THE UNIVERSITY OF CHICAGO

# AUTOMATIC CURRICULUM GENERATION FOR LEARNING ADAPTATION IN NETWORKING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
ZHENGXU XIA

CHICAGO, ILLINOIS
MARCH 22, 2022

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

# ABSTRACT

As deep reinforcement learning (RL) showcases its strengths in networking and systems, its pitfalls also come to the public's attention—when trained to handle a *wide* range of network workloads and previously *unseen* deployment environments, RL policies often manifest suboptimal performance and poor generalizability.

To tackle these problems, we present GENET, a new training framework for learning better RL-based network adaptation algorithms. GENET is built on the concept of *curriculum learning*, which has proved effective against similar issues in other domains where RL is extensively employed. At a high level, curriculum learning gradually presents more difficult environments to the training, rather than choosing them randomly, so that the current RL model can make meaningful progress in training. However, applying curriculum learning in networking is challenging because it remains unknown how to measure the "difficulty" of a network environment.

Instead of relying on handcrafted heuristics to determine the environment's difficulty level, our insight is to utilize traditional rule-based (non-RL) baselines: If the current RL model performs significantly worse in a network environment than the baselines, then the model's potential to improve when further trained in this environment is substantial. Therefore, GENET automatically searches for the environments where the current model falls significantly behind a traditional baseline scheme and iteratively promotes these environments as the training progresses. Through evaluating GENET on three use cases—adaptive video streaming, congestion control, and load balancing, we show that GENET produces RL policies which outperform both regularly trained RL policies and traditional baselines in each context, not only under synthetic workloads but also in real environments.

# CHAPTER 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) Jay et al. [2019], adaptive-bitrate streaming (ABR) Mao et al. [2017], load balancing (LB) Mao et al. [2019a], wireless resource scheduling Chinchali et al. [2018], and cloud scheduling Mao et al. [2019c]. For a given distribution of training network environments (e.g., network connections with certain bandwidth pattern, 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, there have been many efforts to address these issues, by enhancing offline RL training or re-training 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* Narvekar et al. [2020]. Unlike traditional RL training which 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 Mehta et al. [2020], Akkaya et al. [2019], Dennis et al. [2020] as well as *asymptotic* performance Weinshall et al. [2018], Justesen et al. [2018], 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, as we show in §3, several seemingly natural heuristics to identify rewarding environments suffer from limitations.

- First, they use *innate properties* of each environment (e.g., shorter network or workload traces Mao et al. [2019c] and smoother network conditions Gilad et al. [2019] are supposedly easier), but these innate 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.

| **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 1.1: RL use cases in networked systems. Default reward parameters: $\alpha = -10$ (rebuffer 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 .

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.

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 1.1). For each RL use case, GENET parameterizes the network environment space, allowing us to search for rewarding environments in both

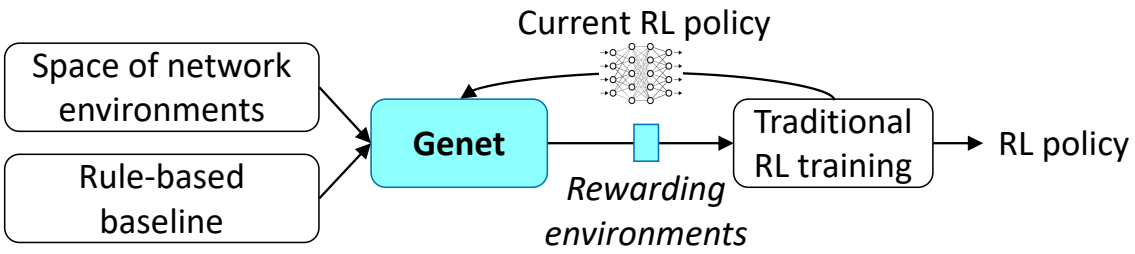



Figure 1.1: GENET creates training curricula by iteratively finding rewarding environments where the current RL policy has high gap-to-baseline.

synthetically instantiated environments and trace-driven environments. GENET also uses Bayesian Optimization to facilitate the search in a large space. 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.[1]

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) pen, congestion control (CC) aur, and load balancing (LB) par.

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 envi-

1. 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.

ronments 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 Raghunathan et al. [2019].

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. We also show that GENET-trained RL policies generalize well to new distributions of network or workload characteristics (different distributions of bandwidth, delay, queue length, etc.).

# CHAPTER 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 Mao et al. [2019a], Haj-Ali et al. [2019].[1] We use three use cases (summarized in Table 1.1) 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 Mao et al. [2017]), choose the next chunk's bitrate (output decision) at the chunk boundary to maximize 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 Jay et al. [2019]) 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 pen for ABR, Aurora aur for CC, and Park par for LB). Our goal is to improve existing RL

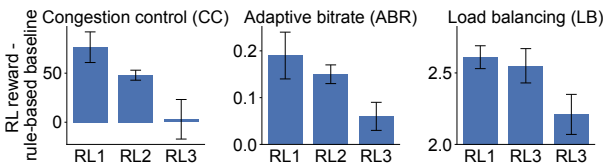
1. There are rule-based alternatives to DNN-based policies, but they are not as expressive and flexible as DNNs, which limits their performance. Oboe Akhtar et al. [2018], for instance, sets optimal hyperparameters for RobustMPC based on the mean and variance of network bandwidth and as shown in §5.4, is a very competitive baseline, but it performs worse than the best RL strategy.

training in networking. Revising the RL algorithm *per se* (input, output, or DNN model) is beyond our scope.

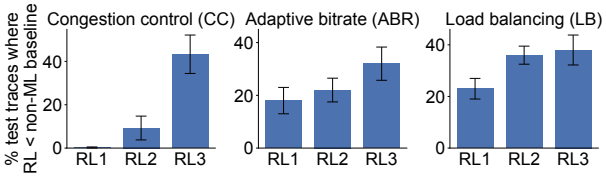
**Network environments:** We generate simulated training environments with a range of parameters, following prior work Mao et al. [2017], Jay et al. [2019], Mao et al. [2019a]. An environment can be synthetically generated using a list of parameters as *configuration*, e.g., in the context of ABR, a 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 §7.2.) In recent papers, both trace-driven (e.g., Mao et al. [2017], Gilad et al. [2019]) and synthetic environments (e.g., Jay et al. [2019], Mao et al. [2019a]) 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 Jay et al. [2019] uses a batch size of 7200 steps (i.e., 30–50 30-second network environments) and applies the PPO1 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 a distribution of target environments, an RL policy can be trained to perform well on these environments (e.g., Mao et al. [2017], Jay et al. [2019]). Unfortunately, this approach falls short on two fronts.

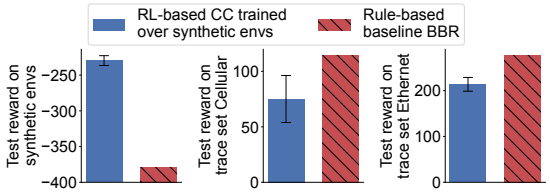



(a) Performance gains of RL schemes over the baselines diminish as the target distribution spans a wide range of environments.

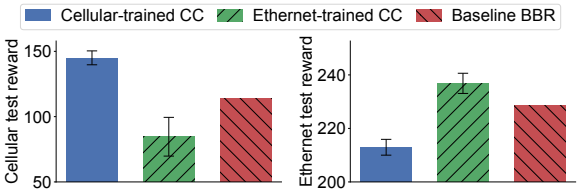



(b) Even if RL schemes perform better on average, they are worse than the baselines on a substantial fraction of test environments.

Figure 2.1: Challenge 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 2.2: Generalization issues of RL-based schemes using CC as an example.

**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 2.1, for each use case, we choose three target distributions (with increasing parameter ranges), labeled RL1/RL2/RL3 ranges of synthetic environment parameters in Table 7.1, 7.2, and 7.3. Figure 2.1(a) compares the asymptotic performance of three RL policies (with different random seeds) with rule-based baselines, MPC Yin et al. [2015] for ABR, BBR Cardwell et al. [2016] 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 2.1(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 Narvekar et al. [2020].

**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 2.2 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 Jay et al. [2019]. We 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 2.2(a); left). Nevertheless, when tested on real-world recorded network traces under the category of "Cellular" and "Ethernet" from Pantheon Yan et al. [2018] (Table 5.1), 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 2.2(b); left), or vice versa (Figure 2.2(b); right). Similarly, its performance degrades significantly when tested on a different trace set.

The observations in Figure 2.2 are not unique to CC. Prior work Gilad et al. [2019] 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 suboptimal, especially when they are trained over a wide range of environments.
- The trained RL policies may generalize poorly to unseen network environments.

# CHAPTER 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.*[1]

**Curriculum learning:** We cast the training of RL-based network adaptation to the well-studied framework of curriculum learning. Unlike the traditional RL training which samples training environments from a fixed distribution in each iteration, curriculum learning gradually increases the difficulty of training environments, so that it always focuses on *training environments that are easier to improve*, i.e., most *rewarding* environments.[2] Prior work has demonstrated the benefits of curriculum learning in other applications of RL, including faster convergence, higher asymptotic performance, and better generalization.

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 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.

1. 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.

2. In deep learning literature, finding the optimal training environments (hypothesized in the seminal paper Bengio et al. [2009] as "not too hard or too easy") in the general setting still remains an open question.

