Estimation of TCP Congestion Control Algorithms by Deep Recurrent Neural Network

Takuya Sawada, Ryo Yamamoto, Satoshi Ohzahata, and Toshihiko Kato
Graduate School of Informatics and Engineering
University of Electro-Communications
Tokyo, Japan
e-mail: sawada@net.lab.uec.ac.jp, ryo-yamamoto@uec.ac.jp, ohzahata@uec.ac.jp, kato@net.lab.uec.ac.jp

***Abstract*— Recently, as various types of networks are introduced, a number of Transmission Control Protocol (TCP) congestion control algorithms have been adopted. Since the TCP congestion control algorithms affect traffic characteristics in the Internet, it is important for network operators to analyze which algorithms are used widely in their backbone networks. In such an analysis, a lot of TCP flows need to be handled and so the automatic processing is indispensable. This paper proposes a machine learning based method for estimating TCP congestion control algorithms. The proposed method uses a passively collected packet traces including both data and ACK segments, and calculates a time sequence of congestion window size for individual TCP flows contained in the traces. We use a classifier based on deep recurrent neural network in the congestion control algorithm estimation. As the results of applying the proposed classifier to ten congestion control algorithms, we obtained high accuracy of classification compared with our previous work using recurrent neural network with one hidden layer.**

Keywords- TCP; Congestion Control; Deep Recurrent Neural Network.

I. Introduction

Along with the introduction of various types of networks, such as a long-haul high speed network and a wireless mobile network, a number of TCP congestion control algorithms have been designed, implemented, and widely spread [1]. Since the congestion control was introduced [2], a few algorithms, such as TCP Tahoe [3], TCP Reno [3], and NewReno [4], have been used commonly for some decades. Recently, new algorithms have been introduced and deployed. For example, HighSpeed TCP [5], Scalable TCP [6], BIC TCP [7], CUBIC TCP [8], and Hamilton TCP [9] are designed for high speed and long delay networks. TCP Westwood+ [10] is designed for lossy wireless links. While those algorithms are based on packet losses, TCP Vegas [11] triggers congestion control against an increase of Round-Trip Time (RTT). TCP Veno [12] combines loss based and delay based approaches in such a way that congestion control is triggered by packet losses but the delay determines how to grow the congestion window (cwnd). In 2016, Google proposed a new algorithm called TCP BBR (Bottleneck Bandwidth and Round-trip propagation time) [13] to solve problems mentioned by conventional algorithms.

Since TCP traffic is a majority in the Internet traffic and the TCP congestion control algorithms characterize the behaviors of individual flows, the estimation of congestion control algorithms for TCP traffic is important for network operators. It can be used in various purposes such as the traffic trend estimation, the planning of Internet backbone links, and the detection of malicious flows violating congestion control algorithms.

The approaches for congestion control algorithm estimation are categorized into the passive approach and the active approach. The former estimates algorithms from packet traces passively collected in the middle of network by network operators. In the latter approach, a test system communicates with a target system with a specially designed test sequence in order to identify the algorithm used in the target system. Although the active approach is capable to identify various congestion control algorithms proposed so far, this approach does not fit the algorithm estimation of real TCP flows by network operators. On the other hand, generally speaking, the detecting capability of passive approaches is relatively low comparing with the active approach.

Previously, we proposed a passive method that can estimate a number of congestion control algorithms [14][15]. In this proposal, we focused on the relationship between the estimated congestion window size and its increment. Their relationship is indicated as a graph and the congestion control algorithm is estimated based on the shape of the graph. Our proposal succeeded to identify eight congestion control algorithms implemented in the Linux operating system, including recently introduced ones.

However, the identification is performed manually by human inspectors, and so it is difficult to deal with a large number of TCP flows. So, we proposed a machine learning based classifier estimating the TCP congestion control algorithms using TCP packet traces [16]. It uses a conventional Recurrent Neural Network (RNN) with one hidden layer. From a packet trace, we estimate the relationship of cwnd values and their increment with congestion control algorithm labels, and apply the results to a RNN classifier for training. Using the RNN classifier, we estimate the algorithms for other packet traces. We obtained a relatively good estimation result from the RNN classifier, but we could not classify similar algorithms, such as TCP Reno and Vegas.

