TTDeep: Time-Triggered Scheduling for Real-Time Ethernet via Deep Reinforcement Learning

Hongyu Jia, Yu Jiang, Chunmeng Zhong, Hai Wan, Xibin Zhao
KLISS, BNRist, School of Software, Tsinghua University, China

***Abstract*—Schedule scheme is essential for real-time Ethernet. Due to the inevitable change of network configurations, the solution requires to be incrementally scheduled in a timely manner. Solver-based methods are time-consuming, while handcrafted scheduling heuristics require domain knowledge and professional expertise, and their application scenarios are usually limited. Instead of designing heuristic strategy manually, we propose TTDeep, a deep reinforcement learning schedule framework, to incrementally schedule Time-Triggered (TT) flows and adapt to various topologies. Our novel framework includes 3 key designs: a period layer to capture the periodical transmission nature of TT flows, the graph neural network to extract and represent topology features, and a 3-step selection paradigm to alleviate the huge action search space issue. Comprehensive experiments show that TTDeep can schedule TT flows much faster than solver-based methods and schedule nearly twice more TT flows on average compared to handcrafted heuristics.**

***Index Terms*—Deep Reinforcement Learning, Time-Triggered Scheduling, Time-Sensitive Networking.**

I. Introduction

The deterministic transmission of real-time data has a strong demand in key fields such as automobiles, trains and aerospace, etc. Ethernet based real-time networks have been proposed to provide real-time data transmission capability with high bandwidth. Time-sensitive networking (TSN [1]) and TTEthernet [2] are enhancements of Ethernet to provide not only traditional best-effort (BE) frame delivery but also time-triggered (TT) frame delivery. TT frame is mainly designed to enable the deterministic transmission of real-time data and its periodic nature relies on two main ingredients: delicate schedule table and precise time synchronization.

The precise arrival and forwarding time of each TT frame are calculated in advance and saved in the so-called schedule tables. Then all network devices (hosts and switches) transmit TT frames periodically according to the table with a synchronized global time. Each specific type of periodically transmitted TT frame is defined as TT flow. Fig.1 shows one TT flow transmits from host H_1 to host H_5. Flow scheduling not only searches the valid route, but also allocates time for each link of the route. We can see that TT flow transmitted on e_1, e_2, e_3 at times 3, 4, 7 respectively.

As the network becomes increasingly flexible, such as node/link failure, new node and switch introduction, high-level application requirement change, they will all result in the variation of network configuration. Therefore, we need to incrementally calculate the schedule table in time.

TT schedule is essentially an NP problem [3]. Proposed solver-based scheduling methods [3]–[6] model the flow requirements, device status, etc., as a set of linear constraints, which ensures that no TT frames transmit through the same link at the same time. Then various solvers are used to search for a solution based on these constraints. However, solver-based methods are time-consuming, thus they are inapplicable in incrementally schedule scenarios. Several improvements [7], [8] were proposed to accelerate the computation process, but they are limited to specific application scenarios.

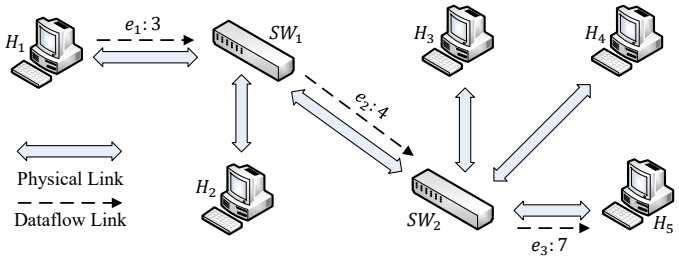

Fig. 1. An example of TT flow schedule with 2 switches and 5 hosts

In contrast, heuristic-based methods can effectively reduce the time of searching solutions. For example, heuristic list scheduler [9] obtains a valid solution by two heuristics: one for TT flow routing, and the other for forwarding time assignment. However, well-designed heuristic strategy requires domain knowledge and professional expertise.

Hence, how to obtain heuristics with good performance becomes a problem worth exploring. In recent years, deep reinforcement learning (DRL) has achieved state-of-the-art performance in many fields. For example, Stampa [10] proposed intelligent routing algorithm, which optimizes network delay of the traditional network. Mao [11] uses policy gradient algorithm to model workflow scheduling problem.

