REDUCING TCP RETRANSMISSIONS BY USING MACHINE LEARNING FOR MORE ACCURATE RTT ESTIMATION

by

Bikramjit DasGupta, B.S.

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Engineering
December 2019

Committee Members:

Stan McClellan, Chair

Damian Valles

Georgios Koutitas

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

I would like to thank Dr. McClellan, who was always there to guide me, Dr. Valles, who had faith in my endeavors, my parents, Subhajit and Indrani, who have always set an example for all virtues I value, my love, Nikki, and of course, Van Jacobson.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

When computers communicate via connection-oriented protocols like TCP, upon receipt of a segment of information, the recipient sends an acknowledgment back to the sender. This facilitates two important traits of connection-oriented protocols: being aware of which segments have been lost in the transfer, and the exact round trip times (RTT). By observing previous RTT values, algorithms are used to estimate what future retransmission time-out (RTO) values should be set to, i.e., if an acknowledgement is not received within the RTO window after sending a segment, then the segment should be considered lost and resent. This paper proposes a neural networks approach to prediction RTTs. By comparing an RNN-LSTM, and a CNN-LSTM, versus Jacobson's Algorithm, both the RNN-LMST and the CNN-LSTM was shown to provide a better RTT estimate. By replacing the predictor used by Jacobson's Algorithm with a neural network predictor, the number of segment retransmissions was reduced by more than 90%.

I. MOTIVATION AND BACKGROUND

In connection-oriented transport protocols such as Transmission Control Protocol (TCP) (1), when a message is sent to a recipient, an acknowledgment packet is sent back from the recipient to the sender. The total time between the the original packet being sent and the acknowledgment packet being received by the sender is known as the Round-Trip Time (RTT). Depending on the the specific implementation of the protocol, a variable known as the Retransmission Time-Out (RTO) threshold is calculated; this RTO threshold defines how long the sender should wait for the acknowledgment packet before it can assume the segment is lost and resend the segment. By analyzing previous RTTs, the sender can anticipate an unusually high RTT time which, for example, could indicate a lost segment. Depending on the implementation of a connection-oriented transport protocol, the forecasted RTT values, and optionally various other parameters, such as a minimum bound on the RTO, are used to generate the RTO values for future segments. The predominantly popular algorithms used in transport protocols are Jacobson's Algorithm and modifications of it.

Thesis Structure

This thesis is organized into 8 main chapters. Chapter 1, titled "Motivation and Background" provides an overview of previous and similar research on the topic of retransmission time-out manipulation, and a brief description of Jacobson's Algorithm. Chapter 2, titled "Collecting Data" describes the methodology used to collect the data used to train and benchmark the proposed algorithm against

existing algorithms. Chapter 3, titled "Using RNN-LSTM to predict RTT", gives an overview of what a RNN-LSTM is and how one was designed to predict RTT values. Chapter 4, titled "Using CNN-LSTM to Predict RTT", describes what a CNN-LSTM is and how one was designed to predict RTT values. Chapter 5, titled "Modifying Jacobson's Algorithm", describes how Jacobson's Flow Control Algorithm was modified to use the proposed LSTM predictors in lieu of the vanilla time series predictor. Chapter 6, titled "Results", presents the RNN-LSTM and CNN-LSTM performances against the vanilla Jacobson Algorithm's performance over several different benchmarks. Chapter 7, titled "Conclusion", wraps up the previous chapters. Finally, Chapter 8, "Future Works", outlines possible implications and opportunities in this field of research.

Literature Review

The following section gives an overview of the research upon which this thesis builds. We then proceed to compare the different approaches suggested by the research.

Manipulating RTO-Min to Reduce SCTP's Time to Complete

In (2), the author provides insight into optimization techniques for SCTP's retransmission mechanisms. The author presents a novel algorithm to dynamically determine minimum bounds for RTO thresholds, which improves performance and reduces retransmissions. This research describes the Fast Retransmission Mechanism and the Time-Out Mechanism. The Fast Retransmission Mechanism does the following: whenever four Selective Acknowledgments (SACKs) report the same missing chunk during a transmission stream, the client bundles the chunk with the next available packet for retransmission. The Fast Retransmission Mechanism,

however, is optimized for congestion avoidance. In contrast, the Time-Out Mechanism focuses on detecting link failures. Where the Fast Retransmission Mechanism cannot distinguish between link error and congestion, the Time-Out mechanism handles congestion and link error differently. However, the author highlights that a major problem with the Time-Out Mechanism is that the lower bound on RTOs - the RTO-min, is set too high. Specifically, the author concludes that the RTO-min should be set just above where the Fast Retransmission mechanism becomes ineffective.

Using Machine Learning to Improve Congestion Control

In (3), the authors present a machine learning solution to congestion control in wireless networks. Congestion is when routers are unable to process the volume of data that arrive to them quickly enough which leads to buffer overflows and ultimately loss of packets. The only way to relieve congestion is to reduce network load. Currently, TCP congestion protocol operates in the following manner: increase traffic rates steadily until loss occurs, at which point reduce it suddenly (1). This works well for wired connections since loss occurs almost entirely because of buffer overflows, but for wireless networks, this occurs mostly because of signal fading. This paper presents several machine learning solutions to create classification models for signal loss causes. Using the results of the experimentation, the group creates a modified TCP kernel module that is optimized to account for the signal loss models. Their module is shown to perform with higher throughput as compared to the original unmodified TCP module.

Enhancement of TCP using Packet Loss Classifiers

In (4), the authors highlight how TCP cannot differentiate between losses due to congestion and losses due to link errors. The authors modify neither TCP's congestion control mechanism, nor TCP's error recovery mechanism. Using the classification method proposed, packet losses due to link error are ignored by TCP's congestion control, but packet losses classified as congestion loss trigger both congestion control and error recovery mechanisms. This research serves to provide a reasonable basis for the assumption that different loss types warrant different approaches for mitigation. However, where this research does not modify TCP's congestion control mechanism, this thesis will attempt to modify SCTP's RTT forecasting method.

Estimating Retransmission Timeouts in IP-Based Transport Protocols

In (5), the authors look at how TCP and SCTP protocols use previous round trip times to estimate future round trip times and so creates a maximum waiting time before attempting to retransmit. Estimation of the maximum round trip time is done by the Jacobson algorithm, which leads to loss in performance due to a "mismatch between theory and the application area." With every chunk of SCTP data transmitted, a retransmission timer is started that is calculated via the Jacobson Algorithm. The author compares the Jacobson Algorithm, the Eifel Algorithm, and a modified Jacobson algorithm and compares the performances.

Jacobson's Algorithm and Causes of Retransmission

Jacobson's Algorithm uses the Chebyshev Bound (6) to create a reasonable RTO value. Chebyshev's Bound guarantees that for any statistical distribution, 75% of data values will lie within two standard deviations from the mean and 89% of all data values will lie between three standard deviations from the mean. There are two main reasons for retransmission: when a packet is legitimately lost in transit, and when, due to delay, the RTT exceeds the RTO bound created by Jacobson's Algorithm using the Chebyshev inequality. However useful this mechanism is for estimating bounds for RTO, Jacobson pointed out that packets were retransmitted due to delays in transit and not lost packets when loads exceeded 30%, leading to poor performance.

As described in (1), Jacobson's Algorithm involves four steps to generate RTO values. First, using Equation 1.1, the error (ERR) is calculated as the difference in the observed RTT and the last predicted RTT or Smooth Round Trip Time (SRTT):

$$ERR = |RTT_n - SRTT_{n-1})| \tag{1.1}$$

Next, the new prediction is generated utilizing Equation 1.2.

$$SRTT_n = 0.85(SRTT_{n-1}) + 0.125(RTT_n) \tag{1.2}$$

The variance (VAR) is then calculated with Equation 1.3.

$$VAR_n = 0.75(SRTT_{n-1}) + 0.25(ERR) \tag{1.3}$$

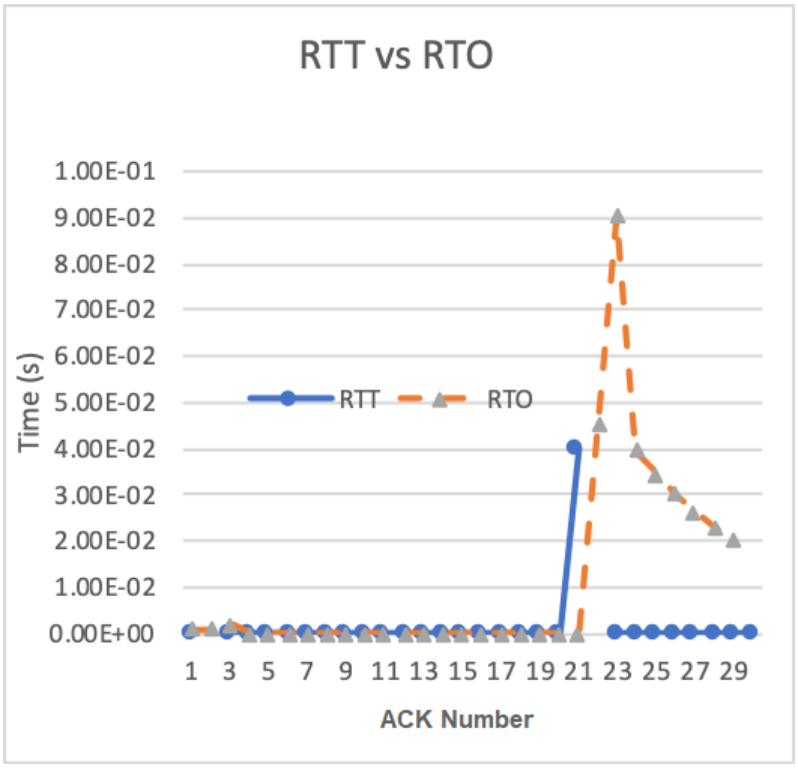

Figure 1.1: RTO adjusting as RTT increases.

Chebyshev's inequality uses the running mean and the standard deviation derived from the variance to state the probability of the difference between the expected value and RTT. Also, it is important to note that provided a distribution has a known variance and known finite mean, Chebyshev's Inequality can be applied to it (6). The RTO is thus generated Equation 1.4 (1)

$$RTO = SRTT_n + 4(VAR_n) \tag{1.4}$$

Figure 1.1 shows a sample of recorded RTT and the subsequent RTO calculations obtained by using Jacobson's Algorithm. As can be seen, Jacobson's Algorithm is purely reactive by changing RTO values only when it sees a change in RTT. RTOmin and RTOmax are bounding parameters for RTO and can also be used to manipulate congestion control efficiency.

Since TCP's implementation in 1988, one form or another of Jacobson's Algorithm has been used for congestion control. However, as useful as it may have proved to be, Jacobson's Algorithm is designed to be fast, but yields accuracy to be so. For example, when the retransmission timer expires, the next RTO is increased exponentially. In the case of a congested network with high latencies, this helps to decrease congestion caused by multiple copies of a segment being sent, but in the case of a link error, the TCP stack will find itself wasting large amounts of time waiting for an acknowledgement. Also, Equation 1.2 which calculates the new predicted RTT, steps up and down gradually, and therefore leads to large amounts of resent segments when RTT spikes dramatically.