This paper proposes a revised version of machine learning classifier for automatic estimation of congestion control algorithms. We adopt a Deep Recurrent Neural Network (DRNN) with multiple hidden layers. We also apply a hyper parameter tuning for the classifier. We pick up ten congestion control algorithms mentioned above and show how those algorithms can be estimated automatically.

The rest of this paper is organized as follows. Section 2 gives some background information including the conventional studies on the congestion control estimation and the machine learning applied for the network areas. Section 3 describes the proposed method and Section 4 gives the performance evaluation results. In the end, Section 5 concludes this paper.

II. Backgrounds

A. *Studies on TCP Congestion Control Algorithm Estimation*

The proposals on the passive approach in the early stage [17-19] estimate the internal state and variables, such as cwnd and ssthresh (slow start threshold), in a TCP sender from bidirectional packet traces. They emulate the TCP sender's behavior from the estimated state/variables according to the predefined TCP state machine. But, they considered only TCP Tahoe, Reno and New Reno and did not handle any of recently introduced algorithms. [20] proposed a method to discriminate one out of two different TCP congestion control algorithms randomly selected from fourteen algorithms implemented in the Linux operating system. This method keeps track of changes of cwnd from a packet trace and to extract several characteristics, such as the ratio of cwnd being incremented by one packet. Although this method targets all of the modern congestion control algorithms, they assumed that the discriminator knows two algorithms contained in the packet trace.

Prior to our previous proposal, the only study that can identify the TCP congestion control algorithms including those introduced recently was a work by Yang et al. [21]. It is an active approach. It makes a web server send 512 data segments under the controlled network environment, and observes the number of data segments contiguously transmitted. From those results, it estimates the window growth function and the decrease parameter to determine the congestion control algorithm.

Our previous proposals [14][15] estimated cwnd in RTT intervals from bidirectional packet traces, in the similar way with the other methods. Different from other methods, we focused on the incrementing situation of estimated cwnd values. From the definition of individual congestion control algorithms, the graph of cwnd increments vs. cwnd has their characteristic forms. For example, in the case of TCP Reno, the cwnd increment is always one segment. In the case of CUBIC TCP, the graph of cwnd increment follows a $\sqrt[3]{cwnd^2}$ curve. In this way, we proposed a way to discriminate eight congestion control algorithms in the Linux operating system.

B. *Studies on Application of Machine Learning to TCP*

Recently, several papers focus on applying the machine learning to TCP analysis. [22] proposes a method to estimate RTT using the fixed-share approach from measured RTT samples. [23] estimates the future throughput of TCP flow using the support vector regression from measured available bandwidth, queueing delay, and packet loss rate. [24] proposes a machine learning based multipath TCP scheduler based on the radio strength in wireless LAN level, wireless LAN data rate, TCP throughput, and RTT with access point, by the random decision forests.

These proposals focused on the control aspects of TCP. As far as we know, our previous work [16] is only an attempt for the congestion control algorithm estimation based on the machine learning.

III. Proposed Method

A. *Estimation of cwnd values at RTT interval*

In the passive approach, packet traces are collected at some monitoring point inside a network. So, the time associated with a packet is not the exact time when the node focused sends/receives the packet. Our scheme adopts the following approach to estimate cwnd values at RTT intervals using the TCP time stamp option.

- Pick up an ACK segment in a packet trace. Denote this ACK segment by *ACK1*.
- Search for the data segment whose TSecr (time stamp echo reply) is equal to TSval (time stamp value) of *ACK1*. Denote this data segment by *Data1*.
- Search for the ACK segment that acknowledges *Data1* for the first time. Denote this ACK segment by *ACK2*. Denote the ACK segment prior to *ACK2* by *ACK1'*.
- Search for the data segment whose TSecr is equal to TSval of *ACK2*. Denote this data segment by *Data2*.