The above work has demonstrated the great potential of using DRL to solve the NP-hard scheduling problems. The essence of DRL is to train an intelligent agent which interacts with the environment and continuously learns how to obtain the highest reward. We carefully design a DRL-based TT schedule framework to effectively capture system status and naturally map actions to schedule solutions. Overall, this paper has made the following contributions:

- We propose a DRL-based framework, TTDeep, to incrementally schedule TT flows in a timely manner.
- Several optimization methods are devised to handle TT domain specific modeling, global information extraction and large search space problems.

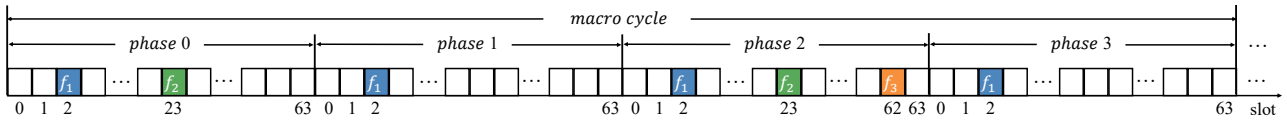

Fig. 2. Simplified slot allocation result. Three TT flows with periods of $1ms$, $2ms$ and $4ms$ are scheduled.

- Comprehensive experiments show that TTDeep outperforms handcrafted scheduling heuristics. And it can well adapt to different topologies and handle network changes.

The remaining parts is organized as follows: section II defines TT scheduling problem formally. The design and implementation details of TTDeep are introduced in sections III and IV respectively. Section V conducts detailed experimental evaluation. Finally, we conclude in section VI.

II. Problem Definition

A. Basic TT Notations

The network topology can be formally defined as an undirected graph $G(V, E)$. All devices and physical links in the network are denoted by vertex set V and edge set E respectively. Each undirected edge is denoted by two bidirectional dataflow links, and thus we can get the dataflow link set L:

$$(v_i, v_j) \in E \implies [v_i, v_j] \in L, [v_j, v_i] \in L \tag{1}$$

The id, source, destination, frame length, maximum end-to-end delay and period of i-th TT flow is denoted by $f_i = \{f_i.id, f_i.src, f_i.dst, f_i.len, f_i.dly, f_i.prd\}$. All TT flows are denoted as the set F. For a TT flow starting from v_1 and ending at v_m, its route is denoted by the path $RT = [[v_1, v_2], \cdots, [v_{m-1}, v_m]]$. Since all TT flows are transmitted periodically, we need the concept of **macro cycle**, which refers to the least common multiple (LCM) of all TT flow periods, $lcm_{prd} = LCM(f_1.prd, \cdots, f_{|F|}.prd)$.

B. Time Slot Assignment Problem

We divide continuous time into discrete **slots**, which are the smallest time units. Following the time granularity of the ILP-based method [7], a slot equals to $\frac{1}{64}ms = 15.625\mu s$. For the link speed of $1Gbit/s$, the transmission of one maximum transmission unit (MTU) frame can be completed within one slot. Moreover, the period of each TT flow is the power of 2 (e.g. $1ms$, $2ms$, $4ms$, etc.) [12]. Thus, f_i needs to transmit $lcm_{prd}/f_i.prd$ frames in a macro cycle.

For each TT flow requirement, the TT scheduling algorithm outputs a route from its source to its destination, and assigns slots to the flow for all links along the route. Suppose three TT flows (f_1, f_2, and f_3 with periods $1ms$, $2ms$, and $4ms$ respectively) are all routed passing through one specific link k. Fig. 2 shows a possible schedule result on the link k.

To handle huge search space issue, we split slot assignment process into two steps: phase selection and offset calculation. A **phase** equals to $1ms$. As shown in Fig. 2, the macro cycle is divided into 4 phases, and each phase further contains 64 slots. When assigning slots for the flow, we first select its phase and then decide which slot in that phase should be assigned to the flow. For example, TT flow f_3 occupies the 63rd slot of the 3rd phase. Then the slot assignment problem is deduced to the problem of finding the **first phase index** in the macro cycle and the **offset** within the phase. As shown in Fig. 2, the $(first\_phase\_index, offset)$ pairs of each flow is $(0, 2)$, $(0, 23)$, and $(2, 62)$. Given the $first\_phase\_index$ and $offset$ of flow f, we can calculate the first slot that f occupies:

$$first\_slot(f) = first\_phase\_index(f) \times 64 + offset(f) \tag{2}$$

Furthermore, the set of slots assigned to f in a macro cycle is: $\{first\_slot(f) + f.prd \times i | i \in \{0, 1, \cdots, lcm_{prd}/f.prd - 1\}\}$.

