Scalable Dynamic Resource Allocation via Domain Randomized Reinforcement Learning

Yiqi Wang
Electrical and Computer Engineering
Carnegie Mellon University
yiqiw2@andrew.cmu.edu

Laixi Shi
Computing and Mathematical Sciences
California Institute of Technology
laixis@caltech.edu

Martin Hyungwoo Lee
Electrical and Computer Engineering
Carnegie Mellon University
martinhwl@cmu.edu

Jaroslaw Sydir
Intel Labs
Intel Corporation
jerry.sydir@intel.com

Zhu Zhou
Intel Labs
Intel Corporation
zhu.zhou@intel.com

Yuejie Chi
Electrical and Computer Engineering
Carnegie Mellon University
yuejiechi@cmu.edu

Bin Li
Intel Labs
Intel Corporation
bin.li@intel.com

***Abstract*—In 5G wireless networks, the User Plane Function (UPF) plays a crucial role in efficiently transferring users' traffic — a series of data packets — to manage internet communications. Setting the server's processor frequency excessively high can easily meet the packet drop requirements but may lead to unnecessary power consumption. Therefore, as user traffic fluctuates, selecting the optimal processor frequency is essential for minimizing power consumption while satisfying packet drop constraints. This challenge motivates us to address the dynamic resource (frequency) allocation problem, where deep reinforcement learning (RL) has shown significant potential. Most existing studies train and evaluate the RL model in the same environment with consistent traffic patterns. However, frequent variations in user traffic can cause the policy trained on the outdated traffic to fail catastrophically on unseen traffic.**

To address such traffic distribution shifts, we propose a two-phase RL approach augmented with Automatic Domain Randomization (RL-ADR). This method includes a training phase that utilizes domain randomization to create a library of policy candidates, and an inference phase that selects the optimal frequency using this policy library alongside a safe data buffer. The proposed RL-ADR achieves zero packet drops on two unseen long-horizon traffics (3 hours) after being trained on 25 synthetic traffics that only span for 18 seconds. Compared to static resource allocation baselines, RL-ADR reduces power consumption by at least 14.5% and performs comparably to the oracle solution.

***Index Terms*—Resource allocation, deep reinforcement learning, domain randomization.**

I. Introduction

A growing number of mobile devices connect to wireless networks on a daily basis, requiring a wireless network to process users' traffic efficiently to meet the quality of service required by different applications (e.g., video streaming [1]). In 5G core network, the 5G User Plane Function (UPF) workload [2] plays an important role in transferring users' traffic (consists a series of packets) to meet strict packet drop requirements from various applications. While it is possible to meet the packet drops requirement easily by increasing the processor frequency of the server (core or uncore frequencies), higher processor frequency will result in a rise in power consumption. This becomes a huge concern of both the electric bills and sustainability regarding that communication technology has contributed 2-2.5% worldwide greenhouse gas emissions [3]. Thus, adaptively choosing the processor frequency based on the user traffic becomes the key to balancing two competing objectives: 1) minimizing packet drops (ideally 0 drops), and 2) minimizing power consumption (only allocating necessary resources).

Since consistently allocating high frequency leads to excessive power consumption when the packet rate is low, adaptively allocating resources to match the dynamically changing traffic rates are crucial for practical and power-efficient networks. For such sequential decision making problems, deep reinforcement learning (RL) has shown significant power when there are multiple competing objectives to be considered. For instance, [4], [5] combines two objectives — packet drops and cache allocations/power consumption — into the reward functions to train a deep Q-network (DQN) [6] to allocate resources for network packet processing workload.

Despite the recent progress in allocating resources via deep RL, we notice that vanilla RL usually trains and evaluates the policy in the same environment (with an identical traffic profile), illustrated in Fig.1. In real-world applications, however, the unseen traffic during inference process could deviate from the one used in training, posing a generalization challenges. Vanilla RL approaches can't achieve zero packet drops and low power consumption in such unseen environments.