**Why sequencing training environments is difficult:** To motivate our design choices, we first introduce three strawman approaches, each with different strengths and weaknesses. A common strategy in curriculum learning for RL is to measure environment difficulty and gradually introduce more difficult environments to training.

*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 Weinshall et al. [2018]. While this method can distinguish any two environments, it does not hint how to improve the *current* RL model during training.

*Strawman 3: performance of the current RL model.* To fix the second strawman solution, one can use the performance of the current RL policy, instead of a traditional algorithm. If the current RL model performs poorly in an environment, it can potentially improve a lot when trained in this environment (or similar ones). However, this approach may fail since some environments are inherently hard for a model to improve on. In CC, examples of such environments include links with frequently varying bandwidth.

**Example:** Figure 3.1 shows a concrete real example in ABR, where "Strawman 3" fails. (In §5.5, we empirically test these three curriculum-learning strategies.) We generate three sets of bandwidth traces $X$, $Y$, and $Z$ using three configurations (details in §7.3). We first train an RL-based ABR policy on trace set $X$ until it performs well in place (on $X$) but poorly on $Y$ and $Z$. Since the performance of the current RL model is lower on $Y$ than

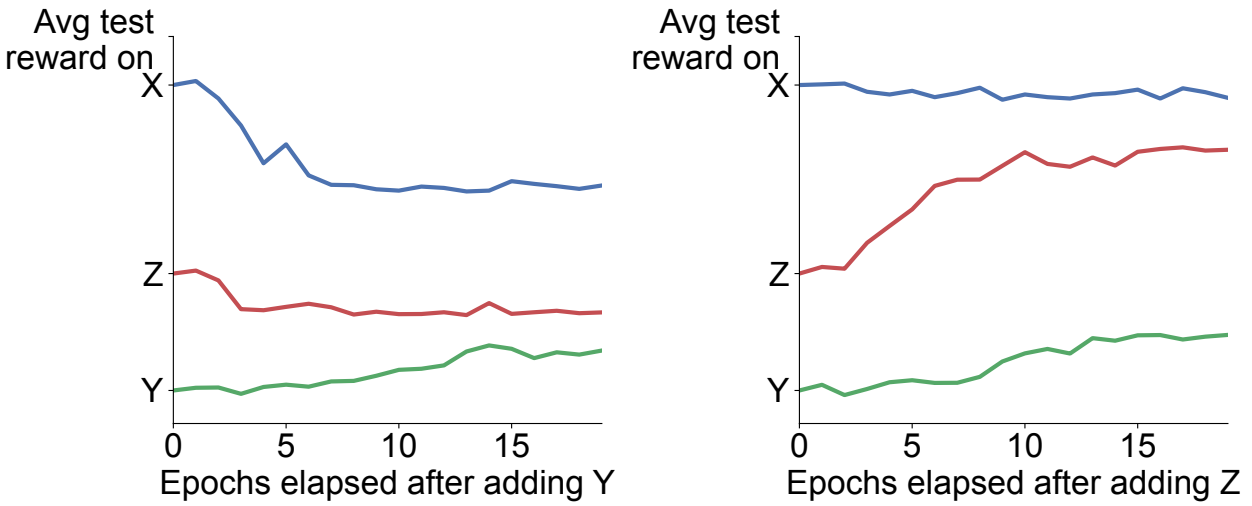
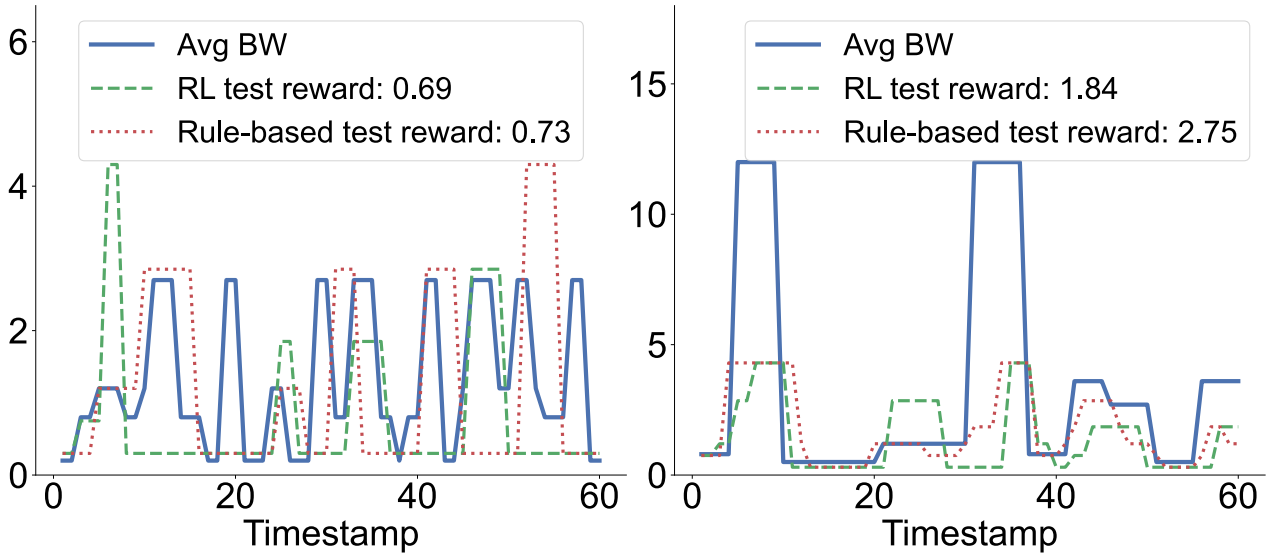



Figure 3.1: A simple example where adding trace set Y to training has a different effect than adding Z. Adding Y to training improves performance on Y only marginally but hurts both X and Z, whereas adding Z improves the performance on both Y and Z without negative impact on X.




(a) Trace in Y (hard)

(b) Trace in Z (improvable)

Figure 3.2: 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).

on $Z$, Strawman 3 opts for adding $Y$ to the training in the next step. However, Figure 3.1 shows that training further on $Y$ worsens the model performance on $X$ and $Z$, although the

in-place performance (on $Y$) is indeed improved. In fact, adding $Z$ to training is better at this point—the performance on $Z$ is improved without negative impact than that on $X$ or $Y$.

To take a closer look, we plot two example traces from $Y$ and $Z$ in Figure 3.2: The trace from $Y$ fluctuates with a smaller magnitude but more frequently, whereas the trace from $Z$ fluctuates with a greater magnitude but much less frequently. However, such observations cannot generalize to an arbitrary pair of environments or a different application.

# CHAPTER 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 training 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[1]. Therefore, a large gap-to-baseline indicates a plausible room for the current RL model to improve. Figure 4.1 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 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 current model's performance ("Strawman 3" in §3) to decide which environments are rewarding.

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-baseline, GENET directly reduces the possibility that the RL policy causes performance regressions.

1. 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 §4.2

In short, 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-baseline) do *not* have particular meanings outside the context of a given pair of RL model and a baseline. For instance, when an RL-based CC model has greater gap-to-baseline in some network environments, it only means that it is easier to improve the RL model by training it in the these environments; it does not indicate if these environments are easy or challenging to any traditional CC algorithm.

## 4.2 Training framework

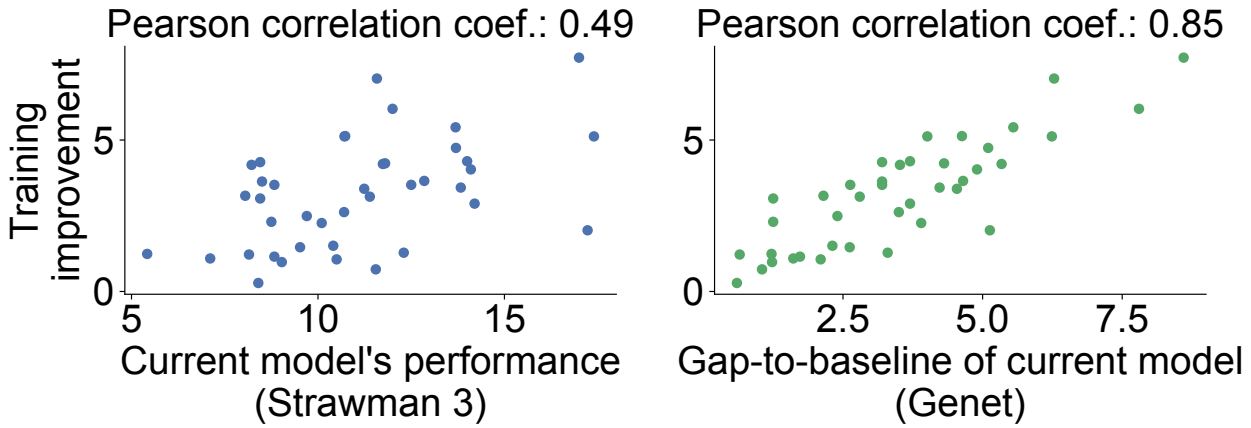
Figure 4.2 depicts GENET's high-level iterative workflow to realize curriculum learning. Each iteration consists of three steps (which will be detailed shortly):

