GENET: Automatic Curriculum Generation for Learning Adaptation in Networking

Zhengxu Xia[1*], Yajie Zhou[2*], Francis Y. Yan[3], Junchen Jiang[1]
[1]University of Chicago, [2]Boston University, [3]Microsoft Research

ABSTRACT

As deep reinforcement learning (RL) showcases its strengths in networking, its pitfalls are also coming to the public's attention. Training on a wide range of network environments leads to suboptimal performance, whereas training on a narrow distribution of environments results in poor generalization.

This work presents GENET, a new training framework for learning better RL-based network adaptation algorithms. GENET is built on curriculum learning, which has proved effective against similar issues in other RL applications. At a high level, curriculum learning gradually feeds more "difficult" environments to the training rather than choosing them uniformly at random. However, applying curriculum learning in networking is nontrivial since the "difficulty" of a network environment is unknown. Our insight is to leverage traditional rule-based (non-RL) baselines: If the current RL model performs significantly worse in a network environment than the rule-based baselines, then further training it in this environment tends to bring substantial improvement. GENET automatically searches for such environments and iteratively promotes them to training. Three case studies—adaptive video streaming, congestion control, and load balancing—demonstrate that GENET produces RL policies that outperform both regularly trained RL policies and traditional baselines.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Transport protocols**; • **Computing methodologies** → **Reinforcement learning**;

KEYWORDS

Reinforcement Learning, Curriculum Learning, Network Adaptation, Congestion Control, Adaptive Bitrate Video

ACM Reference Format:

Zhengxu Xia, Yajie Zhou, Francis Y. Yan, Junchen Jiang. 2022. GENET: Automatic Curriculum Generation for Learning Adaptation in Networking. In *SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3544216.3544243

1 INTRODUCTION

Many recent techniques based on deep reinforcement learning (RL) are now among the state of the arts for various networking and systems adaptation problems, including congestion control (CC) [24], adaptive bitrate streaming (ABR) [32], load balancing (LB) [31], wireless resource scheduling [10], and cloud scheduling [34]. For a given distribution of training network environments (e.g., network connections with certain bandwidth patterns, delay, and queue length), RL trains a policy to optimize performance over these environments.

However, these RL-based techniques face two challenges that can ultimately impede their wide use in practice:

- **Training in a wide range of environments:** When the training distribution spans a wide variety of network environments (e.g., a large range of possible bandwidth), an RL policy may perform poorly even if tested in the environments drawn from the same distribution as training.
- **Generalization:** RL policies trained on one distribution of synthetic or trace-driven environments may have poor performance and even erroneous behavior when tested in a new distribution of environments.

Our analysis in §2 will reveal that, across three RL use cases in networking, these challenges can cause well-trained RL policies to perform much worse than traditional rule-based schemes in a range of settings.

These problems are not unique to networking. In other domains (e.g., robotics, gaming) where RL is widely used, it is also known that RL models have performance issues in both new environments drawn from the training distribution and new environments drawn from an unseen distribution [27, 36, 37, 52, 59]. There have been many efforts to address these issues by enhancing offline RL training or retraining a deployed RL policy online. Since updating a deployed model is not always possible or easy (e.g., loading a new kernel module for congestion control or integrating an ABR logic into a video player), we focus on improving RL training offline.

A well-studied paradigm that underpins many recent techniques to improve RL training is *curriculum learning* [36]. Unlike traditional RL training that samples training environments in a random order, curriculum learning generates a training curriculum that gradually increases the difficulty level of training environments, resembling how humans are guided to comprehend more complex concepts. Curriculum learning has been shown to improve generalization [6, 12, 35] as well as *asymptotic* performance [25, 53], namely the final performance of a model after training runs to convergence. Following an easy-to-difficult routine allows the RL model to make steady progress and reach good performance.

In this work, we present *GENET, the first training framework that systematically introduces curriculum learning to RL-based networking algorithms.* GENET automatically generates training curricula for network adaptation policies. The challenge of curriculum learning in networking is how to sequence network environments in an order that prioritizes highly *rewarding* environments where the current RL policy's reward can be considerably improved. Unfortunately,

*Both authors contributed equally to this research.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544243

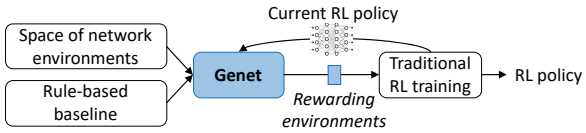


Figure 1: *Genet creates training curricula by iteratively finding rewarding environments where the current RL policy has a large gap-to-baseline.*

as we show in §3, several seemingly natural heuristics to identify rewarding environments suffer from limitations.

- First, they use *intrinsic properties* of each environment (e.g., shorter network or workload traces [34] and smoother network conditions [19] are supposedly easier), but these intrinsic properties fail to indicate whether the current RL model can be improved in an environment.
- Second, they use *handcrafted heuristics* which may not capture all aspects of an environment that affect RL training (e.g., bandwidth smoothness does not capture the impact of router queue length on congestion control, or buffer length on adaptive video streaming). Each new application (e.g., load balancing) also requires a new heuristic.

The idea behind Genet is simple: An environment is considered *rewarding* if the current RL model has a large *gap-to-baseline*, i.e., how much the RL policy's performance falls behind a traditional *rule-based* baseline (e.g., Cubic or BBR for congestion control, MPC or BBA for adaptive bitrate streaming) in the environment. We show in §4.1 that the gap-to-baseline of an environment is highly indicative of an RL model's potential improvement in the environment. Intuitively, since the baseline already shows how to perform better in the environment, the RL model may learn to "imitate" the baseline's known rules while training in the same environment, bringing it on par with—if not better than—the baseline. On the flip side, if an environment has a small or even negative gap-to-baseline, chances are that the environment is intrinsically hard (a possible reason why the rule-based baseline performs badly), or the current RL policy already performs well and thus training on it is unlikely to improve performance by a large margin. A small gap-to-baseline might also arise when the rule-based baseline has poor performance yet the RL model still has a large room for improvement. Genet ignores this case, but we will discuss it in §7.

Inspired by the insight, Genet generates RL training curricula by iteratively identifying rewarding environments where the current RL model has a large gap-to-baseline and then adding them to RL training (Figure 1). For each RL use case, Genet parameterizes the network environment space, allowing us to search for rewarding environments in both synthetically instantiated environments and trace-driven environments. Genet also uses Bayesian Optimization (BO) to facilitate the search in a large space. In particular, we cast the search for environments with a large gap-to-baseline as a maximum-search problem of a blackbox function in a high-dimensional space where each point represents a set of environment configurations and the function value is the gap-to-baseline. BO is then used to find a set of training environments with large gap-to-baselines. Genet is generic, since it does not use handcrafted heuristics to measure the difficulty of a network environment; instead, it uses rule-based algorithms, which are abundant in the literature of many networking and system problems, to generate training curricula. Moreover, by focusing training on places where RL falls behind rule-based baselines, Genet directly minimizes the chance of performance regressions relative to the baselines. This is important, because system operators are more willing to deploy an RL policy if it outperforms the incumbent rule-based algorithm in production without noticeable performance regressions.[2]

We have implemented Genet as a separate module with a unifying abstraction that interacts with the existing codebases of RL training to iteratively select rewarding environments and promote them in the course of training. We have integrated Genet with three existing deep RL codebases in the networking area—adaptive video streaming (ABR) [4], congestion control (CC) [1], and load balancing (LB) [3].

It stands to reason that Genet is not without limitations. For instance, Genet-trained RL policies might not outperform all rule-based baselines (§5.5 shows that when using a naive baseline to guide Genet, the resulting RL policy could still be inferior to stronger baselines). Genet-trained RL policies may also achieve undesirable performance in environments beyond the training ranges (e.g., if we train a congestion-control algorithm on links with bandwidth between 0 and 100 Mbps, Genet will not optimize for the bandwidth of 1 Gbps). Moreover, Genet does not guarantee adversarial robustness which sometimes conflicts with the goal of generalization [41].

Using a combination of trace-driven simulation and real-world tests across three use cases (ABR, CC, LB), we show that Genet improves asymptotic performance by 8–25% for ABR, 14–24% for CC, 15% for LB, compared with traditional RL training methods. Genet aims to optimize an RL model's asymptotic performance (i.e., in-distribution generalizability), and it does not explicitly optimize the generalization in arbitrary test environments (i.e., out-of-distribution generalizability). That said, our empirical test results show that Genet-trained models improve not only asymptotic performance, but also the performance in unseen network environments.

The traces and scripts used in Genet are released at https://github.com/GenetProject/Genet.

2 MOTIVATION

Deep reinforcement learning (RL) trains a deep neural net (DNN) as the decision-making logic (policy) and is well-suited to many sequential decision-making problems in networking [22, 31].[3] We use three use cases (summarized in Table 1) to make our discussion concrete:

- An **adaptive bitrate (ABR)** algorithm adapts the chunk-level video bitrate to the dynamics of throughput and playback buffer (input state) over the course of a video session. ABR policies, including RL-based ones (Pensieve [32]), choose the next chunk's bitrate (output decision) at the chunk boundary to maximize

[2] An example of this mindset is that a new algorithm must compete with the incumbent algorithm in A/B testing before being rolled out to production.

[3] There are rule-based alternatives to DNN-based policies, but they are not as expressive and flexible as DNNs, which limits their performance. Oboe [5], for instance, sets optimal hyperparameters for RobustMPC based on the mean and variance of network bandwidth and as shown in §5.4, is a very competitive baseline, but it performs worse than the best RL strategy.

Use case	**Observed state** (policy input)	**Action** (policy output)	**Reward** (performance)
Adaptive Bitrate (ABR) Streaming	future chunk size, history throughput, current buffer length	bitrate selected for the next video chunk	$\sum_i(\alpha \cdot \text{Rebuf}_i + \beta \cdot \text{Bitrate}_i + \gamma \cdot \text{BitrateChange}_i)/n$
Congestion Control (CC)	RTT inflation, sending/receiving rate, avg RTT in a time window, min RTT	change of sending rate in the next time window	$\sum_i(a \cdot \text{Throughput}_i + b \cdot \text{Latency}_i + c \cdot \text{LossRate}_i)/n$
Load Balancing (LB)	past throughput, current request size, number of queued requests per server	server selection for the current request	$-\sum_i \text{Delay}_i/n$

Table 1: *RL use cases in networked systems. Default reward parameters: $\alpha = -10$ (rebuffering in seconds), $\beta = 1$ (bitrate in Mbps), $\gamma = -1$ (bitrate change in Mbps), $a = 120$ (throughput in kbps), $b = -1000$ (latency in seconds), $c = -2000$. Details in A.5.*

session-wide average bitrate, while minimizing rebuffering and bitrate fluctuation.