With a better predictor algorithm, i.e., a neural network forecaster, patterns in dramatic latency increases can be learned. Instead of stepping up predicted RTT slowly, as with Jacobson's Algorithm, it can immediately set the RTT prediction according to patterns it has seen before. If done correctly, when a segment exceeds the RTO time and is retransmitted, the RTO estimate will be immediately set as high as needed, and the algorithm will wait a longer time for a response. Though this will provide an optimized, less-wasteful sequence of communication, the computational complexity of a large neural network can be too slow for a kernel-level communication module. However, as specialized application-specific integrated circuits (ASICs)(7) are being developed for faster neural networks computation, once unfeasible methods for RTO calculation may soon become a reality. Where the Jacobson's Algorithm is designed to operate fast enough to calculate RTO estimates in between TCP transmissions, it is becoming more and more possible for advanced neural networks to forecast RTO values in similar time.

II. COLLECTING DATA

To compare any proposed solutions to existing TCP implementations, greatly varied control data had to be created. To capture the data, several data transfer patterns were recorded. First, a multiplayer first-person shooter game was started alongside Wireshark, a network packet analyzer. Several minutes of the network data were captured using Wireshark (8). This data included RTT information. The data collected while communicating with the multiplayer game server showed few lag spikes but had slowly increasing and slowly decreasing RTT patterns. Next, a remote file transfer was initiated, and the RTT data of the transfer was recorded. The RTT data collected in this manner showed patterns of periodic and thus predictable latency spikes. Then, several hours of data were collected from interactive web browsing on a single site, and once analyzed, yielded RTT data that was very noisy(9).

Finally, as shown in figure 2.1, to simulate a highly congested network, a File Transfer Protocol (FTP)(10) file transfer was initiated between the Host and Client computers connected via a network switch with varying chunk sizes. While this large file transfer was running, Device A and Device B initiated a Stream Control Transmission Protocol (SCTP)-Test benchmark. SCTP(11) is a computer networking communication protocol similar to TCP and UDP, in the way that it is message oriented like UDP and offers congestion control for in-sequence messages like TCP. These SCTP chunks' RTTs were then recorded. From these RTT values, Jacobson's Algorithm was used to create the RTO values that the TCP and SCTP protocols should be using. Since each communication pattern used was very different in nature, it helped to train the Recurrent Neural Network Long

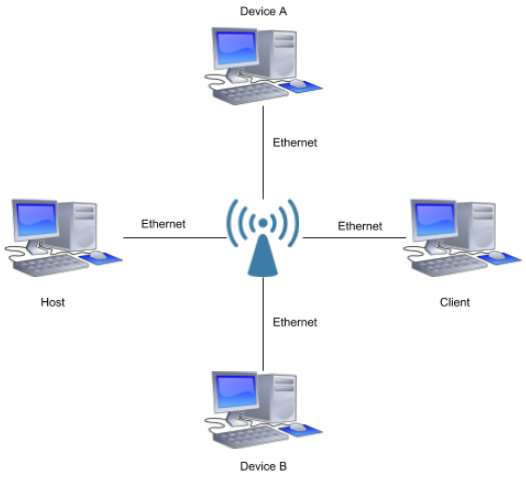

Figure 2.1: Congestion Data Collection.

Short-Term Memory (RNN-LSTM) and Convolutional Neural Network Long Short-Term Memory (CNN-LSTM) models to be able to recognize each pattern. For example, when identifying the leading edge of a latency spike, an LSTM must "remember" what the maximum RTT value was for previous latency spikes but must be able to forget these maximum values when differently shaped latency spikes caused by different communication patterns emerged. Also, the frequency with which latency spikes occur may change over time, and to be able to identify long-term patterns in the data, it needs many different communication patterns in the input data set. Figure 2.2 shows what the entire RTT dataset looks like with the various communication patterns included.

The data collected were combined into one time series and used as the training set (9). The data collection process was run three more times to collect the testing data sets. However, when collecting the testing sets, the order of each communication pattern was changed. This was done to make sure that the testing scores of each LSTM were not affected by the order in which each communication pattern appeared in the time series.

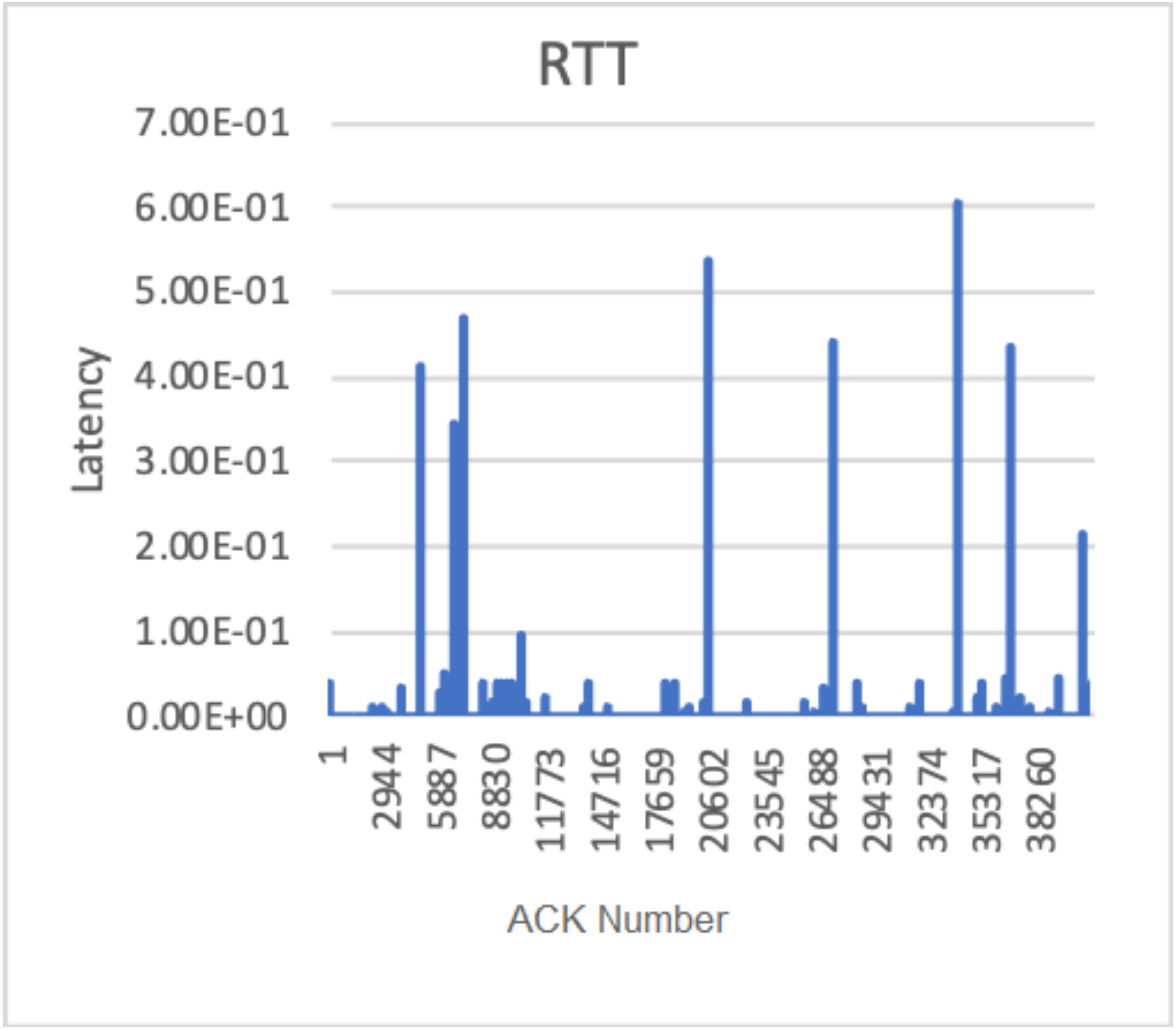

Figure 2.2: Entire RTT dataset.

III. USING RNN-LSTM TO PREDICT RTT

Long short-term memory (LSTM)(12) is a Recurrent Neural Network (RNN)(12) architecture that uses feedback connections that allow it to solve many kinds of general computing problem. They excel at classification and forecasting for time-series data while overcoming the vanishing gradient problem. As shown in (13), since estimating optimal RTO values from previous RTT values is a univariate time-series forecasting task, an RNN-LSTM is well-suited for making such accurate estimates(12).

RNN-LSTM Architecture

As shown in Figure 3.1, a 'Neuron' in an LSTM is actually a logical block that in which a sigmoid activation unit triggers one of three possible gates: a 'forget' gate can remove old information from the block, an input gate can decide which information from the input to store in the block, and an output gate that creates an output using the input and the current information held in the block. By its nature, an RNN is recurrent; hence, information stored in it is kept with minor linear changes. As shown in Figure 3.2, the logical block of the LSTM passes the information stored in it and the last output as inputs to the next iteration. Depending on the combination of inputs, i.e. the last output, the next input, and the current information in the block, the block is then triggered to either store new information, to drop the old information, or to keep the information stored in it. In this manner, patterns in univariate data can be remembered or selectively forgotten, allowing the algorithm to operate with very long time-series.

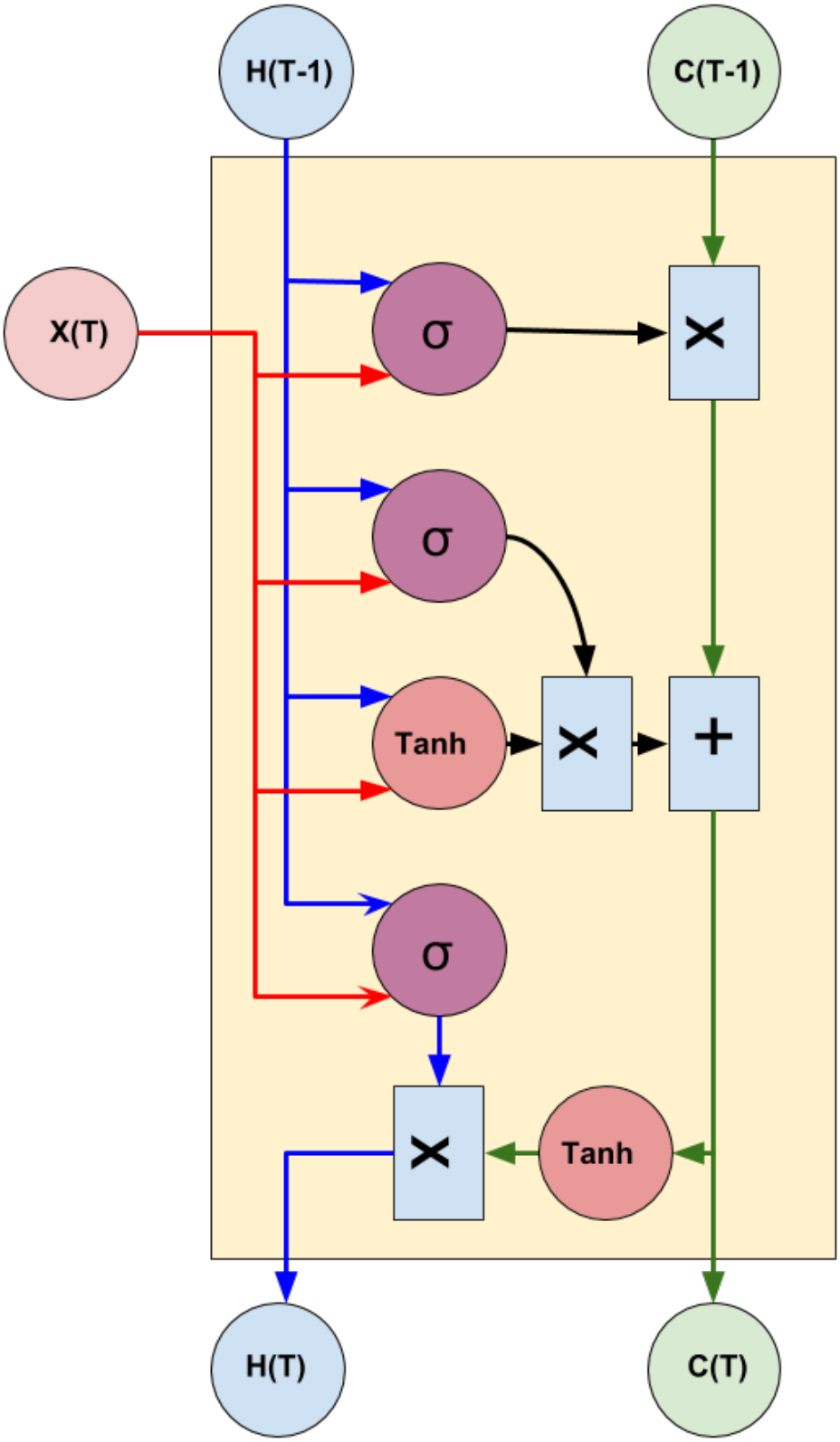

Figure 3.1: An LSTM Block.