From this result, we estimate a cwnd value at the timing of receiving *ACK1* as in (1).

$$cwnd = \left\lfloor \frac{seq\ in\ Data2 - ack\ in\ ACK1'}{MSS} \right\rfloor \text{ (segments)} \qquad (1)$$

Here, *seq* means the sequence number, *ack* means the acknowledgment number of TCP header, and *MSS* is the maximum segment size (MSS). $\lfloor a \rfloor$ is the truncation of a.

Figure 1 shows an example of cwnd estimation. In this figure, MSS is 1024 byte. Data segments are indicated by solid lines with sequence number : sequence number + MSS. ACK segments are indicated by dash lines with acknowledgment number. When "ack 1" is picked up, data segment "1:1024" is focused on as *Data1* above. ACK segment "ack 2049" responding the data segment corresponds to *ACK2*. The ACK segment before this ACK segment (*ACK1'* above) is "ack 1" again. *Data2* in this case is "2049:3073." So, the estimated cwnd is (2049 – 1)/1024 = 2. Similarly, for the following two RTT intervals, the estimated RTT values are (5121 – 2049)/1024 = 3 and (10241 –5121) /1024= 5.

B. *Selection and Normalization of Input Data to Classifier*

When a packet is lost and retransmitted, cwnd is decreased. In order to focus on the cwnd handling in the congestion avoidance phase, we select a time sequence of cwnd between packet losses. We look for a part of packet trace where the sequence number in the TCP header keeps increasing. We call this duration without any packet losses *non-loss duration*. We use the time variation of estimated cwnd values during one non-loss duration as an input to the classifier. However, the length of non-loss duration differs for each duration, and the range of cwnd values in a non-loss

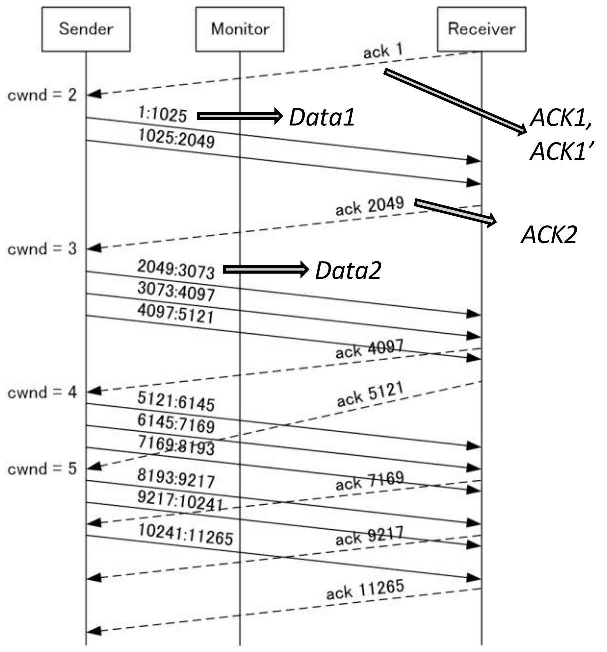


Figure 1. Example of cwnd estimation.

duration also differs from one to another. So, we select and normalize the time scale and the cwnd value scale for one non-loss duration.

The algorithm for selecting and normalizing input to classifier is given in Figure 2. In this algorithm, the input E is as time sequence of cwnd values estimated from one packet trace. The input *InputLength* is a number of samples in one input to the classifier. In this paper, we used 128 as *InputLength*. This is because we think that the cwnd vs time curve can be drawn by 128 points. In the beginning, the time sequence of cwnd is divided at packet losses, and the divided sequences are stored in a two dimensional array *S*. Next, the first sequence *S*[0] is removed, because we focus only on the congestion avoidance phase. Then *S* is reordered according to the length of cwnd sequence. Then the cwnd values for one sequence *S*[t] are normalized between 0 and 1. The normalization is performed in the following way.