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

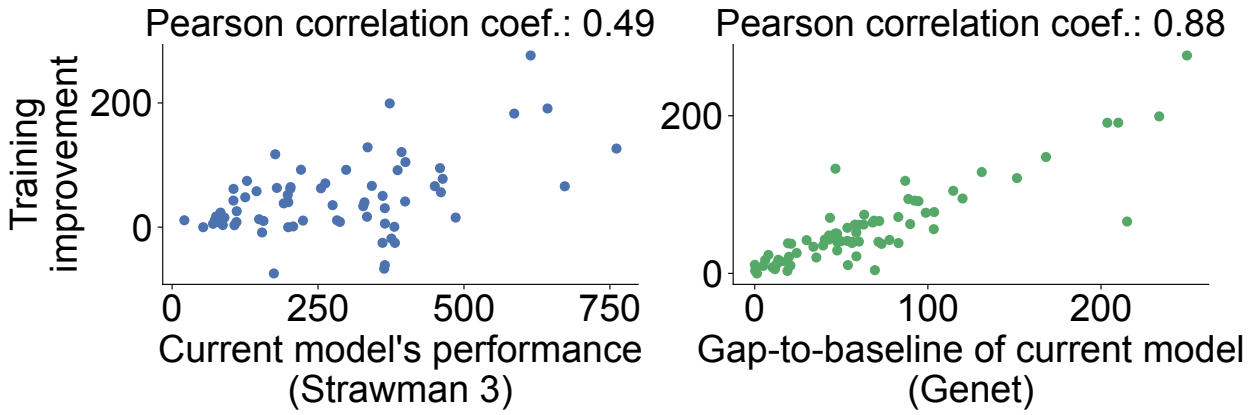
2. Second, we select the environments where the current RL model has a large gap-to-baseline; and

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 5–7




(a) ABR




(b) CC

Figure 4.1: Compared to the current model's performance (left), its gap-to-baseline (right) in an environment is more indicative of the potential training improvement on the environment.

parameters (summarized in Table 7.1, 7.2, 7.3) 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 distribution along 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

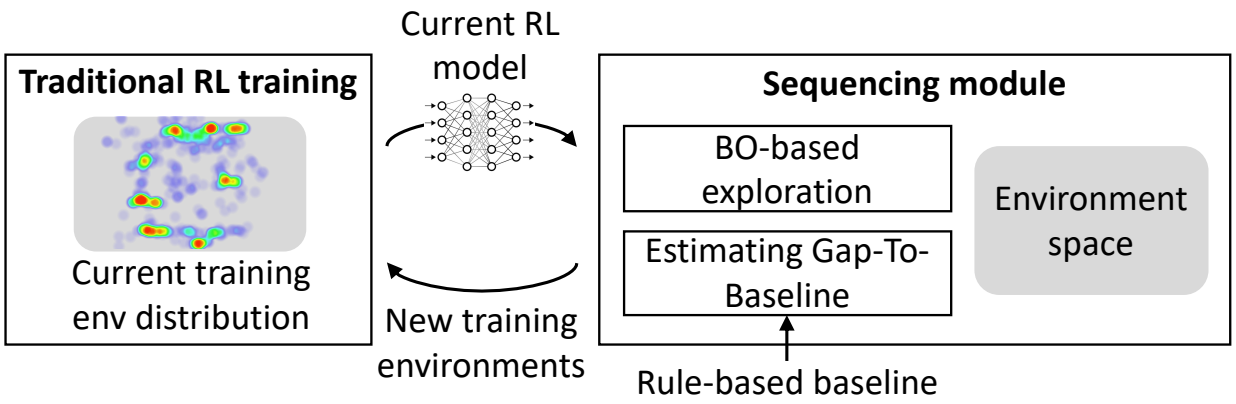



Figure 4.2: Overview of GENET's training process.

training to create new environments, with a probability of $\alpha$ (10% 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 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 over synthetic traces.

**Key components:** Each iteration of GENET starts with training the current model for a fixed number of epochs (defaults to 10). Here, GENET reuses the traditional training method in prior work (i.e., uniform sampling of training environments per epoch), which makes it possible to incrementally apply GENET to existing codebases (see our implementation in §4.3). Recent work on domain randomization Sadeghi and Levine [2016], Tobin et al. [2017a], Peng et al. [2018] also shows that a similar training process can benefit the generalization of RL policies Sadeghi and Levine [2016], Tobin et al. [2017a], Peng et al. [2018]. The details of the training process are described in Algorithm 1.

After a certain number of epochs, 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* Frazier [2018] (BO) as follows. We view the expected gap-to-baseline over the environments created by configuration $p$ as a function of $p$: $Gap(p) = R(\pi^{rule}, p) - R(\pi^{rl}_{\theta}, p)$, where $R(\pi, p)$ is the average reward of a policy $\pi$ (either the rule-based baseline $\pi^{rule}$ or the RL model $\pi^{rl}_{\theta}$) over 10 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 have a significant impact

on the effectiveness of GENET, although better choice tends to yield more improvement (as shown in §5.5).[2]

*Why is BO-based exploration effective?* Admittedly, it can be challenging for BO to search for the rewarding environments in a high-dimensional space. In practice, however, we observe that BO is highly efficient at identifying a good configuration within a relatively small number of steps (15 by default). We empirically validate it in §5.5.

*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%.[3]

## 4.3 Implementation

GENET is fully implemented in Python and Bash, and has been integrated with three existing RL training code. 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 code with two APIs (Figure 4.3): `Train` signals the RL to continue the training using the given distribution of environment configurations and returns a snapshot of model after a specified number of training epochs;

2. 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.

3. When we impose a minimum fraction of "exploration" (i.e., uniformly randomly pick an environment from the original training distribution) in the training (which is a typical strategy to prevent forgetting Zaremba and Sutskever [2014]), GENET's performance actually becomes worse.

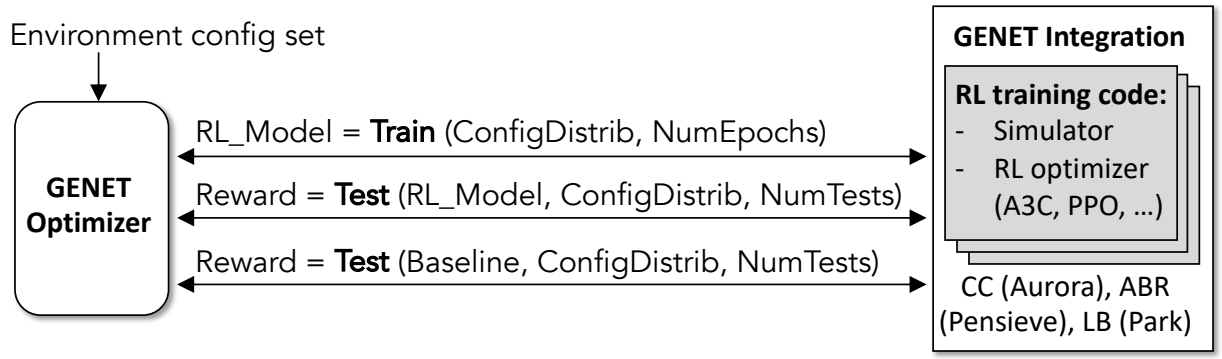



Figure 4.3: Components and interfaces needed to integrate GENET with an existing RL training code.

`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 pen, Aurora CC aur, and Park LB par, 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.

# CHAPTER 5

# EVALUATION

The key takeaways of our evaluation are:

- 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 to outperform various rule-based baselines specified during GENET-based RL training (§5.4).
- Finally, the design choices of GENET, such as its curriculum learning strategy, 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, but by showing them for the first time in facilitating 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) aur, adaptive-bitrate streaming (ABR) pen, and load balancing (LB) par. 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 §7.2 and Table 7.1,7.2,7.3. We choose these environment parameters to

| 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 5.1: Network traces used in ABR and CC tests.

cover a range 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, and queue length, etc.

**Trace-driven environments:** We also use real traces for CC and ABR (summarized in Table 5.1) to create trace-driven environments (in both training and testing), where the bandwidth timeseries 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 Commission (labeled "FCC") and 310 cellular traces Riiser et al. [2013] (labeled "Norway"). We test CC policies on 121 cellular traces (labeled "Cellular") and 112 ethernet traces (labeled "Ethernet") collected by the Pantheon platform Yan et al. [2018].

All used real traces are released in `https://github.com/GenetProject/Genet`.

**Baselines:** We compare GENET-trained RL policies with several baselines.

First, *traditional RL* trains RL policies by uniformly sampling environments from the target distribution per epoch. We train three types of RL policies (RL1, RL2, RL3) over fixed-width uniform distribution of synthetic environments, specified in Table 7.1, 7.2 7.3. From RL1 to RL3, the sizes of their training environment ranges are in ascending order.

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 Gilad

et al. [2019] which generates new training bandwidth traces that maximize the gap between the RL policy and optimal adaptation with an unsmoothness penalty (§5.5).

Second, *traditional rule-based algorithms* include BBA Huang et al. [2014] and RobustMPC Yin et al. [2015] for ABR, PCC-Vivace Dong et al. [2018], BBR Cardwell et al. [2016] and CUBIC for CC, and least-load-first (LLF) for LB.[1] They can be viewed as a reference point of traditional non-ML solutions.