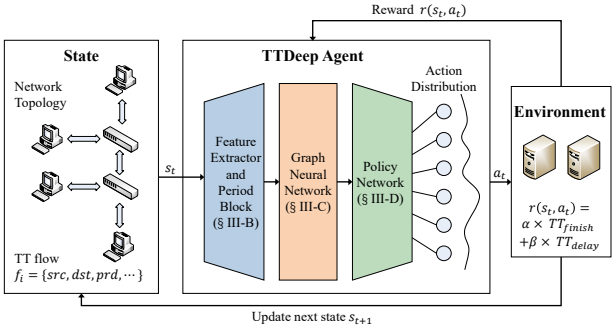

Fig. 3. TTDeep framework, which contains three main ingredients: state, TTDeep agent and environment.

III. Detailed Design of TTDeep

A. Overview

The architecture of TTDeep is illustrated in Fig. 3. During the t-th step, TTDeep observes state s_t, which contains topology information and the requirement of current flow. Then TTDeep encodes state into embedding vectors through graph neural network (GNN). Finally, the TTDeep agent leverages policy network to select action a_t. After executing action a_t, the agent can obtain the feedback reward $r(s_t, a_t)$ from the environment. TTDeep expects to schedule as many flows as possible while ensuring low transmission delay. When the scheduling process reaches terminal conditions, we decay rewards of the entire trajectory and update network parameters.

Fig. 4 shows TTDeep model architecture, which contains three parts: feature extractor, GNN and policy network. Feature extractor encodes the basic information of each link to a corresponding feature vector. It expresses the characteristics of a single link, while GNN propagates the information of each link to its neighbors, so that TTDeep can learn richer knowledge. Finally, to select a well-performed action, the policy network scores each link and the corresponding slots.

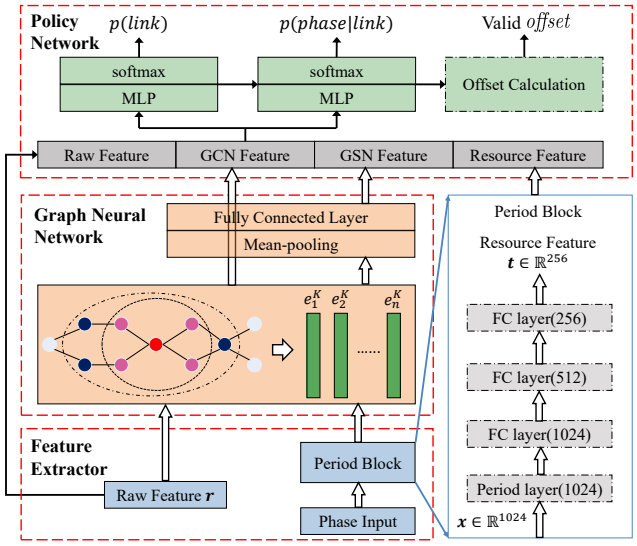

Fig. 4. Network architecture of TTDeep, which can be divided into three parts: feature extractor, graph neural network and policy network.

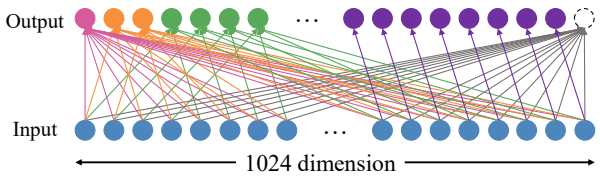

Fig. 5. Period layer structure. Output neurons with the same color are grouped together. For example, the green group has 4 neurons, each neuron connects to 256 input neurons. The last grey neuron is a dummy neuron.

B. State Modeling: Basic Feature and Period Block

We present each link v as a feature vector $\boldsymbol{e_v}$, which captures the information related to TT schedule. This feature vector consists of two parts: raw feature vector $\boldsymbol{r_v}$ and available resource vector $\boldsymbol{t_v}$, where $\boldsymbol{r_v} \in \mathbb{R}^6$ is the basic feature and contains the following elements: the source of flow or not, the destination of flow or not, whether current link has been visited, the proportion of legal phase, the used rate of bandwidth, and the period of flow.