Let $w_{max}[t] = \max(S[t][u])$
for $u = 0 \cdots \mathrm{Len}(S[t]) - 1$, and

$$w_{min}[t] = \min(S[t][u])$$

for $u = 0 \cdots \mathrm{Len}(S[t]) - 1$.

Each cwnd value in S[t] is normalized by

$$S[t][u] \leftarrow \frac{S[t][u] - w_{min}[t]}{w_{max}[t] - w_{min}[t]}.$$

After that, the cwnd values are resampled into the number of *InputLength* (128 in this paper). This is done by the loop between step 11 and step 15. As a result, a cwnd sequence in *S*[t] is converted to an array *I*[t] with 128 elements. By this algorithm, all of the time sequences of cwnd values are the arrays with 128 elements whose value is between 0 and 1.

Figure 3 shows some examples of cwnd estimation. Figure 3 (a) and (b) show the estimated cwnd time sequences for TCP Reno and CUBIC TCP, respectively. We focus on the non-loss durations as described above. Reno 1, Reno 2,

```
Algorithm 1
1. function Normalize (E, InputLength)
2.        S <- DivideAtLoss(E)
3.        Delete(S[0])
4.        S <- SortBySequenceLength(S)
5.        for t = 0 to Len(S) − 1 do
6.                S <- MinMaxNormalization(S)
7.        end for
8.        I <- Array(Len(S))
9.        for t = 0 to Len(S) − 1 do
10.               I[t] <- Array(InputLength)
11.               for u = 0 to InputLength − 1 do
12.                       SurjectiveMap <- InputLength/Len(S[t])
13.                       Index <- Trunc(u / SurjectiveMap)
14.                       I[t][u] <- S[Index]
15.               end for
16.       end for
17.       return I
18. end function
```

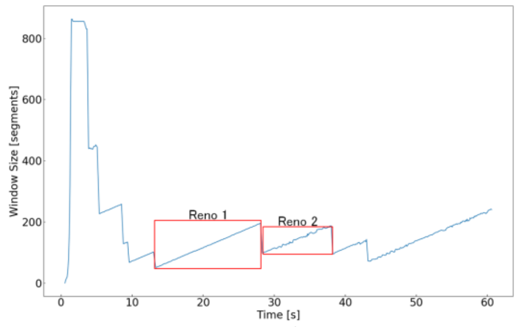
Figure 2. Selection/normalization algorithm.

CUBIC 1, and CUBIC 2 in the figure are examples. The size of these sequences differ from each other, both for the time scale and the scale of cwnd. Therefore, it is necessary to normalize these sequences.

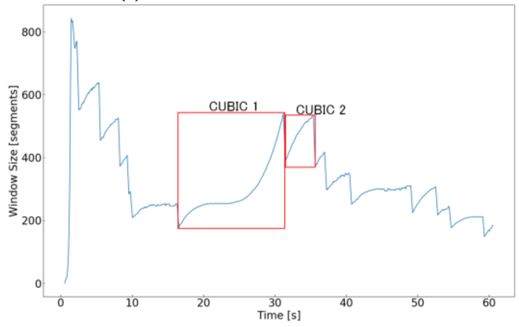
Figure 4 shows the results of the normalization for the examples shown in Figure 3. Different scale of cwnd time sequences are transformed into a canonical form with 128 samples in the range of 0 through 1.

C. DRNN Based Classifier for Congestion Control Algotithm Estimation

We used DRNN for constructing the classifier, which has three hidden layers and whose output layer defines the TCP



(a) Estimated cwnd for TCP Reno



(b) Estimated cwnd for CUBIC TCP

Figure 3. Examples of cwnd estimation.