## 5.2 Asymptotic performance

We first compare GENET-trained RL policies and traditional RL-trained 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 its full range) as the target distribution. As shown in Figure 2.1, in these training ranges, traditional RL training yields little performance improvement over the rule-based baselines. Figure 5.1 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 consistently GENET 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

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

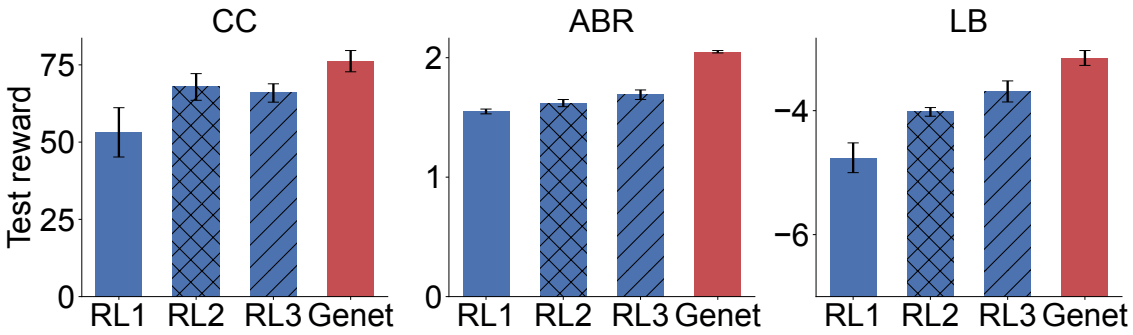



Figure 5.1: Comparing performance of GENET-trained RL policies for CC, ABR, and LB, with baselines in unseen synthetic environments drawn from the training distribution, which sets all environment parameters at their full ranges.

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 5.2 picks ABR as an example and shows the performance across different values along six environment parameters. We vary one parameter at a time while fixing other parameters at the same default values (see Table 7.1, 7.2, 7.3). We see that GENET-trained RL policies enjoy consistent performance advantages (in reward) over the RL policies trained by traditional RL-trained models. This suggests that the improvement of GENET shown in Figure 5.1 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.

We also run CC emulation on a Dell Inspiron 5521 machine with a Mahimahi-emulated link with controlled bandwidth, delay, and queue length. We run LB emulation on a local Cassandra testbed with three machines hosted in the Chameleon cluster cloud server fed with key-value requests at Poisson arrival intervals. Figure 5.3 shows in synthetic environments, 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

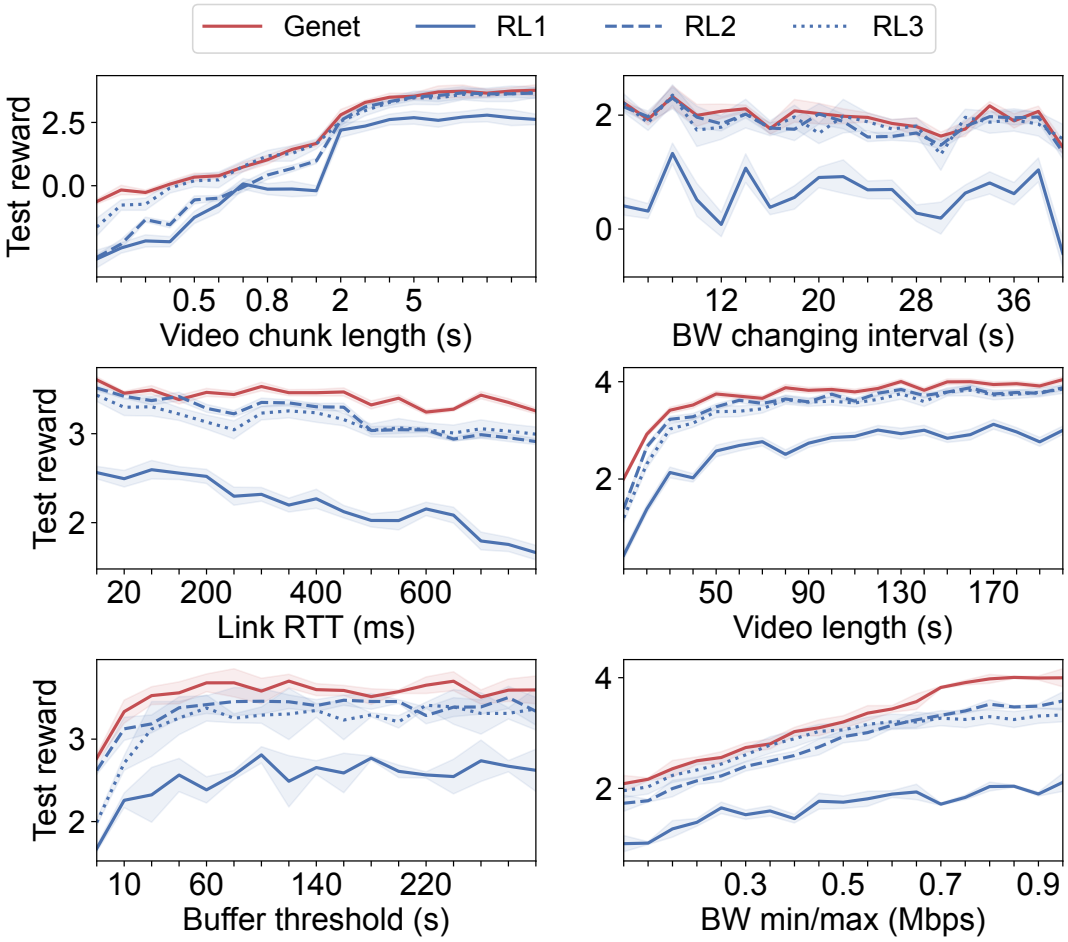



Figure 5.2: Test of ABR policies along individual environment parameters.

Table 5.1. GENET trains ABR and CC policies by combining the trace-driven environments and the 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 5.4 tests GENET-trained ABR and CC policies with their respective traditional RL-trained base-

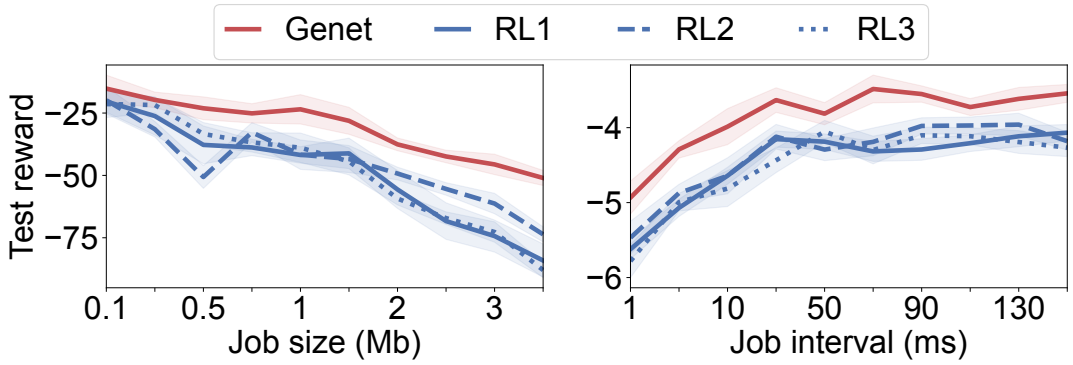
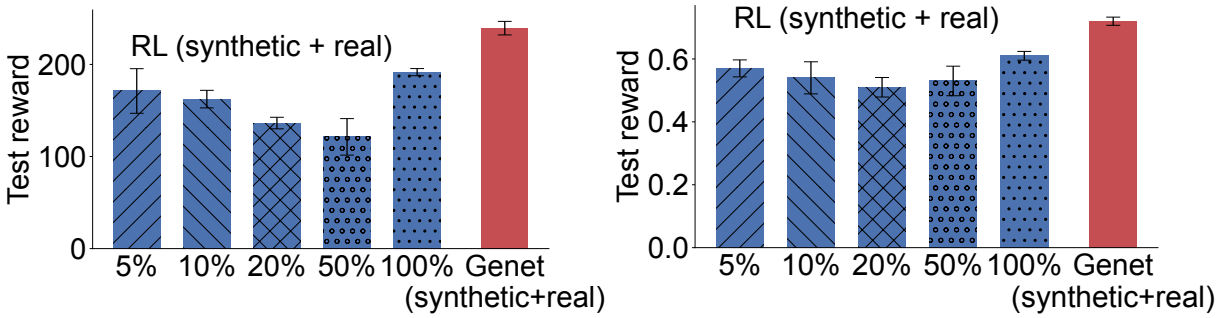



Figure 5.3: Test of LB policies along individual environment parameters.




(a) Congestion control (CC)

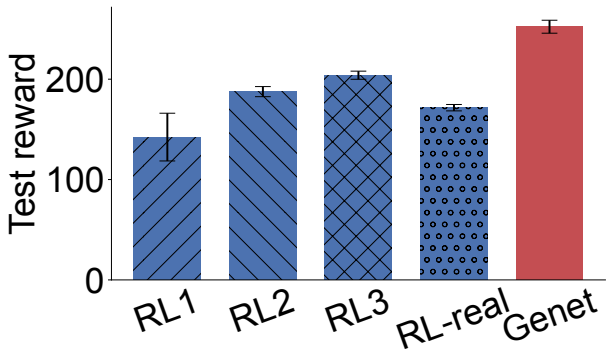
(b) Adaptive bitrate (ABR)

Figure 5.4: Asymptotic performance of GENET-trained CC policies (a) and ABR policies (b) and baselines, when the real network traces are randomly split to training set and test set.

