Recurrent Neural Network-based Prediction of TCP Transmission States from Passive Measurements

Desta Haileselassie Hagos*, Paal E.Engelstad†, Anis Yazidi‡, Øivind Kure§
*†§University of Oslo, Department of Technology Systems, Kjeller, Norway
*†‡Oslo Metropolitan University, Department of Computer Science, Oslo, Norway
§Norwegian University of Science and Technology, Department of Telematics, Trondheim, Norway
Email: *destahh@ifi.uio.no, {*desta.hagos, †paal.engelstad, ‡anis.yazidi}@oslomet.no, §okure@item.ntnu.no

***Abstract*—Long Short-Term Memory (LSTM) neural networks are a *state-of-the-art* techniques when it comes to sequence learning and time series prediction models. In this paper, we have used LSTM-based Recurrent Neural Networks (RNN) for building a generic prediction model for Transmission Control Protocol (TCP) connection characteristics from passive measurements. To the best of our knowledge, this is the first work that attempts to apply LSTM for demonstrating how a network operator can identify the most important system-wide TCP per-connection states of a TCP client that determine a network condition (e.g., *cwnd*) from passive traffic measured at an intermediate node of the network without having access to the sender. We found out that LSTM learners outperform the *state-of-the-art* classical machine learning prediction models. Through an extensive experimental evaluation on multiple scenarios, we demonstrate the scalability and robustness of our approach and its potential for monitoring TCP transmission states related to network congestion from passive measurements. Our results based on *emulated* and *realistic* settings suggest that *Deep Learning* is a promising tool for monitoring system-wide TCP states from passive measurements and we believe that the methodology presented in our paper may strengthen future research work in the computer networking community.**

***Keywords*—Long Short-Term Memory, TCP Congestion Control, Passive Measurement, Recurrent Neural Networks**

I. Introduction

Deep Neural Networks (DNN) [20, 28] are a deep learning architecture trained using new machine learning methods that have shown advancements in a wide range of supervised and unsupervised machine intelligence tasks. In recent years, RNN have become popular focus of research topic in the areas of DNN as diverse as, for example, *speech recognition* [8], *music generation* [4], *text generation* [30], *sentiment classification* [31], and other areas of major advancements. RNNs use input sequences to solve both for prediction [5] as well as classification [2, 19, 21] problems. LSTM [14] is a special kind of RNN *state-of-the-art* architecture designed for a wide range of sequence modeling tasks and time series prediction models. The LSTM unit [14] is a powerful and flexible RNN tool that has a memory cell which gives previous hidden state containing connection information through the hidden layer activations from the past for a long period of time. LSTM in its recurrent hidden layer has a special unit called *memory blocks* consisting of memory cell units that are responsible for remembering the temporal states of the network for an arbitrary time intervals [14]. In each layer of the LSTM architecture [14], there is a forward propagation step with is a corresponding backward propagation through time step. In addition to this, there is a *cache* that passes information from one layer to another. This ability of LSTM [14] allows us to solve the vanishing gradient problem by dynamically controlling the information flow within the layers and capture the long-term dependencies of the connections in a sequence effectively. LSTM [14] is used to address difficult sequence learning and prediction problems in machine learning and have achieved *state-of-the-art* results. One of the main benefits of using an LSTM model for challenges that involve time series data is to avoid the *vanishing gradient* problem. RNN model scans through the training data from left to right and the parameters it uses to govern the connection in the hidden layer for each time-step, learned features during the training are shared and this significantly improves the prediction. An LSTM model computes a mapping from an input feature vector $x = (x_{(1)}, x_{(2)}, x_{(3)}, ..., x_{(n)})$ where $x_i \in \mathbb{R}^n$ to an output sequence $y = (y_{(1)}, y_{(2)}, y_{(3)}, ..., y_{(n)})$ where $y_i \in \mathbb{R}^n$ by calculating the network unit activations of a weighted sum using the Equations 1-6 iteratively from $t = 1$ to n. As it is shown in Equations 1, 2, and 4, LSTM [14] uses *three* adaptive, an *input*, *forget* and *output*, gates shared by all cells in the LSTM block in order to learn long-term dependencies and control the flow of information. The output of these gates multiplicatively influences connections within the memory units. The *input* gate determines the flow of input activations into the memory cell whereas the *output* gate determines the output flow of cell activations into the rest of the network. The *forget* gate determines the extent to which the current value remains in the memory cell of the LSTM unit before it gets gradually discarded when its data is no longer needed.

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$
$$f_t = \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \quad (3)$$
$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \quad (4)$$
$$m_t = o_t \odot h(c_t) \quad (5)$$
$$y_t = \phi(W_{ym}m_t + b_y) \quad (6)$$

Where the i, f, c, o are *input*, *forget*, *memory state*, and *output* gate activation vectors respectively at each time step t. σ is the logistic sigmoid non-linearity while $\odot$, g and h are element-wise product of the vectors, the cell input and output non-linearity activation functions of the entire neural network, *ReLU* in our case, applied to each layer of the deep network respectively. W and b represents a vector of weighted recurrent connections and the bias vector. m_t is the hidden state output of the LSTM layer. Finally, ϕ is the activation function in the hidden layer applied to the network output. Figure 1 describes the basic unit of an LSTM network where the input sequence to the LSTM cell is carried over each *time step* of $t+1$, t and

t-1. As shown in Figure 1, the hidden state, at time step *t*, is a function of the current input sequence x_t at the same time step. C_t and C_{t-1} are the memory cell state activation vectors from the current and previous block at time *t* and *t-1* respectively.

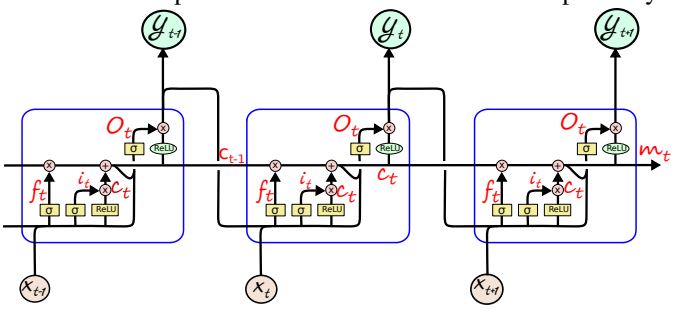


Fig. 1: LSTM Networks. For more thorough details, refer [23]

In this paper, we are interested in the capability of RNN model based on emulated and realistic networks for estimating TCP *cwnd*, as well as the underlying TCP variants within a flow. Hence, we have explored an LSTM architecture for RNN-based prediction approaches to monitor the most important TCP per-connection states from passive measurements related to network congestion. In our paper, we have demonstrated that LSTM can use its memory blocks and a series of gates to effectively capture the patterns of a TCP *cwnd* from passive measurements. Congestion control is a fundamental problem in computer networks. The TCP congestion control algorithms that are widely deployed today perform the most important functionalities related to congestion control such as handling the *cwnd* from the sender-side. In this paper, we investigate and explore questions quantitatively as they apply to problems of network congestion that include: *(i)* How well can we infer the most important TCP per-connection transmission states that determine a network condition from a passive traffic collected at an intermediate node of the network? *(ii)* How can we uniquely track the underlying TCP variant that the TCP client is using from passive measurements? *(iii)* What is the motivation why we need to know which algorithm the TCP sender is using? *(iv)* Is there some action that we would take based on knowing the information of the underlying TCP variant of the sender? *(v)* Which user is responsible for the majority of heavy flow traffic in the network? etc.?