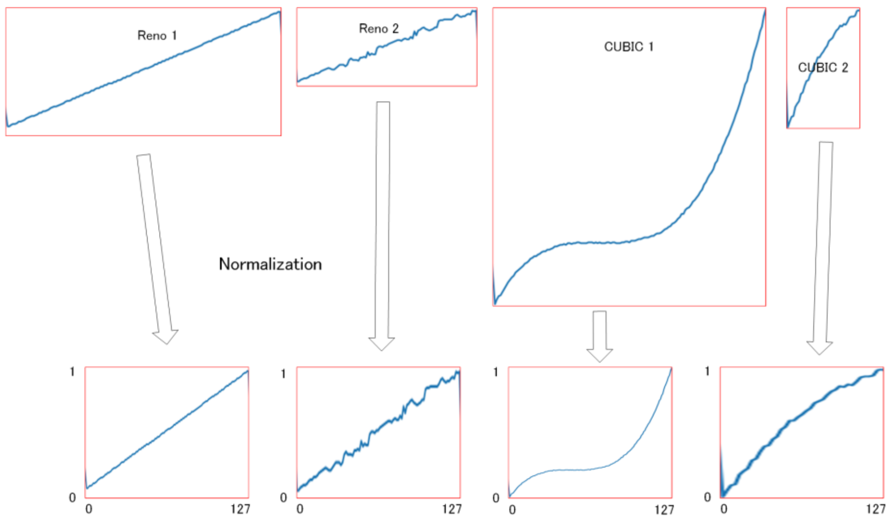


Figure 4. Examples of normalization.

congestion control algorithms. Among the RNN technologies, we pick up the long short-term memory mechanism [25], which was proposed to handle a relatively long time sequence of data. The input is a normalized time sequence of cwnd as described above, with using labels of congestion control algorithms represented by one-hot vector.

In our previous work, we selected the hyper parameters given in Table I. In the work presented in this paper, we select the hyper parameter ranges shown in Table II. The input length is the same as that of the previous work. We use three hidden layers and the number of neurons are as specified in the table. As for the optimizer, the learning rate and the weight decay, we propose the alternatives shown in the table. We perform the hyper parameter tuning based on the target area in this table.

In the training of the classifier, we use the mini-batch method, which selects a specified number of inputs randomly

TABLE I. HYPER PARAMETERS OF CLASSIFIER IN OUR PREVIOS WORK.

Parameter	Value
Input Length	128
Hidden Layers	1
Hidden Neurons	512
Optimizer	Adam
Learning Rate	2×10^{-4}

TABLE II. HYPER PARAMETER RANGES IN THIS WORK.

Parameter	Value
Input Length	128
Hidden Layers	3
Hidden Neurons	1st./2nd: 512, 3rd: 256
Optimizer	Adam or MomentumSGD
Learning Rate	$[10^{-5}, 10^{-1}]$
Weight Decay	$[10^{-10}, 10^{-3}]$

from the prepared training data. The mini-batch size will be determined for individual training. The training will be continued until the result of the loss function becomes smaller than the learning rate.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

Figure 5 shows the experimental configuration for collecting time sequence of cwnd values. A data sender, a data receiver, and a bridge are connected via 100 Mbps Ethernet links. In the bridge, 50 msec delay for each direction is inserted. As a result, the RTT value between the sender and the receiver is 100 msec. In order to generate packet losses that will invoke the congestion control algorithm, packet losses are inserted randomly at the bridge. The average packet loss ratio is 0.01%. The data transfer is performed by use of iperf3 [26], executed in both the sender and the receiver. The packet traces are collected by use of tcpdump at the sender's Ethernet interface. We use the Python 3 dpkt module [27] for the packet trace analysis. We changed the congestion control algorithm at the receiver by use of the sysctl command provided by the Linux operating system.