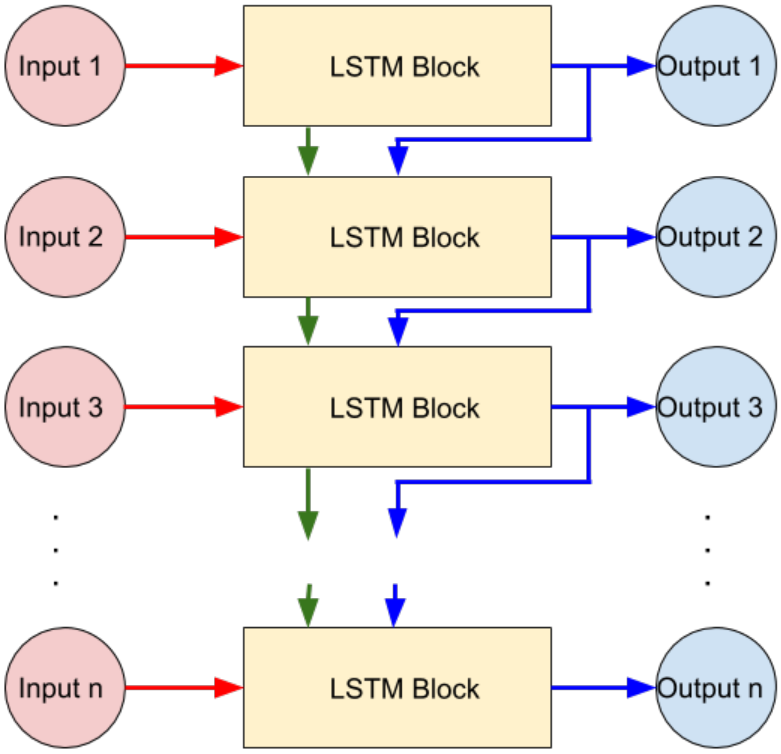

Figure 3.2: RNN-LSTM Architecture.

RNN-LSTM Hyperparameters

All time-series forecasting methods require a look-back variable. The look-back variable determines how many previous inputs to consider when predicting the next output. By increasing the look-back, more accurate predictions can be made, at the cost of computational complexity. Due to the high density of the RTT data, the look-back size had to be relatively large, as too small of a look-back would not allow the LSTM to recognize patterns. However, since dramatic changes in RTT values occurred infrequently, increasing look-back was not a good way to identify patterns either. As shown in Table 3.1, various look-back sizes were tested, and ultimately, an input layer of size 188 was chosen. One hundred eighty-eight input nodes allowed the LSTM to identify patterns in sparse data while still requiring a reasonable computational complexity. Starting at 10, the number of input nodes was

Table 3.1: RNN-LSTM Hyperparameters

Parameter	**Value**
Batch Size	200
Input Layer Nodes	188
Hidden Layer Nodes	36
Output Layer Nodes	1
Learning Rate	0.020

incremented, and accuracy was recorded. Below 188 input nodes, the model's accuracy was still increasing and the past 188 input nodes, the model was not performing any more accurately.

Table 3.1 shows the hyperparameters for the RNN LSTM. The batch size of a network is the number of samples that can be shown to the network before a weight is updated. The batch size was kept at 200 samples, since smaller values led to overfitting and more volatile learning and larger values led to the network not learning to recognize patterns quickly enough. The learning rate of a network controls the rate at which a model learns from input data - more specifically, the amount that weights are updated by back-propagation. Low learning rates lead to linear improvement in accuracy. Higher learning rates lead to faster decay of loss but generally converge to lower accuracy. The learning rate was initialized at 1.0 and decreased by decrements of 0.1 until the optimal learning rate was achieved. Since the number of test samples was relatively high, the optimal learning rate converged to 0.020.

IV. USING CNN-LSTM TO PREDICT RTT

A CNN-LSTM uses a convolutional neural network (CNN) for feature extraction and feeds them to an LSTM which then carries out the sequence prediction. CNN-LSTMs are best for time-series prediction problems involving computer vision, but they are powerful univariate forecasters as well. As shown in figure 3.1, a CNN-LSTM involves a convolutional neural that feeds extracted data into an LSTM, which then forecasts the univariate data. The key difference between an RNN-LSTM and a CNN-LSTM is the convolution block that can take minimally preprocessed data, creates its own filters that would otherwise have to be hand-engineered and identify patterns. A CNN can be particularly useful when data sizes are massive - which makes it great for analyzing patterns in RTT data from hundreds of thousands of segments, similar to time series forecasting techniques detailed in (14).

CNN-LSTM Architecture

The structure of the CNN portion of the CNN-LSTM is shown in Figure 3.2. Applying a convolution operation to the input data allows a neural network to learn features and classify data with only a few neurons in a shallow architecture, where a fully-connected feedforward neural network would have to be very deep with many more neurons to match the accuracy performance of a CNN. The first convolution layer effectively extracts low-level features. In 2D image recognition problems, the first convolution layer helps to identify features such as edges; in time-series forecasting, they help to identify edges in the data patterns of a univariate dataset.

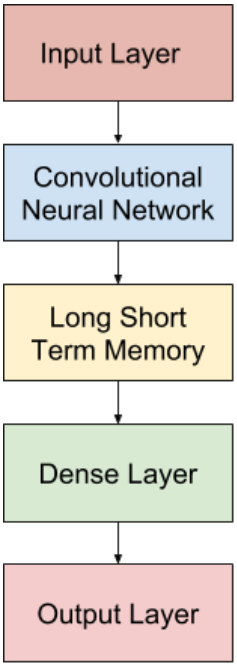

Figure 4.1: CNN-LSTM Architecture.

Subsequent convolution layers help to extract higher-level features, which, in this case, is the long-term patterns in between latency spikes. Each down-sampling layer suppresses noise in the data, i.e., there are minor fluctuations in RTT data that the CNN should not be focusing on; instead, with down-sampling, only the significant latency spikes are analyzed. Furthermore, down-sampling reduces the dimensionality, allowing the next layer to analyze patterns over a longer time-scale. The fully-connected (dense) layers allows the CNN to learn non-linear combinations. Table 3.1 shows the final CNN-LSTM hyperparameters.

LSTM Block

The CNN-LSTM is essentially an encoder-decoder model with the CNN as the encoder and the LSTM as the decoder. The CNN is responsible for encoding the entire time-series into a single very long context vector. The LSTM, being the decoder, is responsible for stepping through the context vector and analyzing the

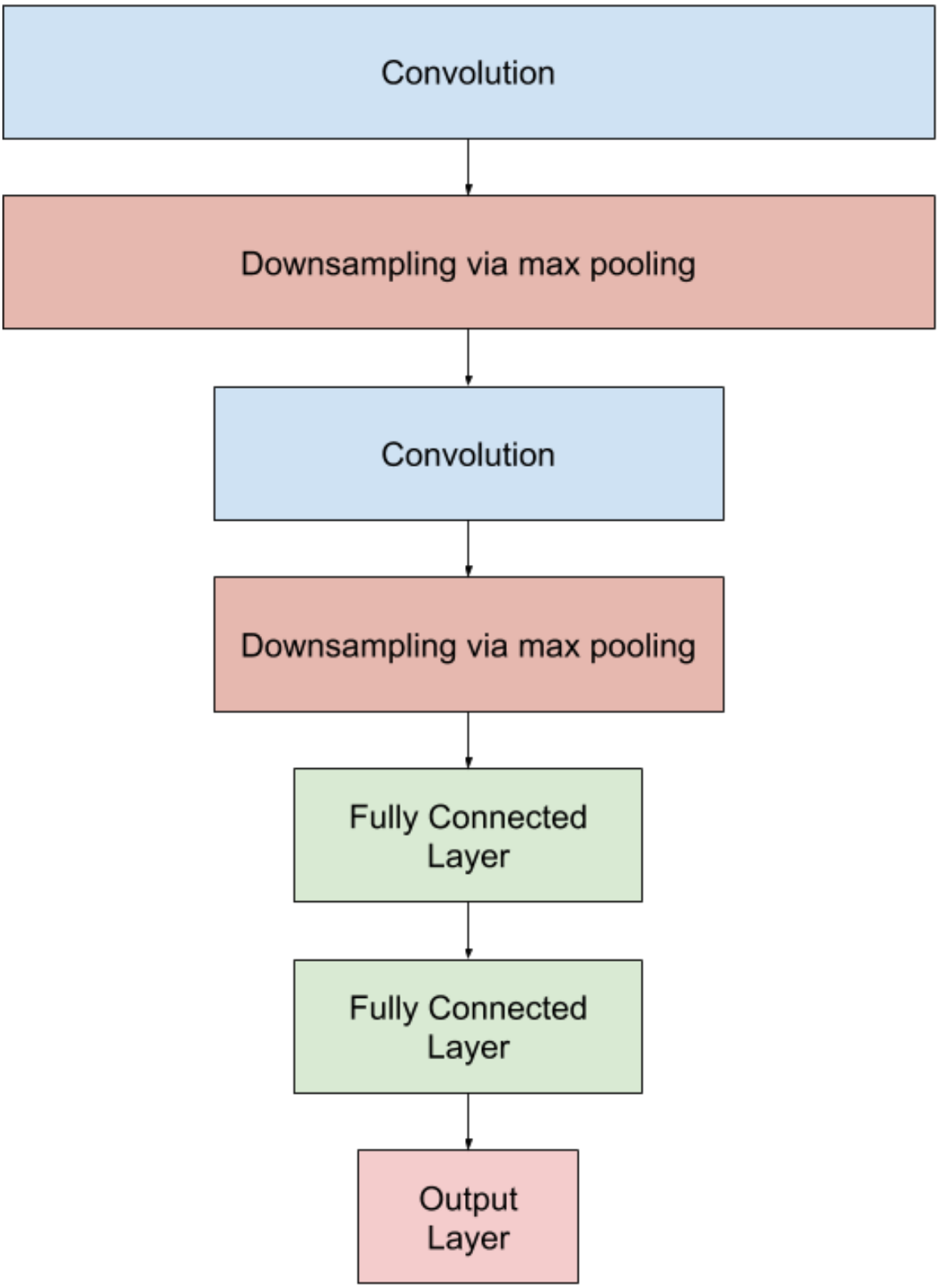

Figure 4.2: CNN-LSTM Architecture.