- A **congestion control (CC)** algorithm at the transport layer adapts the sending rate based on the sender's observations of the network conditions on a path (input state). An example of RL-based CC policy (Aurora [24]) makes sending rate decisions at the beginning of each interval (of length proportional to RTT), to maximize the reward (a combination of throughput, latency, and packet loss rate).
- A **load balancing (LB)** algorithm in a key-replicated distributed database reroutes each request to one of the servers (whose real-time resource utilization is unknown), based on the request arrival intervals, resource demand of past requests, and the number of outstanding requests currently assigned to each server.

We choose these use cases because they have open-source implementations (Pensieve [4] for ABR, Aurora [1] for CC, and Park [3] for LB). Our goal is to improve existing RL training in networking. Revising the RL algorithm *per se* (input, output, or DNN model) is beyond our scope.

Network environments: We generate simulated training environments with a range of parameters, following prior work [24, 31, 32]. An environment can be synthetically generated using a list of parameters as *configuration*, e.g., in the context of ABR, a configuration encompasses bandwidth range, frequency of bandwidth change, chunk length, etc. Meanwhile, when recorded bandwidth traces are available (for CC and ABR experiments), we can also create *trace-driven* environments where the recorded bandwidth is replayed. Note that bandwidth is only one dimension of an environment and must be complemented with other synthetic parameters in order to create a simulated environment. (Our environment generator and a full list of parameters are documented in §A.2.) In recent papers, both trace-driven (e.g., [19, 32]) and synthetic environments (e.g., [24, 31]) are used to train RL-based network algorithms. We will explain in §4.2 how our technique applies to both types of environments.

Traditional RL training: Given a user-specified distribution of (trace-driven or synthetic) training environments, the traditional RL training method works in iterations. Each iteration randomly samples a subset of environments from the provided distribution and then updates the DNN-based RL policy (via forward and backward passes). For instance, Aurora [24] uses an iteration of 7200 steps (i.e., 30–50 30-second network environments) and applies the PPO algorithm to update the policy network by simulating the network environments in each batch.

Several previous efforts have demonstrated the promise of the traditional RL training—given the distribution of target environments, an RL policy can be trained to perform well in these environments (e.g., [24, 32]). Unfortunately, this approach falls short on two fronts.

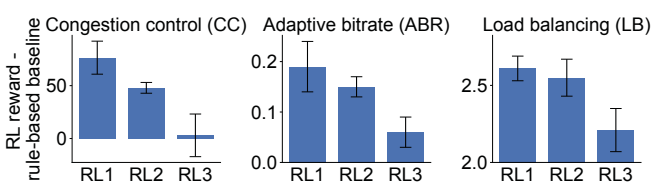
Challenge 1: Training over wide environment distributions. When the training distribution of network environments has a wide-spread (e.g., a large range of possible bandwidth values), RL training tends to result in poor *asymptotic* performance (model performance after reaching convergence) even when the test environments are drawn from the *same* distribution as training.

In Figure 2, for each use case, we choose three target distributions (with increasing parameter ranges), labeled RL1/RL2/RL3 ranges of synthetic environment parameters in Table 3, 4, and 5. Figure 2(a) compares the asymptotic performance of three RL policies (with different random seeds) with rule-based baselines, MPC [57] for ABR, BBR [8] for CC, and least-load-first (LLF) policy for LB, in test environments randomly sampled from the *same* ranges. It shows that RL's performance advantage over the baselines diminishes rapidly when the range of target environments expands. Even though RL-based policies still outperform the baselines on average, Figure 2(b) reveals a more striking reality—their performance falls behind the baselines in a substantial fraction of test environments.

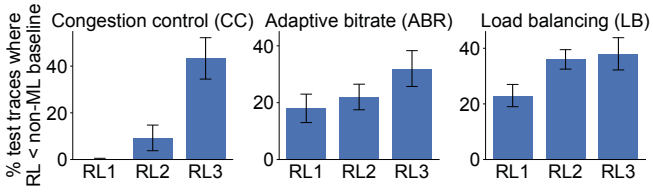
An intuitive explanation is that in each RL training iteration, only a batch of randomly sampled environments (typically 20–50) is used to update the model, and when the entire training set spans a wide range of environments, the batches between two iterations may have dramatically different distributions which potentially push the RL model to different directions. This causes the training to converge slowly and makes it difficult to obtain a good policy [36]. Although this problem is not completely avoided in our solution, it is mitigated by curriculum learning which draws the environments of a batch from a "narrower" training environment distribution, thus reducing the discrepancies between batches.

Challenge 2: Low generalizability. Another practical challenge arises when the training process does not have access to the target environment distribution. This calls for models with good generalization, i.e., the RL policies trained on one distribution also perform well on a different environment distribution during testing. Unfortunately, existing RL training methods often fall short of this ideal. Figure 3 evaluates the generalizability of RL-based CC schemes in two ways.

- First, we train an RL-based CC algorithm on the same range of synthetic environments as specified in its original paper [24]. We

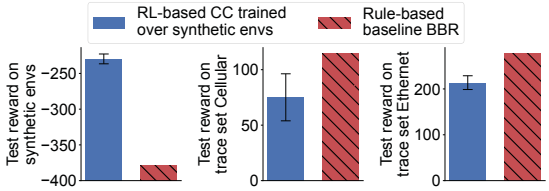


(a) Performance gains of RL schemes over the baselines diminish as the target distribution spans a wide range of environments.

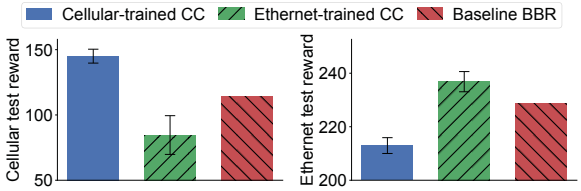


(b) Even if RL schemes perform better on average, they are worse than the baselines on a substantial fraction of test environments.

Figure 2: *Challenges of RL training over a wider range of environments from small (RL1), medium (RL2), to large (RL3).*



(a) RL-based CC trained in synthetic network environments performs worse on real network traces than the rule-based baseline.



(b) RL-based CC trained over one real trace set performs worse on another real trace set than the rule-based baseline.

Figure 3: *Generalization issues of RL-based schemes using CC as an example.*

first validate the model by confirming its performance against a rule-based baseline BBR, in environments that are independently generated from the same range as training (Figure 3(a); left). Nevertheless, when tested on real-world recorded network traces under the category of "Cellular" and "Ethernet" from Pantheon [56] (Table 2), the RL-based policy yields much worse performance than the rule-based baseline.

- Second, we train the RL-based CC algorithm on the "Cellular" trace set and test it on the "Ethernet" trace set (Figure 3(b); left), or vice versa (Figure 3(b); right). Similarly, its performance degrades significantly when tested on a different trace set.

The observations in Figure 3 are not unique to CC. Prior work [19] also shows a lack of generalization of RL-based ABR algorithms.

Summary: In short, we observe two challenges faced by the traditional RL training mechanism:

- The asymptotic performance of the learned policies can be sub-optimal, especially when they are trained over a wide range of environments.
- The trained RL policies may generalize poorly to unseen network environments.

3 CURRICULUM LEARNING FOR NETWORKING

Given these observations regarding the limitations of RL training in networking, a natural question to ask is *how to improve RL training such that the learned adaptation policies achieve good asymptotic performance across a broad range of target network environments.*[4]

Curriculum learning: We cast the training of RL-based network adaptation to the well-studied framework of curriculum learning. Unlike the traditional RL training that samples training environments from a *fixed* distribution in each iteration, curriculum learning *varies* the training environment distribution to gradually increase the difficulty of training environments, so that training will see more environments *that are more likely to improve*, which we refer to as *rewarding* environments. In many RL applications, prior work has shown the promise of curriculum learning, including faster convergence, higher asymptotic performance, and better generalization (§6).

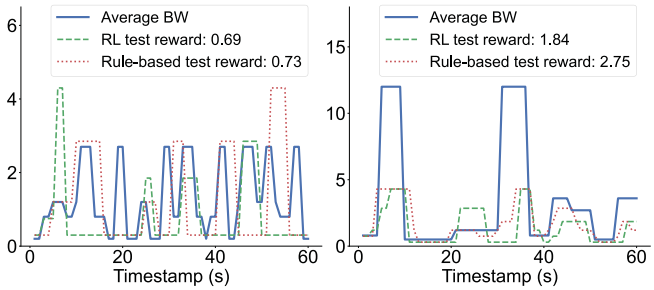
The theoretical intuition behind curriculum learning is that a curriculum allows the model to optimize a family of gradually less smooth loss functions and prevents it from being trapped in local minima [7]. In the early stage of the curriculum, easier training samples are selected to comprise a smoothed loss function that reveals the big picture and is easier to optimize. The resulting model serves as a good starting point when more difficult samples are introduced to the training, reducing the smoothness of the loss function and making it harder to optimize. By optimizing the model on a sequence of loss functions with decreasing smoothness, the curriculum is able to gradually bring the model parameters close to the global optimum.

However, the challenge of employing curriculum learning lies in determining which environments are rewarding. Apparently, the answer to this question varies with applications, but three general approaches exist: (1) training the current model on a set of environments individually to determine in which environment the training progresses faster; (2) using heuristics to quantify the easiness of achieving model improvement an environment; and (3) jointly training another model (typically DNN) to select rewarding environments. Among them, the first option is prohibitively expensive and thus not widely used, whereas the third introduces the extra complexity of training a second DNN. Therefore, we take a pragmatic stance and explore the second approach, while leaving the other two for future work.

[4] An alternative is to retrain the deployed RL policy whenever it meets a new domain (e.g., a new network connection with unseen characteristics), but this does not apply when the RL policy cannot be updated frequently. Besides, it is also challenging to precisely detect model drift in the network conditions that necessitate retraining the RL policy.