Without loss of generality, we assume the macro cycle n is $1024ms$ in the following discussion. Note that one phase contains 64 slots, so that this macro cycle has 65,536 slots. To ensure no frame loss during the flow transmission, the slots that previously occupied can not be scheduled once more. So that we use $\boldsymbol{t_v}$ to represent available slots resource. However, simply encoding all available slot status will result in unacceptable memory consumption. Here, we take into consideration the periodic nature of TT transmission and propose **period layer**. Fig. 5 shows that each neuron of the input layer $\boldsymbol{x_v} \in \mathbb{R}^{1024}$ represents a phase. The value of each input neuron is the available slot rate of corresponding phase.

For the traditional fully connected (FC) layer, each output neuron connects with all input neurons directly. However, we observe that flow f with period $f.prd$ just occupy $lcm_{prd}/f.prd$ slots in a macro cycle, which means TT flow with a higher period influences less phase status. Hence, we design a specific connection pattern as illustrated in Fig. 5. For $i \in \{0, 1, \cdots, k-1\}$, where $k = log_2(n)$, we group every 2^i output neurons together, and connect them to 2^{k-i} input neurons separately. So far we have $\sum_{i=0}^{k-1} 2^i = n-1$ output neurons connected with the input layer. To keep the input and output dimension consistent, we add a dummy neuron at the end of the output layer. The number of parameters of period layer is $O(\sum_{i=0}^{k-1} 2^i \times 2^{k-i} + n) = O(nlogn)$. Compared with FC layer whose parameters is $O(n^2)$, period layer can save a great number of parameters. We further design **period block** (cf. Fig. 4) which encodes the input vector $\boldsymbol{x_v}$ to available resource vector $\boldsymbol{t_v}$. Period block helps TTDeep learn adequate feature knowledge in a relatively fast time and accelerate model convergence.

C. Neighbor Knowledge gathering: Graph Neural Network

To adapt to various network topologies, we take advantage of the graph convolutional network (GCN) to get embedding vector of each link and use graph summarization network (GSN) to extract the global information.

For any link v, GCN takes feature vector $\boldsymbol{e_v^0}$ as input and performs per-link embedding. Link v absorbs information from its neighbors and the output embedding vector is denoted by:

$$\boldsymbol{e_v^k} = g\left[\sum_{u \sim \xi(v)} f(\boldsymbol{e_u^{k-1}}) + \boldsymbol{e_v^{k-1}}\right], k = 1, 2, \cdots \quad (3)$$

Where f and g indicate activation functions (Relu, PRelu, etc.), and $\xi(v)$ denotes all neighbor links of v. Each link collects all information of its k-order neighbors by iterating such computation k times.

GSN calculates the average of all link embedding vectors and then encodes it through the FC layer and activation function f and g. $\boldsymbol{W}$ denotes all parameters of GSN, then the process of GSN can be formulated by:

$$\boldsymbol{e_{global}} = g\left(\boldsymbol{W}\left(\frac{\sum_{v \sim V} f(\boldsymbol{e_v^k})}{count(v \sim V)}\right)\right) \quad (4)$$

For any link v, we can obtain the final vector $\boldsymbol{z_v}$, which is a concatenation of raw feature vector, available resource vector, per-link embedding vector and global vector:

$$\boldsymbol{z_v} = concat(\boldsymbol{r_v}||\boldsymbol{t_v}||\boldsymbol{e_v^k}||\boldsymbol{e_{global}}) \quad (5)$$

D. Action modeling: 3-step Selection Paradigm

When TTDeep agent performs action selection, there are at most hundreds of options for the selection of the next-hop link in the route. Meanwhile, we have over 10^4 choices for slot assignment of a TT flow. Such huge search space makes it difficult for TTDeep to learn an effective strategy.

Hence, we devise a 3-step action paradigm and split the action into three parts $a_t = (link, phase, offset)$. $link$ denotes the routing option, i.e., which link the TT flow should go through to reach the next node. The second part denotes the

forwarding $phase$ index. $offset$ denotes the slot offset within $phase$. Then policy network sequentially calculates these three parts as illustrated in the Fig. 4.