Table 4.1: CNN-LSTM Hyperparameters

Parameter	**Value**
Batch Size	650
Input Layer Nodes	196
MaxPool-1 Size	3x3
MaxPool-2 Size	3x3
Hidden Layer Nodes	105
Output Layer Nodes	1
Learning Rate	0.010

data in time steps. As in the RNN-LSTM, the LSTM stage used after the CNN is essentially the same.

V. MODIFYING JACOBSON'S ALGORITHM

Simply forecasting RTT values does not help the congestion problems when creating a Flow Control algorithm. Rules must be set that decide what happens when an RTO timer expires. The Chebyshev Bound dictates that the probability for a random value exceeding a range around a mean depends on the standard deviation. Jacobson's Algorithm uses the Chebyshev Bound to produce reasonable RTO values as shown in (5). However, since conventional parameter estimation algorithms require complex calculations and storage for historic values, Jacobson's Algorithm uses simple prediction algorithms to predict mean and standard deviation (5). Since abrupt changes in RTT causes Jacobson's Algorithm to overshoot RTO estimations, replacing the predictor with a much more accurate predictor can help a flow control algorithm to anticipate latency spikes that would otherwise be unpredictable. As shown in the following equation, the $\mathrm{PRED_n}$ is the output of the LSTM predictor:

$$PRED_n = LSTM(PRED_{n-188}, ..., PRED_{n-1}) \tag{5.1}$$

The $\mathrm{PRED_n}$ dictates the error as shown in the following equation:

$$ERR = |RTT_n - PRED_{n-1})| \tag{5.2}$$

The error changes the variance in the following equation:

$$VAR_n = 0.75(SRTT_{n-1}) + 0.25(ERR) \tag{5.3}$$

This ultimately generates the RTO in as shown in the following equation:

$$RTO = PRED_n + 4(VAR_n) \tag{5.4}$$

VI. RESULTS

Jacobson's Algorithm uses the Chebyshev Bound to predict expected RTO values. By replacing the predictor used in Jacobson's Algorithm with a neural network, two important traits emerged in the results: the LSTMs were both able to identify latency spikes by the starting edge of the spike and, to a certain degree, predict the time intervals between spikes. The CNN even "remembered" the maximum RTT values and preemptively set the prediction high when it identified the leading edges.

RNN-LSTM Performance

The RNN-LSTM was able to identify patterns in the data best when only one communication pattern was present. This model would be best in a hypothetical situation where a new RNN-LSTM is trained for each new data stream. For example, after the first 188 segments in a new connection, the RNN-LSTM can begin training and forecasting the subsequent segments' RTTs in the stream. However, when the stream is closed, the RNN-LSTM state information is discarded as it will not be relevant for a different kind of data-stream. As shown in figure 6.1, the RNN-LSTM was best at predicting RTT when only one data pattern was observed - in this case, the data pattern is recorded RTT values from a stream captured during multiplayer gameplay. Figure 6.2 shows a closer view at the algorithm correctly anticipating the periodic changes in RTT. The RNN-LSTM was able to also identify a data pattern characteristic of a sine wave, as shown in figure 6.3. Figures 6.4 and 6.5 show the RNN-LSTM able to predict triangle waves and square waves respectively.