The work in [16] presented an approach to estimate TCP parameters at the sender-side based on packets captured at the monitoring point using a *finite state machine*. The authors have pointed out that the estimation of *cwnd* may have potential errors primarily due to an over-estimation of the Round-trip Time (RTT) and estimation of incorrect window sizes [16]. Another limitation of this work, given the many existing variants of TCP, is that the use of a separate *state machine* for each TCP variant is *unscalable* and we also believe that the constructed *replica* may not manage to reverse or backtrack the transitions taking the tremendous amount of data into consideration. In addition to this, the *replica* may also not observe the same sequence of packets as the sender and ACKs observed at the intermediate node may not also reach the sender. Within the computer networking research community, RNN techniques are potentially useful. After we extensively survey the existing works for monitoring of TCP transmission states from passive measurements, we believe there is not much work on a *scalable* method of predicting the *cwnd* and uniquely identifying the type of the underlying TCP control algorithm from a passive traffic without the knowledge of the sender's *cwnd* for most of the widely used TCP variants using RNN-based techniques. Hence, in this paper, we demonstrate how an intermediate node (e.g., a network operator) can identify the transmission states of the TCP client associated with a TCP flow related to network congestion from a traffic passively measured at an intermediate node using LSTM [14]. Our experimental results demonstrate the feasibility of our prediction model. We believe that our study will be potentially useful to network operators, researchers and scientists in the networking community from both academia and industry who want to assess the characteristics of TCP transmission states related to network congestion from passive measurements. To the best of our knowledge, this is the first work that attempts to apply LSTM [14] for inferring the most important TCP per-connection states that determine a network condition from a passive traffic collected at an intermediate node of the network without having access to the sender. Our prediction model has several benefits over other approaches as we demonstrate in our experimental results.

Our Contributions: The main contributions of our paper are the following:

- We demonstrate how the intermediate node (e.g., a network operator) can identify the transmission state of the TCP client associated with a TCP flow and predict the *Congestion Window (cwnd)* size of the sender from passive measurements using an LSTM recurrent model.
- We explore the applicability of our LSTM-based prediction model by presenting a *robust* and *scalable* methodology to uniquely identify the widely deployed underlying TCP variants that the TCP client is using.
- We show that the learned prediction model performs reasonably well by leveraging a trained knowledge from the *emulated network* when it is applied and transferred on a *real-life scenario* setting. Thus our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community [26].
- We validate the *robustness* and *scalability* approach of our prediction model extensively through several controlled experiments and experimentally verified across an *emulated*, *realistic* and *combined* scenario settings.

II. Motivation

Our work is mainly motivated by the questions presented on Section I. Congestion control algorithms have a critical role in improving the performance of TCP on the Internet [6]. However, when different variants of TCP algorithms coexist on a network, they can potentially influence the performance of each other. One approach to solve this issue is to control the TCP flows individually by predicting the *cwnd* and uniquely identifying the underlying TCP variant.

Benefits: From an *operational perspective*, this information is useful for network operators to monitor if major content providers (e.g., *Google*, *Facebook*, *Netflix*, *Akamai* etc.) are manipulating their congestion windows in their servers to achieve more than their fair share of available bandwidth. Another scenario where operators might find this information useful is if they have a path that they know is congested

due to customer complaints, but the links using that path are not especially over-subscribed. In that case, details about the congestion window behavior of all the users on that path might be helpful in trying to diagnose the cause. From an *ISP perspective*, we believe knowledge about the TCP stack in use in the endpoints is useful for operators of big ISP networks that do much traffic engineering and anomaly detection [12].

Methodological Challenges: In practice; however, predicting TCP per-connection states from passive measurement has a number of difficulties. One of the challenges is, for example, TCP packets can be lost between the sender and the intermediate monitor, or between the monitor and the receiver. If a TCP packet is lost before it reaches the intermediate node and is somehow retransmitted in order, there is no way we can determine whether a packet loss has occurred or not. Therefore, what the intermediate monitor sees may not be exactly what the sender or the receiver sees. The set of methodological challenges we identify involved in performing inference of TCP per-connection states related to network congestion from passive measurements are presented more in detail in [10]. In this paper, we advocate that LSTM-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements collected at an intermediate node by addressing the aforementioned practical challenges.

Roadmap: The rest of the paper is organized as follows: In Section III, we review and give a detailed overview of the closely related research works of TCP passive measurements considered as a *state-of-the-art*. In Section IV, we describe our experimental setup for the evaluation. Section V gives an overview of our methodology highlighting the machine learning techniques, performance measurement metrics used in our paper. Section VI presents detailed experimental results and the multiple scenario settings used to validate our prediction model. Finally, Section VII concludes the paper and outlines directions of research for future extensions.

III. Related Work

This section briefly discusses closely related research works on inferring TCP per-connection states related to network congestion from passive measurements. The techniques to monitor TCP per-connection characteristics are divided into *two* categories: *active* and *passive measurements*.

Active Measurement: Many existing research works that have been proposed rely on an *active* approach to measuring the characteristics of TCP. This technique actively measures the TCP behaviors of Internet flows by injecting an artificial traffic into the network between at least two endpoints [22, 25]. It focuses mainly on active network monitoring and relies on the capability to inject specific traffic which is then monitored so as to measure service obtained from the network.

Passive Measurement: In a passive measurement, passively collected packet traces are examined to measure TCP behaviors of Internet flows [16]. Passive measurement, unlike an active measurement, doesn't inject an artificial traffic into the network. It only measures the network without creating or modifying any real traffic on the network. Passive monitoring measurements are increasingly used by network operators and researchers in the networking community. A work of interest that is most closely related to our work is [16] which provides a passive measurement methodology to infer and keep track of the values of the sender variables: end-to-end RTT and *cwnd*. Their idea is to emulate a *state transition* by detecting Retransmission Timeout (RTO) events at the sender and observing the ACKs which cause the sender to change the value of the *cwnd*. This work [16] considers only the predominant implementations of TCP and the basic idea is it constructs a *replica* of the TCP sender's state for each TCP connection observed at the intermediate node. The replica takes the form of a *finite state machine*. However, the use of a separate *state machine* for each variant is *unscalable* taking the many existing TCP variants into consideration. We also believe that the constructed *replica* [16] cannot manage to reverse or backtrack the transitions taking the tremendous amount of data into consideration. Another limitation is that the *replica* may not observe the same sequence of packets as the sender and ACKs observed at the intermediate node may not also reach the sender. The authors of the study [27] developed a tool, called *tcpflows* that attempts to *passively* estimate the value of *cwnd* and identify TCP congestion control algorithms by analyzing the ACK stream to detect the occurrence of TCP congestion events. However, the *state machine* implemented with *tcpflows* is limited to old TCP variants and hence it cannot uniquely identify new TCP congestion control algorithms.