To address the generalization challenges, in this paper, we propose a novel 2-phase algorithm named RL-ADR to train a deep RL control policy with Automatic Domain Randomization (ADR) technique (see Fig. 1) to enable resource allocations on completely unseen traffics. The main contributions are summarized as below:

This material is based upon work supported by the National Science Foundation under grant no. CNS-2148212 and is supported in part by funds from federal agency and industry partners as specified in the Resilient & Intelligent NextG Systems (RINGS) program.

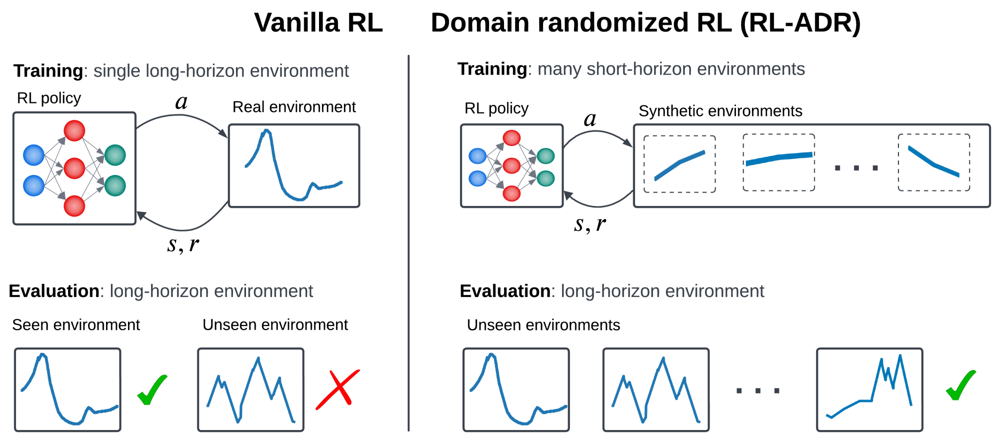
Vanilla RL
Domain randomized RL (RL-ADR)
Training: single long-horizon environment
RL policy
a
Real environment
s, r
Evaluation: long-horizon environment
Seen environment
Unseen environment
Training: many short-horizon environments
RL policy
a
Synthetic environments
s, r
Evaluation: long-horizon environment
Unseen environments

Fig. 1. Vanilla RL (left) VS. Domain Randomized RL (right). Originally, a RL policy is trained and tested in the same environment characterized by the same traffic pattern. It is not prepared to apply to any new environment during evaluation. Different from the original RL, we train a RL policy with a sequence of short-horizon environments characterized by synthetic traffics only spans for 18 seconds. The resulted policies are able to dynamically allocate resource for unseen environments (traffics) without dropping packets and consuming power close to the achievable minimum power in the environments.

• RL-ADR leverages the idea that long-horizon traffic can be decomposed into short-horizon traffic sequences for dynamic resource allocation. In phases 1 of RL-ADR, we train a RL agent to solve a sequence of short-horizon synthetic traffics (domain randomization) and build a library of RL policies specialized in resource allocations in diverse traffic patterns. In phase 2, the policies in the library are composed together to solve an unseen long-horizon traffics during inference.

• In evaluation, RL-ADR reduces power consumption by 15.5% on a 3-hour synthetic traffic and 14.5% on a 24-hour traffic profile scaled to 3 hours, with comparison to the static resource allocation baseline.

II. Background and Problem Formulation

A. Reinforcement Learning & Problem Formulation

Reinforcement learning (RL) enables an agent to improve its decision-making by interacting with the environment. RL could be formulated as Markov Decision Processes, described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ [7]. At each time step t, the agent observes its current state s_t from the state space $\mathcal{S}$, and chooses an action a_t from the action space $\mathcal{A}$ based on its policy $\pi(a_t|s_t)$. The environment transmits the agent to the next state s_{t+1} determined by the transition kernel $\mathcal{T}(s_{t+1}|s_t, a_t)$, as the consequence of taking an action a_t. An immediate reward will be given to the agent after it executes the action a_t, represented by $r_t = \mathcal{R}(s_t, a_t)$. Given a discount factor $\gamma \in [0, 1)$ and a horizon length of T, the objective of RL is to find a policy π that maximizes the long-term reward: $\mathbb{E}_{a_t \sim \pi(\cdot|s_t), s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)} [\sum_t^T \gamma^{t-1} r_t(s_t, a_t)]$.