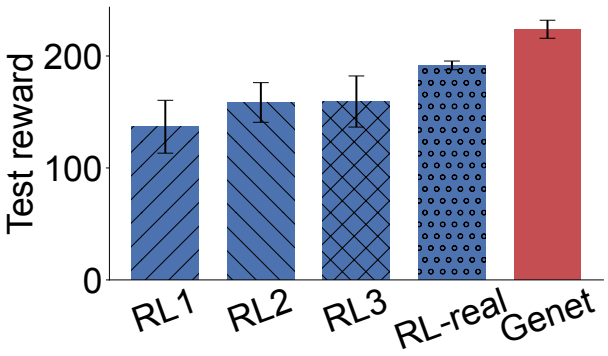
lines over new environments generated from the traces in the testing set. Figure 5.4 shows that GENET-trained policies outperform traditional RL training by 17-18%, regardless of the ratio of real traces, including when training the model entirely on real traces.

## 5.3 Generalization

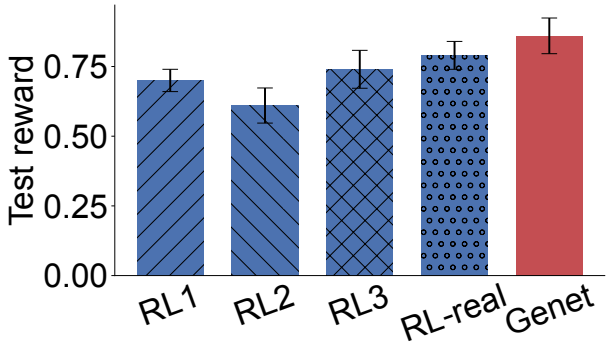
Next, we take the RL policies of ABR and CC trained (by GENET and other baselines) entirely over synthetic environments (the RL3 synthetic environment range) and test their generalization in trace-driven environments generated by the ABR (and CC) testing traces in Table 5.1.



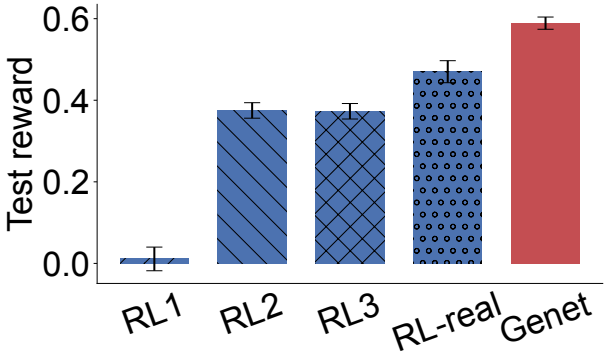

(a) CC test in trace-driven environments (Cellular)




(b) CC test in trace-driven environments (Ethernet)



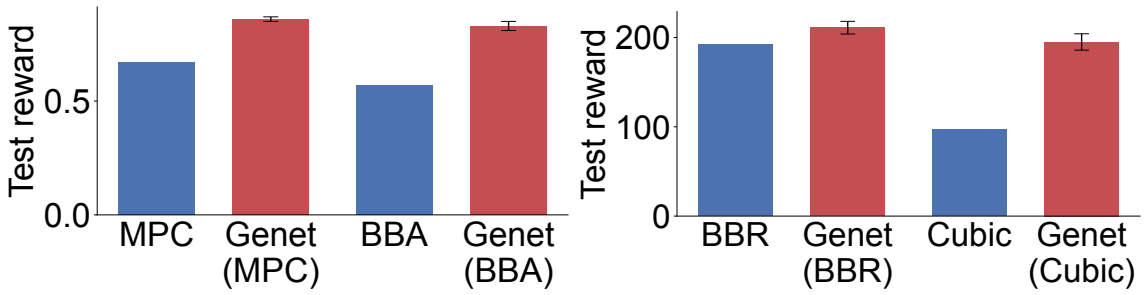

(c) ABR test in trace-driven environments (FCC)




(d) ABR test in trace-driven environments (Norway)

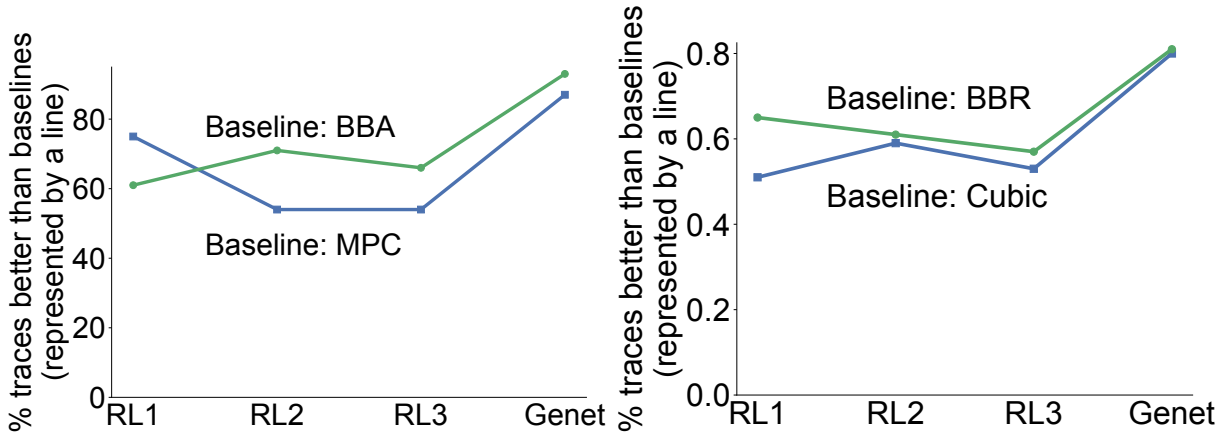
Figure 5.5: Generalization test: Training of various methods is done entirely in synthetic environments, but the testing is over various real network trace sets.

Figure 5.5 shows that they perform better than traditional RL baselines trained over the same synthetic environment distribution. Though Figure 5.5 uses the same testing environments as Figure 5.4 and has a similar relative ranking between GENET and traditional RL training, the implications are different: Figure 5.4 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 5.5 shows their performance when the training real traces are actively used in training of GENET and the baselines.




(a) ABR (b) CC

Figure 5.6: GENET can outperform the rule-based baselines used in its training.



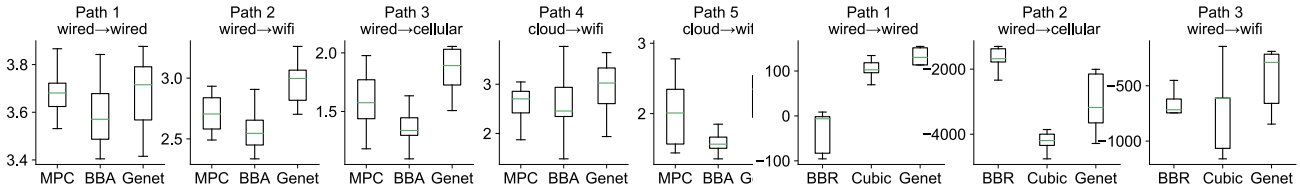

(a) ABR (b) CC

Figure 5.7: Fraction of real traces where GENET trained policies (and traditional RL) are better than the rule-based baseline.

## 5.4 Comparison with rule-based baselines

**Impact of the choice of rule-based baselines:** Figure 5.6 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




(a) ABR (b) CC

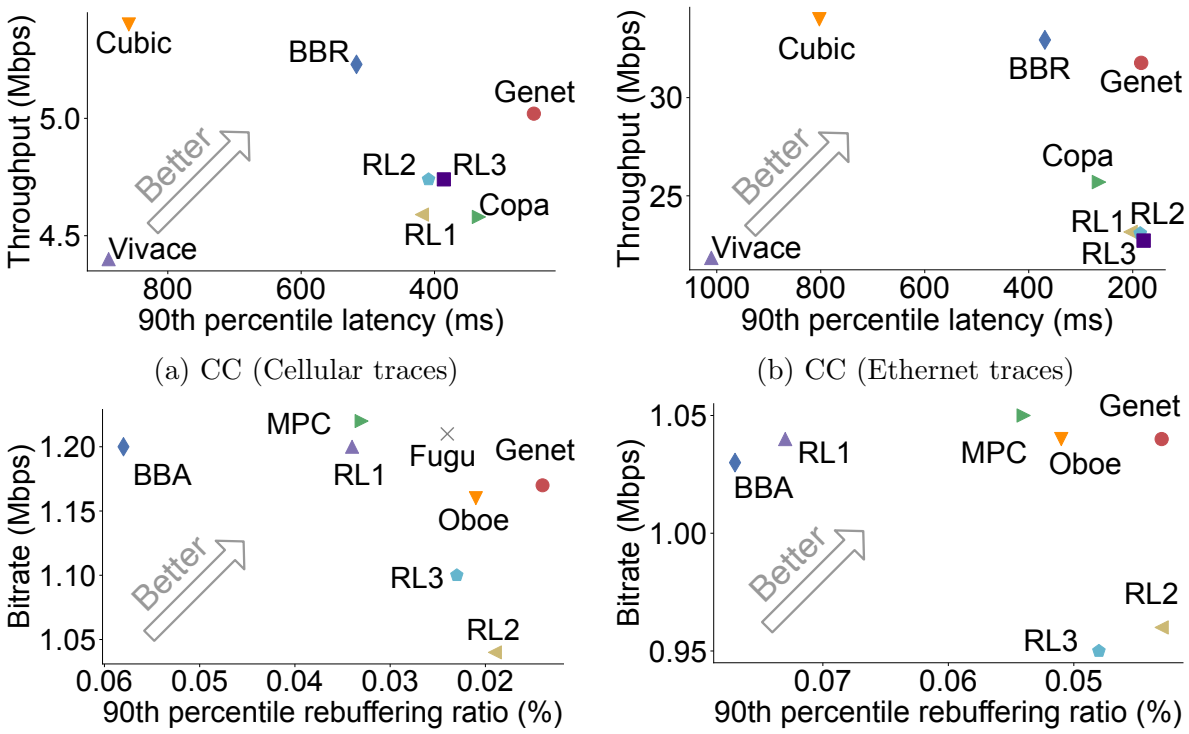
Figure 5.8: Testing ABR and CC policies in real-world environments.