Our work mainly differs from the previous works in that our main goal is more fundamentally to develop a *scalable* LSTM-based prediction model for inferring TCP per-connection states for the most widely used *loss-based* congestion algorithms. Different TCP stacks come with a variety of features that will violate the assumptions we might make if we only look at one or two TCP variants. Hence, a list of the most widely used *loss-based* variations of TCP algorithms we consider in our work so as to cover the whole scope of the problem are BIC [32], CUBIC [9] and Reno [15].

IV. Experimental Setup and Discussion

In this section, we provide a detailed overview of our experimental testbed.

A. Experimental Testbed

Figure 2 shows the experimental setup that we use for all of our experiments in this paper. In order to introduce congestion, we first created an emulated network and put a communication tunnel across the network and simultaneously push TCP cross-traffic to the network using an *iperf* traffic generator [7]. We carried out the experiment by capturing all sessions on the network when the client and server are sending TCP packets. During a single TCP flow of our experiment, the parameters *bandwidth*, and *delay* are *constant* with a *uniform* distribution. However, since we have the *jitter* given as an average, its distribution is *normal*. We created an identical regular *tcpdump* of the TCP packets on the client node including information about the per-connection *states* so that we can match the *tcpdump* with the TCP *states*. As shown in Figure 2, we used the *measured* TCP data as an *input* to our methodology for a prediction of the TCP per-connection states. Finally, we verified the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel used only for training and generate a new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions.

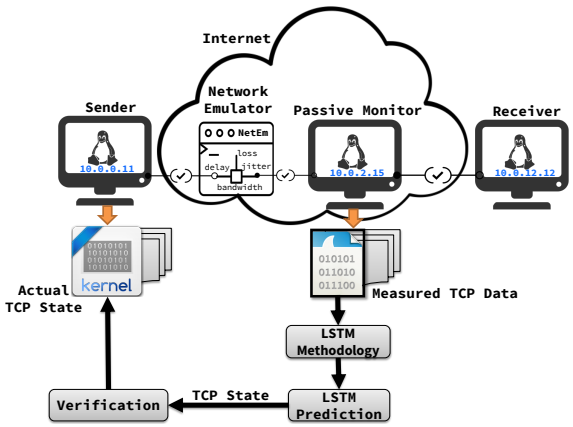


Fig. 2: Experimental Setup

B. Testbed Hardware

We have carried out our experiments using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.4.0-75-generic* kernel release. The prediction model is performed on an NVIDIA Tesla K80 GPU accelerator computing with the following characteristics: Intel(R) Xeon(R) CPU E5-2670 v3 @2.30GHz, 64 CPU processors, 128GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s *Infiniband*, *gigabit* Ethernet and have access to 600 TiB of *BeeGFS* parallel file system storage.

C. Network Emulation and Verification of the emulator

For the network emulation, we used the popular Linux-based network emulator, *Network Emulator (NetEm)* [13] on a separate node, that supports an end-to-end variability of *bandwidth*, *delay*, *jitter*, *packet loss*, and other parameters which the *cwnd* is highly influenced by to an outgoing packets of a selected network interface. Given that the software emulator is not precise, can we trust the network emulator for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* parameters that we change for our evaluation irrespective of the measurement we get from TCP stream? In order to use the network emulator with great care in an extremely well-contained environment for all the variations of the parameters, we created a filter that sets the parameter variation of each packet. As the precision of the emulator cannot be measured from TCP streams, we set up a different experiment using *UDP* to evaluate and measure the precision where both the emulator and traffic generator create variations. We verified the raw performance by measuring the *bandwidth*, *delay*, *jitter* and *packet loss* variations created by the traffic generator and network emulator at the receiver side.

D. Impact of Cross-traffic Variability

We ran *NetEm* [13] with variations in the data rate and the emulation parameters between the client and the server. We have carefully studied and validated the impact of cross-traffic variability from the same TCP congestion protocol on our results by emulating other UDP traffic and we found out that each variation run by the emulator doesn't affect our results. We believe that the variability of the cross-traffic in our current setup will not impact our analysis. In general, when it comes to the *cwnd* variability, it will depend on the particular TCP congestion control in use. We also believe the emulator may be impacted by network elements outside of its scope e.g., CPU load, network card buffers, hardware architectural factors etc.

E. Network Traces

To evaluate our prediction model on both the emulated and realistic network conditions, we have generated our own dataset using *tcptrace* [24]. The data traces for all our experiments are generated using the *iperf* [7] traffic generator on an emulated LAN link where we run each TCP variant with variation of the parameters *bandwidth*, *delay*, *jitter* and *packet loss* as shown below in Table I where the *cwnd* is highly influenced by. However, the kernel might keep the TCP per-connection states of the packets in the buffer and waits for enough amount of packets before sending the TCP states to the userspace. TCP per-connection states might also get lost due to a slow process of TCP by the userspace process. Therefore, the first thing we did as a sanity check is to capture the packets at both the sender and the receiver for it helps us to know whether a packet was lost or just never sent as the ACKs from receiver to sender are just as important as the data packets for inferring packet loss. This way, it is possible to verify if the traffic captures are identical and there are no missing per-connection TCP states. The second thing we carried out in order to avoid missing of packets and capture exactly the same number of packets on the sender and the monitor is tuning the buffer size and flush the buffer to the userspace. We carried out our experiment over a path that is jumbo-frame clean by disabling TCP segmentation offloading so that we can avoid packet sizes way over the regular legitimate size.

F. Network Emulation Parameters

TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT, etc. Therefore, in order to create a realistic scenario, we have emulated the network in our setup as it is shown in Figure 2 by adding variability within a flow to the important network emulation parameters presented in Table I.