This paper investigates dynamic resource allocation (i.e., processor frequency) using a RL agent, trained in an environment characterized by a traffic profile [2] of horizon T. At each time step t, an agent observes a state s_t including performance counters and a packet rate $p \in [p_{\min}, p_{\max}]$. The agent's policy chooses an action a_t to allocate resources by predicting processor core and uncore frequency. At $t+1$, the agent observe s_{t+1} and a scalar reward determined by the objectives of minimizing packet drops and power consumption caused by taking the action a_t at state s_t. Readers interested in details could refer to Section IV-A.

B. Domain Randomization

In our algorithm design, we have drawn inspiration from the technique Automatic Domain Randomization (ADR), which primarily used in robotics and RL to improve generalization of the machine learning models to new environments [8]. By systematically changing key factors of the environment, ADR aims to expose a wide range of scenarios to a model, thereby improving its adaptability to novel environments during inference. The systematically changing of environmental factors will be done in a progressive manner. Once a model demonstrates proficiency in handling the current level of environmental complexity, it's encouraged to tackle more diverse and challenging scenarios to further enhance its robustness and generalization capabilities. In our task, we focus on progressively randomizing user traffic patterns (1-d time series, different in scales and steepness) to help the policy generalize across various real-world networking scenarios characterized by inherently unpredictable and highly variable user traffic.

III. Methodology

To address the generalization concerns raised in Section I, we proposed a two-phase algorithm — RL-ADR. Inspired by the process of ADR, the first phase is called domain randomized RL, which trains a library of policies specialized in resource allocation of diverse short-horizon traffics. Then, the second phase of our algorithm named policy library inference will compose the policies from the library trained on short-horizon traffics to choose frequency (allocate resources) for a long-horizon traffic. Finally, we highlight our design of the neural network architecture for deep RL, tailored for the resource allocation tasks with different types of states.

A. Domain Randomized RL

Motivated by the capability of ADR to enable generalization (introduced in Section II-B), we designs an algorithm to randomly generate traffics (a sequence of packet rates) with constrained values and steepness. The packet rates range $[p_{\min}, p_{\max}]$ is divided into K intervals uniformly to construct K^2 traffics. For the interval k and its combination with all K intervals, there're K number of interval pairs. For each pair of the intervals, we will 1) uniformly samples a start and end rate from the first and second interval respectively; 2) uniformly sampling a rate between the start and end rate; and lastly 3) generates the full traffics by connecting the selected 3 rates smoothly. Note that all the traffics generated by the above steps are simple traffics that are flat, monotonically increasing or decreasing as time goes. Empirically, we found learning from these simple traffics patterns are sufficient to handle unseen real world traffics and challenging synthetic traffics.

The overall procedure is described by Algorithm 1. The policy of the RL agent is trained to solve K^2 number of short-horizon traffics generated by traffic generator $\mathcal{G}$, each with a total number of T steps. We reinitialize replay buffer $\mathcal{B}$ when a new traffic is generated from $\mathcal{G}$, while not the policy π since the knowledge from previous training process could be helpful. Given $N = K^2$ number of randomized traffics (environments), N corresponding trained policies will be added to the library L, each is registered with a key *(start rate, end rate)* used for policy retrieval.