(a) Add X to training (b) Add Y to training

Figure 4: *A simple example where adding trace set X to training has a different effect than adding Y. Adding X to training improves performance on X only marginally but hurts Y, whereas adding Y improves the performance on both X and Y.*



(a) Trace in X (hard) (b) Trace in Y (improvable)

Figure 5: *Contrasting (a) an inherently hard (possibly unsolvable) environment with (b) an improvable environment. The difference is that the rule-based policy's reward is higher than the RL policy in (b), whereas their rewards are similar in (a).*

Why sequencing training environments is difficult: A common strategy in curriculum learning for RL is to measure environment difficulty and gradually introduce more difficult environments to training. To motivate our design choices, we first introduce three strawman approaches, with different strengths and weaknesses. They are used to determine how rewarding an environment is. A good approach should always select network environments in which the RL model has a large improvement in reward when trained in them.

Strawman 1: inherent properties. The first idea is to quantify the difficulty level of an environment using some of its inherent properties. In congestion control, for instance, network traces with higher bandwidth variance are intuitively more difficult. This approach, however, only distinguishes environments that differ in the hand-picked properties and may not suffice under complex environments (e.g., adding bandwidth traces with similar variance to training can have different effects).

Strawman 2: performance of rule-based baselines. Alternatively, one can use the test performance of a traditional algorithm to indicate the difficulty of an environment. Lower performance may suggest a more difficult environment [53]. While this method can distinguish any two environments, it does not hint at how to improve the *current* RL model during training.

Strawman 3: performance gap to the optimum. To fix the problem of Strawman 2, one can use the performance gap between the current RL policy and the optimum instead [19]. If the current model performs much worse than the optimum in an environment (e.g., obtained by using ground-truth bandwidth as the bandwidth prediction), its performance might improve when trained in this environment. A caveat of this approach is that the computation of the optimal performance could be prohibitively expensive or even infeasible. This approach may also fail to improve RL's performance in environments that are inherently hard (e.g., highly fluctuating bandwidth in ABR and CC).

Example: Figure 4 shows a concrete example in ABR, where "Strawman 3" leads to a suboptimal outcome. (§5.5 will empirically test these three strawman approaches.) We first pretrain an RL-based ABR policy which performs poorly on X and Y (two sets of bandwidth traces from two different environment configurations, details in §A.3). Since the performance gap between the current RL model and the optimum is larger on X than on Y, Strawman 3 opts for adding X to the training in the next step. However, Figure 4 shows that training further on X yields only a marginal reward improvement on X (and also hurts the performance on Y).

Instead, adding Y to training is a better choice at this point—the performance on Y is significantly improved (and it also benefits the performance on X though to a less extent).

To take a closer look, we plot two example traces from X and Y in Figure 5: The trace from X fluctuates with a smaller magnitude but more frequently, whereas the trace from Y fluctuates with a greater magnitude but much less frequently. However, such observations cannot generalize to an arbitrary pair of environments or a different application.

4 DESIGN AND IMPLEMENTATION OF GENET

4.1 Curriculum generation

To identify rewarding environments, the idea of GENET is to find environments with a large *gap-to-baseline*, i.e., the RL policy is worse than a given rule-based baseline by a large margin. At a high level, adding such environments to training has three practical benefits.

First, when a rule-based baseline performs much better than the RL policy in an environment, it means that the RL model may learn to "imitate" the baseline's known rules while training in the environment, bringing it on par with—if not better than—the baseline.[5] Therefore, a large gap-to-baseline indicates plausible room for the current RL model to improve. Figure 6 empirically confirms this with one example ABR policy and CC policy (both are intermediate models during GENET-based training). For example, among 73 randomly chosen synthetic environment configurations in CC, a configuration with a larger gap-to-baseline is likely to yield more improvement when adding its environments to the RL training. Moreover, this correlation is stronger than using the performance gap between the current model and the optimum ("Strawman 3" in §3) to decide which environments are rewarding. Nonetheless, the model's training improvement does not only depend on the gap-to-baseline. Other factors such as training hyperparameters can affect the reward improvement of an RL model. For example, too large a learning rate causes the RL model to jump over the

[5]This may not be true when the behavior of the rule-based algorithm cannot be approximated by RL's policy DNN, and we will discuss this issue in §7.

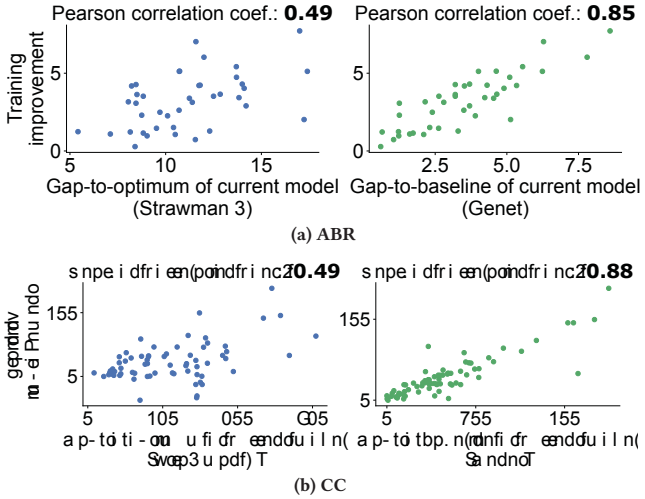


Figure 6: *Compared with the gap-to-optimum (left), the current model's gap-to-baseline (right) in an environment is more indicative of its potential training improvement in the environment.*

optima while too small a learning rate slows down the convergence. In this work, we only focus on the gap-to-baseline and keep the training hyperparameters (e.g., learning rate, batch size of each iteration) unchanged in all the experiments.

Second, although not all rule-based algorithms are easily interpretable or completely fail-proof, many of them have traditionally been used in networked systems long before the RL-based approaches and are considered more trustworthy than black-box RL algorithms. Therefore, operators tend to scrutinize any performance disadvantages of the RL policy compared with the rule-based baselines currently deployed in the system. By promoting environments with large gap-to-baselines, Genet directly reduces the possibility that the RL policy causes performance regressions.

In short, the gap-to-baseline builds on the insight that rule-based baselines are *complementary* to RL policies—they are less susceptible to any discrepancies between training and test environments, whereas the performance of an RL policy is potentially sensitive to the environments seen during training. In §5.5, we will discuss the impact of different choices of rule-based baselines and why gap-to-baseline is a better way of using the rule-based baseline than alternatives. It is worth noting that the rewarding environments (those with large gap-to-baselines) do *not* have particular meanings outside the context of a given pair of RL model and baseline. For instance, when an RL-based CC model has a greater gap-to-baseline in some network environments, it only means that it is easier to improve the RL model by training it in these environments; it does not indicate if these environments are easy or challenging to any traditional CC algorithm.

4.2 Training framework

Figure 7 depicts Genet's high-level iterative workflow to realize curriculum learning. Each iteration consists of three steps (which will be detailed shortly):

1. First, we update the current RL model for a fixed number of iterations over the current training environment distribution;

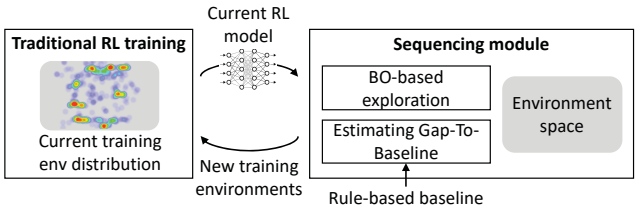


Figure 7: *Overview of Genet's training process.*

2. Second, we select the environments where the current RL model has a large gap-to-baseline; and
3. Third, we promote these selected environments in the training environments distribution used by the RL training process in the next iteration.

Training environment distribution: We define a distribution of training environments as a probability distribution over the space of *configurations*, each being a vector of 5–6 parameters (summarized in Table 3, 4, 5) used to generate network environments. An example configuration is: [BW: 2–3Mbps, BW changing frequency: 0–20s, Buffer length: 5–10s]. Genet sets the initial training environment distribution to be a uniform or exponential distribution along with each parameter, and automatically updates the distribution used in each iteration, effectively generating a training curriculum.

When recorded traces are available, Genet can augment the training with trace-driven environments as follows. Here we use bandwidth traces as an example. The first step is to categorize each bandwidth trace along with the bandwidth-related parameters (i.e., bandwidth range and variance in our case). Each time a configuration is selected by RL training to create new environments, with a probability of w (30% by default), Genet samples a bandwidth trace whose bandwidth-related parameters fall into the range of the selected configuration.

In §5.2, we will show that adding trace-driven environments to training improves the performance of RL policies, especially when tested in unseen real traces from the same distribution. That said, even if we do not use trace-driven environments in RL training, our trained RL policies still outperform the traditional method of training RL over real traces or synthetic traces.

Key components: Each round of Genet starts with training the current model for a fixed number of iterations (defaults to 10). Here, Genet reuses the traditional training method in prior work (i.e., uniform sampling of training environments per iteration), which makes it possible to incrementally apply Genet to existing codebases (see our implementation in §4.3). Recent work on domain randomization [39, 45, 52] also shows that a similar training process can benefit the generalization of RL policies [39, 45, 52]. The details of the training process are described in Algorithm 1.

After a certain number of iterations, the current RL model and a pre-determined rule-based baseline are given to a *sequencing module* to search for the environments where the current RL model has a large gap-to-baseline. Ideally, we want to test the current RL model on all possible environments and identify the ones with the largest gap-to-baseline, but this is prohibitively expensive. Instead, we use *Bayesian Optimization* [17] (BO) as follows. We view the expected gap-to-baseline over the environments created by configuration p as a function of p: $Gap(p) = R(\pi^{rule}, p) - R(\pi_\theta^{rl}, p)$, where $R(\pi, p)$

is the average reward of a policy π (either the rule-based baseline π^{rule} or the RL model π_θ^{rl}) over k (10 by default) environments randomly generated by configuration p. BO then searches in the environment space for the configuration that maximizes $Gap(p)$.

Once a new configuration is selected, the environments generated by this configuration are then added to the training distribution as follows. When the RL training process samples a new training environment, it will choose the new configuration with w probability (30% by default) or uniformly sample a configuration from the old distribution with $1-w$ probability (70% by default), and then create an environment based on the selected configuration. Next, training is resumed over the new environment distribution.

It is important to notice that the BO-based search does *not* carry its states when searching rewarding environments for a new RL model. Instead, GENET restarts the BO search every time the RL model is updated. The reason is that the rewarding environments can change once the RL model changes.

Design rationale: The process described above embeds several design decisions that make it efficient.