TABLE I: Network Emulation Parameters

	Bandwidth (in *mbit*)	**Delay** (in *ms*)	**Jitter** (in *ms*)	**Packet Loss** (%)
1	10	1	0.001	0.01
2	100	2	0.1	0.05
3	300	3	0.2	0.1
4	500	5	0.5	1
5	700	7	1	1.5
6	1000	10	2	2
		[×6]	[×6]	[×6]

G. Assumptions

In TCP, the *cwnd* is one of the main factors that determine the number of *bytes* that can be outstanding at any time. Hence, we assume that using the observed outstanding sequence of *unacknowledged bytes* on the network seen at any point in time in the lifetime of the connection as an estimate of the sending TCP's *cwnd* from *tcptrace* [24] when there is variability of *bandwidth*, *delay*, *loss* and *RTT* is a better approach to estimate the *cwnd* and how fast the recovery is. Firstly, since we are estimating *cwnd* from *bytes in flight*, we have also considered that *cwnd* must be the limiting factor for the sender and it has to be less than the receiver side window. Secondly, we

assume that we don't know what TCP variant is running in the network and the per-connection state within the variant. Lastly, the results we present in this paper assume that the endpoints have the same *receiver window* set by the operating system independent of the underlying TCP variant.

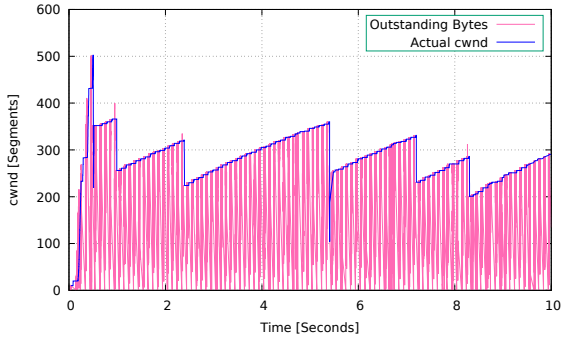


Fig. 3: *Outstanding bytes* calculated from the monitor before applying LSTM technique vs. the actual *cwnd* from the sender

V. Methodology

This section explains the general methodology we have used to experimentally infer both the *cwnd* and uniquely identifying the underlying TCP variant from passive measurement using RNN-based techniques.

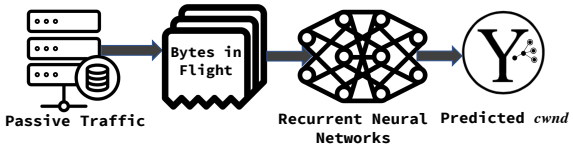


Fig. 4: Methodology for *cwnd* prediction

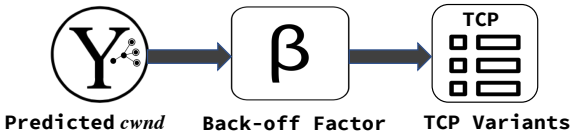


Fig. 5: Methodology for *TCP Variant* prediction

A. Passive Monitoring of bytes in flight

TCP congestion control algorithms govern the TCP sender's sending rate by employing the *cwnd* that limits the number of cumulatively *unacknowledged bytes* that are allowed at any given time. The measured passive TCP data collected at the intermediate node as shown in Figure 2 is used for a training experiment of our model. The TCP implementation details and use of TCP options are not visible at the monitoring point. The TCP sender also keeps track of *outstanding bytes* by two variables in the kernel: *snd_nxt* (the sequence number of the next packet to be sent) and *snd_una* (the smallest unacknowledged sequence number).

B. Prediction of TCP cwnd from Passive Traffic

The *cwnd* is a TCP per-connection state internal variable that represents the maximum amount of data a sender can potentially transmit at any given point in time based on the sender's network capacity and conditions. TCP [15] uses *cwnd* that determine the maximum number of *bytes* that can be *outstanding* without being acknowledged at any given time maintained independently by the sender to do congestion avoidance. Figure 3 shows the comparison between the number of *outstanding bytes* from the intermediate node before running the *neural* model and applying the LSTM techniques versus the actual *cwnd* tracked from the kernel of the sender-side with respect to time. Taking the nature of TCP, accurately inferring *cwnd* of the sender by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node is a challenging task as it is not advertised. One initial approach we tried to estimate the *cwnd* was to process the packet headers of the flows in the *tcpdump* and calculate an aggregate TCP cross-traffic from the trace sets and add that as a feature. We, however, found out during our experiment that turns out to be an insufficient detail for an accurate prediction. In this paper, we argue that training a classifier and prediction model utilizing RNN-based algorithms to predict the *cwnd* from passive measurements is very important.

Learning Context: We built and trained a highly *robust* and *scalable* RNN-based prediction model in *Python* using the *Keras* deep learning framework with a *TensorFlow* backend [1] where we apply an LSTM-based architecture to estimate the *cwnd* trained over multiple epochs with a batch size of *32*. As shown in Figure 1, at each time-step of t, the LSTM model takes an entire array of *outstanding bytes in flight* as an input feature vector (x) indexed by *time stamp* obtained from the kernel. We propagate the input to the model through a multilayer LSTM cell followed by a dense layer of 15-dimensional hidden states with *ReLU* activation that generates an output of a sequence dimensional vector of predicted *cwnd* (y) of the same size indexed by *time stamp*.

Our LSTM network is trained using the *Truncated Back Propagation Through Time* (*TBPTT*) training algorithm for modern RNNs applied to sequence prediction problems [29]. We used this training algorithm to minimize LSTM's total prediction error between the expected output and the predicted output for a given input of the *bytes in flight*. We trained our LSTM-based learning algorithm without the knowledge of the input features from the sender-side during the learning phase. We validated our methodology using the experimental testbed shown in Figure 2 over a LAN link. In order to train and test our prediction model, we employed a single trained network that adapts to all experiments with variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. We have trained our recurrent model on a GPU using the *Adam* stochastic optimization algorithm [18] with the default *learning rate* of *0.001*. We optimize the hyper-parameters (e.g., *Number of epochs*, *batch size*, *the number of time steps to unroll the LSTM during training*, *cell hidden state size* and *the number of LSTM layers*) related to the neural network topology so as to improve the performance of our prediction model. In order to boost our neural network implementation, we used the *ReLU* activation function for the hidden layer. We learn the model from the training data and then finally predict the test labels from the testing instances on all variations of the emulation parameters. Finally, in order to evaluate and measure how well our LSTM-based prediction model performs in terms of capturing the *cwnd* pattern, all neural networks are trained, as it is shown in Section VI, by employing both the *Root Mean Square Error (RMSE)* and *Mean Absolute Percentage Error (MAPE)* loss functions.