Algorithm 1 Domain Randomized RL

```
1:  procedure ADR($K, T$)
2:      $N \leftarrow K^2$                        ▷ Number of traffics in ADR
3:      Initialize a RL policy $\pi$ and $a_0$.        ▷ default action
4:      Initialize a traffic generator $\mathcal{G}$.
5:      Initialize a RL environment to be interacted $E$
6:      while $N > 0$ do
7:          key, traffic $\leftarrow \mathcal{G}$   ▷ yields the next traffic to train
8:          Initialize an empty replay buffer $\mathcal{B}$
9:          $s_1, \_ \leftarrow E$(traffic)($a_0$)
10:         for $t = 1$ to $T$ do
11:             $a_t = \pi(s_t)$.
12:             $s_{t+1}, r_t \leftarrow E$(traffic)($a_t$)
13:             Add experience $(s_t, a_t, r_t, s_{t+1})$ to $\mathcal{B}$
14:             Optimizes $\pi$ by sampling experience from $\mathcal{B}$
15:         end for
16:         Add { key: $\pi$ } to the policy library $L$.
17:         $N \leftarrow N - 1$
18:     end while
19:     return policy library $L$
20: end procedure
```

B. Policy Library Inference

Our inference algorithm combines the policy library L obtained through the training process with a safe buffer b. Although a library of policies from phase 1 is trained to allocate resources for many different rates, the training process cannot cover all possible rates given limited computation resources and time. Thus, we introduce a safe buffer constructed based on the the training data to 1) output safe actions (no packet drops with reasonable power) when the rate is unseen for the library or 2) output a lower bound action to remove trivially bad actions from the policy's predictions.

In buffer b, discrete rates are keys, and power-efficient actions are values. Regarding that the prior data during training is noisy, an action corresponding to a key rate p of b in a low-rate regime is determined by: 1) finding all actions from the training data with some rate $p' \geq p$, and 2) chooses the action with the reward as high as the 98% percentile one among all the actions. The lower rate range is defined by rate ≤ 5 millions of packets per seconds (mpps). For higher rate regimes (> 5 mpps), we use the 30% percentile instead of 98%. The intuition behind this is that dropping packets is more likely to take place when packet rates are high. Therefore, we choose a safer action associated with smaller rewards (i.e., 30%), which is equal to higher frequency. For the lower rate, we choose the action that almost achieves minimum power consumption. The hyperparameters including 5 mpps, 98%, and 30% percentile are determined based on domain knowledge and empirical performance.

Armed with the prior brought by the safe buffer b, the policy library L inference procedure is summarized in Algorithm 2. At each time step, the proposed algorithm tries to 1) pick up a candidate policy from the library L to output power-efficient action and 2) combine with the information in the safe buffer $\mathbf{b}$ to make the action power-efficient and safe. For the first step (line 9-20, Algorithm 2), a policy from L will be a candidate when the rate pattern matches the key of the policy library. Specifically, we look through all the keys in libaray L and consider those cases that the key (i.e., (start rate, end rate)) and the current rates (last rate and current rate) have the same tendency, namely increasing (i.e., key.start $<$ key.end and last rate $<$ rate) or decreasing. In such cases, the policy trained on the corresponding traffic is assumed to be capable of allocating resources with efficient power consumption for the current step. If such cases occur, we only output one candidate (the policy corresponds to one key) whose packet range is the smallest one (line 13, Algorithm 2). The reason is that a smaller packet rate range indicates the policy is trained on flatter traffic, which empirically yields better performance given the traffic is easier to solve. After finalized the chosen policy from L, it will predict an action. Then, a lower-bounded action is retrieved from the buffer (line 21-22, Algorithm 2) to make sure the action to be executed is not trivially unsafe (dropping packets). The lower-bound action will be retrieved by rounding the current rate down to the nearest key rate in the safe buffer. The final action will be the maximum