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 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 Genet outperforms 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 5.7, 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 5.9 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 individual performance metrics. We see that the Genet-trained ABR and CC policies stay




(a) CC (Cellular traces)

(b) CC (Ethernet traces)

(c) ABR (FCC traces)

(d) ABR (Norway traces)

Figure 5.9: RL-based ABR and CC vs. rule-based baselines.

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 onl, one laptop at home, and two cloud servers (§7.4), 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 with at least five times, and show their performance in Figure 5.8. 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

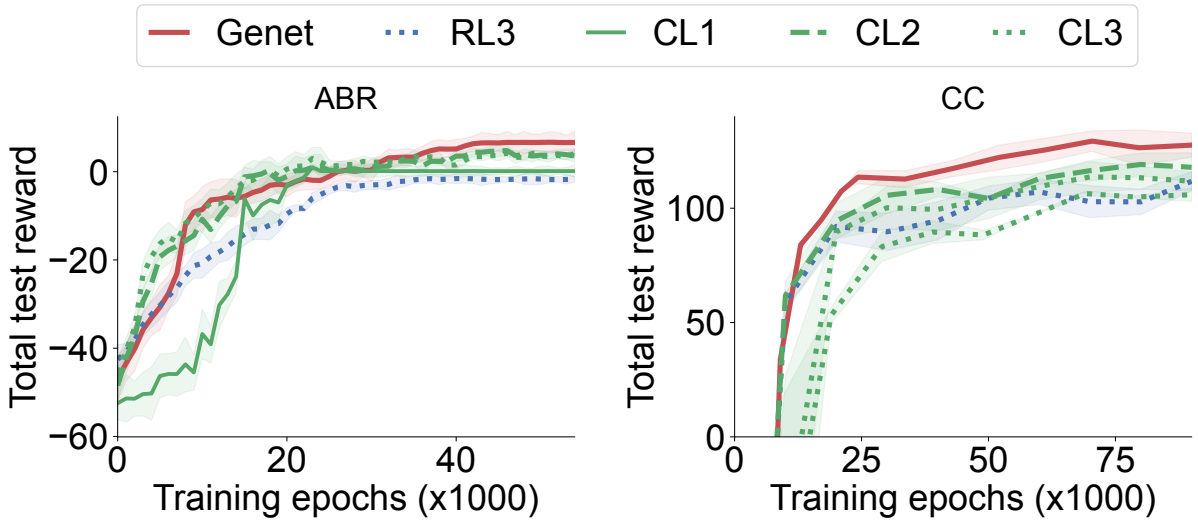



Figure 5.10: Genet's training ramps up faster than better than alternative curriculum learning strategies.

results do not prove 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 5.10 compares Genet's training curve with that of traditional RL training and three alternatives of 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 performs badly (whereas Genet picks the traces where the current RL model is much worse than a rule-based baseline). Compared to these baselines, In Figure 5.10, we show that Genet's training curves have faster ramp-ups, suggesting that with the same number of training epochs, Genet can arrive at a much better policy, which corroborates the reasoning in §3.

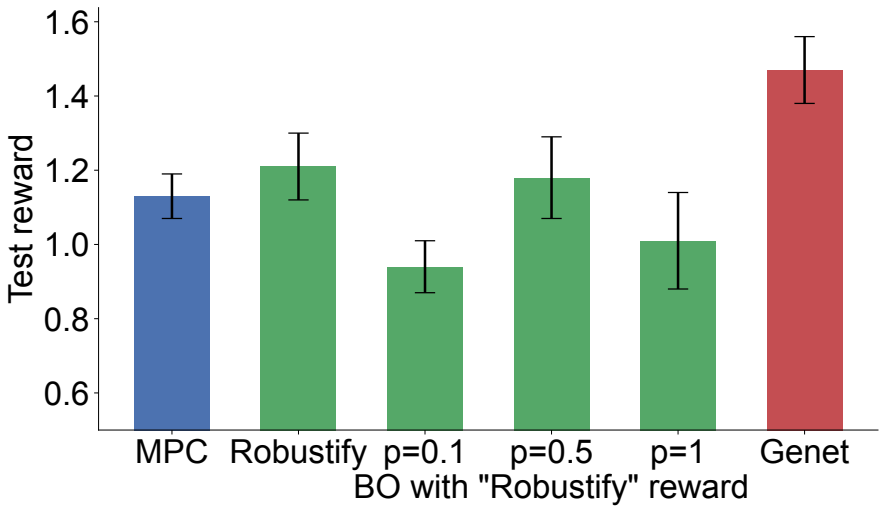
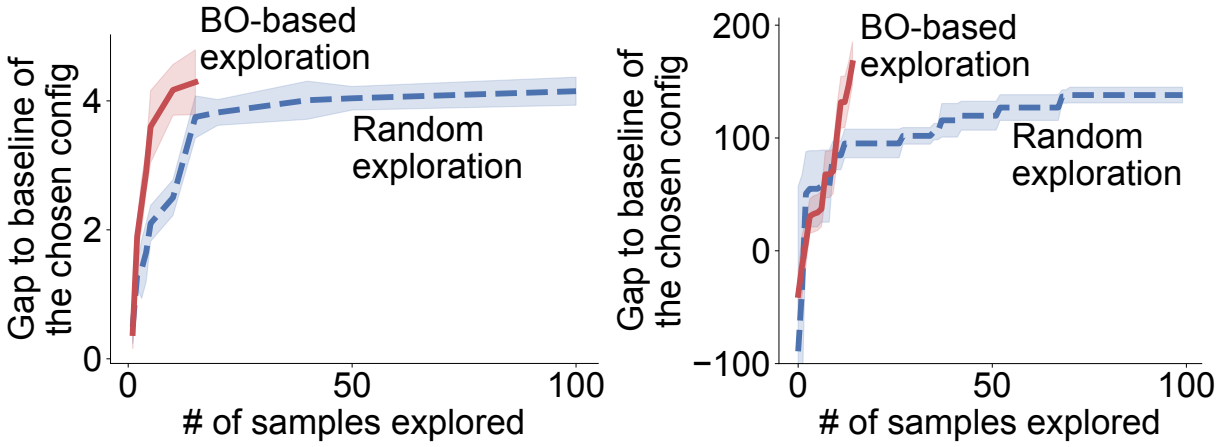



Figure 5.11: GENET outperforms Robustifying Gilad et al. [2019] which improves RL performance by generating adversarial bandwidth traces, and variants of GENET which uses the Robustifying's criteria in BO-based environment selection.

In addition, "Robustifying" Gilad et al. [2019][2] (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 unsmoothness with different weights of $p$). Figure 5.11 compares the resulting RL policies with GENET-trained RL policy and MPC as a baseline on the synthetic traces in Figure 5.2. 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 unsmoothness metric used in Gilad et al. [2019] may not completely capture the inherent difficulty of bandwidth traces (Figure 3.2 shows a concrete example).

**BO-based search efficiency:** GENET uses BO to explore the multi-dimensional environ-

2. In lack of a public implementation, we follow the description in Gilad et al. [2019] (e.g., unsmoothness 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 7.6.




(a) ABR (b) CC

Figure 5.12: BO-based search is more efficient at finding environments with high gap-to-baseline than random exploration in the environment configuration space.

ment space environment to find the environment configuration with a high gap-to-baseline. While BO may not identify the single optimal point in arbitrarily complex relationships between environment parameters and gap-to-baseline, we found it to be a highly pragmatic solution, within a small number of steps (by default, 15), it can identify a configuration that is almost as good as the randomly searching for many more points. To show it, we randomly choose an intermediate RL models during the GENET training of ABR and CC. Figure 5.12 shows the gap-to-baseline of the configuration selected by BO for each model within 15 search steps, and it compares the values with the maximum gap-to-baseline of 100 randomly selected points (which represents a much more expensive alternative).