The targeted congestion control algorithms are TCP Reno, HighSpeed TCP, BIC TCP, CUBIC TCP, Scalable TCP, Hamilton TCP, TCP Westwood+, TCP Vegas, TCP Veno,

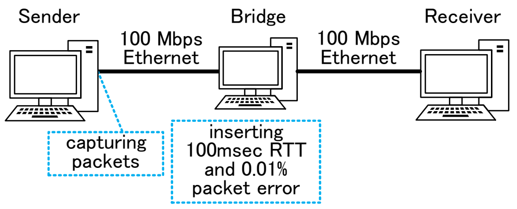


Figure 5. Experiment configuration.

and BBR. We collected more than 1,500 samples for

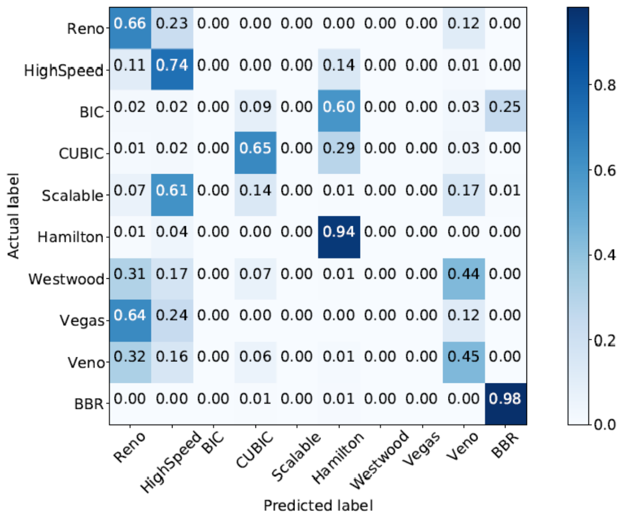


Figure 6. Confusion matrix for previous approach.

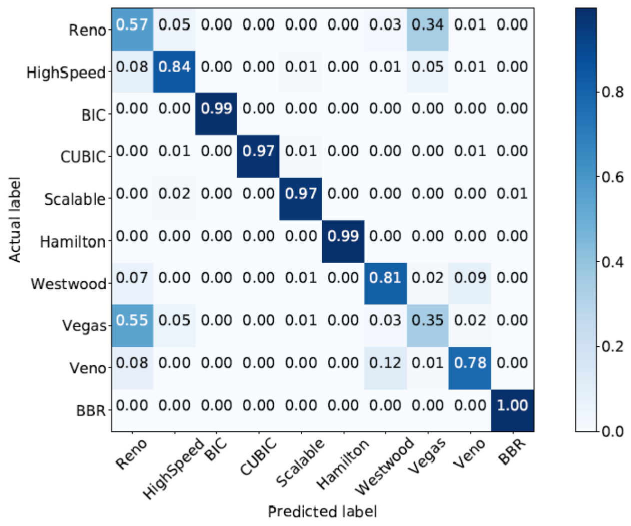


Figure 8. Confusion matrix for this approach.

individual algorithms, and prepared 1,000 samples as training data, 250 samples as verifying data, and 250 samples as test data.

B. *Results of Congestion Control Algorithm Estimation*

First, we re-evaluated the performance of our previous approach. The result is shown in Figure 6. The total accuracy for ten congestion control algorithms was 42.8%, which is rather worse than the result described in our previous paper [19]. This means that our previous classifier will depend largely on the prepared training data.

So, we applied the same training data and verifying data for a DRNN based classifier with three hidden layers and selected optimal values for the hyper parameters mentioned in Table II. We tried to look for optimal values 100 times by the mini-batch method using 256 as the mini-batch size. Table III shows the values of hyper parameters obtained by this tuning.

TABLE III. TUNED UP HYPER PARAMETER VALUES.

Parameter	Value
Optimizer	Adam
Learning Rate	0.0015967736
Weight Decay	2.967486×10^{-8}

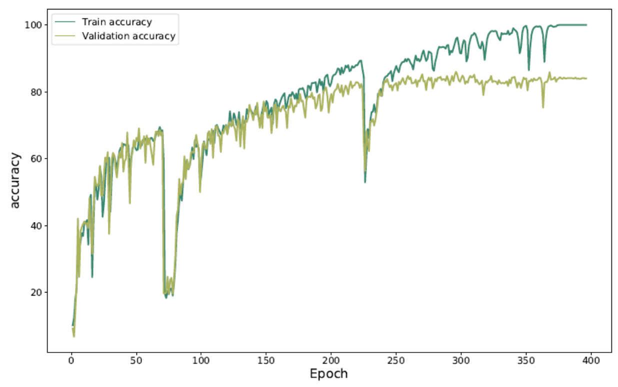


Figure 7. Confusion matrix for previous approach.