Algorithm 2 Policy Library Inference

```
 1: procedure INFERENCE($L$, traffic, $T$, $b$)
 2:     Initialize a RL environment wrapper $E$
 3:     $p' \leftarrow 1$ mpps  ▷ initializes previous rate to minimum
 4:     for $t = 1$ to $T$ do
 5:         $idx \leftarrow 0$     ▷ Index to retrieve $\pi$ from Library $L$
 6:         $\pi \leftarrow$ None
 7:         $p \leftarrow$ traffic$[t]$                    ▷ current packet rate
 8:         range $\leftarrow +\infty$
 9:         while idx $< len$(L.keys) do
10:             key $\leftarrow$ L.keys[idx]
11:             if sign$(p' - p)$ = sign(key.start − key.end)
12:               and min(key) $\leq p \leq$ max(key) and
13:               max(key) − min(key) $\leq$ range then
14:                 range $\leftarrow$ max(key) − min(key)
15:                 $\pi \leftarrow$ L[key]          ▷ prefers $\pi$ trained on
16:                                                flatter traffics
17:             end if
18:             idx $\leftarrow$ idx + 1
19:         end while
20:         if $\pi$ is not None then
21:             $\pi$ predicts an action.
22:             Lower bound the action by checking $b$.
23:         else
24:             Replays a safe action from buffer $b$.
25:         end if
26:         Execute the action in $E$(traffic).
27:         $p' \leftarrow p$
28:     end for
29: end procedure
```

core and uncore frequencies between lower-bound action and the predicted action since any action lower than the action corresponding to a lower rate is prone to drop packets. If a rate at test time cannot be matched to any policy in the library, an action corresponding to the nearest round-up key rate in the safe buffer will be replayed (line 24, Algorithm 2).

C. *Feature-Aware Architecture Design*

Recall that deep RL usually parameterizes the policy using a deep neural network, such as one of the widely used method DQN [6]. Tailored to our tasks, we propose a new neural network architecture named DQN-FiLM, since a state is composed of two types of inputs: 1) 8 performance counters (large integers) and 2) one incoming packet rate (a floating point number). Known from domain knowledge, the packet rate involves more information for predicting the next action than the performance counters. To address such information bias, we insert FiLM [9] layers to the architecture used in DQN (fully connected layers) to emphasize on packet rates, shown in Fig. 2. Each FiLM layer transforms the representation X_i from the previous layer by a γ_i and a β_i (both are scalars predicted by the FiLM generator) for $i = 0, 1, 2$. The parameters $\{\gamma_i\}$ and $\{\beta_i\}$ from all FiLM

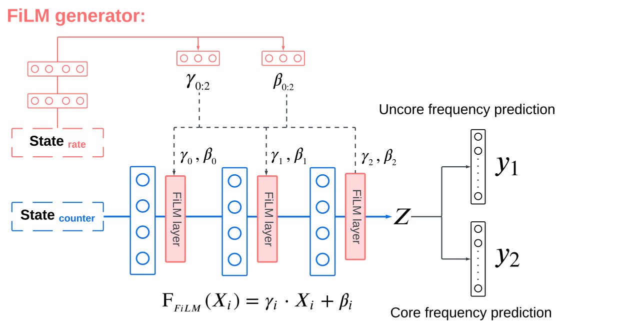

Fig. 2. Illustration of the architecture used in DQN with FiLM layers. The representation learned from counters will be transformed by the γ, β learned by a separate network.

layers are predicted by a separate neural network (shown in Fig. 2 in red), taking packet rate as the input. Lastly, the decision backbone shown in red and blue in Fig. 2 will output Z, which will be fed into the two prediction heads for state-action value predictions, similar to Tavakoli et al. [10].

IV. Experiments and Evaluation

A. *Experimental Setups*

Task environmental settings. The state observed by an RL policy on the UPF server is based on performance counters and packet rates. In the experiments, we consider 8 performance counters (shown in Table I) selected through the technique in [5]. We assume the RL policy can directly observe the current ground truth packet rates as part of the state during both training and inference. The policy chooses a core frequency from the set $[800, 900, ..., 2300]$ MHz and an uncore frequency from $[800, 900, ..., 2400]$ MHz as an action. Frequency adjustments are made through the CPU governor interface [11], while non-utilized cores are set to idle.

The reward signal is computed from both packet drops D and power consumption P, shown in Algorithm 3. It penalizes packet drops with negative values constructed from current packet drops for 1 second, scaled by the maximum possible packet drops profiled in advance. If there are no packet drops, it will output a positive value to encourage the actions with smaller power consumption. We use Intel PCM Power monitor [12] to measure the power consumption of the socket where the UPF workload runs on, and measure the packet drops at the Network Interface Card (NIC).