Link Selection Sub-action. The input of the policy network can be denoted as a matrix $\boldsymbol{M} = [\boldsymbol{z_1^T}, \boldsymbol{z_2^T}, \cdots, \boldsymbol{z_m^T}] \in \mathbb{R}^{m \times n}$, where m is the number of links and $\boldsymbol{z_i} \in \mathbb{R}^n$ is the input vector of the i-th link. Then the policy network exploits multiple layer perceptron (MLP) to score each link and obtains $\boldsymbol{scr} = [s_1, s_2, \cdots, s_m]$. Finally, it turns $\boldsymbol{scr}$ into the probability distribution of the link through softmax function:

$$p_i = \frac{exp(s_i)}{\sum_{j=1}^{m} exp(s_j)} \tag{6}$$

The link selection will be sampled according to this probability distribution, and the link with a higher score has a greater probability of being selected as the next hop of the route.

Phase Selection Sub-action. After selecting the next link $k \in [1, m]$, we utilize the input vector $\boldsymbol{z_k}$ and build another MLP to score all phases of link k. Similar to the link selection stage, we also use softmax function to regularize the probability distribution and select the forwarding phase h.

Offset Calculation Sub-action. Leveraging MLP to select the offset within the phase will unnecessarily increase the difficulty of training model. Therefore, we traverse from slot 0 to the latest slot 63 of phase h, and directly assign the earliest valid $offset$ to the phase h.

E. Model Updating: policy gradient algorithm

As flow requirements continuously arrive and they are incrementally scheduled, TTDeep will reach terminal condition when it fails to schedule current flow (find wrong route or cannot allocate valid slot). Then we use the policy gradient algorithm [13] to update TTDeep model.

The policy gradient algorithm updates model by calculating the gradient of the expected reward value relative to the parameter. The state and action sequences in the scheduling process can be considered as a trajectory $\tau = (s_0, a_0, \cdots, s_{n-1}, a_{n-1}, s_n)$. Assuming θ indicates all parameters of TTDeep, α and γ denote the learning rate and decay factor respectively, the goal of TTDeep agent is to find an optimal strategy $\pi : s \times a \rightarrow [0, 1]$, so as to maximize the reward value expectation $R_\pi = E_\tau[\sum_{t=0}^{n-1} \gamma^t r(s_t, a_t)]$. we can derive the gradient and the corresponding update formula:

$$\nabla_\theta R(\pi_\theta) = E_{s \sim p_\pi, a \sim \pi_\theta} \left[\nabla_\theta log(\pi_\theta(a|s))(r(s, a) - b)\right] \tag{7}$$

$$\theta = \theta + \alpha \nabla_\theta R(\pi_\theta) \tag{8}$$

Where b is estimated based on the mean value of all rewards under the same trajectory, thus reducing the variance in the model training process without introducing bias.

To calculate $\pi_\theta(a|s)$, as we calculate $offset$ directly without using neural network, the probability of the action triplet is $P(a = [link, phase, offset]|s) = p(link) \times p(phase|link)$. When t-th step action a_t reaches the destination of current flow, then TTDeep will obtain next flow requirement until it reaches terminal condition. Otherwise, it indicates that a_t are moving forward to its destination and TTDeep will update slot occupancy situation and transit to the next state s_{t+1}.

We can convert the action triplet $(link, phase, offset)$ into schedule solution directly. $link$ indicates the next-hop, and the forwarding time of each flow follows Eq. 2. After action reaches the flow destination, we can obtain the whole route of current flow, and slots of each link along the route.

Algorithm 1 TTDeep Training Procedure

Input: network topology NT, TT flow requirement set F

Initialize TTDeep parameter θ, dual replay buffer R
for episode = $1, \cdots, M$ **do**
  Reset NT, F and get initial state s_0
  for $t = 0, \cdots, T$ **do**
    Sample link $link$ according to $p(link|s_t)$
    Sample phase $phase$ based on $p(phase|s_t, link)$
    Calculate the earliest valid slot $offset$
    Take action $a_t = [link, phase, offset]$, get reward $r_t = r(s_t, a_t)$ and transit to the next state s_{t+1}
    Store transition (s_t, a_t, r_t) in R
    Get new TT flow from F if a_t reaches destination.
  end for
  for $t = 0, \cdots, T$ **do**
    Decay and update reward $r_t = \sum_{q=t}^{T} \gamma^{q-t} r_q$
  end for
  for $i = 1, \cdots, N$ **do**
    Sample transitions (s_i, a_i, r_i) from replay buffer R
    Exploit Eq. 7 and Eq. 8 to update parameter θ
  end for
end for

IV. Implementation Details

Alg. 1 shows the pseudo-code of the action selection and TTDeep training process. At each timestep t, TTDeep encodes state s_t through feature extractor and GNN first, then it calculates the probability distribution and serially selects sub-actions. When the current trajectory reaches the terminal condition, TTDeep will decay the reward and save each transition (s_t, a_t, r_t) in the replay buffer.