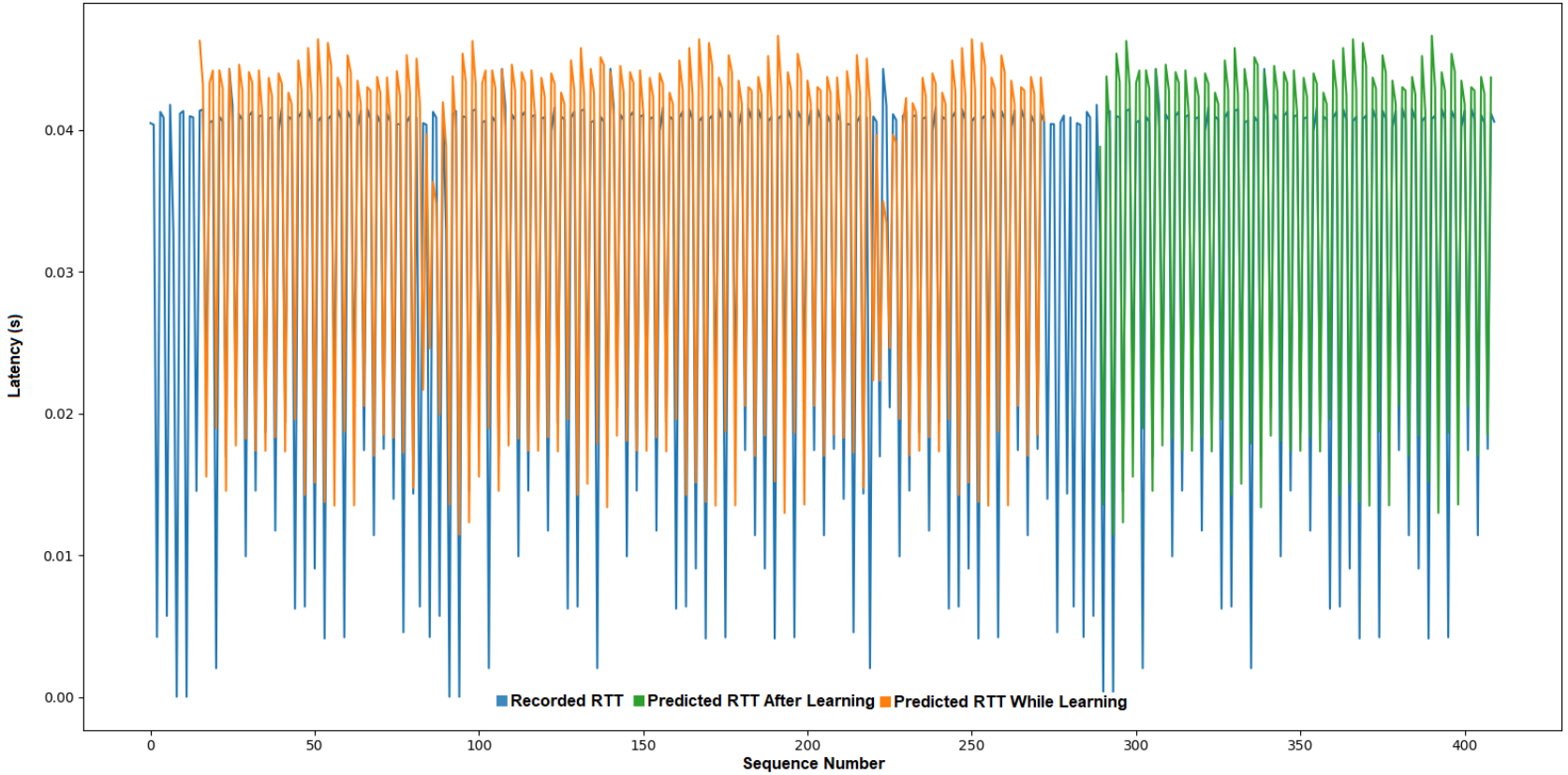

Figure 6.1: RNN-LSTM Predicting MMORPG stream RTT

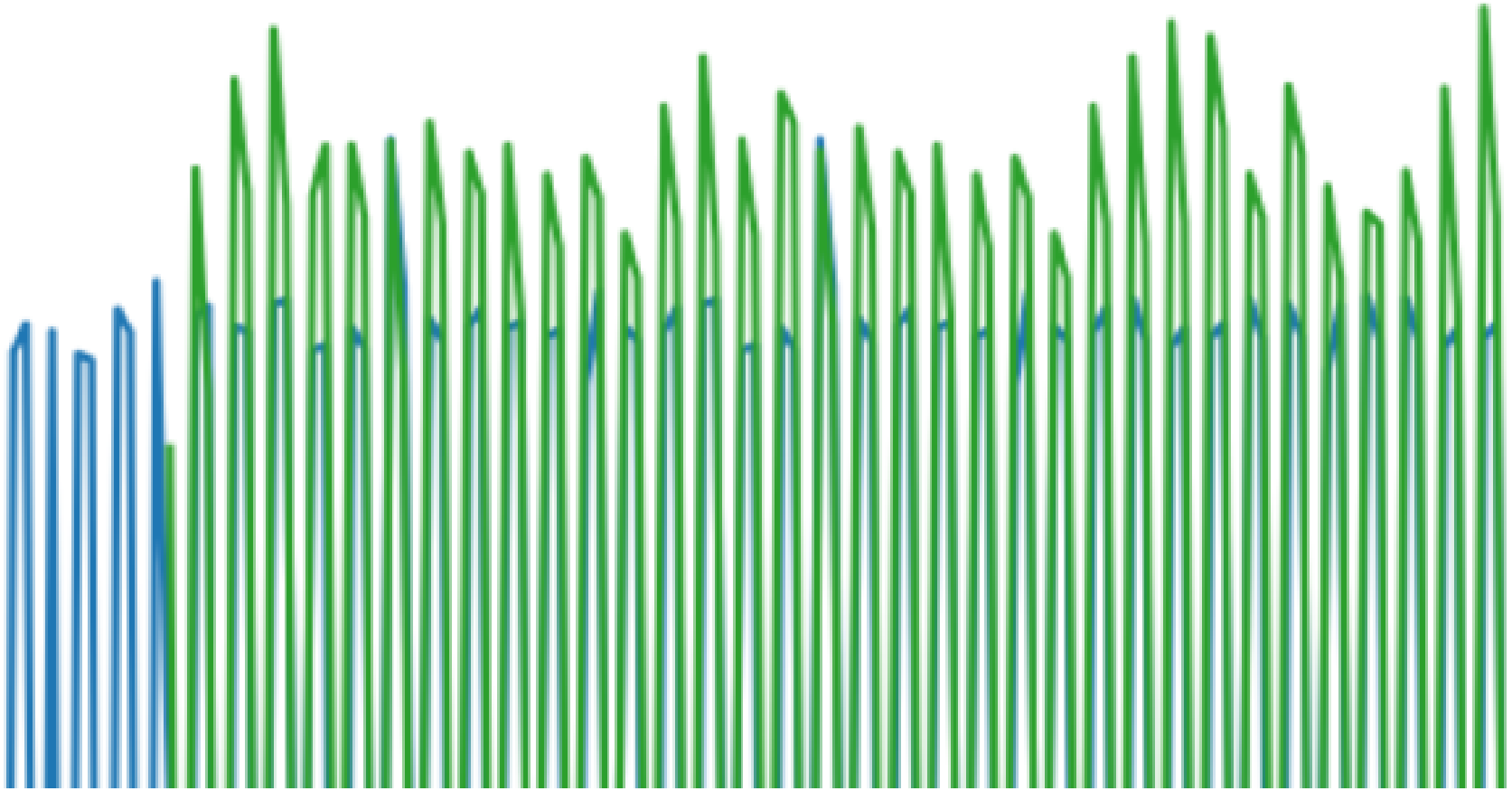

Figure 6.2: Closeup of Figure 6.1: RNN-LSTM Anticipating Period of RTT Change

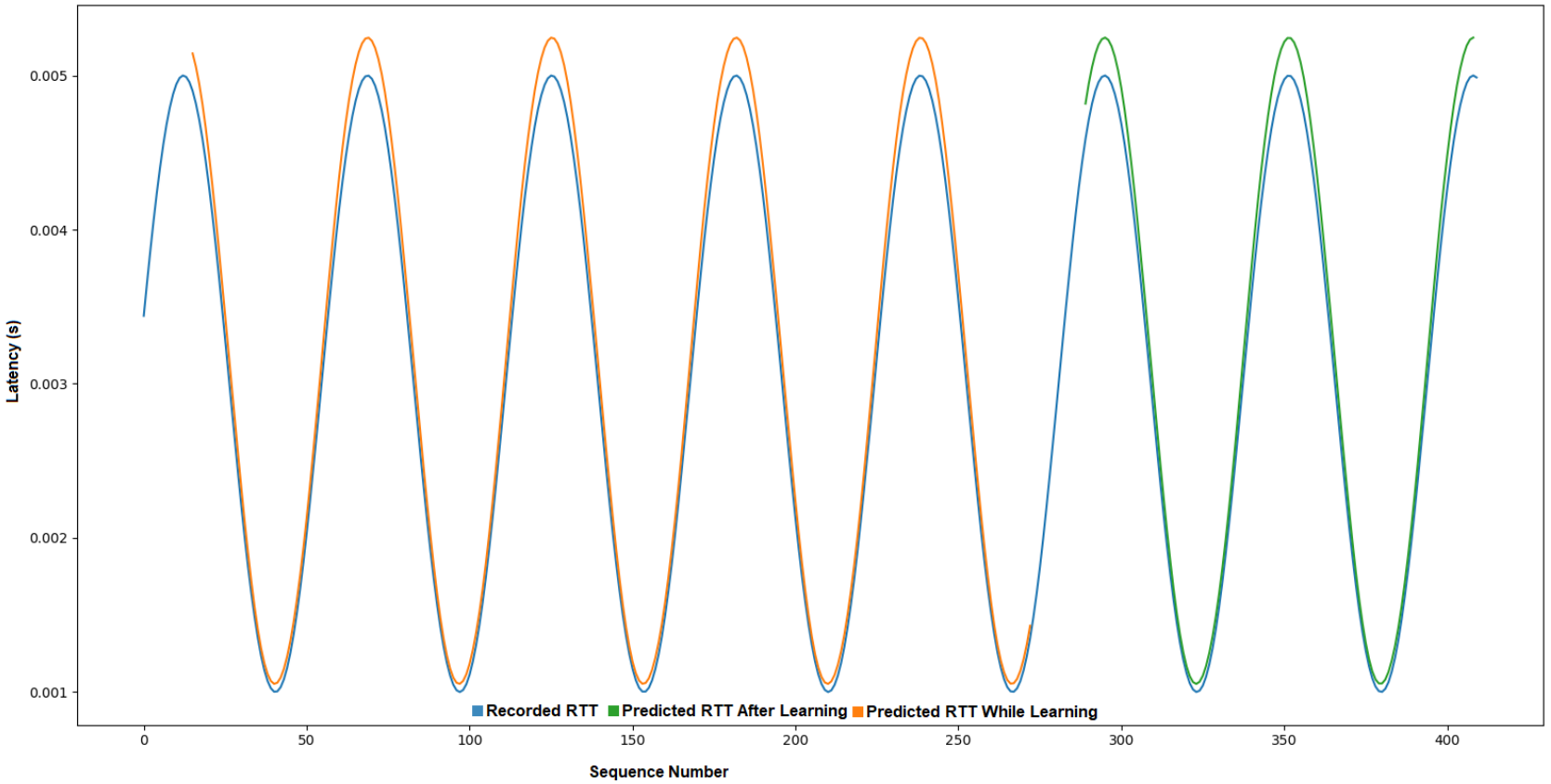

Figure 6.3: RNN-LSTM Estimating RTT Change of Sine Wave

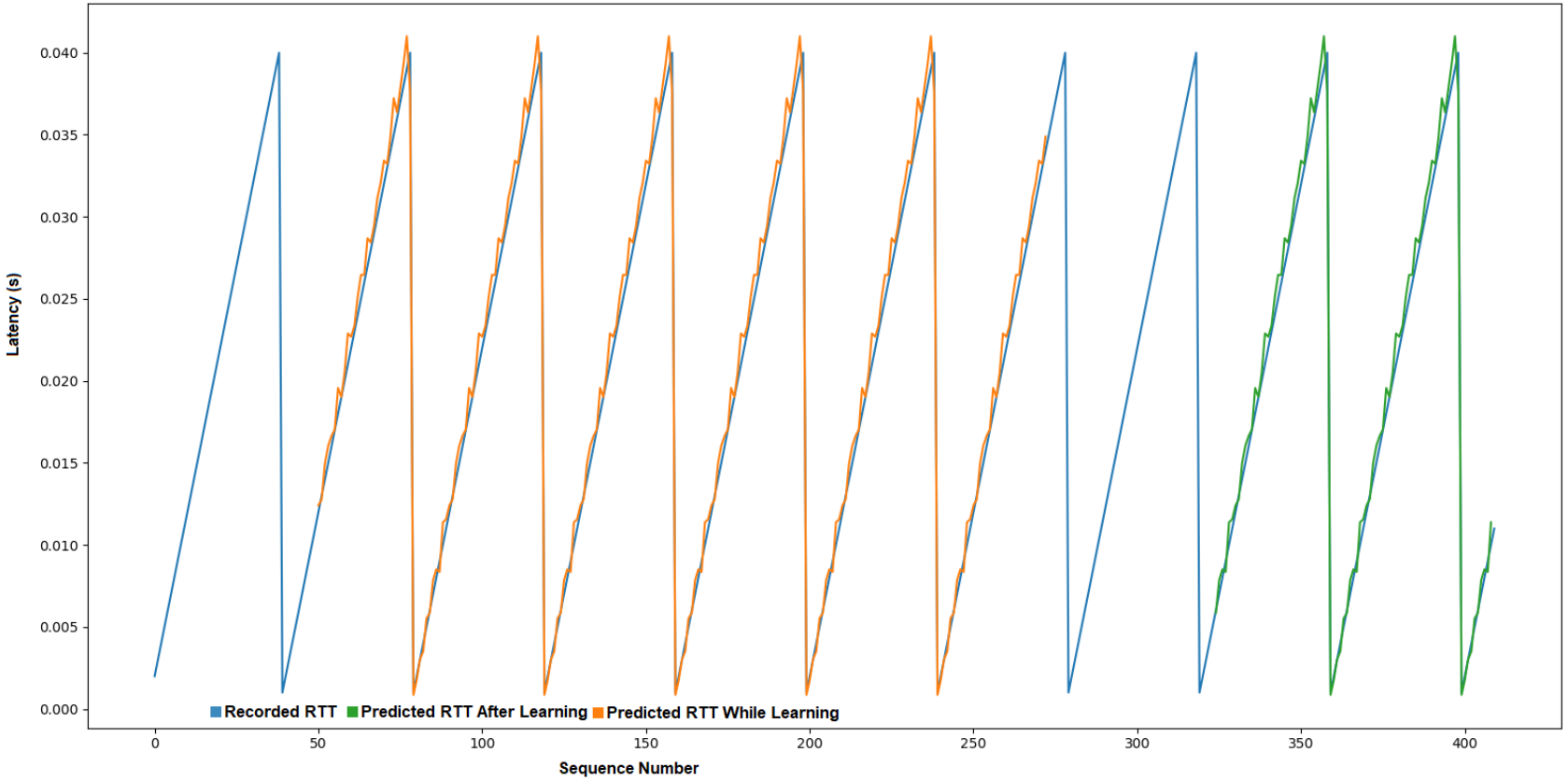

Figure 6.4: RNN-LSTM Estimating RTT Change of Triangle Wave

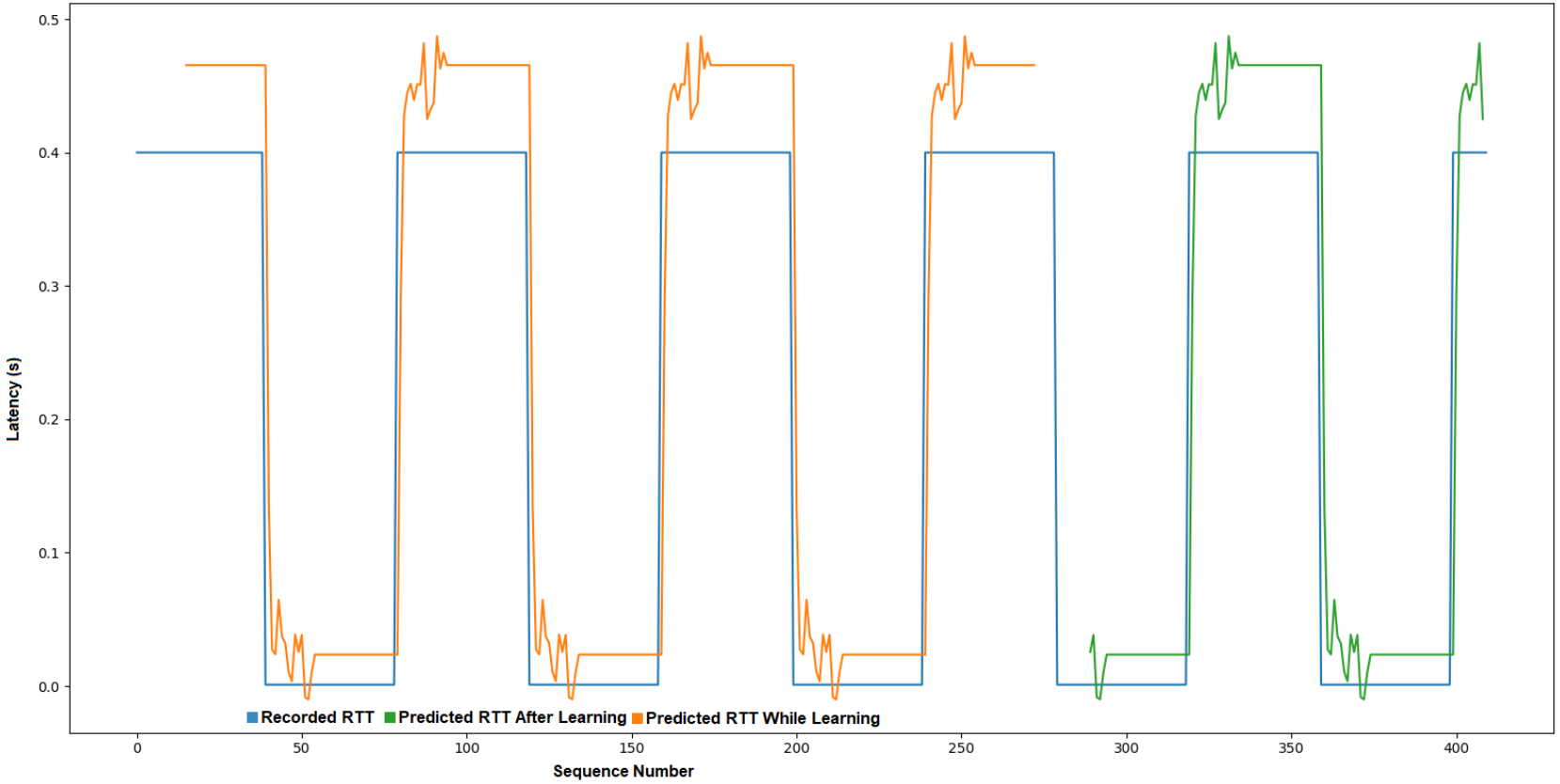

Figure 6.5: RNN-LSTM Estimating RTT Change of Square Wave

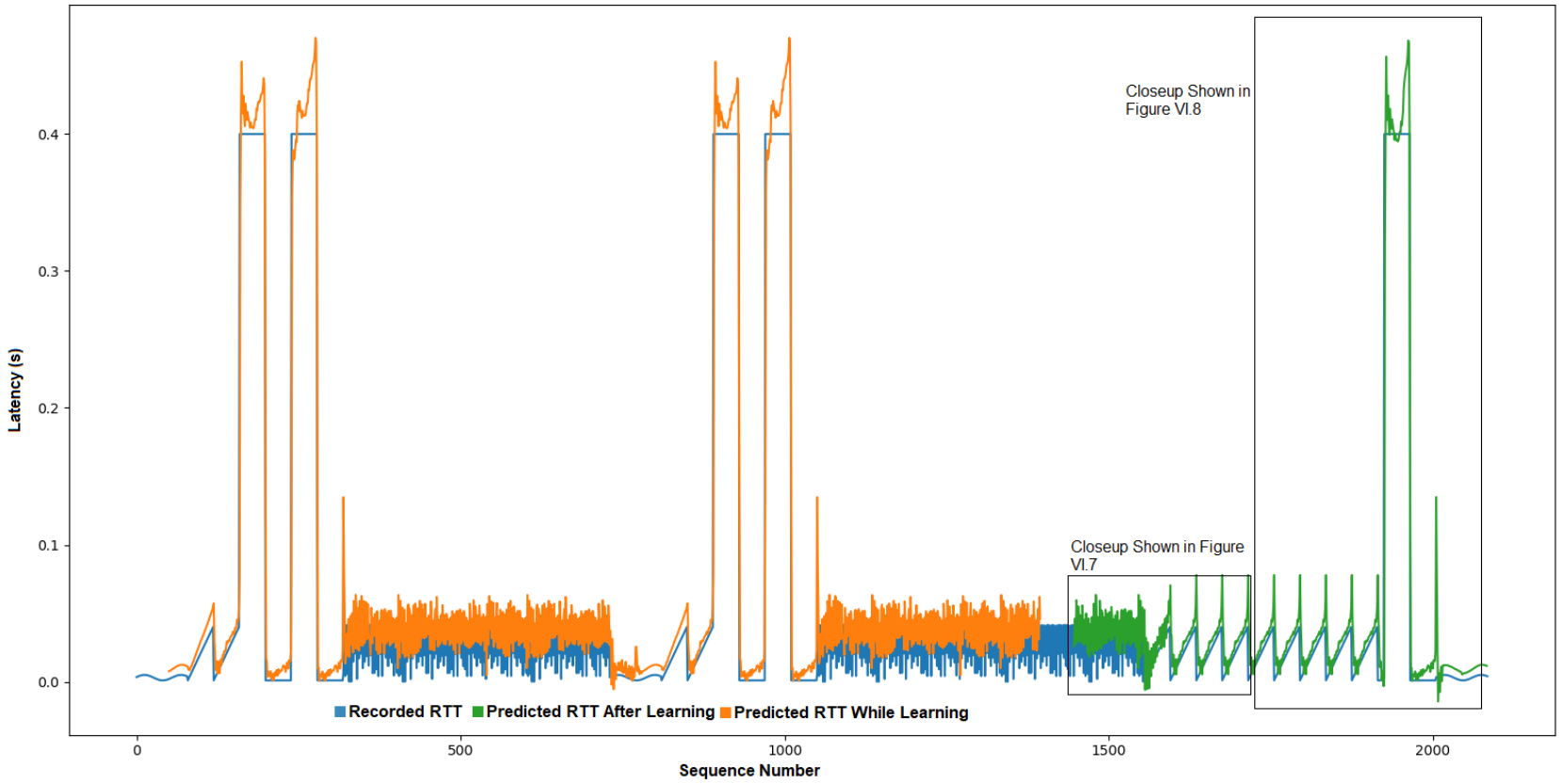

Figure 6.6: RNN-LSTM Estimating RTT Change of Random Permutation

Finally, several permutations of patterns were joined together and were used to train the RNN-LSTM. And example data pattern and the RNN-LSTM's RTT estimations are shown in figure 6.6.

As shown in figure 6.7, a closeup of figure 6.6, as soon as the training dataset ended and the random permutation of patterns began (in this example with a random MMO stream's RTT dataset), the RNN-LSTM was quick to remember the pattern and adjust the RTT estimations accordingly. However, when the pattern changed to a triangle wave, the