Finally, The experiments are conducted on a two-server system, where the UPF workload runs on dedicated cores on one socket of an Intel Ice Lake Xeon® server. This server is

TABLE I
CPU PERFORMANCE COUNTERS USED BY RL.

cycle_activity.stalls_mem_any
cycle_activity.stalls_l2_miss
frontend_retired.latency_ge_32
offcore_requests_outstanding.cycles_with_demand_code_rd
uops_executed.core_cycles_ge_3
rs_events.empty_cycles
offcore_requests.demand_data_rd
mem_load_retired.l2_miss

Algorithm 3 Reward function

```
1: procedure REWARD( $D_{\text{current}}, D_{\max}, P_{\text{current}}, P_{\min}, P_{\max}$)
2:     if $D_{\text{current}} > 0$ then                ▷ If packet drops $> 0$
3:         $r = -D_{\text{current}}/D_{\max}$
4:     else                         ▷ minimizing power consumption
5:         $r = (P_{\max} - P_{\text{current}})/(P_{\max} - P_{\min})$
6:     end if
7:     return $r$
8: end procedure
```

connected to a TREX [13] traffic generation server (Intel® Xeon® server) through 100 Gb/s Ethernet links.

Our proposed method RL-ADR. RL-ADR is trained on $N = K^2 = 25$ randomized traffics following the procedure described in Algorithm 1. The traffic generation process in line 7 of Algorithm 1 generates traffic with packet rates ranged from $[p_{\min}, p_{\max}]$ = [1mpps to 15mpps], with each traffic only spans for 18 seconds. The starting rates of the generated traffics will gradually increased from 1 to 15 mpps, to make the traffic more and more challenging. We use the DQN-FiLM architecture proposed in Section III-C as the policy network. The training takes around two and a half days.

Vanilla RL baseline. As described in the Fig. 1 (left), a vanilla RL policy uses the same architecture as the proposed RL-ADR (the proposed DQN-FiLM) and is trained and tested in a same environment, detailed in the next sub-section. It is trained by 334 episodes which takes around two days. For training efficiency, we scale the 3 hour long-horizon traffic to 104 seconds to significantly reduce the training time for each episode. This is motivated by that RL agent generally benefits from training more shorter episodes rather than a small number of longer-horizon episode.

Oracle. The oracle represents the accessible optimal reward (i.e., the smallest amount of power without dropping packets) for packet rates within the range [1mpps to 15mpps]. It is constructed by brute-forcing all possible resource configurations (core and uncore frequencies) for a discrete set of packet rates uniformly chosen within the range. Since the packet rate space is continuous, we match each incoming packet rate with the closest higher rate from the oracle set during inference.

Max power. A static baseline always outputs actions that minimizes power consumption (minimal core and uncore frequency) and maintains 0 packet drops for the maximum packet rate (15 mpps). It is used to normalize the power consumption for evaluation.

B. Evaluations in Unseen Traffics

Synthetic traffic of 3 hours. A random traffic is created as a 1-dimensional time series that lasts for 3 hours and later tunes the peaks to be higher or lower. All packet rates are limited in $[p_{\min}, p_{\max}]$ = [1mpps to 15mpps]. The resulting traffic pattern is designed to have lots of steep spikes in order to evaluate the algorithm performance on the traffic with abrupt noise or changes (Fig. 3, top).