The transitions of the flow f_i are considered to be failed if TTDeep reaches terminal condition. By contrast, all transitions of f_i will be stored in the success replay buffer if TTDeep finds a valid solution. To cope with the unbalance between the success and failure transitions, we exploit double replay buffer [14] to sample these two type of transitions individually.

TTDeep aims to meet as many TT flow requirements as possible while keeps relatively low transmission delay, thus we set reward function as Eq. 9 and Eq. 10.

$$r(s_t, a_t) = \alpha \times TT_{finish} + \beta \times TT_{delay} \tag{9}$$

$$TT_{finish} = \begin{cases} 1, & \text{action reaches destination} \\ 0, & \text{scheduling is in progress} \\ -1, & \text{schedule failed} \end{cases} \tag{10}$$

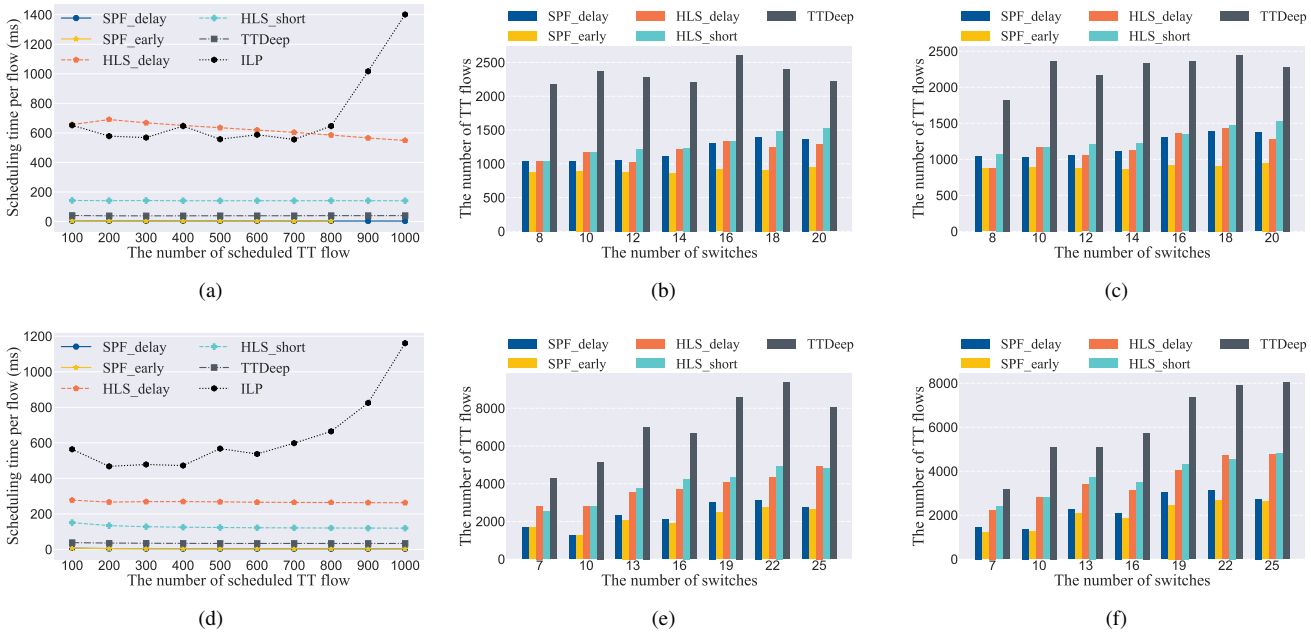

Fig. 6. The first and second rows depict the experiment results of ladder topology and RRG topology respectively. (a)(d) Schedule running time per flow of various approaches. (b)(e) Comparison of the number of scheduled TT flows. (c)(f) Comparison of link failure adaptation capability.