How to choose rule-based baselines? For GENET to be effective, the baselines should not fail in simple environments; otherwise, GENET would ignore them given that the RL policy could easily beat the baselines. For instance, when using Cubic as the baseline in training RL-based CC policies, we observe that the RL policy is rarely worse than Cubic along the dimension of random loss rate, because Cubic's performance is susceptible to random packet losses. That said, we find that the choice of baselines does not significantly impact the effectiveness of GENET, although a better choice tends to yield more improvement (as shown in §5.5).[6]

Why is BO-based exploration effective? GENET models the selection of network environments that maximize gap-to-baseline as a parameter search procedure in a high-dimensional space—each dimension of the space is a configuration of the network environment (e.g., link latency), each point in the space is a set of network environments with the same configurations, and the desired points are those whose environments have large gap-to-baselines. This problem has two features: (1) the environment search space is high-dimensional, and (2) evaluating the gap-to-baseline of a point in the space is computationally expensive (partly due to the variance among the environments with the same configurations). In this context, BO is merely one of the candidate solutions among several others to perform the parameter search. In §5.5, we will compare BO's efficiency with other candidate solutions and show that BO is efficient at identifying rewarding environments.

Why not set a threshold for the gap-to-baseline of the selected environments? While GENET uses BO to search rewarding environments with a fixed number of steps (default is 15), an alternative is to run BO until it finds an environment configuration whose gap-to-baseline is above a threshold. However, the latter strategy may not end (or take a long time to finish) if the RL model is already better than the baseline in most environments, which is possible during training. Moreover, the threshold introduces another hyper-parameter to be tuned with domain knowledge.

[6] One possible refinement in this regard is to use an "ensemble" of rule-based heuristics, and let the training scheduler focus on environments where the RL policy falls short of any one of a set of rule-based heuristics.

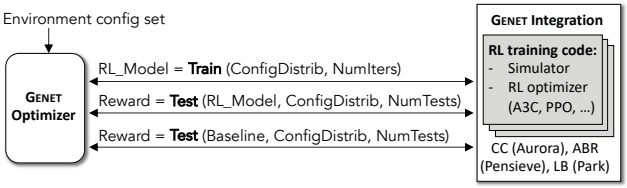


Figure 8: *Components and interfaces needed to integrate GENET with an existing RL training codebase.*

Impact of forgetting? It is important that we train models over the *full* range of environments. GENET does begin the training over the whole space of environment in the first iteration, but each subsequent iteration introduces a new configuration, thus diluting the percentage of random environments in training. This might lead to the classic problem of forgetting—the trained model may forget how to handle environments seen before. While we do not address this problem directly, we have found that GENET is affected by this issue only mildly. The reason is that GENET stops the training after changing the training distribution for 9 times, and by then, the original environment distribution still accounts for about 10%.[7]

4.3 Implementation

GENET is fully implemented in Python and Bash, and has been integrated with three existing RL training codebases. Next, we describe the interface and implementation of GENET, as well as optimizations for eliminating GENET's performance bottlenecks.

API: GENET interacts with an existing RL training codebase with two APIs (Figure 8): `Train` signals the RL to continue the training using the given distribution of environment configurations and returns a snapshot of the model after a specified number of training iterations; `Test` calculates the average reward of a given algorithm (RL model or a baseline) over a specified number of environments drawn from the given distribution of configurations.

Integration with RL training: We have integrated GENET with Pensieve ABR [4], Aurora CC [1], and Park LB [3], which use different RL algorithms (e.g., A3C, PPO) and network simulators (e.g., packet level, chunk level). We implement the two APIs above using functionalities provided in the existing codebase.

Rule-based baselines: GENET takes advantage of the fact that many RL training codebases (including our three use cases) have already implemented at least one rule-based baseline (e.g., MPC in ABR, Cubic in CC) that runs in their simulators. In addition, we also implemented a few baselines by ourselves, including the shortest-job-first in LB, and BBR in CC. The implementation is generally straightforward, but sometimes the simulator (though sufficient for the RL policy) lacks crucial features for a faithful implementation of the rule-based logic. Fortunately, GENET-based RL training merely uses the baseline to select training environments, so the consequence of having a suboptimal baseline is not considerable.

5 EVALUATION

The key takeaways of our evaluation are:

[7] When we impose a minimum fraction of "exploration" (i.e., uniformly randomly picking an environment from the original training distribution) in the training (which is a typical strategy to prevent forgetting [58]), GENET's performance becomes worse.

Name	Use case	Training # traces, total length (s)	Testing # traces, total length (s)
FCC	ABR	85, 105.8k	290, 89.9k
Norway	ABR	115, 30.5k	310, 96.1k
Ethernet	CC	64, 1.92k	112, 3.35k
Cellular	CC	136, 4.08k	121, 3.64k

Table 2: *Network traces used in ABR and CC tests.*

- Across three RL use cases in networking, GENET improves the performance of RL algorithms when tested in new environments drawn from the training distributions that include wide ranges of environments (§5.2).
- GENET improves the generalization of RL performance, allowing models trained over synthetic environments to perform well even in various trace-driven environments as well as on real-world network connections (§5.3).
- GENET-trained RL policies have a much higher chance of outperforming various rule-based baselines specified during GENET-based RL training (§5.4).
- Finally, the design choices of GENET, such as its curriculum learning strategy and BO-based search, are shown to be effective compared to seemingly natural alternatives (§5.5).

Given the success of curriculum learning in other RL domains, these improvements are not particularly surprising. However, by showing for the first time that curriculum learning facilitates RL training in networking, we hope to inspire more follow-up research in this direction.

5.1 Setup

We train GENET for three RL use cases in networking, using their original simulators: congestion control (CC) [1], adaptive bitrate streaming (ABR) [4], and load balancing (LB) [3]. As discussed in §4.1, we train and test RL policies over two types of environments.

Synthetic environments: We generate synthetic environments using the parameters described in detail in §A.2 and Table 3,4,5. We choose these environment parameters to cover a variety of factors that affect RL performance. For instance, in CC tests, our environment parameters specify bandwidth (e.g., the range, variance, and how often it changes), delay, queue length, etc.

Trace-driven environments: We also use real traces for CC and ABR (summarized in Table 2) to create trace-driven environments (in both training and testing), where the bandwidth time series are set by the real traces, but the remaining environment parameters (e.g., queue length or target video buffer length) are set as in the synthetic environments. We test ABR policies by streaming a pre-recorded video over 290 traces from FCC broadband measurements [11] (labeled "FCC") and 310 cellular traces [43] (labeled "Norway"). We test CC policies on 121 cellular traces (labeled "Cellular") and 112 Ethernet traces (labeled "Ethernet") collected by the Pantheon platform [56].

Baselines: We compare GENET-trained policies with several baselines. First, *traditional RL* trains RL policies by uniformly sampling environments from the target distribution per iteration. We train three types of RL policies (RL1, RL2, RL3) over fixed-width uniform distribution of synthetic environments, specified in Table 3, 4, 5. From RL1 to RL3, the sizes of their training environment ranges are in ascending order.

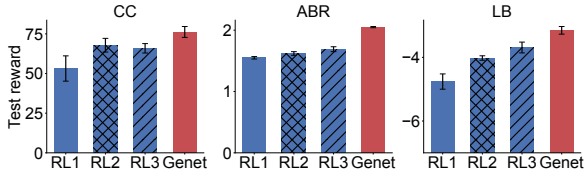


Figure 9: *Comparing the performance of GENET-trained RL policies for CC, ABR, and LB, with baselines in unseen synthetic environments drawn from the training distribution, which sets all environment parameters to their full ranges.*

We also train RL policies over trace-driven environments, i.e., randomly picking bandwidth traces from one of the recorded sets. This is the same as prior work, except that we also vary non-bandwidth-related parameters (e.g., queue length, buffer length, video length, etc) to increase its robustness. In addition, we test an early attempt to improve RL [19] which generates new training bandwidth traces that maximize the gap between the RL policy and optimal adaptation with a non-smoothness penalty (§5.5).

Second, *traditional rule-based algorithms* include BBA [23] and RobustMPC [57] for ABR, PCC-Vivace [14], BBR [8] and CUBIC for CC, and least-load-first (LLF) for LB.[8] They can be viewed as a reference point for traditional non-ML solutions.

5.2 Asymptotic performance

We first compare GENET-trained policies and traditionally trained RL policies, in terms of their *asymptotic performance* (i.e., test performance over new test environments drawn independently from the training distribution). In other words, we train RL policies over environments from the target distribution and test them in new environments from the same distribution.

Synthetic environments: We first test GENET-trained CC, ABR, and LB policies under their perspective RL3 synthetic ranges (where all parameters are set to their full ranges) as the target distribution. As shown in Figure 2, in these training ranges, traditional RL training yields little performance improvement over the rule-based baselines. Figure 9 compares GENET-trained CC, ABR, and LB policies with their respective baselines over 200 new synthetic environments randomly drawn with the target distribution.

Across three use cases, we can see that GENET consistently improves over traditional RL-trained policies by 8–25% for ABR, 14–24% for CC, 15% for LB, compared with traditional RL training methods. We notice that there is no clear ranking among the three traditional RL-trained policies. This is because RL1 helps training to converge better but only sees a small slice of the target distribution, whereas RL3 sees the whole distribution but cannot train a good model. In contrast, GENET outperforms them, as curriculum learning allows it to learn more efficiently from the large target distribution.

To show the performance more thoroughly, Figure 10 picks ABR as an example and shows the performance across different values along with six environment parameters. We vary one parameter at a time while fixing other parameters at the same default values (see Table 3, 4, 5). We see that GENET-trained RL policies enjoy consistent performance advantages (in reward) over the RL policies trained by

[8]By default, we use RobustMPC as MPC and PCC Vivace-latency as Vivace, since they appear to perform better than their perspective variants.