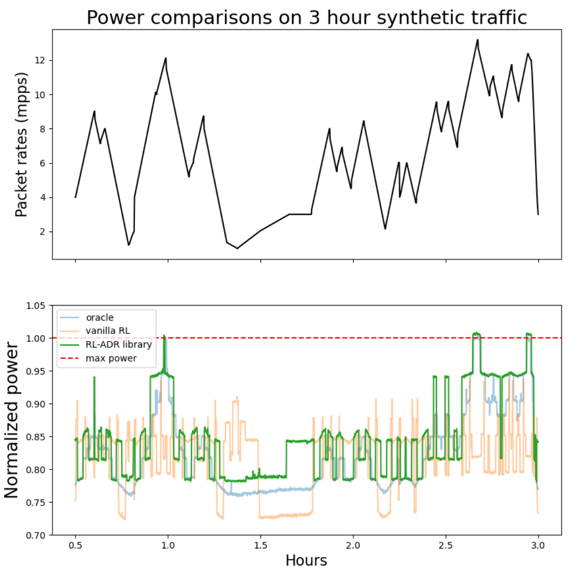

Fig. 3. Comparing power consumption (bottom) on synthetic traffics for 3 hours (top). Note that our proposed method RL-ADR library achieves power consumption similar to the oracle. The power is normalized based on the power achieved by maximum power baseline: power / max_power.

Both vanilla RL and RL-ADR are going to test on the 3-hour version of the synthetic traffic. The difference is that RL-ADR hasn't seen this traffic during training process, while vanilla RL is trained on a version of the traffic scaled to a shorter time. The performance of oracle, RL-ADR, and vanilla RL will be reported on packet drop counts and normalized power saving based on the max power baseline.

The results are shown at the Fig. 3 (bottom), where RL-ADR's power consumption is close to oracle on a totally unseen traffic pattern. Although vanilla RL has seen the synthetic traffic at inference during its training, it still can't consistently allocate sufficient resources similar to the oracle and drops 780 million packets in total. The proposed RL-ADR successfully achieves 0 packet drops and saves 15.5% of power whereas oracle saves 17.7%, as shown in Table II.

Real traffic of 3 hours. The real traffic originally spans for 24 hours, and we scale it to 3 hours for evaluation purposes, shown in the Fig. 4 (top). All packet rates are scaled between 1 mpps to 15 mpps. The traffic has fewer peaks, reflecting the real-world scenario.

Even though vanilla RL has seen the traffic it will be evaluated on, it drops 1.13 million packets. RL-ADR achieves zero packet drops and comparable power consumption as the oracle, shown in the Fig. 4 and Table II. We notice that RL-ADR sometimes consumes unnecessary power when the rate is relatively low (i.e., $\leq$ 5 mpps), as shown in the Fig. 4. This is because the maximum between a suboptimal policy's predicted action and its lower bound action may result in a very safe but high-frequency action. This issue could be mitigated by devising better ways to lower bound the action predicted by the policy, which we leave to future work.

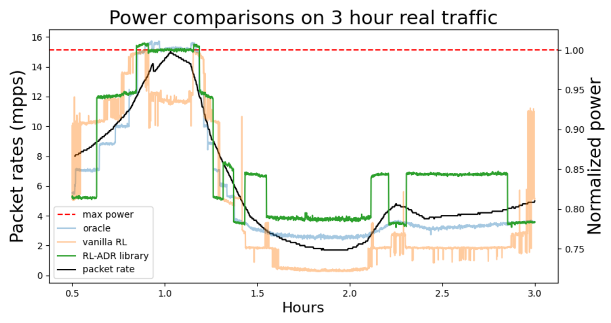

Fig. 4. Comparing power consumption on real traffic for 3 hours. Our proposed RL-ADR method achieves power consumption similar to the oracle. The peaks that are away from the oracle are the scenarios where the predicted action is replaced by a safe action replayed from the buffer. The power is normalized based on the power achieved by the maximum power baseline: power / max_power.

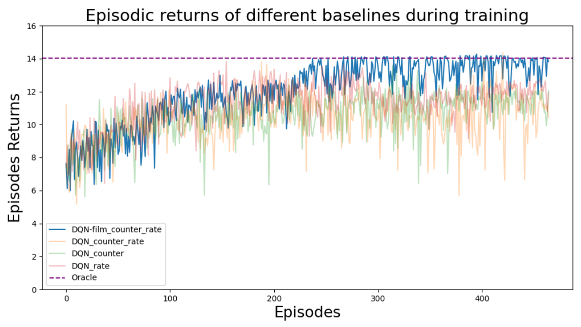

Fig. 5. The ablation study on architecture is shown in Fig. 2. In comparison to the DQN variants lacking FiLM, the DQN with FiLM taking counters and packet rates as input achieves most of the rewards per episode.