Figure 7 shows the learning curve for ten congestion control algorithms using the DRNN based classifier with the selected hyper parameter values. The horizontal axis of this figure indicates the epoch, which is the number of training and verifying trials. The vertical axis indicates the accuracy for the training process and the verifying process. The blue line is the accuracy for the training process and the green line is for the verification process. This result shows that the classifier learns the model for estimating congestion control algorithms. Figure 8 shows the confusion matrix for this experiment. By comparing Figures 6 and 8, we can conclude that the DRNN based classifier estimates the congestion control algorithms much better than our previous classifier. The total accuracy was 82.9%, which is higher than that of our previous work. The only problem is that it still confuses TCP Reno and TCP Vegas. The further studies are required.

As the last analysis, we evaluated the generalization accuracy for our previous work and this work using the 10-fold cross-validation. We divided the training data into ten folds, and selected one fold for the validation and used the rest folds for the training. Figure 9 shows the result. The vertical axis is the validation accuracy. In our previous classifier, the validation accuracy sometimes drops to 40%, although it goes up 70%. On the other hand, our new classifier provides 70% through 80% accuracy stably.

V. CONCLUSIONS

In this paper, we showed a result of TCP congestion control algorithm estimation using a Deep Recurrent Neural Network (DRNN) based classifier. From packet traces including both data segments and ACK segments, we derived a time sequence of cwnd values at RTT intervals without any packet retransmissions. By ordering the time sequences and

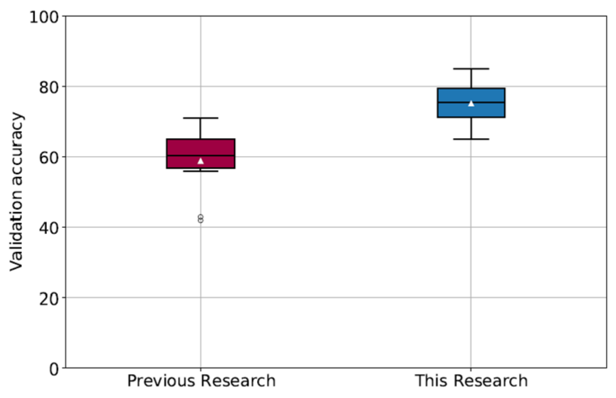


Figure 9. Generalization accuracy of previous work and this work.

normalizing in the time dimension and the cwnd value dimension, we obtained the input for the DRNN classifier. As the results of applying the proposed classifier for ten congestion control algorithms implemented in the Linux operating system, we showed that the DRNN based classifier can estimate ten algorithms effectively, with a problem that TCP Reno and TCP Vegas are difficult to discriminate. This result is much better than our previous classifier that used a simple recurrent neural network.

REFERENCES

[1] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, “Host-to-Host Congestion Control for TCP,” IEEE Commun. Surveys & Tutorials, vol. 12, no. 3, pp. 304-342, 2010.

[2] V. Jacobson, “Congestion Avoidance and Control,” ACM SIGCOMM Comp. Commun. Review, vol. 18, no. 4, pp. 314-329, 1988.

[3] W. R. Stevens, “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algotithms,” IETF RFC 2001, 1997.

[4] S. Floyd, T. Henderson, and A. Gurtov, “The NewReno Modification to TCP’s Fast Recovery Algorithm,” IETF RFC 3728, 2004.

[5] S. Floyd, “HighSpeed TCP for Large Congestion Windows,” IETF RFC 3649, 2003

[6] T. Kelly, “Scalable TCP: Improving Performance in High-speed Wide Area Networks,” ACM SIGCOMM Comp. Commun. Review, vol. 33, no. 2, pp. 83-91, 2003.