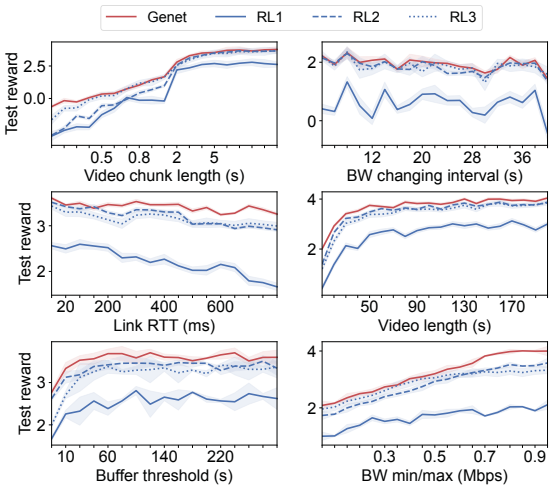


Figure 10: *Test of ABR policies along individual env-parameters.*

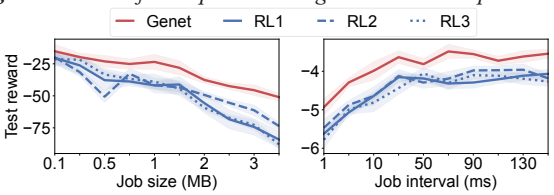


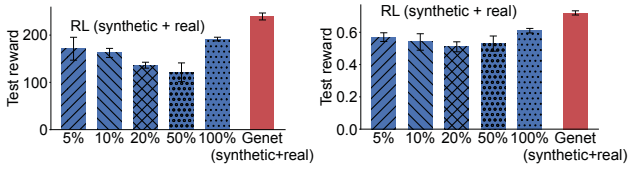
Figure 11: *Test of LB policies along individual env-parameters.*

traditional RL-trained models. This suggests that the improvement of GENET shown in Figure 9 is not a result of improving rewards in some environments at the cost of degrading rewards in others; instead, GENET improves rewards in most cases. Figure 11 shows that in the simulated environments [3], the GENET-trained LB policy outperforms its baselines by 15%.

Trace-driven environments: Next, we set the target environment distributions of ABR and CC to be the environments generated from multiple real-world trace sets (FCC and Norway for ABR, Ethernet and Cellular for CC). We partition each trace set as listed in Table 2. GENET trains ABR and CC policies by combining trace-driven environments and synthetic environments (described in §4.2). For a thorough comparison, both GENET and the traditional RL training have access to the training portion of the real traces as well as the synthetic environments. We vary the ratio of real traces and synthetic environments and feed them to the traditional RL training method, e.g., if the ratio of real traces is 20%, then the traditional RL training randomly draws a trace-driven environment with 20% probability and synthetic environments with 80% probability. That is, we test different ways for the traditional RL training to combine the training traces and synthetic environments. Figure 12 tests GENET-trained ABR and CC policies with their respective traditional RL-trained baselines over new environments generated from the traces in the testing set. Figure 12 shows that GENET-trained policies outperform traditional RL training by 17–18%, regardless of the ratio of real traces, including when training the model entirely on real traces.

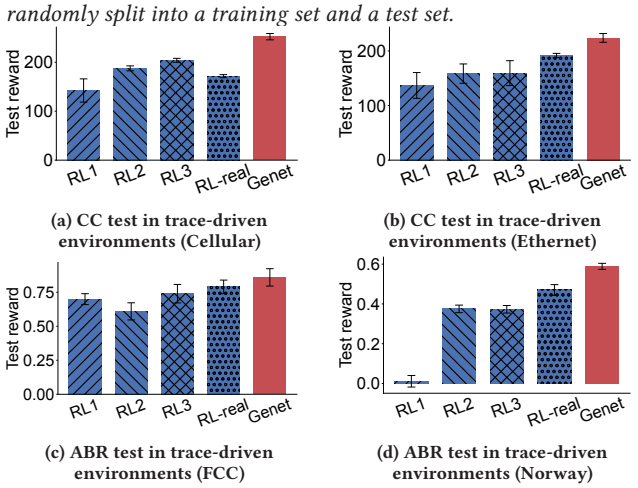
5.3 Generalization

Next, we take the RL policies of ABR and CC trained (by GENET and other baselines) entirely over synthetic environments (the RL3



(a) Congestion control (CC) **(b) Adaptive bitrate (ABR)**

Figure 12: *Asymptotic performance of GENET-trained CC policies (a) and ABR policies (b) and baselines, when the real network traces are randomly split into a training set and a test set.*



(a) CC test in trace-driven environments (Cellular) **(b) CC test in trace-driven environments (Ethernet)**

(c) ABR test in trace-driven environments (FCC) **(d) ABR test in trace-driven environments (Norway)**

Figure 13: *Generalization test: Training of various methods is done entirely in synthetic environments, but the testing is over various real network trace sets.*

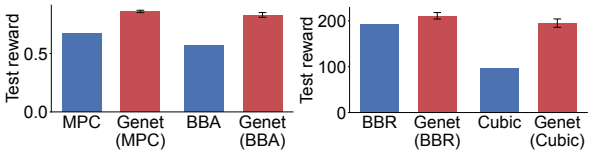
synthetic environment range) and test their generalization in trace-driven environments generated by the ABR (and CC) testing traces in Table 2.

Figure 13 shows that they perform better than traditional RL baselines trained over the same synthetic environment distribution. Though Figure 13 uses the same testing environments as Figure 12 and has a similar relative ranking between GENET and traditional RL training, the implications are different: Figure 13 also shows that when the real traces are *not* accessible in training, GENET can produce models with better generalization in real-trace-driven environments than the baselines, whereas Figure 12 shows their performance when the real traces are actively used in training of GENET and the baselines.

5.4 Comparison with rule-based baselines

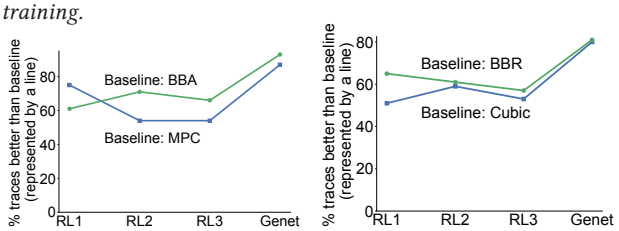
Impact of the choice of rule-based baselines: Figure 14 shows the performance of GENET-trained policies when using different rule-based baselines. We choose MPC and BBA as baselines in the ABR experiments and BBR and Cubic as baselines in CC experiments, respectively. We observe that in all cases, GENET-trained policies outperform their respective rule-based baselines.

What if GENET uses naive rule-based baselines? As explained in §4.2, the rule-based baseline should have a reasonable (though not necessarily optimal) performance; otherwise, it would be unable to indicate when the RL policy can be improved. To empirically verify it, we use two unreasonable baselines: choosing the highest bitrate when rebuffer in ABR, and choosing the highest loaded server in LB. In both cases, the BO-based search fails to find useful training



(a) ABR (b) CC

Figure 14: *Genet outperforms the rule-based baselines used in its training.*



(a) ABR (b) CC

Figure 15: *Fraction of real traces where Genet-trained policies (and traditional RL) are better than the rule-based baselines.*

environments, because the RL policy very quickly outperforms the naive baseline everywhere. That said, the negative impact of using a naive baseline is restricted to the selection of training environments, rather than the RL training itself (a benefit of decoupling baseline-driven environment selection and RL training), so in the worst case, Genet would be roughly as good as traditional RL training.

How likely is Genet to outperform rule-based baselines? One of Genet's benefits is to increase how often the RL policy is better than the rule-based baseline used in Genet. In Figure 15, we create various versions of Genet-trained RL policies by setting the rule-based baselines to be Cubic and BBR (for CC), and MPC and BBA (for ABR). Compared to RL1, RL2, RL3 (unaware of rule-based baselines), Genet-trained policies remarkably increase the fraction of real-world traces (emulated) where the RL policy outperforms the baseline used to train them. This suggests that operators can specify a rule-based baseline, and Genet will train an RL policy that outperforms it with high probability.

Breakdown of performance: Figure 17 takes one Genet-trained ABR policy (with MPC as the rule-based baseline) and one Genet-trained CC policy (with BBR as the rule-based baseline) and compares their performance with a range of rule-based baselines along with individual performance metrics. We see that the Genet-trained ABR and CC policies stay on the frontier and outperform other baselines.

Real-world tests: We also test the Genet-trained ABR and CC policies in five real wide-area network paths (without emulated delay/loss), between four nodes reserved from OpenNetLab [2, 16], one laptop at home, and two cloud servers (§A.4), allowing us to observe their interactions with real network traffic. For statistical confidence, we run the Genet-trained policies and their baselines back-to-back, each at least five times, and show their performance in Figure 16. The system metrics behind each reward value are shown in Table 6 and Table 7. In all but two cases, Genet outperforms the baselines. On Path-2, Genet-trained ABR has little improvement, because the bandwidth is always much higher than the highest bitrate, and the baselines will simply use the highest bitrate, leaving no room for improvement. On Path-3, Genet-trained CC has negative improvement, because the network has a deeper queue than used in training, so RL cannot handle it well. This is an example where Genet can fail when tested out of the range of training environments. These results do not prove that the policies generalize to all environments; instead, they show Genet's performance in a range of network settings.

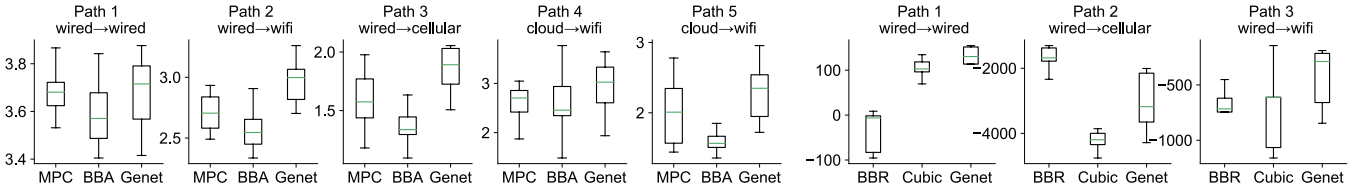
5.5 Understanding Genet's design choices

Alternative curriculum-learning schemes: Figure 18 compares Genet's training curve with that of traditional RL training and three alternatives for selecting training environments described in §3. **CL1** uses hand-picked heuristics (gradually increasing the bandwidth fluctuation frequency in the training environments), **CL2** uses the performance of a rule-based baseline (gradually adding environments where BBR for CC and MPC for ABR performs badly), and **CL3** adds traces where the current RL model is much worse than the optimum (whereas Genet picks the traces where the current RL model is much worse than a rule-based baseline). Compared to these baselines, In Figure 18, we show that Genet's training curves have faster ramp-ups, suggesting that with the same number of training iterations, Genet can arrive at a much better policy, which corroborates the reasoning in §3.

In addition, "Robustifying" [19][9] (which learns an adversarial bandwidth generator) also tries to improve ABR logic by adding more challenging environments to training. For a more direct comparison with Genet, we implement a variant of Genet where BO picks configurations that maximize the gap between RL and the optimal reward (penalized by bandwidth non-smoothness with different weights of p). Figure 19 compares the resulting RL policies with Genet-trained RL policy and MPC as a baseline on the synthetic traces in Figure 10. We see that they perform worse than Genet-trained ones and that by changing the BO's environment selection criteria, Genet becomes less effective. Genet outperforms Robustifying, because the non-smoothness metric used in [19] may not completely capture the inherent difficulty of bandwidth traces (Figure 5 shows a concrete example).