# CHAPTER 6

# RELATED WORK

**Improving RL for networking:** Some of our findings regarding the lack of generalization corroborate those in previous work Winstein and Balakrishnan [2013], Mao et al. [2017], Jay et al. [2019], Gilad et al. [2019], Rotman et al. [2020b], Dethise et al. [2019]. To improve RL for networking use cases, prior work has attempted to apply and customize techniques from the ML literature. For instance, Gilad et al. [2019] applies adversarial learning by generating relatively smooth bandwidth traces that maximize the RL regret w.r.t. optimal outcomes, Eliyahu et al. [2021], Kazak et al. [2019] show that the generalization of RL can be improved by incorporating training environments where a given RL policy violates pre-defined safety conditions, Schaarschmidt et al. [2019], Schaarschmidt [2020] incorporate randomization in the evaluation of RL-based systems, and Fugu Yan et al. [2020] 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 Mao et al. [2019b], Rotman et al. [2020b], Shi et al. [2021]. 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.

**Curriculum learning for RL:** There is a substantial literature on improving deep RL with curricula (Narvekar et al. [2020], Hacohen and Weinshall [2019], Portelas et al. [2020] 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 Silva and Costa [2018], Schmidhuber [2013], how to sequence tasks Ren et al. [2018], Sukhbaatar et al. [2017], 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 Kumar et al. [2010] advocates the use of easier training examples first, while the other approaches prefer to use harder examples first Chang et al. [2017]. Recent work tries to bridge the gap by suggesting that an ideal next training task should be difficult with respect to the current model's hypothesis, while it is also beneficial to prefer easier points with respect to the target hypothesis Hacohen and Weinshall [2019]. 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 Bengio et al. [2009], which hypothesizes that "it would be beneficial to make learning focus on 'interesting' examples that are neither too hard or 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 Peng et al. [2018]. Several schemes boost RL's training efficiency by iteratively creating a curriculum of challenging training environments (e.g., Dennis et al. [2020], Mehta et al. [2020]) 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 Dennis et al. [2020] to approximate the optimal policy or a metric Mehta et al. [2020] 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 Ghavamzadeh et al. [2016], Laroche et al. [2019], 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.

# CHAPTER 7

# APPENDIX

## 7.1 Details of RL implementation

The input of RL algorithm consists of a space of configurations, an initial policy parameters and predefined total number of epochs 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 epoch, 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*N environments to update the policy. When the policy is updated for the predefined number of epochs, the RL algorithm stops training and outputs a trained policy.

**Algorithm 1** Traditional Reinforecment Learning (RL)

**Input:** $\Omega$: space of configurations, $\theta$: initial policy parameters, $N_{epochs}$: # of epochs
**Output:** $\theta$: returned policy parameters

**for** $i$ from 1 to $N_{epochs}$ **do**
  $\Phi_{rand} \leftarrow \emptyset$
  **for** 1 **to** $K$ **do** ▷ $K$: # configs per epoch
    $p_i \sim Random(\Omega)$ ▷ Uniformly sampled config in $\Omega$
    **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 $\pi_\theta$ on $E$
      $\Phi_{rand} \leftarrow \Phi_{rand} \cup \phi$
    **end for**
  **end for**
  **with** $\Phi_{rand}$ update:
  $\theta \leftarrow \theta + \nu \bigtriangledown_\theta J(\pi_\theta)$ ▷ Gradient update with rate $\nu$
**end for**
**return** $\theta$

**Algorithm 2** GENET training framework

**Input:** $\Omega$: uniform configuration distribution (equal probability on each configuration), $\pi^{rule}$: rule-based policy.
**Output:** $\theta$: final RL policy parameters

**function** GENET($\Omega, \pi^{rule}$)
  $\theta \leftarrow$ Random initial policy parameters
  $\Omega_{cur} \leftarrow \Omega$ ▷ $\Omega_{cur}$ will be updated and used for training
  **for** from 1 to $N_{iter}$ **do** ▷ # of exploration iterations
    BO.INITIALIZE($\Omega$) ▷ Initialize with full config space $\Omega$
    **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($\Omega_{cur}, \theta, N_{epochs}$)
  **end for**
  **return** $\theta$
**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 $\pi^{rl}$
    **rollout** $\phi^{rule} \sim \pi^{rule}(\cdot; E)$ ▷ Rollout rule-based $\pi^{rule}$
    add $Reward(\phi^{rule}) - Reward(\phi^{rl})$ to $samples$
  **end for**
  **return** MEAN($samples$)
**end function**

## 7.2 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 does 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 drew 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 par project, where jobs arrive according to a Poisson process, and the job sizes follow a Pareto distribution with parameters [shape, scale]. In the simulation, all servers process jobs from their queues at identical rates.

## 7.3 Sequencing training trace adding example

Trace sets in Figure 3.1 was generated by three configurations. For trace set X, we used BW range: 0-4Mbps, BW changing frequency: 4-10s. For trace set Y, we used BW range: 0-1Mbps, BW changing frequency: 0-2s. For trace set Z, we used BW range: 0-3Mbps, BW changing frequency: 2-15s. As a motivation example, each trace set contains 20 traces to show the testing reward trend.

## 7.4 Testbed setup

**ABR:** To test our model on a client-side system, we first leverage the testbed from Pensieve pen, which modifies dash.js (version 2.4) [2] 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 Netravali et al. [2015] to emulate the network conditions from our pre-recorded FCC Mao, cellular Riiser et al. [2013], Puffer Yan et al. [2020] network traces, along with an 80 ms RTT, between the client and server. All above experiments are performed on UChicago servers.

To compare with Fugu, we then modify the interface to connect GENET trained model with Puffer's system Yan et al. [2020]. This experiment was performed on Azure Virtual Machines.

**CC:** We build up CC testbed on Pantheon Yan et al. [2018] platform. Pantheon uses network emulator Mahimahi Netravali et al. [2015] 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 delopying pantheon platform on the nodes shown in Figure 7.1. Among all the CC algorithms tested, BBR Cardwell et al. [2016] and TCP Cubic Ha et al. [2008] are provided by Linux kernel and are called via iperf3. PCC-Aurora Jay et al. [2019] and PCC-Vivace Dong et al. [2018] are implemented on top of UDP. We train our models in python and Tensorflow framework and port the models into the Aurora C++ code.

**LB:** We implement the testbed within Cassandra (distributed database), and use different scheduling (GENET trained, LLF) policies to select the replica. We modify Cassandra's internal read-request routing mechanism (originally every Cassandra node is both a client and server), this means that one of Cassandra nodes is a client and three others are servers. We generate each key size as 100B and value size as 1KB, which served as the dataset. In the Cassandra testbed, each request job size follows the same Pareto distribution as in the simulator, which is replicated to three replicas (three Emulab nodes), so each replica has a

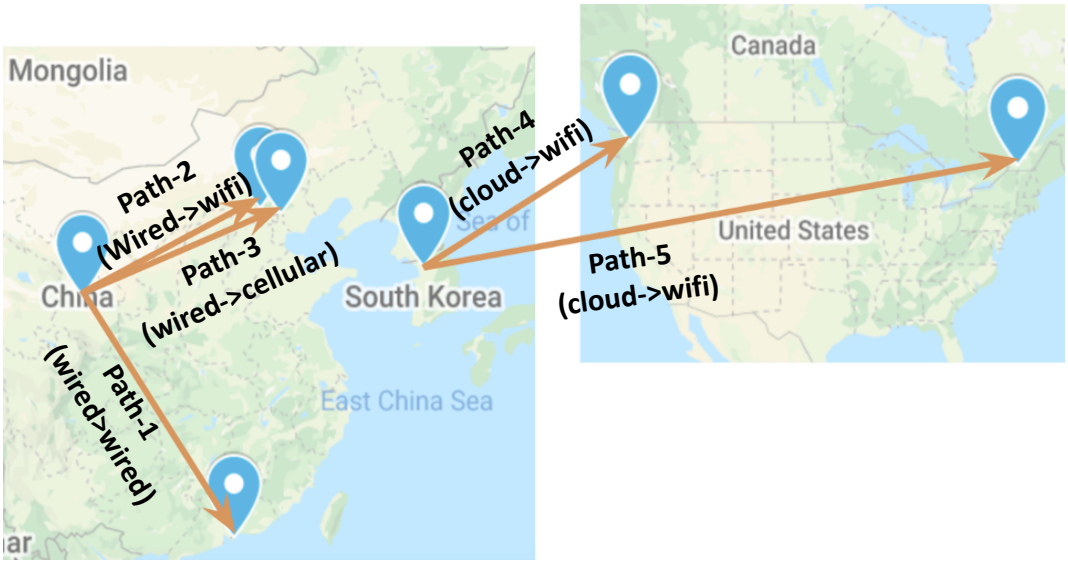



Figure 7.1: Real-world network paths used to test ABR and CC policies.

copy of each key. Each experiment involves 10000 operations of the workload and is repeated 10 times. All experiments were performed on Chameleon servers cloud server.

**Real network testbed:** We also test the GENET-trained ABR and CC policies in real wide-area network paths (depicted in Figure 7.1), including four nodes reserved from onl, one laptop at home, and two cloud servers.

## 7.5 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 second, 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 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

| ABR Parameter | RL1 | RL2 | RL3 | Default | Original |
|---|---|---|---|---|---|
| Max playback buffer (s) | [5, 10] | [5, 105] | [5, 500] | 60 | 60 |
| Total video length (s) | [40, 45] | [40, 240] | [40, 800] | 200 | 196 |
| Video chunk length (s) | [1, 6] | [1, 11] | [1, 100] | 4 | 4 |
| Min link RTT (ms) | [20, 30] | [20, 220] | [20, 1000] | 80 | 80 |
| Bandwidth change frequency (s) | [0, 2] | [0, 20] | [0, 100] | 5 | |
| Max link bandwidth (Mbps) | [0, 5] | [0, 20] | [0, 100] | 5 | |
| Min link bandwidth (Mbps) | [0, 0.7×max] | [0, 0.4×max] | [0, 0.1×max] | 0.1 | |

Table 7.1: 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 §7.2.

| 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 7.2: 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 §7.2. 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.