C. Prediction of TCP Variants

Our methodology for uniquely identifying the underlying TCP variant from passive measurements by inferring the *multiplicative decrease* parameter, denoted by (β), from the predicted TCP *cwnd* is shown in Figure 5. The standard TCP congestion algorithm employs an Additive Increase and Multiplicative Decrease (AIMD) scheme that backs off in response to a single congestion indication [3]. The AIMD has a linear growth function for *increasing* the *cwnd* at the receipt of an ACK packet and β on encountering a TCP packet loss at the receipt of *triple duplicate ACKs*. This scheme adjusts the *cwnd* by the *increase-by-one decrease-to-half* strategy. The aspect of the AIMD algorithm is generalized and controlled by adding *two* variables, α and β. α indicates the increase in the window size if there is no packet loss in round-trip time and β indicates the fraction of the window size that it is decreased to when packet loss is detected [3]. Let *f(t)* be the sending rate (e.g., the congestion window) during time slot *t*, $\alpha(\alpha>0)$, be the *additive increase* parameter, and $\beta(0<\beta<1)$ be the *multiplicative decrease* factor.

$$f(t+1) = \begin{cases} f(t) + \alpha, & \text{If congestion is detected} \\ f(t) \times \beta, & \text{If congestion is not detected} \end{cases} \quad (7)$$

For the underlying TCP variant prediction task, we consider only *loss-based* TCP congestion control algorithms (e.g., CUBIC [9] BIC [32], and Reno [15]) [11] that consider packet loss as an implicit indication of congestion by the network for a proof of concept. Congestion control in any IP stack doesn't have much information available to drive its algorithm. It has to infer congestion from the history of packet loss and RTT. The β value especially for *loss-based* congestion control algorithms is one of the most important TCP characteristics which determines important conditions of a network congestion like the *cwnd* and *ssthresh* [33]. There are two approaches to measure the β value of a TCP congestion control algorithm: *(i)* using a packet loss event, and *(ii)* using a timeout event. In the presence of a packet loss event, TCP sets both its *ssthresh* and the *cwnd* size to $\beta \times cwnd\_loss$ where *cwnd_loss* is the *cwnd* size before a packet loss event or a timeout occurs. When timeout occurs, TCP sets its *ssthresh* to $\beta \times cwnd\_loss$ and its *cwnd* size to its initial congestion window (*init_cwnd*) size. The *back-off* parameter along with other TCP characteristics can be used to predict the underlying TCP congestion control algorithms. Hence, here we use the β value so as to uniquely predict the underlying TCP variant of the selected *loss-based* TCP congestion control algorithms summarized in Table II.

TABLE II: β Values of *Loss-based* TCP Variants

TCP Congestion Control Algorithm	β Value
BIC	0.8
CUBIC	0.7
Reno	0.5

VI. Experiments and Results

In this section, we summarize in detail the several experimental results that illustrate our main contributions under multiple scenarios using an LSTM-based RNN architecture. In the experimental evaluations, we choose a testing scenario configurations and present CUBIC [9], BIC [32] and Reno [15] in order to make our obtained evaluation results easily readable. We have experimented with several variations (36 configurations for each TCP variant, 216 in total as presented in Table I). Due to space limitation in this paper, we cannot present all the evaluation plots for a total of 216 configurations. Hence, the results reported in this paper for all the scenario settings are for a subset of the selected configurations for a proof of concept as shown in Figures 6, 7, 8, and 9 to verify the accuracy of our LSTM RNN-based prediction model.

The TCP *cwnd* pattern prediction model is evaluated under different configurations of training and testing sample size ratios. As it is shown in the plots below, we found out the *RNN-based* model we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. Figures 6*(a)* and *(b)* don't share the same *bandwidth*, *delay*, *loss* and *jitter* configurations which cause the difference on the maximum number of segments over the course of the connection. For example, if we see on Figures 6*(b)*, it has a *Bandwidth-Delay Product (BDP)* [17] of *700mb*0.01s = 875,000 bytes*. At *1500 byte* segments, that's 583 segments and our emulation shows a maximum of 500-600 segments for *cwnd*. In all the plots shown below we can see, once the timeout occurs, all the packet losses are handled with *fast recovery* in response to 3 *duplicate ACKs*. This is because the *cwnd* does not drop below half of its previous peak. In the results, there is a linear-increase phase followed by a packet loss event where the *cwnd* increases with new arriving ACK. This also demonstrates how the TCP congestion control algorithm responds to congestion events. We can see that the pattern of the predicted *cwnd* generally matches the actual *cwnd* quite well with a small prediction error. We matched both the increasing and decreasing parts of the sawtooth pattern using the precise *timestamp* obtained from the kernel.

A. Emulated Network Setup

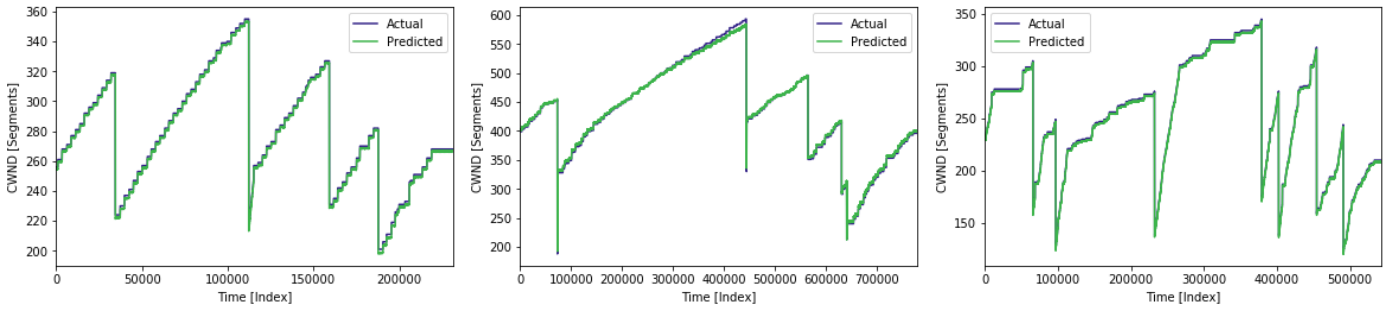
In Figure 6, the comparison of the *predicted* TCP *cwnd* and the actual *cwnd* of the sender in an *emulated* setup is presented. We found out our prediction model captures the ratio of the *cwnd* drop very accurately. We evaluate our TCP *cwnd* prediction model and the performance results with different configurations are presented in Table III. For the TCP variant prediction, we analyzed the β value by averaging out the window size of AIMD algorithm every time we have a peak so that we don't do the computation of the multiplicative decrease factor only on a *slow start* phase. The accuracy of uniquely identifying the underlying TCP variant prediction result in the *emulated* environment as shown in Table V is *97.22%*.

