.

Adding back the third unchanged-rate action would still allow all three diagnostic views. If the policy then has more than two parameters, V1 and V2 could use projections, as in §5.1. Future work could compare the RL policy with Vegas under the same network conditions, varying the observations, window versus rate control, and decision timing one at a time. This would help distinguish training problems from differences in the control formulation.

Adapting the litmus test to other control policies. Heuristic rediscovery does not require a single input, two actions, or a monotonic response. We use linear logit scores based on one signal to keep our examples easy to inspect. Other heuristics may use observations, actions, or decision timing that differ from the RL formulation. Adapting the test therefore requires domain expertise to choose a compatible target and a suitable compact policy class, which may use different input features. When this involves simplifying a heuristic, we must check that the simplified target preserves the selected behavior and remains useful in the tested setting.

Several well-known policies in networked systems offer concrete starting points. One example is join-the-shortest-queue (JSQ) load balancing, which selects a server with the fewest unfinished requests at each request arrival [10]. An RL policy could score all servers from their queue lengths using a shared linear function. A negative slope reproduces JSQ's ranking while keeping the parameter count independent of the number of servers. Kubernetes HPA can also act on one user-specified metric, such as CPU utilization [28]. Its basic rule multiplies the current replica count by the ratio of the measured value to its target and rounds up to an integer. A compact RL policy could represent this proportional rule. Similarly, least-recently-used (LRU) cache eviction removes the cached item that has gone longest without access [18]. An RL policy could observe the time since each item's last access and choose an item whenever eviction is needed. A shared linear scoring function that assigns higher scores to longer elapsed times can express the LRU choice. These heuristics offer candidate Case 1 targets that can be represented by compact policies with only a few parameters.

Choosing a compact representation becomes harder when the controller depends on history or internal state. For Case 2, one possible approach is to identify a component whose core behavior is governed by feedback control over a single input variable. For example, in TCP CUBIC, a possible target is window reduction in response to packet loss [11]. This would isolate its loss response without reproducing its time-dependent window growth. For BBR, we could target its decision to continue draining until the amount of data in flight falls to the estimated bandwidth-delay product (BDP) [5]. Within the draining phase, a compact policy could use the ratio of data in flight to estimated BDP as its input. These examples suggest possible directions for constructing compact rediscovery targets. Developing litmus tests for these heuristics and evaluating whether the compact targets preserve the selected behavior are left to future work.

Applying the litmus test in practical training. Once we have a compatible target, we must determine what rediscovery can tell us about the training pipeline. The training environment affects this interpretation. The policy needs to encounter states where the target behavior is useful. In both of our case studies, trace bandwidth, initialization, and episode length all affect which states it visits. We currently choose these training environments manually; automatically choosing or rebalancing training environments is a separate research problem [24, 29].

We must also account for what the diagnostic views can miss. In our six-bitrate example (§5.1), a projection reveals a failure but can hide behavior outside the plotted plane. Larger policy classes may require similar approximations for V1 and V2 [2]. Stochastic traces and longer episodes may also require estimating returns from sampled trajectories. V3 needs a behavioral check matched to the chosen target.

Future work could apply the test to an existing RL pipeline. We would replace its policy with a compatible compact class while keeping the simulator, reward, traces, episode length, and trainer unchanged. This would test whether reward-only training can rediscover the heuristic's behavior under practical training conditions. We would then introduce the same types of changes that produced the failure types F1 and F2 in our experiments. Recurrence of these patterns would provide evidence that the diagnostic signatures extend beyond our controlled setting.

Conclusion. Heuristic rediscovery helps us inspect RL training in networked systems: what it learns and why it fails. Our case studies in ABR and CC show how the return landscape, training path, and learned behavior can reveal problems in RL rewards and optimization settings.

Acknowledgements

We thank our shepherd, Vyas Sekar, and the ACM HotNets reviewers for their helpful feedback. This work was supported by NSF CAREER Award No. 2237474.

References

[1] Neil Agarwal, Rui Pan, Francis Y. Yan, and Ravi Netravali. 2025. Mowgli: Passively Learned Rate Control for Real-Time Video. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*.

[2] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. 2019. Understanding the Impact of Entropy on Policy Optimization. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR.

[3] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[4] L. S. Brakmo and L. L. Peterson. 1995. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications* (1995), 1465–1480.

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *Commun. ACM* 60, 2 (Jan. 2017), 58–66.

[6] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. 2018. Cellular Network Traffic Scheduling with Deep Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '18)*.

[7] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. 2021. Verifying Learning-Augmented Systems. In *Proceedings of the ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. ACM, New York, NY, USA.

[8] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. 2019. Robustifying Network Protocols with Adversarial Examples. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*.

[9] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. 2014. Qualitatively Characterizing Neural Network Optimization Problems. *arXiv preprint arXiv:1412.6544* (2014).

[10] Varun Gupta, Mor Harchol-Balter, Karl Sigman, and Ward Whitt. 2007. Analysis of Join-the-Shortest-Queue Routing for Web Server Farms. *Perform. Eval.* 64, 9–12 (Oct. 2007), 1062–1081.

[11] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.

[12] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2014. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*. ACM, New York, NY, USA.

[13] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. 2022. AI/ML for Network Security: The Emperor Has No Clothes. In *Proceedings of the 2022 ACM Conference on Computer and Communications Security (CCS '22)*.

[14] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 3050–3059.

[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[16] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA.

[17] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA.

[18] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*. USENIX Association.

[19] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. 2020. Interpreting Deep Learning-Based Networking Systems. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*. ACM, New York, NY, USA.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*. PMLR.

[21] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. 2000. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Transactions on Networking* 8, 2 (2000).

[22] Lorenzo Pappone, Alessio Sacco, and Flavio Esposito. 2025. Mutant: Learning Congestion Control from Existing Protocols via Online Reinforcement Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*.

[23] Sagar Patel, Dongsu Han, Nina Narodytska, and Sangeetha Abdu Jyothi. 2025. Agua: A Concept-Based Explainer for Learning-Enabled Systems. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '25)*. ACM, New York, NY, USA.

[24] Sagar Patel, Junyang Zhang, Nina Narodytska, and Sangeetha Abdu Jyothi. 2024. Practically High Performant Neural Adaptive Video Streaming. *Proc. ACM Netw.* 2, CoNEXT4, Article 30 (Nov. 2024).

[25] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2016. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *International Conference on Learning Representations (ICLR '16)*.

[26] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. 2020. BOLA: Near-Optimal Bitrate Adaptation for Online Videos. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1698–1711.

[27] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems* 12 (1999).

[28] The Kubernetes Authors. [n. d.]. Horizontal Pod Autoscaling. https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/ Accessed: October 19, 2026.

[29] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. 2022. Genet: Automatic Curriculum Generation for Learning Adaptation in Networking. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '22)*. ACM, New York, NY, USA.

[30] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. 2023. Teal: Learning-Accelerated Optimization of WAN Traffic Engineering. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '23)*. ACM, New York, NY, USA.

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

[32] Mohammad Zangooei, Jannis Weil, Amr Rizk, Mina Tahmasbi Arashloo, and Raouf Boutaba. 2026. Analyzing Symbolic Properties for DRL Agents in Systems and Networking. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 2, Article 29 (2026).

[33] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. 2020. OnRL: Improving Mobile Video Telephony via Online Reinforcement Learning. In *ACM MobiCom '20*.