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 being assigned, the server would summarize and update the delay of all active jobs.

## 7.6 Baseline implementation

According to the paperGilad et al. [2019], 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

| LB Parameter | RL1 | RL2 | RL3 | Default | Original |
|---|---|---|---|---|---|
| Service rate | [0.1, 2] | [0.1, 10] | [0.1, 100] | 1.5 | [2, 4] |
| Job size (byte) | [1, 100] | [1, $10^4$] | [1, $10^6$] | $10^4$ | [100, 1000] |
| Job interval (ms) | [0.1, 10] | [0.1, 100] | [0.1, 1000] | 100 | 100 |
| Number of jobs | [1, 100] | [1, $10^4$] | [1, $10^6$] | 1000 | 1000 |
| Queue shuffled frequency (episodes) | [1, 10] | [1, 100] | [1, 1000] | 100 | |
| Queue shuffled probability | [0.1, 0.2] | [0.1, 0.5] | [0.1, 1] | 0.5 | |

Table 7.3: 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 §7.2.

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.

## 7.7 BO search behavior

Figure 7.2 projects an example trajectory of configurations chosen by BO on a 2-D configuration space ("max link bandwidth" and "bandwidth change interval" in Table 7.1). It

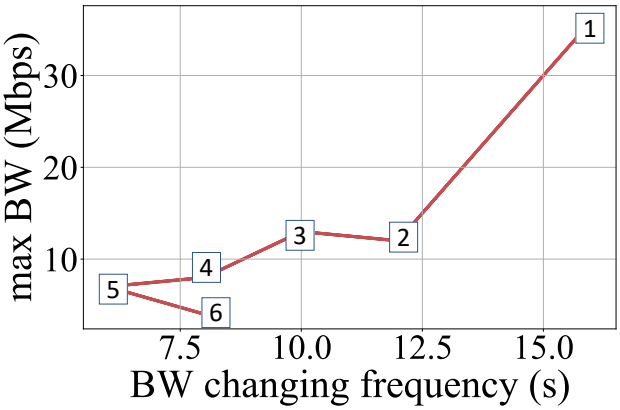



Figure 7.2: Exploration by GENET's Bayesian Optimization in a 2-D configuration space.

starts with an easy configuration (a large maximum bandwidth value and low bandwidth change frequency). After that, BO gradually lowers the bandwidth value while increasing the bandwidth change frequency, effectively raising the difficulty of the chosen configuration each time. In other words, GENET *automatically* chooses a sequence of environments with "emergent complexity," a desirable behavior of RL training Dennis et al. [2020].

# BIBLIOGRAPHY


Aurora implementation. https://github.com/PCCproject/PCC-RL.

UChicago Computer Science Department BPC Website. https://www.cs.uchicago.edu/diversity/#bpcplan, a.

UChicago CS BPC Plan. https://drive.google.com/file/d/1YG4RPV1Oflq8p1gxlzusoxxBjX566TQh/view, b.

DDS streaming pipeline codebase. https://github.com/KuntaiDu/dds.

The Fisk-Vanderbilt Master's to PhD Bridge Program. https://www.fisk-vanderbilt-bridge.org/.

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. https://github.com/openai/gym.

workshop on ml for systems at neurips 2018, a.

Workshop on ML for Systems at NeurIPS 2019. http://mlforsystems.org/neurips2019/, b.

Workshop on ml for systems at isca 2019, c.

workshop on ml for systems at neurips 2020, d.

OpenNetLab. https://opennetlab.org/.

Park implementation. https://github.com/park-project/park.

Pensieve implementation. https://github.com/hongzimao/pensieve.

Yoda: Video analytics workload benchmark for performance clarity. https://github.com/zxxia/benchmarking, 2021.

Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

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.

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.

OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

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.

Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Nikolaos Karianakis, Yuanchao Shu, Kevin Hsieh, Victor Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. *arXiv preprint arXiv:2012.10557*, 2020.

Konstantinos Bousmalis, Nathan Silberman, David Dohan, Dumitru Erhan, and Dilip Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3722–3731, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

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.

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.

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.

Chameleon cloud server. Chameleon cloud server. `https://www.chameleoncloud.org/`.

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, pages 1282–1289. PMLR, 2019.

Federal Communications Commission. Measuring Broadband America. `https://www.fcc.gov/general/measuring-broadband-america`.

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.

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.

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.

Kurt Driessens and Sašo Džeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304, 2004.

Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2020.

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.

Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

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.

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.

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.

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.

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.

Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *International Conference on Machine Learning*, pages 1480–1490. PMLR, 2017.

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.

Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019.

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.

Junchen Jiang. *Enabling Data-Driven Optimization of Quality of Experience in Internet Applications*. PhD thesis, Carnegie Mellon University, 2017.

Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, and Hui Zhang. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 286–299, 2016a.

Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. Cfa: A practical prediction system for video qoe optimization. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 137–150, 2016b.

Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 393–406, 2017.

Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A Chien. Networked cameras are the new big data clusters. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 1–7, 2019.

Nan Jiang and Lihong Li. Doubly robust off-policy value evaluation for reinforcement learning. In *International Conference on Machine Learning*, pages 652–661. PMLR, 2016.

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.

Daniel Kahneman, Jack L Knetsch, and Richard H Thaler. Anomalies: The endowment effect, loss aversion, and status quo bias. *Journal of Economic perspectives*, 5(1):193–206, 1991.

Nikhil Kansal. *Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On-the-Fly*. PhD thesis, UCLA, 2019.

Nikhil Kansal, Murali Ramanujam, and Ravi Netravali. Alohamora: Reviving http/2 push and preload by adapting policies on the fly. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 269–287, 2021.

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.

Rawal Khirodkar, Donghyun Yoo, and Kris M Kitani. Vadra: Visual adversarial domain randomization and augmentation. *arXiv preprint arXiv:1812.00491*, 2018.

M Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *NIPS*, volume 1, page 2, 2010.

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.

Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 178–184, 2017.

Qiang Liu, Lihong Li, Ziyang Tang, and Dengyong Zhou. Breaking the curse of horizon: Infinite-horizon off-policy estimation. *arXiv preprint arXiv:1810.12429*, 2018.

Mohammadhossein Malmir, Josip Josifovski, Noah Klarmann, and Alois Knoll. Robust sim2real transfer by learning inverse dynamics of simulated systems. In *2nd Workshop on Closing the Reality Gap in Sim2Real Transfer for Robotics*, 2020.

Hongzi Mao. Pensieve collected data. `https://www.dropbox.com/sh/ss0zs1lc4cklu3u/AAB-8WC3cHD4PTtYT0E4M19Ja?dl=0`.

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.

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. 2019a.

Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. Towards safe online reinforcement learning in computer systems. In *NeurIPS Machine Learning for Systems Workshop*, 2019b.

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. 2019c.

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.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 311–324, 2015.

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.

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.

Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

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.

Lerrel Pinto, James Davidson, and Abhinav Gupta. Supervision via competition: Robot adversaries for learning tasks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1601–1608. IEEE, 2017.

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.

Aditi Raghunathan, Sang Michael Xie, Fanny Yang, John C Duchi, and Percy Liang. Adversarial training can hurt generalization. *arXiv preprint arXiv:1906.06032*, 2019.

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.

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.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P Lillicrap, and Greg Wayne. Experience replay for continual learning. *arXiv preprint arXiv:1811.11682*, 2018.

Noga H Rotman, Michael Schapira, and Aviv Tamar. Online safety assurance for learning-augmented systems. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 88–95, 2020a.

Noga H Rotman, Michael Schapira, and Aviv Tamar. Online safety assurance for deep reinforcement learning. *arXiv preprint arXiv:2010.03625*, 2020b.

Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

Michael Schaarschmidt. End-to-end deep reinforcement learning in computer systems. Technical report, University of Cambridge, Computer Laboratory, 2020.

Michael Schaarschmidt, Kai Fricke, and Eiko Yoneki. Wield: Systematic reinforcement learning with progressive randomization. *arXiv preprint arXiv:1909.06844*, 2019.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

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.

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.

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.

Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, 2015.

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, 2017a.

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, 2017b.

Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018.

Joanne Truong, Sonia Chernova, and Dhruv Batra. Bi-directional domain adaptation for sim2real transfer of embodied navigation agents. *IEEE Robotics and Automation Letters*, 6(2):2634–2641, 2021.

Cameron Voloshin, Hoang M Le, Nan Jiang, and Yisong Yue. Empirical study of off-policy policy evaluation for reinforcement learning. *arXiv preprint arXiv:1911.06854*, 2019.

Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *Ieee Network*, 32(2):92–99, 2017.

Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

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.

Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

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. ISBN 978-1-939133-01-4. URL `https://www.usenix.org/conference/atc18/presentation/yan-francis`.

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. ISBN 978-1-939133-13-7. URL `https://www.usenix.org/conference/nsdi20/presentation/yan`.

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.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 258–271, 2021.

Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*, 2020.