C. Effectiveness of Feature-Aware Architecture

We conduct an ablation study on the DQN-FiLM architecture proposed in Section III-C, following the vanilla RL procedure (see Section IV-A). For convenience, we refer DQN-FiLM as DQN-film_counter_rate in this subsection given the usage of FiLM (shown in Fig. 2) with packet rate and performance counters as input. To understand the effectiveness of FiLM, a variant is constructed without FiLM and using the standard architecture in DQN [6] to take the concatenation of normalized counter and packet rate as input (i.e., DQN_counter_rate). To understand the roles of performance counter and packet rate, another two variants (DQN_counter and DQN_rate) are created with the standard architecture but take either the counters or the packet rate as input.

As shown in Fig. 5, DQN-Film_counter_rate using the FiLM architecture with both counter and rates performs the best. While DQN_counter_rate also takes packet rate and counters as input, it doesn't have the ability to leverage two types of inputs in different ways. As expected, leveraging solely counters or rates doesn't bring benefits.

V. Conclusion and Discussion

In this work, we focus on minimizing the power consumption while simultaneously satisfying the packet drop requirement (usually zero drop) when the input users' traffic (a series of packets) is unseen from the training process. To achieve this, we propose a two-phase RL algorithm based on Automatic Domain Randomization called RL-ADR, which achieves zero dropping packets while maintaining a desired power consumption close to the oracle on two conducted long-horizon unseen traffics for testing during inference. In addition, considering the time cost, the training time of RL-ADR only depends on the total number of synthetic short-horizon traffics constructed during the training, while that of vanilla RL heavily counts on the real traffic length which can be much longer. It shows that RL-ADR is more scalable than RL in vanilla setting.

TABLE II
Summary table on power and packet drops.

Traffic types	Baselines	Packet drops total	Power saved on average
Synthetic traffic 3 hour	Oracle	0	17.7%
	RL-ADR library	0	15.5%
	Vanilla RL	780 million	18.2%
Real traffic 3 hour	Oracle	0	17.5%
	RL-ADR library	0	14.5%
	Vanilla RL	1.13 million	18.9%

[a]The saved power is normalized by the maximum power baseline with the formula: (max_power - power) / max_power.

References

[1] O. Oyman, J. Foerster, Y.-j. Tcha, and S.-C. Lee, "Toward enhanced mobile video services over wimax and lte [wimax/lte update]," *IEEE Communications Magazine*, vol. 48, no. 8, pp. 68–76, 2010.

[2] D. Lee, J. Park, C. Hiremath, J. Mangan, and M. Lynch, "Towards achieving high performance in 5g mobile packet core's user plane function," *Intel Corporation: Mountain View, CA, USA*, 2018.

[3] R. Miftakhutdinov, *Energy saving drives new approaches to telecommunications power system.* IntechOpen, 2010.

[4] B. Li, Y. Wang, R. Wang, C. Tai, R. Iyer, Z. Zhou, A. Herdrich, T. Zhang, A. Haj-Ali, I. Stoica, *et al.*, "Rldrm: Closed loop dynamic cache allocation with deep reinforcement learning for network function virtualization," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pp. 335–343, IEEE, 2020.

[5] D. Penney, B. Li, J. J. Sydir, L. Chen, C. Tai, S. Lee, E. Walsh, and T. Long, "Prompt: Learning dynamic resource allocation policies for network applications," *Future Generation Computer Systems*, vol. 145, pp. 164–175, 2023.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[8] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[9] E. Perez, F. Strub, H. De Vries, V. Dumoulin, and A. Courville, "Film: Visual reasoning with a general conditioning layer," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[10] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 32, 2018.

[11] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the linux symposium*, vol. 2, pp. 215–230, 2006.

[12] "Intel performance counter monitor."

[13] "Cisco systems traffic generators."