RNN-LSTM was slow to recognize the pattern and adjust. Also, as shown in figure 6.8, though the RNN-LSTM was able to recognize and adjust its estimations for the square wave pattern, at the end of the "low" phase of the square wave at around packet number 2000, the neural network expected the leading edge of a square wave, and not finding said leading edge, was slow to recognize the sine wave.

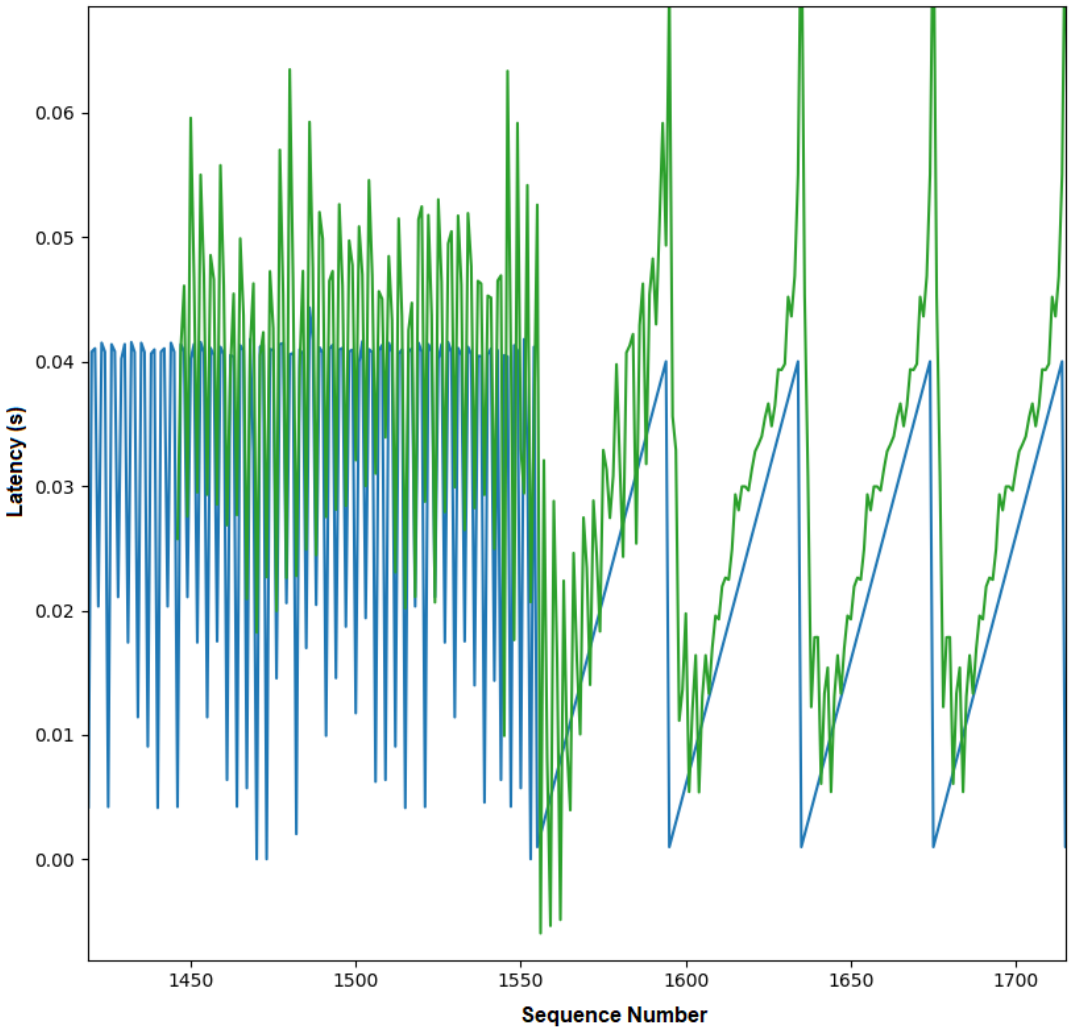

Figure 6.7: Closeup of Figure 6.6: MMO to Triangle Wave

CNN-LSTM Performance

The CNN-LSTM was able to identify patterns over a wide variety of communication patterns. In the case of the high-congestion environment shown in figure II.1, it was able to estimate the maximum RTT values for each latency spike and was able to immediately set the RTO as high as previous maxima instead of stepping it up slowly. Then, when the communication pattern changed to multiplayer gaming, the gradually increasing and decreasing RTT values were properly predicted, and RTO

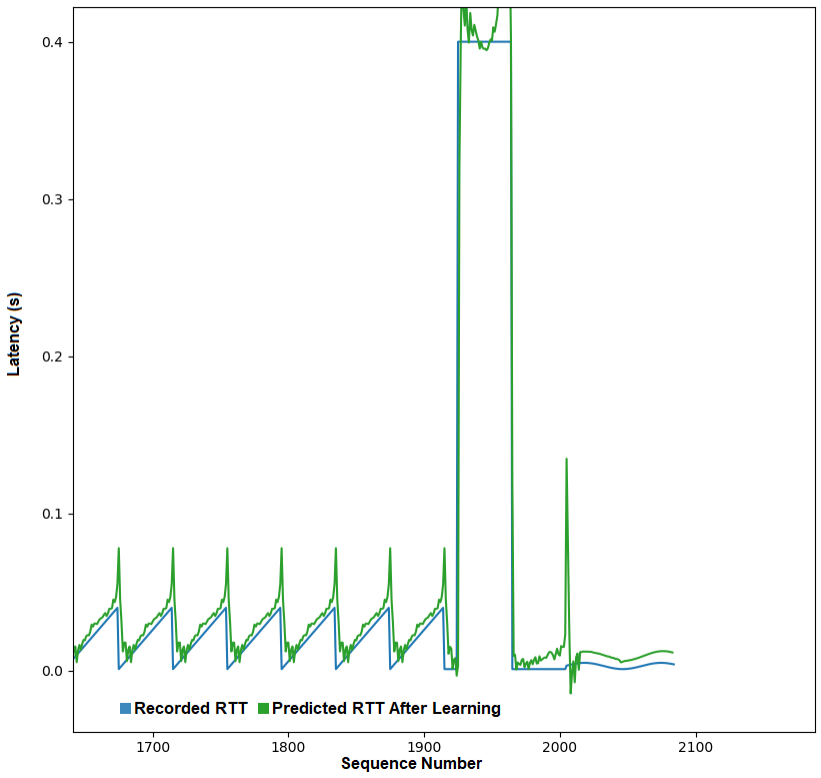

Figure 6.8: Closeup of Figure 6.6: RNN-LSTM Estimating RTT Change

values were set accordingly.