TT_{delay} indicates transmission delay of the current flow, usually we set the hyperparameters $\alpha = 1$, $\beta = 1 \times 10^{-7}$.

The hyperparameter settings of the TTDeep model are as follows. The number of iterations in GCN is 2. The length of GCN and GSN embedding vectors are both 32. In the policy network, the MLP that scores $link$ contains four FC layers with dimensions 128, 64, 16, 1. The MLP of $phase$ includes two FC layers with dimensions 512 and 1024. In addition, the activation function of all layers uniformly uses Leaky_Relu. Adam optimizer with learning rate 1×10^{-4} is adopted when updating the parameters. The reward decay factor is set to 0.99 and we train TTDeep at least 1000 episodes.

V. Evaluations

A. Experiment Setup

TTDeep training stage is deployed on a server with Nvidia Tesla T4 GPU. Note that all experiments of TTDeep evaluation stage and baseline algorithms are evaluated on a workstation with an Intel Core i7-10710U CPU and 16GB RAM.

1) Network Topology We select two real application topologies and one randomly generated topology.

Ladder topology is deployed in train communication networks and its switch number varies from 8 to 20.

Orion Crew Exploration Vehicle (CEV) topology [15] is a fixed topology with 13 switches, 31 hosts and 53 links.

Random Regular Graph (RRG) is randomly generated by the python library NetworkX and each switch randomly connected to five other devices.

We trained TTDeep agents for each type of topology. In each scenario, the corresponding agent is used for evaluation. All flow requirements are generated randomly. The source and destination of each flow are randomly selected. In addition, we randomly select frame length between 64 bytes to 1518 bytes. The period of TT flows is uniformly selected from $\{2ms,\ 4ms,\ 8ms,\ 16ms,\ 32ms,\ 64ms\}$ and their delays are restricted randomly from $512ms$ to $1024ms$.

2) Baseline Scheduling Methods

SPF_delay Shortest path first delay (SPF_delay) [16] calculates the shortest route first. When assigning time slots to each link, SPF_delay considers a time slot assignment scheme that minimizes the transmission delay of the entire route.

SPF_early SPF_early searches the same routes as SPF_delay. However, SPF_early allocates the earliest valid slot directly.

HLS_delay Heuristic list scheduler delay (HLS_delay) [9] considers all routes from source to the destination. Then HLS_delay selects the valid slot which leads to the minimal transmission delay. Finally, we select the minimal-delay route.

HLS_short The basic idea of HLS_short is similar to the HLS_delay. But HLS_short considers the route with the shortest hops when it searching the final decision.

ILP The integer linear programming (ILP) method [8] is a solver-based method, which constructs constraints on the fixed route and enumerate all forwarding time of flows.

B. Scheduling Time Comparison

For ladder topology and RRG topology with 16 switches, we schedule 100 flows at first and then iteratively add 100 flows as step size until the number of flows reach 1000. Fig. 6(a) and Fig. 6(d) show the average time of scheduling one flow on ladder topology and RRG topology respectively. SPF_delay and SPF_early consume the shortest time because they only consider the shortest route. But SPF_early fails when it schedules more than 800 TT flows on the ladder topology. The running time of TTDeep costs $36.846ms$ on average.

TABLE I
THE NUMBER OF SCHEDULED TT FLOWS ON ORION CEV TOPOLOGY

Scenario	SPF_delay	SPF_early	HLS_delay	HLS_short	**TTDeep**
Incremental	1566	1403	1483	1789	**4051**
Link Failure	1566	1393	1444	1673	**3545**

Since HLS_delay and HLS_short have to consider all possible routes, they take longer time to complete the scheduling process. In addition, the ILP method needs to enumerate all slot assignment schemes, which leads to the longest running time and its scheduling time increases exponentially when the number of TT flows increases. Overall, the scheduling time of TTDeep is less than $100ms$, which is applicable [17].

C. Incremental Scheduling Capability Comparison

To evaluate incremental schedule ability, we continuously generate random flow requirements and feed them to each algorithm one by one. We get the maximum number of scheduled flows when one method fails to find legal route or allocates invalid slot. Since ILP's running time is usually unacceptable when facing huge flow requirements, we ignore ILP method in the following experiments.