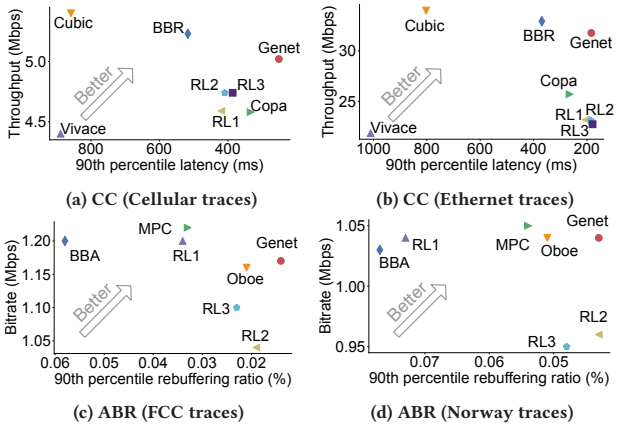
BO-based search efficiency: Genet uses BO to explore the multi-dimensional environment space environment to find the environment configuration with a large gap-to-baseline. While BO may not always find the single optimal point in arbitrary blackbox function between environment parameters and gap-to-baseline, we found it to be a pragmatic solution. To show it, we randomly choose an intermediate RL model during the Genet training of ABR and CC. Figure 20 shows the gap-to-baseline of the configuration selected by BO for each model within 15 search steps. Within a small number of steps, it can identify a configuration that is almost as good as randomly searching for 100 points, which is much more expensive. Figure 20 also includes the grid search as a reference, which starts with all configurations initialized to their respective midpoints and then searches and updates the best value for each configuration one by one. We observe that it does not converge as fast as BO.

[9] In lack of a public implementation, we follow the description in [19] (e.g., non-smoothness weight) and apply it to Pensieve (with the only difference being that for fair comparisons with other baselines, we apply it on Pensieve trained on our synthetic training environments). We have verified that our implementation of Robustifying achieves similar improvements in the setting of original paper. More details are in Appendix A.6.



(a) ABR (b) CC

Figure 16: *Testing ABR and CC policies in real-world environments.*



(a) CC (Cellular traces) (b) CC (Ethernet traces)

(c) ABR (FCC traces) (d) ABR (Norway traces)

Figure 17: *RL-based ABR and CC vs. rule-based baselines.*

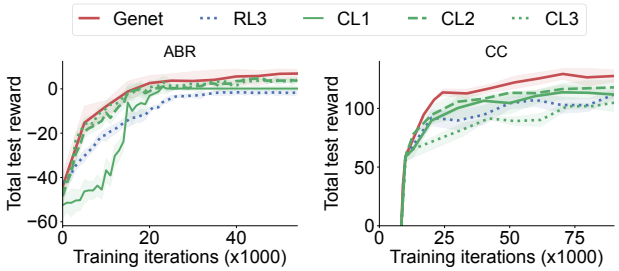


Figure 18: *GENET's training ramps up faster than alternative curriculum learning strategies.*

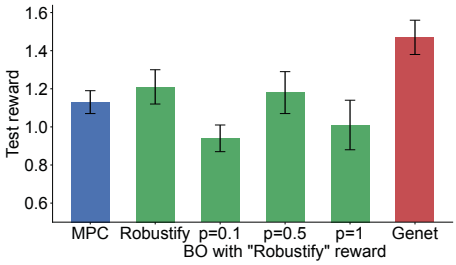
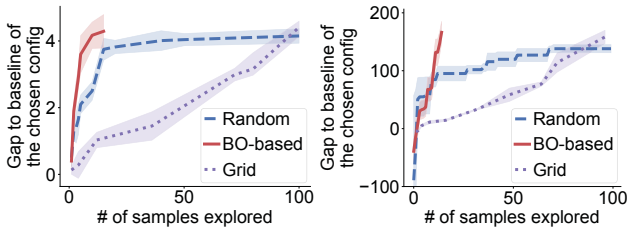


Figure 19: *GENET outperforms Robustifying [19] that improves RL performance by generating adversarial bandwidth traces, and variants of GENET using Robustifying's criteria in BO-based environment selection.*

6 RELATED WORK

Improving RL for networking: Some of our findings regarding the lack of generalization corroborate those in previous work [13, 19, 24, 32, 44, 54]. To improve RL for networking use cases, prior work has attempted to apply and customize techniques from the ML literature. For instance, [19] applies adversarial learning by generating relatively smooth bandwidth traces that maximize the RL regret w.r.t. optimal outcomes, [15, 26] show that the generalization of RL can be improved by incorporating training environments where a given RL policy violates pre-defined safety conditions, [46, 47] incorporate randomization in the evaluation of RL-based systems, and Fugu [55] achieves a similar goal through learning a transmission time predictor *in situ*. Other proposals seek to safely deploy a given RL policy in new environments [33, 44, 49]. In many ways, GENET follows this line of work, but it is different in that it systematically introduces curriculum learning, which has underpinned many recent enhancements of RL and demonstrates its benefits across multiple applications.



(a) ABR (b) CC

Figure 20: *BO-based search is more efficient at finding environments with large gap-to-baselines than random exploration in the environment configuration space.*

Curriculum learning for RL: There is a substantial literature on improving deep RL with curricula ([21, 36, 40] give more comprehensive surveys on this subject). Each component of curriculum learning has been extensively studied, including how to generate tasks (environments) with potentially various difficulties [48, 50], how to sequence tasks [42, 51], and how to add a new task to training (transfer learning). In this work, we focus on sequencing tasks to facilitate RL training. It is noticed that, for general tasks that do not have a clear definition of difficulty (like networking tasks), optimal task sequencing is still an open question. Some approaches, such as self-paced learning [28] advocate the use of easier training examples first, while the other approaches prefer to use harder examples first [9]. Recent work tries to bridge the gap by suggesting that an ideal next training task should be difficult for the current model's hypothesis, while it is also beneficial to prefer easier points with respect to the target hypothesis [21]. In other words, we should prefer an easy environment that the current RL model cannot handle well, which confirms the intuition elaborated in Bengio's seminal paper [7], which hypothesizes that "it would be beneficial to make learning focus on 'interesting' examples that are neither too hard nor too easy." GENET is an instantiation of this

idea in the context of networking adaptation, and the way to identify the rewarding (or "interesting") environments is by using the domain-specific rule-based schemes to identify where the current RL policy has a large room for improvement.

Automatic generation of curricula also benefits generalization, particularly when used together with domain randomization [39]. Several schemes boost RL's training efficiency by iteratively creating a curriculum of challenging training environments (e.g., [12, 35]) where the RL performance is much worse than the optimal outcome (i.e., maximal regret). When the optimal policy is unavailable, they learn a competitive baseline [12] to approximate the optimal policy or a metric [35] to approximate the regret. Genet falls in this category, but proposes a domain-specific way of identifying rewarding environments using rule-based algorithms.

Some proposals in safe policy improvement (SPI) for RL also use rule-based schemes [18, 29], though for different purposes than Genet. While Genet uses the performance of rule-based schemes to identify where the RL policy can be maximally improved, SPI uses the decisions of rule-based algorithms to avoid violation of failures during training.

7 DISCUSSION

Does a small gap-to-baseline always mean that an RL model has small improvement when trained on it?
Although a small gap-to-baseline on a network environment indicates that the RL model already performs quite closely with the rule-based baseline, there is still a chance that the RL model could be greatly improved when trained in that environment. This is because if the rule-based baseline performs very badly in an environment, the gap-to-baseline will no longer be indicative of the potential improvement of RL training. For example, Cubic may perform poorly on a high-bandwidth link with occasional random packet loss, as Cubic does not differentiate random packet loss and congestion-induced loss, causing it to lower congestion window size when the available bandwidth does not drop. In such cases, even if an RL model has a small gap-to-baseline with Cubic, there *could* still be room for the RL model to improve performance, but Genet may not choose to prioritize such environments. That said, this problem could be mitigated by using a more performant baseline or an "ensemble" of existing baselines (i.e., measuring the maximum gap to *any* baseline from a set).

Does training in environments of large gap-to-baseline always lead to large RL model improvement?
Unfortunately, the answer is not always. RL models may not always be able to approximate the performance of rule-based baselines, e.g., due to an RL model's coarse decision granularity. For instance, Aurora (an RL-based CC) is a monitor-interval-based CC algorithm. Each monitor interval needs to be long enough to accumulate enough packet acks (e.g., 10–50) to compute the features (throughput, latency, etc.) for the RL model to select the sending rate. In contrast, traditional TCP algorithms like Cubic and BBR can update sending rate (cwnd) on the arrival of each packet ack. Thus, Aurora has a much coarser decision granularity than traditional TCPs, rendering it hard for the RL model to approximate the traditional TCP's behavior when the network condition suddenly changes. For instance, during sudden bandwidth drops and rapid queue buildups, the inter-packet interval dramatically increases, and so does Aurora's monitor interval, whereas TCP Cubic or BBR can still update its sending rate on each packet ack. In these cases, Aurora will never ramp up or reduce sending rate as fast as its rule-based baselines, so even with a large gap-to-baseline in such environments, Aurora may not see a large reward improvement.

What if a rule-based baseline does not exist?
The current Genet training framework requires the existence of a rule-based baseline for the target networking problem. If the problem does not have a well-studied rule-based baseline, there are three alternative training methods that Genet can fall back to. First, Genet can fall back on traditional RL training. Although it loses the benefits of curriculum learning, it may still produce a reasonable RL-based policy. Second, we can use the performance gap between an optimal solution based on ground truth knowledge (such as future bandwidth variation) and the current RL model as the guidance of rewarding network environment selection. [19] trains an ABR RL model using network traces from a bandwidth-generating model. The training of the bandwidth-generating model is then guided by the performance gap between the optimal solution and the current RL model. This training method works well when the optimal solution is feasible and computationally cheap. Third, a trained RL model can be treated as a rule-based baseline. [12] trains two RL models (with identical model architecture) competitively on the environments produced by an adversarial generator. The adversarial generator is a neural network that aims to maximize the reward difference between the two RL models. However, the training complexity increases due to the increased number of models to be trained. Even though Genet can fall back on alternative training methods, how to extend it to work in applications domains that do not have an existing rule-based baseline remains to be investigated.