TABLE III: Prediction of *cwnd* on an *emulated* network

Congestion Algorithms	Sample Configurations	RMSE	MAPE (%)
TCP CUBIC	Predicted *cwnd* - C_1	2.181	2.846%
	Predicted *cwnd* - C_2	2.855	3.103%
TCP Reno	Predicted *cwnd* - R_1	2.013	2.815%

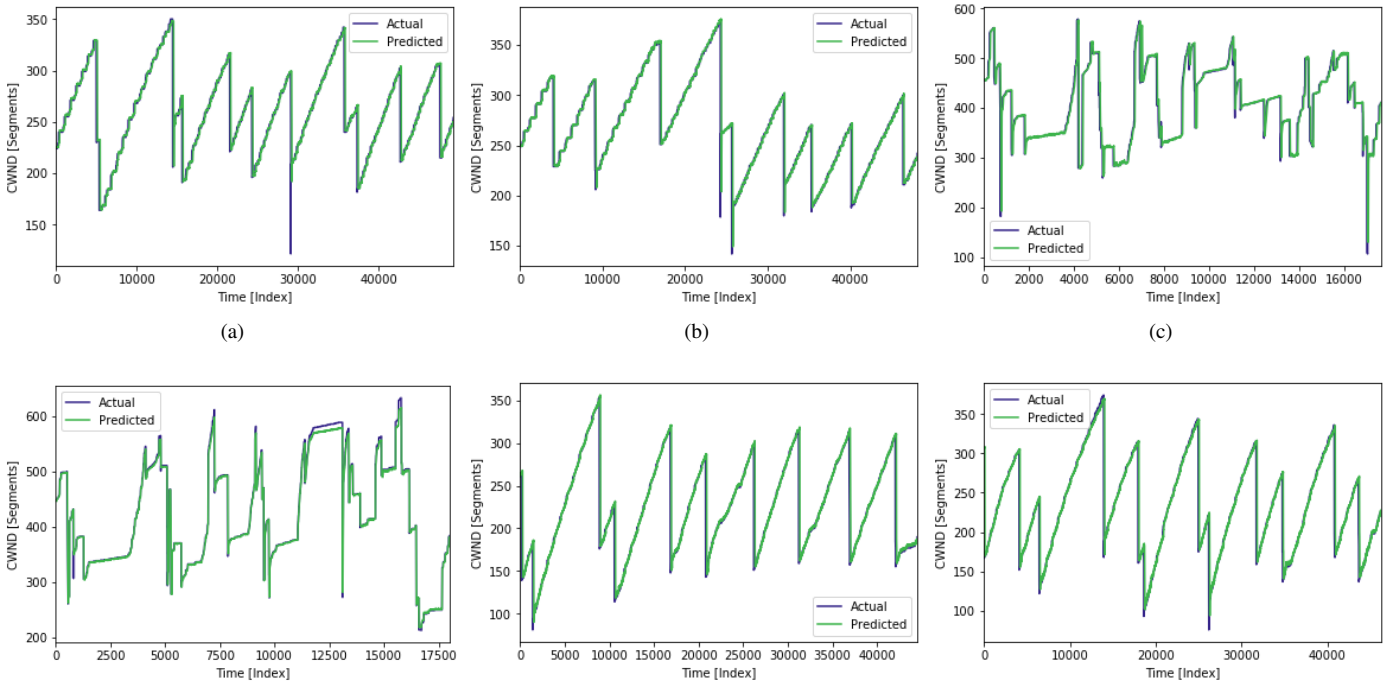
TABLE IV: TCP Variant Prediction of an *emulated network* setting: Confusion Matrix

		Predicted		
Actual	BIC	CUBIC	Reno
BIC	34	0	0
CUBIC	1	35	0
Reno	1	1	36



(a) (b) (c)

Fig. 6: TCP *cwnd* prediction with different configurations in an *emulated network* setting. **(a)** CUBIC [9] Configuration C_1, **(b)** CUBIC [9] Configuration C_2, **(c)** Reno [15] Configuration R_1



(a) (b) (c)

(d) (e) (f)

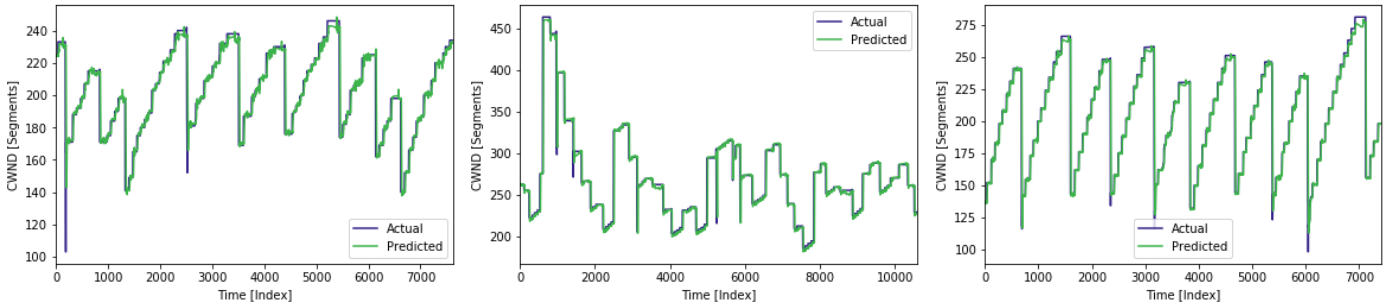
Fig. 7: TCP *cwnd* prediction from a *realistic* scenario setting on different *zones* of *Google Cloud* platform (East coast USA (North Carolina) and Northeast Asia (Tokyo, Japan) sites). **(a)** CUBIC [9], USA site. **(b)** CUBIC [9], Japan site. **(c)** BIC [32], USA site. **(d)** BIC [32], Japan site. **(e)** Reno [15], USA site. **(f)** Reno [15], Japan site.

TABLE V: TCP Variant Prediction of an *emulated network* setting: Performance metrics

	Precision	**Recall**	**F1-Score**	**Support**
BIC	0.94	1.00	0.97	34
CUBIC	0.97	0.97	0.97	36
Reno	1.00	0.95	0.97	38
Average/Total	0.97	0.97	0.97	108
Accuracy	**0.9722**			

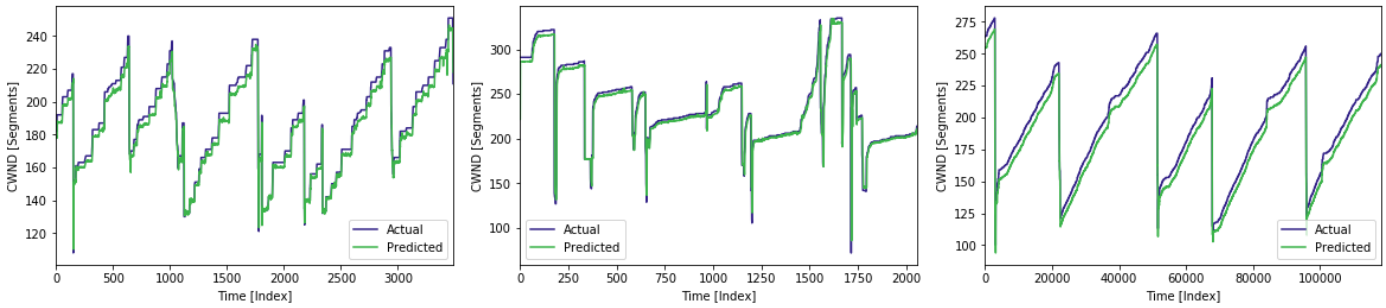
B. Realistic Scenario Setup

In order to demonstrate the *transferability* [26] approach of our proposed machine learning-based prediction model and further validate our results presented in Section VI by conducting a series of controlled experiments against other scenarios, we believe it is necessary to carefully test how well our model using an emulated network works with realistic scenarios by leveraging the knowledge of the emulated network. This guarantees that our prediction model is able to discern the results to unforeseen scenarios. In this experimental scenario, the prediction model is trained where the passive monitor is placed between the sender and the receiver. From an experimental viewpoint, this helps us to justify and guarantee how our model could predict the development of a *cwnd* and the underlying TCP variant with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic testbed where we experiment from *Google Cloud* platform nodes by running our resources on the East coast