Fig. 6(b) and Fig. 6(e) show that SPF_delay and SPF_early have the least number of scheduled flows because they only consider one single route. In contrast, HLS_delay and HLS_short improved their performance by selecting routes from a global view. TTDeep schedules the highest number of TT flows since it can not only select routes intelligently, but also consider the slot distribution of each link. Note that TTDeep only uses the the dataset of 16 switches in the training stage. However, it can still achieve fantastic performance when scheduling various topologies with different numbers of switches, which reflects the generality of the TTDeep model.

As for the Orion CEV topology, since the number of devices and links is fixed, we conducted multiple experiments with different flow requirements, table I shows the performance of TTDeep has increased by 126.4% on average compared to the other algorithms. Faced with the Orion CEV topology, other algorithms are prone to encounter bottleneck link problem: many previously scheduled flows pass through certain key links and result in the congestion of these links. In contrast, we have observed that TTDeep can selectively bypass some bottleneck links by sacrificing transmission delay and obtain trade-off with higher number of scheduled TT flows.

D. Link Failure Adaptation Capability Comparison

Link failure is the main reason for network topology changes, which affect TT transmission seriously. We randomly delete a link when the algorithm completes the scheduling of 500 flows. Then all flows passed through the failure link need to be rescheduled. We test whether each method can recover from the link failure first, and then it will continue to schedule new flow requirements until it reaches terminal condition. Fig. 6(c), Fig. 6(f) and table I show that all five methods can recover from link failures and TTDeep still has the best performance and good generality.

VI. CONCLUSION

This paper proposes an incremental schedule model TTDeep, which provides the scheduling scheme for critical frames in real-time Ethernet. We devise a 3-step action selection paradigm to cope with the issue of huge action space and propose the period layer to capture the temporal features efficiently. Comprehensive experiments show that TTDeep can schedule more TT flows than handcrafted heuristics methods and it can well adapt to topology change scenarios.

REFERENCES

[1] IEEE, "Ieee standard for local and metropolitan area networks—bridges and bridged networks—amendment 25: enhancements for scheduled traffic: 802.1 qbv-2015," 2016.

[2] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and R. Obermaisser, "Ttethernet: Time-triggered ethernet," in *Time-Triggered Communication*. CRC Press, 2011.

[3] W. Steiner, "An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 375–384.

[4] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in ieee 802.1 qbv time sensitive networks," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192.

[5] R. S. Oliver, S. S. Craciunas, and W. Steiner, "Ieee 802.1 qbv gate control list synthesis using array theory encoding," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 13–24.

[6] Z. Hanzalek, P. Burget, and P. Sucha, "Profinet io irt message scheduling with temporal constraints," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 3, pp. 369–380, 2010.

[7] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental flow scheduling and routing in time-sensitive software-defined networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, 2017.

[8] N. Wang, Q. Yu, H. Wan, X. Song, and X. Zhao, "Adaptive scheduling for multicluster time-triggered train communication networks," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1120–1130, 2018.

[9] M. Pahlevan, N. Tabassam, and R. Obermaisser, "Heuristic list scheduler for time triggered traffic in time sensitive networks," *ACM Sigbed Review*, vol. 16, no. 1, pp. 15–20, 2019.

[10] G. Stampa, M. Arias, D. Sánchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[11] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[12] Q. Yu and M. Gu, "Adaptive group routing and scheduling in multicast time-sensitive networks," *IEEE Access*, vol. 8, pp. 37 855–37 865, 2020.

[13] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, pp. 1057–1063, 1999.

[14] K. Narasimhan, T. Kulkarni, and R. Barzilay, "Language understanding for text-based games using deep reinforcement learning," *arXiv preprint arXiv:1506.08941*, 2015.

[15] Z. Pang, X. Huang, Z. Li, S. Zhang, Y. Xu, H. Wan, and X. Zhao, "Flow scheduling for conflict-free network updates in time-sensitive software-defined networks," *IEEE Transactions on Industrial Informatics*, 2020.

[16] Wikipedia contributors, "Open shortest path first — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Open_Shortest_Path_First&oldid=992275185, 2020, [Online; accessed 15-December-2020].

[17] Z. Li, H. Wan, Y. Deng, X. Zhao, Y. Gao, X. Song, and M. Gu, "Time-triggered switch-memory-switch architecture for time-sensitive networking switches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 185–198, 2018.