8 CONCLUSION

We present Genet, a new training framework to improve the training of deep RL-based network adaptation algorithms. For the first time, we introduce curriculum learning to the networking domain as the key to reaching better RL performance and generalization. To make curriculum learning efficient in networking, the main challenge is how to automatically identify the "rewarding" environments that can maximally benefit from retraining. Genet addresses this challenge with a simple-yet-efficient idea that highly rewarding network environments are where the current RL performance falls significantly behind that of a rule-based baseline scheme. Our evaluation on three RL use cases shows that Genet improves RL policies (in both performance and generalization) in various environments and workloads.

Ethics: This work does not raise any ethical issues.

9 ACKNOWLEDGEMENTS

We thank Microsoft Research and OpenNetLab for their generous support of server nodes. We thank the SIGCOMM reviewers and our shepherd, Michael Schapira, for their invaluable feedback. This research is partly supported by NSF (2146496, 1901466, 2131826), UChicago CERES Center, a Google Faculty Research Award.

REFERENCES

[1] Aurora implementation. https://github.com/PCCproject/PCC-RL.
[2] OpenNetLab. https://opennetlab.org/.
[3] Park implementation. https://github.com/park-project/park.
[4] Pensieve implementation. https://github.com/hongzimao/pensieve.
[5] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video ABR algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.
[6] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.
[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.
[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.
[9] Haw-Shiuan Chang, Erik Learned-Miller, and Andrew McCallum. Active bias: Training more accurate neural networks by emphasizing high variance samples. *Advances in Neural Information Processing Systems*, 30:1002–1012, 2017.
[10] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. Cellular network traffic scheduling with deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.
[11] Federal Communications Commission. Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.
[12] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *arXiv preprint arXiv:2012.02096*, 2020.
[13] Arnaud Dethise, Marco Canini, and Srikanth Kandula. Cracking open the black box: What observations can tell us about reinforcement learning agents. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 29–36, 2019.
[14] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation NSDI 18)*, pages 343–356, 2018.
[15] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 305–318, 2021.
[16] Jeongyoon Eo, Zhixiong Niu, Wenxue Cheng, Francis Y. Yan, Rui Gao, Jorina Kardhashi, Scott Inglis, Michael Revow, Byung-Gon Chun, Peng Cheng, and Yongqiang Xiong. OpenNetLab: Open platform for RL-based congestion control for real-time communications. In *6th Asia-Pacific Workshop on Networking (APNet 2022)*, 2022.
[17] Peter I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.
[18] Mohammad Ghavamzadeh, Marek Petrik, and Yinlam Chow. Safe policy improvement by minimizing robust baseline regret. *Advances in Neural Information Processing Systems*, 29:2298–2306, 2016.
[19] Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 85–92, 2019.
[20] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.
[21] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In *International Conference on Machine Learning*, pages 2535–2544. PMLR, 2019.
[22] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*, 2019.
[23] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.
[24] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.
[25] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.
[26] Yafim Kazak, Clark Barrett, Guy Katz, and Michael Schapira. Verifying deep-rl-driven systems. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 83–89, 2019.
[27] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *arXiv preprint arXiv:2111.09794*, 2021.
[28] M. Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *NIPS*, volume 1, page 2, 2010.
[29] Romain Laroche, Paul Trichelair, and Remi Tachet Des Combes. Safe policy improvement with baseline bootstrapping. In *International Conference on Machine Learning*, pages 3652–3661. PMLR, 2019.
[30] Hongzi Mao. Pensieve collected data. https://www.dropbox.com/sh/ss0zs1lc4cklu3u/AAB-8WC3cHD4PTtYT0E4M19Ja?dl=0.
[31] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An open platform for learning augmented computer systems. 2019.
[32] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.
[33] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. Towards safe online reinforcement learning in computer systems. In *NeurIPS Machine Learning for Systems Workshop*, 2019.
[34] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.
[35] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active domain randomization. In *Conference on Robot Learning*, pages 1162–1176. PMLR, 2020.
[36] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint arXiv:2003.04960*, 2020.
[37] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 international conference on autonomous agents & multiagent systems*, pages 566–574, 2016.
[38] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.
[39] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE, 2018.
[40] Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020.
[41] Aditi Raghunathan, Sang Michael Xie, Fanny Yang, John C Duchi, and Percy Liang. Adversarial training can hurt generalization. *arXiv preprint arXiv:1906.06032*, 2019.
[42] Zhipeng Ren, Daoyi Dong, Huaxiong Li, and Chunlin Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE transactions on neural networks and learning systems*, 29(6):2216–2226, 2018.
[43] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3g networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 114–118, 2013.
[44] Noga H. Rotman, Michael Schapira, and Aviv Tamar. Online safety assurance for deep reinforcement learning. *arXiv preprint arXiv:2010.03625*, 2020.
[45] Fereshteh Sadeghi and Sergey Levine. CAD2R: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.
[46] Michael Schaarschmidt. End-to-end deep reinforcement learning in computer systems. Technical report, University of Cambridge, Computer Laboratory, 2020.
[47] Michael Schaarschmidt, Kai Fricke, and Eiko Yoneki. Wield: Systematic reinforcement learning with progressive randomization. *arXiv preprint arXiv:1909.06844*, 2019.
[48] Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.
[49] Junyang Shi, Mo Sha, and Xi Peng. Adapting wireless mesh network configuration from simulation to reality via deep learning based domain adaptation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.
[50] Felipe Leno Da Silva and Anna Helena Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1026–1034, 2018.
[51] Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

[52] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.
[53] Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. In *International Conference on Machine Learning*, pages 5238–5246. PMLR, 2018.
[54] Keith Winstein and Hari Balakrishnan. TCP ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.
[55] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.
[56] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, July 2018. USENIX Association.
[57] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.
[58] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.
[59] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*, 2020.

A APPENDICES

Appendices are supporting material that has not been peer-reviewed.

A.1 Details of RL implementation

The input of RL algorithm consists of a space of configurations, an initial policy parameters and predefined total number of iterations to train. The space of configurations is constructed by ranges of environment configurations. Each range is marked by the configuration's min and max values. Within a training iteration, each dimension of the space of configurations is uniformly sampled to create K configurations. For each configuration, N random environments are created. Thus, rollouts are collected by running the policy on total $K \times N$ environments to update the policy. When the policy is updated for the predefined number of iterations, the RL algorithm stops training and outputs a trained policy.

Algorithm 1 Traditional Reinforecment Learning (RL)

Input: Ω: space of configurations, θ: initial policy parameters, N_{iters}: # of iterations
Output: θ: returned policy parameters

for i from 1 to N_{iters} **do**
  $\Phi_{rand} \leftarrow \emptyset$
  for 1 **to** K **do** ▷ K: # configs per iteration
    $p_i \sim Random(\Omega)$ ▷ Uniformly sampled config in Ω
    for 1 **to** N **do** ▷ N: # random envs per config
      $E \leftarrow S(p_i)$ ▷ Create a simulated env by p_i
      rollout $\phi \sim \pi_\theta(\cdot; E)$ ▷ Rollout policy π_θ on E
      $\Phi_{rand} \leftarrow \Phi_{rand} \cup \phi$
    end for
  end for
  with Φ_{rand} update:
  $\theta \leftarrow \theta + \nu \bigtriangledown_\theta J(\pi_\theta)$ ▷ Gradient update with rate ν
end for
return θ

Algorithm 2 GENET training framework

Input: Ω: uniform configuration distribution (equal probability on each configuration), π^{rule}: rule-based policy.
Output: θ: final RL policy parameters

function GENET(Ω, π^{rule})
  $\theta \leftarrow$ Random initial policy parameters
  $\Omega_{cur} \leftarrow \Omega$ ▷ Ω_{cur} will be updated and used for training
  for from 1 to N_{iter} **do** ▷ # of exploration iterations
    BO.INITIALIZE(Ω) ▷ Initialize with full config space Ω
    for from 1 to $N_{boTrials}$ **do** ▷ # of trial configs by BO
      $p \leftarrow$ BO.GETNEXTCHOICE()
      $adv \leftarrow$ CALCBASELINEGAP($p, \pi_\theta^{rl}, \pi^{rule}$)
      BO.UPDATE(p, adv)
    end for
    $p_{new} \leftarrow$BO.GETDECISION()
    ▷ Weight new config p_{new} by w and old configs by $1 - w$
    $\Omega_{cur} \leftarrow (1 - w) \cdot \Omega_{cur} + w \cdot \{p_{new}\}$
    $\theta \leftarrow$UNIFORMDOMAINRAND($\Omega_{cur}, \theta, N_{iters}$)
  end for
  return θ
end function
function CALCBASELINEGAP($p, \pi_\theta^{rl}, \pi^{rule}$)
  Initialize: $samples \leftarrow \emptyset$
  for 1 **to** N_{Tests} **do** ▷ # of reward comparisons
    $E \leftarrow S(p)$ ▷ Create a simulated env by p_i
    rollout $\phi^{rl} \sim \pi_\theta^{rl}(\cdot; E)$ ▷ Rollout RL π^{rl}
    rollout $\phi^{rule} \sim \pi^{rule}(\cdot; E)$ ▷ Rollout rule-based π^{rule}
    add $Reward(\phi^{rule}) - Reward(\phi^{rl})$ to $samples$
  end for
  return MEAN($samples$)
end function

A.2 Trace generator logic

ABR: For the simulation in ABR, the link bandwidth trace has the format of [timestamp (s), throughput (Mbps)]. Our synthetic trace generator includes 4 parameters: minimum BW (Mbps), maximum BW (Mbps), BW changing interval (s), and trace duration (s). Each timestamp represents one second with a uniform [-0.5, 0.5] noise. Each throughput follows a uniform distribution between [min BW, max BW]. The BW changing interval controls how often throughput change over time, with uniform [1, 3] noise. Trace duration represents the total time length of the current trace.

CC: The trace generator in the CC simulation takes 6 inputs: maximum BW (Mbps), BW changing interval (s), link one-way latency (ms), queue size (packets), link random loss rate, delay noise (ms), and duration (s). It outputs a series of timestamps with 0.1s step length and dynamic bandwidth series. Each bandwidth value is drawn from a uniform distribution of range [1, max BW] Mbps. The BW changing interval allows bandwidth to change every certain seconds. The link one-way latency is used to simulate packet RTT. The queue size simulates a single queue in a sender-receiver network. Link random loss rate determines the chance of random packet loss in the network. Delay noise determines how large a Gaussian noise is added to a packet. The trace duration is determined by the duration input.