(a) (b) (c)

Fig. 8: TCP *cwnd* prediction with different configurations in a *combined network* setting. **(a)** CUBIC [9] Configuration C_1, **(b)** BIC [32] Configuration B_1, **(c)** Reno [15] Configuration R_1



(a) (b) (c)

Fig. 9: TCP *cwnd* prediction across different Google Cloud settings where the *passive monitor* is closer to the *receiver*. **(a)** CUBIC [9] USA Zone, $C_{Receiver}$, **(b)** BIC [32] USA Zone $B_{Receiver}$, **(c)** Reno [15] USA Zone, $R_{Receiver}$

of the USA and Japan as shown in Figure 7. In order to create a realistic TCP session, we uploaded a big *Ubuntu* image to *Google Cloud* platform sites so that we have a full control of the underlying TCP variant on the sender-side and at the same time run a *tcpdump* in the background and capture the whole TCP traffic flow for testing on the source node. We filtered out the host where we send the TCP traffic to. Finally, we calculated the number of *outstanding bytes* from the captured network traffic and run it through our learning model to predict the development of the TCP *cwnd* and variant. As it is shown in Figure 7, we confirm that our prediction model operates correctly and accurately recognizes the sawtooth pattern for realistic scenario settings across different *Google Cloud zones*. This shows that our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community. The *cwnd* prediction performance result of the realistic scenario setting across the *Google Cloud* platforms is presented in Table VI. As it is shown in Table VIII, the accuracy of the TCP variant prediction for this scenario setting is *96.66%*.

TABLE VI: Prediction of *cwnd* on a *realistic* scenario

Congestion Algorithm	Google Cloud Zone	RMSE	MAPE (%)
TCP CUBIC	*USA* Zone	1.752	2.517%
	Japan Zone	1.964	2.852%
TCP BIC	*USA* Zone	2.219	2.979%
	Japan Zone	2.527	3.097%
TCP Reno	*USA* Zone	2.057	3.143%
	Japan Zone	2.975	2.861%

TABLE VII: TCP Variant Prediction of a *realistic scenario* setting: Confusion Matrix

		Predicted		
Actual	BIC	CUBIC	Reno
BIC	20	0	0
CUBIC	0	19	1
Reno	0	1	19

TABLE VIII: TCP Variant Prediction of a *realistic scenario* setting: Performance metrics

	Precision	**Recall**	**F1-Score**	**Support**
BIC	1.00	1.00	1.00	20
CUBIC	0.95	0.95	0.95	20
Reno	0.95	0.95	0.95	20
Average/Total	0.97	0.97	0.97	60
Accuracy	**0.9666**			

C. Intermediate Node Closer to the Receiver Scenario

Our experimental setup for this scenario setting across different *Google Cloud zones* is presented in Figure 10. It is fundamentally difficult to infer the sender's *cwnd* accurately from passive measurements collected close to the receiver. If we try to measure the *cwnd* for the end-to-end path between the sender and the receiver basing our inference on the total amount of *outstanding* bytes, the further away from sender that our passive monitor is, the less likely it is that the packets that our monitor observes will match the packets that are used by the sending host to adjust its *cwnd*. For example, more

hops between the sender and our passive monitor create more opportunities for packets to be lost, reordered or delayed. This means that the information we are using to infer congestion behavior is less reliable and may introduce more opportunities for prediction algorithms to make false inferences. In this scenario, the number of hops are 18 with an average RTT of *137ms* whereas in the *emulated* scenario, the number hops are 3 with an average RTT of *1.8ms*. We believe the data wouldn't reveal what additional packets are in flight from the sender, or which ACKs from the receiver have been received. Because placing the monitor close to the receiver means, we will be seeing the ACKs before the sender does and so we may have more trouble estimating which of the data packets we capture were liberated by which of the ACKs we see. As it is shown in Figure 9, we can see that our prediction model correctly recognizes the sawtooth pattern of the *cwnd*. However, as shown in Table IX, the prediction error is relatively higher as compared to the other scenario settings. This is because of the cases mentioned earlier. For predicting the underlying TCP variant, we can use the same evaluation methodology, applied on the other presented scenario settings, based on measuring the change in *cwnd* size.

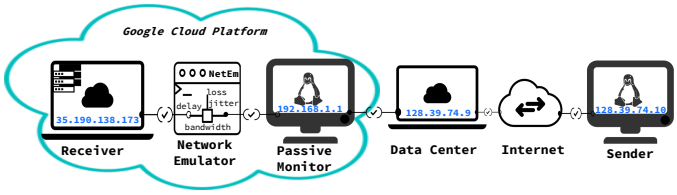


Fig. 10: Intermediate node closer to the receiver scenario setup

TABLE IX: Prediction of *cwnd* across different Google Cloud Zones when the monitor is closer to the receiver

Congestion Algorithms	Google Cloud Zones	RMSE	MAPE (%)
TCP CUBIC	*USA Zone,* $C_{Receiver}$	6.341	9.057%
TCP BIC	*USA Zone,* $B_{Receiver}$	5.185	8.680%
TCP Reno	*USA Zone,* $R_{Receiver}$	6.937	9.238%

D. Combined Scenario Setting

Real networks behave in a more complex manner than emulated networks. The TCP control loop affects the *loss* and *delay* of packets. We believe, there are queue dynamics in the network which cause packet trains and other behaviors which software emulators like *NetEm* [13] can't reproduce well enough. In Section VI-B, we performed a realistic experiment when the random packet loss comes from the dynamics of multiple TCP connections sharing a link (congestion) rather than an injected packet loss. In this section, we address the scalability approach by conducting an experiment of our model under a broader range by combining the realistic and emulated scenario settings to justify the applicability and robustness of our prediction model. Our experimental setup for this scenario setting is presented in Figure 11.