The CNN-LSTM was able to also identify a data pattern characteristic of a sine wave, as shown in figure 6.9. Figures 6.10 and 6.11 show the CNN-LSTM able to predict triangle waves and square waves respectively.

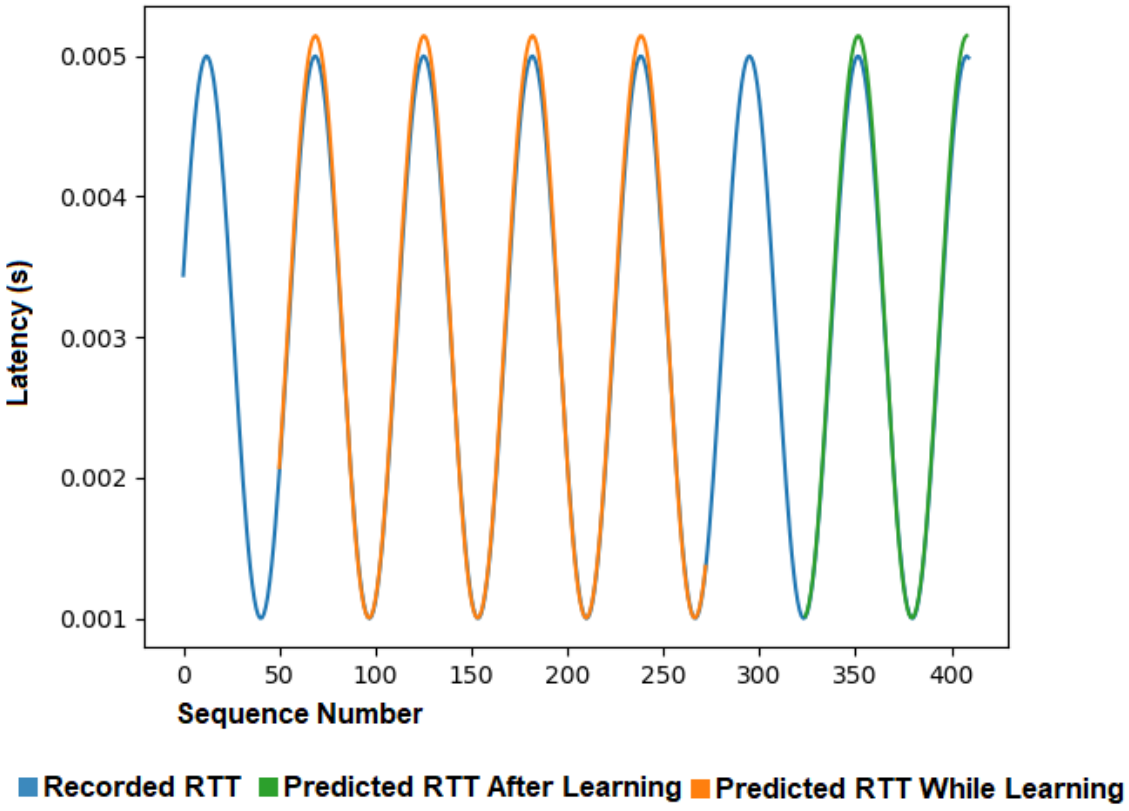

Figure 6.9: CNN-LSTM Estimating RTT Change of Sine Wave

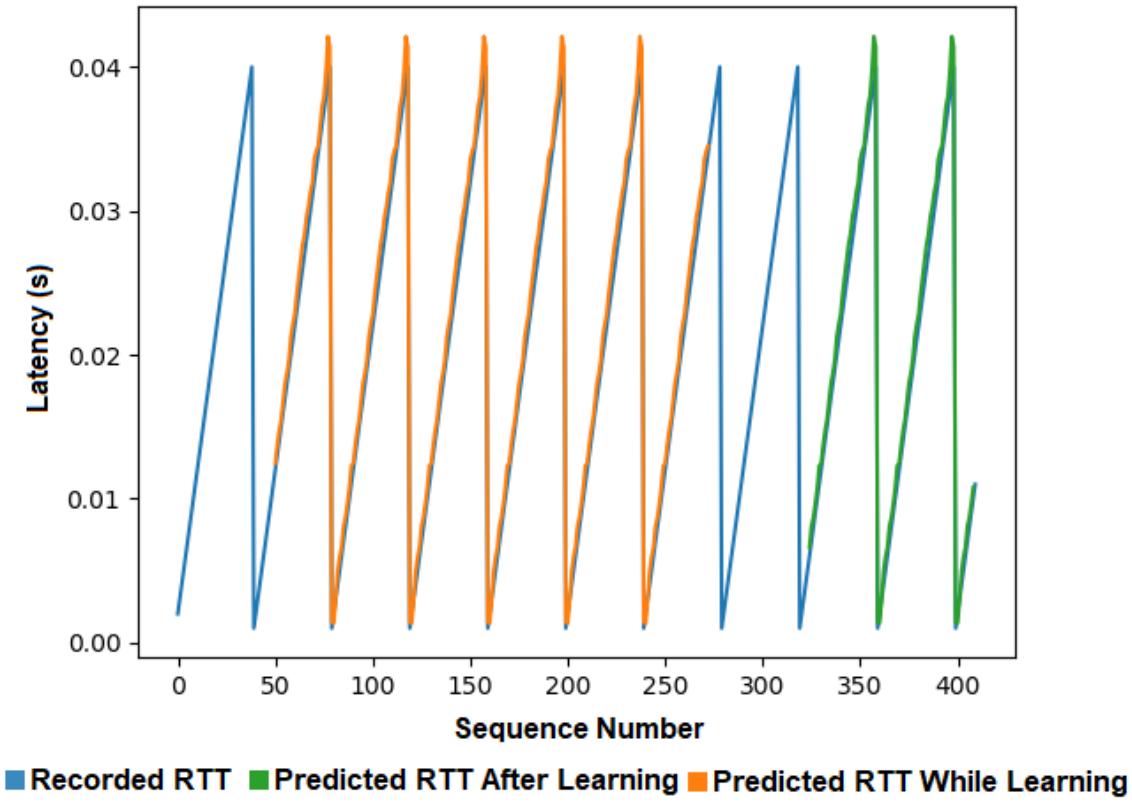

Figure 6.10: CNN-LSTM Estimating RTT Change of Triangle Wave

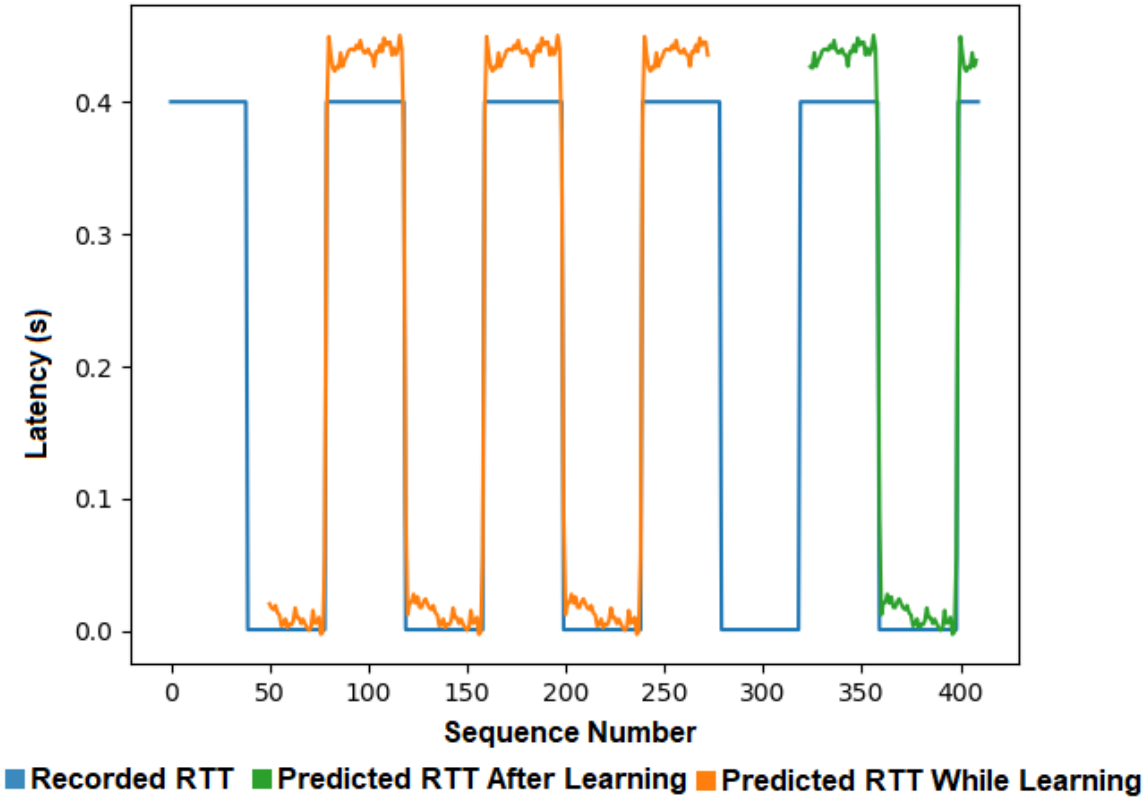

Figure 6.11: CNN-LSTM Estimating RTT Change of Square Wave

Finally, several permutations of patterns were joined together and were used to train the CNN-LSTM. And example data pattern and the CNN-LSTM's RTT estimations are shown in figure 6.12.