[7] L. Xu, K. Harfoush, and I. Rhee, “Binary increase congestion control (BIC) for fast long-distance networks,” Proc. IEEE INFOCOM 2004, vol. 4, pp. 2514-2524, 2004.

[8] S. Ha, I. Rhee, and L. Xu, “CUBIC: A New TCP-Friendly High-Speed TCP Variant,” ACM SIGOPS Operating Systems Review, vol. 42, no. 5, pp. 64-74, 2008.

[9] D. Leith and R. Shorten, “H-TCP: TCP for high-speed and long distance networks,” Proc. Int. Workshop on PFLDnet, pp. 1-16, 2004.

[10] L. Grieco and S. Mascolo, “Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control,” ACM Computer Communication Review, vol. 34, no. 2, pp. 25-38, 2004.

[11] L. Brakmo and L. Perterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet,” IEEE J. Selected Areas in Commun., vol. 13, no. 8, pp. 1465-1480, 1995.

[12] C. Fu and S. Liew, “TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks,” IEEE J. Sel. Areas in Commun., vol. 21, no. 2, pp. 216-228, 2003.

[13] N. Cardwell, Y. Cheng, C. S. Gumm, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-Based Congestion Control,” ACM Queue vol. 14 no. 5, pp. 20-53, 2016.

[14] T. Kato, A. Oda, S. Ayukawa, C. Wu, and S. Ohzahata, “Inferring TCP Congestion Control Algorithms by Correlating Congestion Window Sizes and their Differences,” Proc. IARIA ICSNC 2014, pp.42-47, 2014.

[15] T. Kato, A. Oda, C. Wu, and S. Ohzahata, “Comparing TCP Congestion Control Algorithms Based on Passively Collected Packet Traces,” Proc. IARIA ICSNC 2015, pp. 145-151, 2015.

[16] N. Ohzeki, R. Yamamoto, S. Ohzahata, and T. Kato, “Estimating TCP Congestion Control Algorithms from Passively Collected Packet Traces using Recurrent Neural Network,” Proc. ICETE DCNET 2019, pp. 33-42, 2019.

[17] V. Paxson, “Automated Packet Trace Analysis of TCP Implementations,” ACM Comp. Commun. Review, vol. 27, no. 4, pp.167-179, 1997.

[18] T. Kato, T. Ogishi, A. Idoue, and K. Suzuki, “Design of Protocol Monitor Emulating Behaviors of TCP/IP Protocols,” Proc. IWTCS ’97, pp. 416-431, 1997.

[19] S. Jaiswel, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, “Inferring TCP Connection Characteristics Through Passive Measurements,” Proc. INFOCOM 2004, pp. 1582-1592, 2004.

[20] J. Oshio, S. Ata, and I. Oka, “Identification of Different TCP Versions Based on Cluster Analysis,” Proc. ICCCN 2009, pp. 1-6, 2009.

[21] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu, “TCP Congestion Avoidance Algorithm Identification,” In Proc. ICDCS ’11, pp. 310-321, 2011.

[22] Y. Edalat, J. Ahn, and K. Obraczka, “Smart Experts for Network State Estimation,” IEEE Trans. Network and Service Management, vol. 13, no. 3, pp. 622-635, 2016.

[23] M. Mirza, J. Sommers, P. Barford, and X. Zhu, “A Macine Learning Approach to TCP Throughput Prediction,” IEEE/ATM Trans. Networking, vol. 18, no. 4, pp. 1026-1039, 2010.

[24] J. Chung, D. Han, J. Kim, and C. Kim, “Machine Learning based Path Management for Mobile Devices over MPTCP,” Proc. 2017 IEEE International Conference on Big Data and Smart Computing (BigComp 2017), pp. 206-209, 2017.

[25] S. Hochreiter and J. Schimidhuber, “Long short-term memory,” Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997.

[26] iPerf3, “iPerf - The ultimate speed test tool for TCP, UDP and SCTP,” https://iperf.fr/.

[27] dpkt, “dpkt,” https://dpkt.readthedocs.io/en/latest/.