LB: We use the similar workload traces generator as the Park [3] project, where jobs arrive according to a Poisson process, and the

ABR Parameter	RL1	RL2	RL3	Default	Original
Max playback buffer (s)	[2, 10]	[2, 50]	[2, 100]	60	60
Video chunk length (s)	[1, 4]	[1, 6]	[1, 10]	4	4
Min link RTT (ms)	[20, 30]	[20, 220]	[20, 1000]	80	80
Video length (s)	[40, 45]	[40, 200]	[40, 400]	196	196
Bandwidth change interval (s)	[2, 2]	[2, 20]	[2, 100]	5	
Max link bandwidth (Mbps)	[2, 5]	[2, 100]	[2, 1000]	5	

Table 3: *Parameters in ABR simulation. Colored rows show the configurations (and their ranges) used in the simulator in the original paper. The synthetic trace generator is described in §A.2.*

CC Parameter	RL1	RL2	RL3	Default	Original
Maximum link bandwidth (Mbps)	[0.5, 7]	[0.4, 14]	[0.1, 100]	3.16	[1.2, 6]
Minimum link RTT (ms)	[205, 250]	[156, 288]	[10, 400]	100	[100, 500]
Bandwidth change interval (s)	[11, 13]	[8, 3]	[0, 30]	7.5	
Random loss rate	[0.01, 0.014]	[0.007, 0.02]	[0, 0.05]	0	[0, 0.05]
Queue (packets)	[2, 6]	[2, 11]	[2, 200]	10	[2, 2981]

Table 4: *Parameters in CC simulation. Colored rows show the configurations (and their ranges) used in the simulator in the original paper. The synthetic trace generator is described in §A.2. The range of RL1 is defined as 1/9 of the range of RL3 and the range of RL2 is defined as 1/3 of RL3. The CC parameters shown here for RL1 and RL2 are example sets.*

LB Parameter	RL1	RL2	RL3	Default	Original
Service rate	[0.1, 2]	[0.1, 5]	[0.1, 10]	[0.5, 1.0, 2.0]	[2, 4]
Job size (byte)	[100, 200]	[100, 10^3]	[1, 10^4]	2000	[100, 1000]
Job interval (ms)	[0.01, 0.05]	[0.01, 0.1]	[0.1, 1]	0.1	0.2
Number of jobs	[10, 100]	[10, 1000]	[10, 5000]	2000	1000
Queue shuffled probability	[0.1, 0.2]	[0.1, 0.5]	[0.1, 1]	0.5	

Table 5: *Parameters in LB simulation. Colored rows show the configurations (and their ranges) used in the simulator in the original paper. The synthetic trace generator is described in §A.2.*

job sizes follow a Pareto distribution with parameters [shape, scale]. In the simulation, all servers process jobs from their queues at identical rates.

A.3 Details of Figure 4

Trace sets in Figure 4 was generated by two configurations. For trace set X, we used BW range: 0–5Mbps, BW changing frequency: 0–2s. For trace set Y, we used BW range: 0–10Mbps, BW changing frequency: 4–15s. As a motivation example, each trace set contains 20 traces to show the testing reward trend.

A.4 Testbed setup

ABR: To test our model on a client-side system, we first leverage the testbed from Pensieve [4], which modifies dash.js (version 2.4) to support MPC, BBA, and RL-based ABR algorithms. We use the "Envivio- Dash3" video which format follows the Pensieve settings. In this emulation setup, the client video player is a Google Chrome browser (version 85) and the video server (Apache version 2.4.7) run on the same machine as the client. We use Mahimahi [38] to emulate the network environments from our pre-recorded FCC [30], cellular [43], Puffer [55] network traces, along with an 80 ms RTT, between the client and server. All above experiments are performed on UChicago servers.

CC: We build up CC testbed on Pantheon [56] platform on a Dell Inspiron 5521 machine. Pantheon uses network emulator Mahimahi [38] and a network tunnel which records packet status inside the network link. We run local customized network emulation in Mahimahi by providing a bandwidth trace and network configurations. We run remote network experiment by deplopying pantheon platform on the nodes shown in Figure 21. Among all the CC algorithms tested, BBR [8] and TCP Cubic [20] are provided by Linux kernel and are called via iperf3. PCC-Aurora [24] and PCC-Vivace [14] are implemented on top of UDP. We train our models in python and Tensorflow framework and port the models into the Aurora C++ code.

Real network testbed: We also test the GENET-trained ABR and CC policies in real wide-area network paths (depicted in Figure 21), including four nodes reserved from [2], one laptop at home, and two cloud servers.

A.5 Details on reward definition

ABR: The reward function of ABR is a linear combination of bitrate, rebuffering time, and bitrate change. The bitrate is observed in kbps, and the rebuffering time is in seconds, and bitrate change is the bitrate change between bitrate of current video chunk and that of the previous video chunk. Therefore, a reward value can be

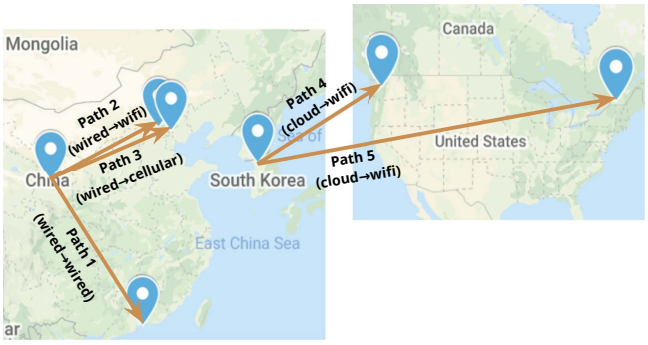


Figure 21: *Real-world network paths used to test ABR and CC policies.*

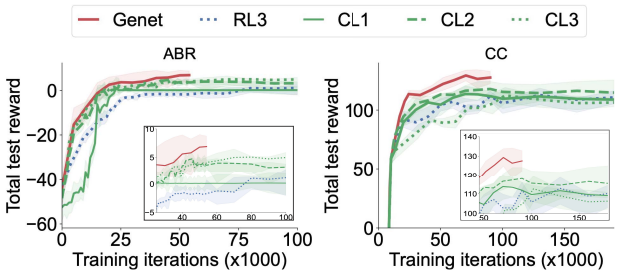


Figure 22: *Training RL and CL with more iterations still cannot outperform* Genet.

computed for a video chunk. The total reward of a video is the sum of the rewards of all video chunks.

CC: The reward function of CC is a linear combination of the throughput (packets per second), average latency (s), and packet loss (percentage) over a network connection. In training, a reward value is computed using the above metrics observed within a monitor interval. The total reward is the sum of the rewards of all monitor intervals in a connection.

LB: The reward function of LB is the average runtime delay of a job set, which is measured by milliseconds. For each server, we observe its total work waiting time in the queue and the remaining work currently being processed. After the incoming job is assigned, the server would summarize and update the delay of all active jobs.

A.6 Baseline implementation

According to the paper [19], we train an additional RL model for Robustify to improve the main RL-policy model by generating adversarial network traces inside ABR. The state of the adversary model contains the bitrate chosen by the protocol for the previous chunk, the client buffer occupancy, the possible sizes of the next chunk, the number of remaining chunks, and the throughput and download time for the last downloaded video chunk. The action is to generate the next bandwidth in the networking trace, in order to optimize the gap between the ABR optimal policy, RL-policy, and the unsmoothness, which is the absolute difference between the last two chosen bandwidths. Here, the penalty of unsmoothness is set as 1, same as the paper.

We use PPO as the training algorithm, and train the Robustify adversary model with a RL model until they both converge. Afterward, we add the traces Robustify model generated into the RL training process to retrain the RL. The PPO parameter settings follow the original paper.

As an alternative implementation, we also use the reward defined in Robustify as the training signal for BO to search and update environments. For the unsmoothness penalty here, we empirically tried three numbers: 0.1, 0.5, 1. From our results, penalty=0.5 works better than others.

A.7 Reward value breakdown

Table 6 and Table 7 contain the system metrics behind the reward values in Figure 16 for ABR and CC, respectively. The breakdown is done by decomposing the reward equations introduced in Table 1. For ABR, Table 6 shows that Genet tends to train a model that leads to less rebuffering and more smoothed bitrate selection without significantly sacrificing the average bitrate. For CC, Table 7 shows that Genet-trained model tends to have a lower 90th percentile latency and packet loss rate while not reducing throughput too much on Path 2 and 3. On Path 1, the performance gain is mainly from the larger throughput that Genet-trained model enables.

A.8 Train RLs and CLs with more iterations

To understand whether baselines like RLs and CLs can outperform Genet if they are given more training iterations, we trained RLs and CLs with twice as many training iterations as Genet. We empirically found that training with more iterations did not help the models trained by RLs and CLs as much as those trained by Genet. Their learning curves are shown in Figure 22.

	ABR	Bitrate (Mbps)	Rebuffering (s)	Bitrate change (Mbps)	Reward
Path 1	MPC	3.98	0.03	0.02	3.66
	BBA	3.84	0.018	0.15	3.51
	GENET	3.87	0.006	0.04	3.77
Path 2	MPC	3.22	0.041	0.07	2.74
	BBA	2.81	0.014	0.12	2.55
	GENET	3.15	0.008	0.07	3.01
Path 3	MPC	2.24	0.042	0.04	1.78
	BBA	1.75	0.03	0.05	1.40
	GENET	2.26	0.033	0.02	1.91
Path 4	MPC	2.93	0.013	0.04	2.76
	BBA	2.96	0.05	0.03	2.43
	GENET	2.88	0.002	0.02	2.84
Path 5	MPC	2.35	0.027	0.05	2.03
	BBA	1.82	0.022	0.04	1.56
	GENET	2.32	0.004	0.03	2.25

Table 6: *Reward breakdown of Figure 16(a) in ABR real-world experiment.*

Path	CC	Throughput (Mbps)	90th percentile latency (ms)	Packet loss rate	Reward
Path 1	BBR	164.2	57.25	0.0906	-35.62
	Cubic	158.2	56.60	0.0072	104.2
	GENET	180.5	55.54	0.0063	152.1
Path 2	BBR	0.2108	3346	0.0407	-1721
	Cubic	0.2149	6978	0.2206	-4273
	GENET	0.1975	6381	0.0267	-3178
Path 3	BBR	5.40	1581	0.0136	-705.9
	Cubic	6.63	1400	0.0382	-719.1
	GENET	4.91	1180	0.0075	-439.9

Table 7: *Reward breakdown of Figure 16(b) in CC real-world experiments.*