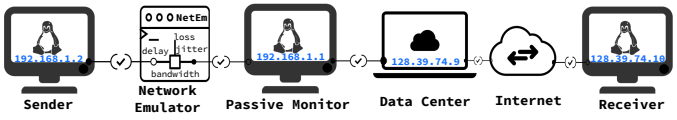


Fig. 11: Combined scenario setup

In this experiment, we combine the two scenario settings (one with an emulator and one with no emulator but Internet) where our intermediate node acts as a router. We get the traffic to the intermediate node, wrap and forward it to the network so that we can add more delay and the number of hops in the network on both sides. In this scenario, as it is shown in Figure 8, both the *increasing* and *decreasing* portions of the sawtooth pattern across different TCP variants is potentially accurate. The TCP variant prediction accuracy of the combined scenario setting, as it is presented in Table XII, is *94.44%* and this justifies that our prediction model can handle multiple scenario settings.

TABLE X: Prediction of *cwnd* on a *combined* setting

Congestion Algorithms	Per Configuration	RMSE	MAPE (%)
TCP CUBIC	*Sample Configuration* C_1	2.072	3.262%
TCP BIC	*Sample Configuration* B_1	3.506	4.846%
TCP Reno	*Sample Configuration* R_1	2.096	3.829%

TABLE XI: TCP Variant Prediction of a *combined scenario* setting: Confusion Matrix

		Predicted		
Actual	BIC	CUBIC	Reno
BIC	33	0	0
CUBIC	2	33	0
Reno	1	3	36

TABLE XII: TCP Variant Prediction of a *combined scenario* setting: Performance metrics

	Precision	Recall	F1-Score	Support
BIC	0.92	1.00	0.96	33
CUBIC	0.92	0.94	0.93	35
Reno	1.00	0.90	0.95	40
Average/Total	0.95	0.94	0.94	108
Accuracy	**0.9444**			

Transfer Learning: In our work, we are able to train in one scenario setting and apply it as a pre-training in another scenario setting. Therefore, we are able to show that the learned prediction model by leveraging a trained knowledge from the *emulated* network performs reasonably well as it is shown above when it is applied and transferred to a *realistic* scenario setting bearing similarity to the concept of *transfer learning* in the machine learning community [26].

Optimality: As it is shown in Tables XIII and XIV, the experimental results show that our LSTM-based prediction model is able to outperform our previous approach using machine learning techniques [10]. Our LSTM-based TCP variant prediction model achieves accuracies of *97.22%*, *96.66%* and *94.44%* on the *emulated*, *realistic* and *combined* scenario settings, outperforming the standard ML-based which yields accuracies of *93.51%*, *95%* and *91.66%* respectively.

TABLE XIII: TCP *cwnd* prediction comparison

Scenario Settings	TCP Algorithms	Configuration	Techniques			
			Machine Learning		*LSTM*	
			RMSE	MAPE	RMSE	MAPE
Emulated	CUBIC	C_1	5.839	6.953%	2.181	2.846%
		C_2	3.075	3.725%	2.855	3.103%
	Reno	R_1	3.511	3.140%	2.013	2.815%
Realistic	CUBIC	USA	4.265	5.134%	1.752	2.517%
		Japan	3.522	4.738%	1.964	2.852%
	BIC	USA	2.952	3.809%	2.219	2.979%
		Japan	2.694	3.761%	2.527	3.097%
	Reno	USA	3.170	5.068%	2.057	3.143%
		Japan	3.396	5.197%	2.975	2.861%

TABLE XIV: TCP variant prediction accuracy comparison

	Scenario Settings		
Techniques Accuracy	Emulated	Realistic	Combined
Machine Learning-based	93.51%	95%	91.66%
LSTM-based	97.22%	96.66%	94.44%

VII. Conclusion and Future Work

In this paper, we have demonstrated the capability of a deep neural network architecture based on a learning LSTM recurrent predictive models to capture the pattern of a TCP *cwnd* with small prediction errors from passive traffic collected at an intermediate node. We have also uniquely identified the underlying TCP variants based on the *multiplicative decrease* window of the *cwnd* and the per-connection states within the variant from passive measurements. Our goal in this work was to implement a learning predictive model that generates the pattern of *cwnd* from passive measurements using an LSTM architecture and finally justify if our previous machine learning based-based experiments are valid. The experimental results show the effectiveness of our LSTM-based prediction approach. We found out that our LSTM-based model outperforms our previous work carried out using the *state-of-the-art* machine learning-based prediction models by a reasonably significant margin. We show that the learned prediction model by leveraging knowledge from the *emulated network* performs reasonably well when it is applied on a *real-life scenario* setting bearing similarity to the concept of transfer learning in the machine learning community. Finally, we believe that our work can open up the path to a number of future research work directions in the computer networking community.

In this work, we consider only *loss-based* TCP congestion control algorithms that consider packet loss as an implicit indication of congestion by the network for a proof of concept. By design, unlike *loss-based* algorithms, the *multiplicative decrease* parameter of *delay-based* congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from a passive traffic when there is variability in *delay*. As a future work, it would be interesting to develop a *delay-based* model using both machine learning and deep learning techniques so as to verify how delay changes and look into how the TCP variants of delay-based congestion control algorithms can be predicted both from passively measured traffic and real measurements over the Internet. We plan to investigate these issues further and extend the approaches in our future work.

Acknowledgment

We greatly acknowledge the anonymous reviewers for their helpful feedback and detailed comments on our paper.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu. Recurrent neural networks for multivariate time series with missing values. *Scientific reports*, 8(1):6085, 2018.

[3] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 1989.

[4] K. Choi, G. Fazekas, M. Sandler, and K. Cho. Convolutional recurrent neural networks for music classification. IEEE, 2017.

[5] J. T. Connor, R. D. Martin, and L. E. Atlas. Recurrent neural networks and robust time series prediction. *IEEE*, 1994.

[6] N. Dukkipati, Y. Cheng, and A. Vahdat. Research Impacting the Practice of Congestion Control, 2016.

[7] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[8] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Machine Learning*, 2014.

[9] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS*, 2008.

[10] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A Machine Learning Approach to TCP State Monitoring from Passive Measurements. In *Wireless Days (WD)*. IEEE, 2018.

[11] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. In *to apppear in ICCCN 2018*. IEEE, 2018.

[12] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing Security Attacks Analysis Using Regularized Machine Learning Techniques. In *AINA 2017*. IEEE, 2017.

[13] S. Hemminger et al. Network emulation with NetEm. 2005.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*. ACM, 1988.

[16] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *INFOCOM*. IEEE, 2004.

[17] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[18] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, 2015.

[20] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[21] P. Liu, X. Qiu, and X. Huang. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101*, 2016.

[22] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM*, 2005.

[23] C. Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs, 2015.

[24] S. Ostermann. Tcptrace. http://www.tcptrace.org, 2000.

[25] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[26] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions*, 2010.

[27] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[28] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[29] I. Sutskever. Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*, 2013.

[30] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *Machine Learning*, 2011.

[31] D. Tang, B. Qin, and T. Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.

[32] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. IEEE, 2004.

[33] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE*, 2014.