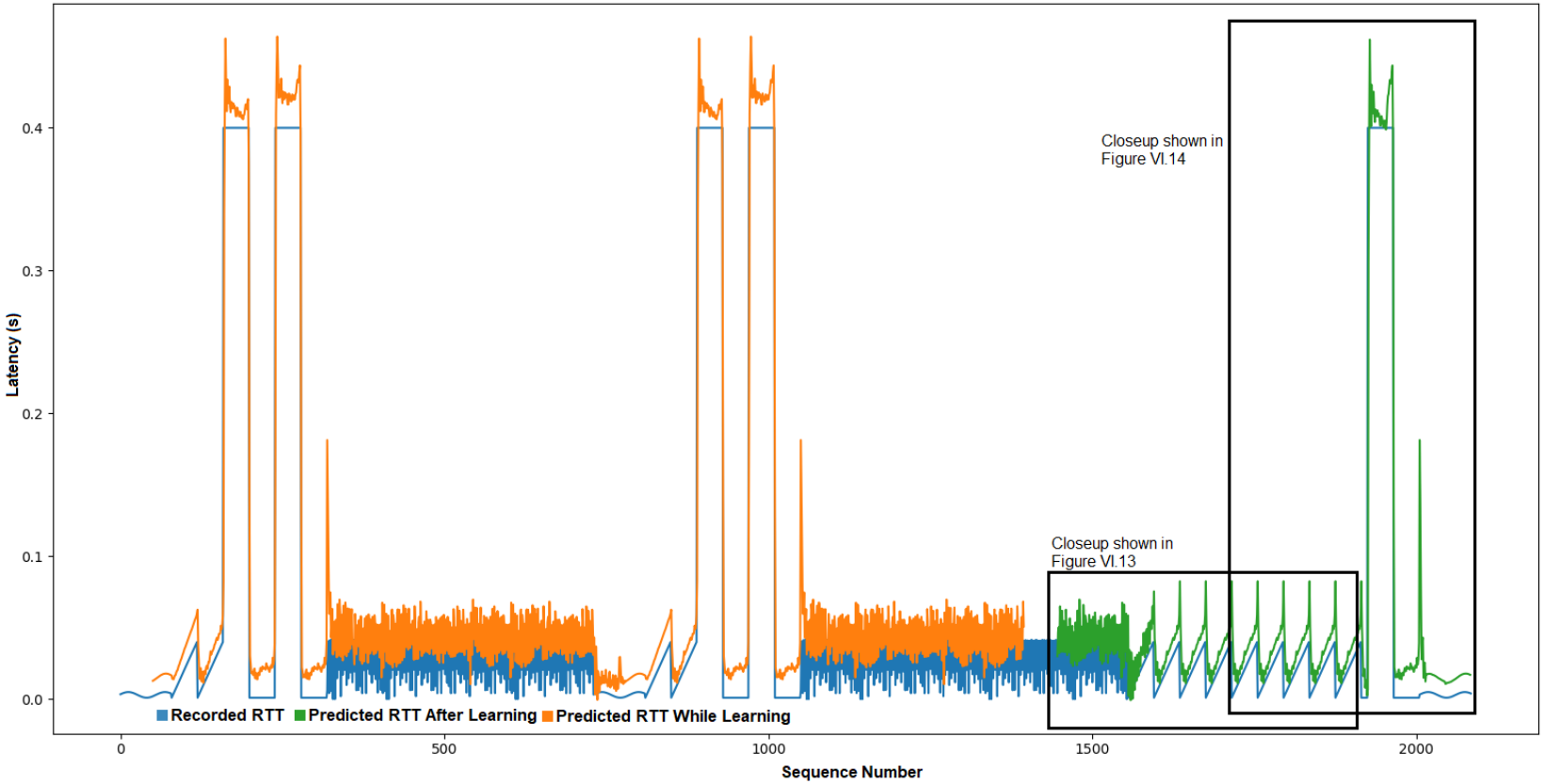

Figure 6.12: CNN-LSTM Estimating RTT Change of Random Permutation

As shown in figure 6.13, a closeup of figure 6.12, as soon as the training dataset ended and the random permutation of patterns began (in this example with a random MMO stream's RTT dataset), the CNN-LSTM was quick to remember the pattern and adjust the RTT estimations accordingly. Notably, when the pattern changed to a triangle wave, the CNN-LSTM was much faster than the RNN-LSTM to recognize the pattern and adjust. Also, as shown in figure 6.14, the CNN-LSTM was able to recognize and adjust its estimations for the square wave pattern, but performed similarly to the RNN-LSTM at recognizing the sine wave.

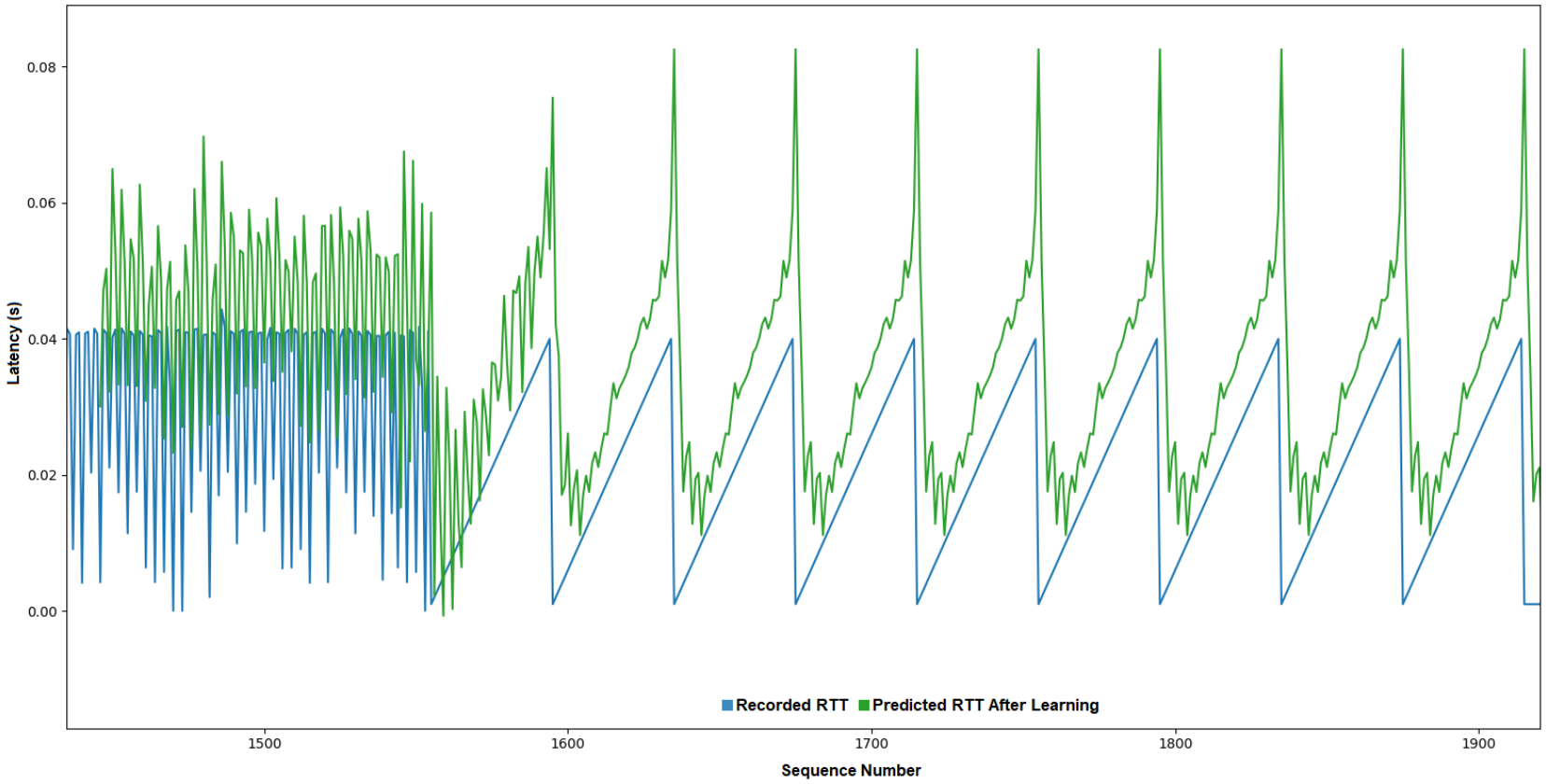

Figure 6.13: CNN MMO to Triangle Wave Closeup

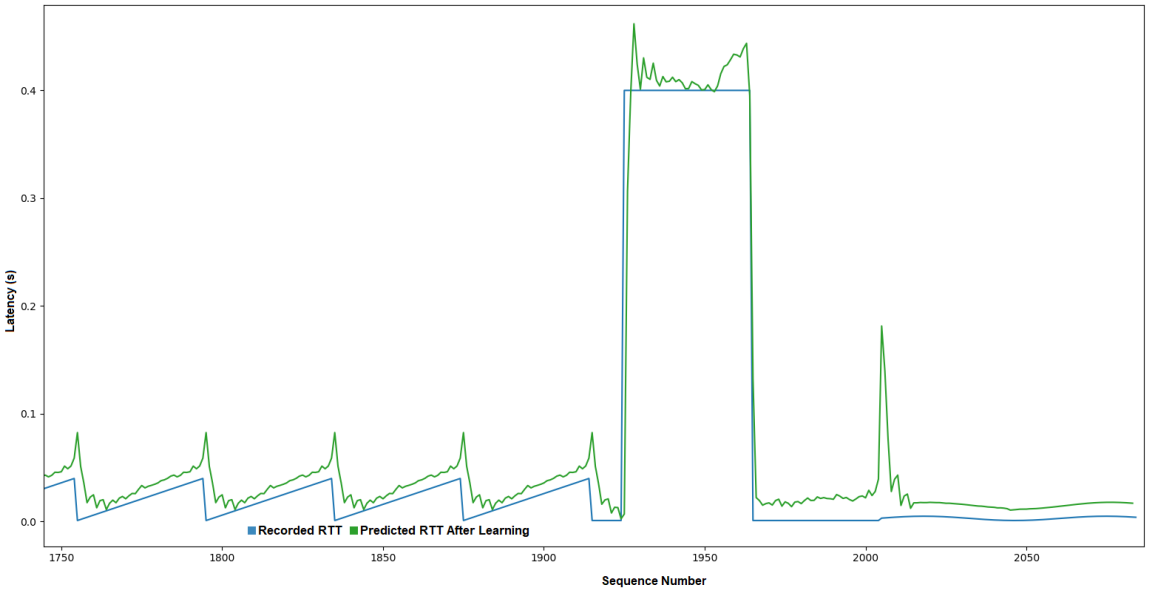

Figure 6.14: CNN-LSTM Estimating RTT Change of Random Permutation

Accuracy versus Jacobson's Algorithm

Since there was a total of three training sets (each communication pattern's order of appearance was changed), the CNN-LSTM's and RNN-LSTM's ability to detect patterns varied. Table 6.1 shows the CNN-LSTM's ability to detect long term patterns in latency spikes and therefore decrease the number of retransmissions and the RNN-LSTM's ability to predict short term patterns in increasing and decreasing RTT changes and reduce the number of retransmissions in that manner. As shown, Jacobson's Algorithm modified to use a CNN-LSTM forecaster instead of the Chebyshev Bound showed significantly fewer RTO expirations, albeit requiring around 100 epochs of training. However, as the number of epochs was increased to 200, severe overfitting became a problem and many more segments were dropped.

Table 6.1: Number of False Retransmissions (Lower is Better)

		Epochs			
		10	**50**	**100**	**200**
TCP Implementation of Jacobson's Algorithm	First Testing Set	1088	1088	1088	1088
	Second Testing Set	1211	1211	1211	1211
	Third Testing Set	1036	1036	1036	1036
Modified Jacobson's Algorithm with RNN -LSTM	First Testing Set	891	1255	1510	28144
	Second Testing Set	914	1045	1165	30271
	Third Testing Set	810	1250	1440	21033
Modified Jacobson's Algorithm with CNN -LSTM	First Testing Set	671	325	101	1812
	Second Testing Set	671	441	86	666
	Third Testing Set	840	610	223	1545

The Jacobson's Algorithm that was modified to use an RNN-LSTM forecaster instead of the Chebyshev Bound showed a minor improvement in RTO expirations when given a combination of each communication pattern. It was able to decrease the number of RTO expirations only in the first two communication patterns, after which overfitting led to a high number of dropped segments when transitioning to the third communication pattern. This was compounded when put through higher epochs of training. Since the unmodified Jacobson's Algorithm is deterministic, the number of RTO expirations were the exact same no matter how many times it was applied to the same data set, and therefore was a constant for each individual data set.

Table 6.2: Error Calculations for Combination Dataset (Lower is Better)

	Combination Test		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-15.736628166	-32.117609026	-17.143076981
RMSE (dB)	-7.8683114682	-15.9889081	-8.5715332612
Weighted MSE (dB)	-14.129212981	-15.071865493	-16.973184865

Table 6.3: Error Calculations for FTP Dataset (Lower is Better)

	FTP Test		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-62.510371387	-62.494916051	-62.494916051
RMSE (dB)	-31.255187621	-31.247459527	-31.247458369
Weighted MSE (dB)	-61.676188398	-62.358238676	-61.821042428

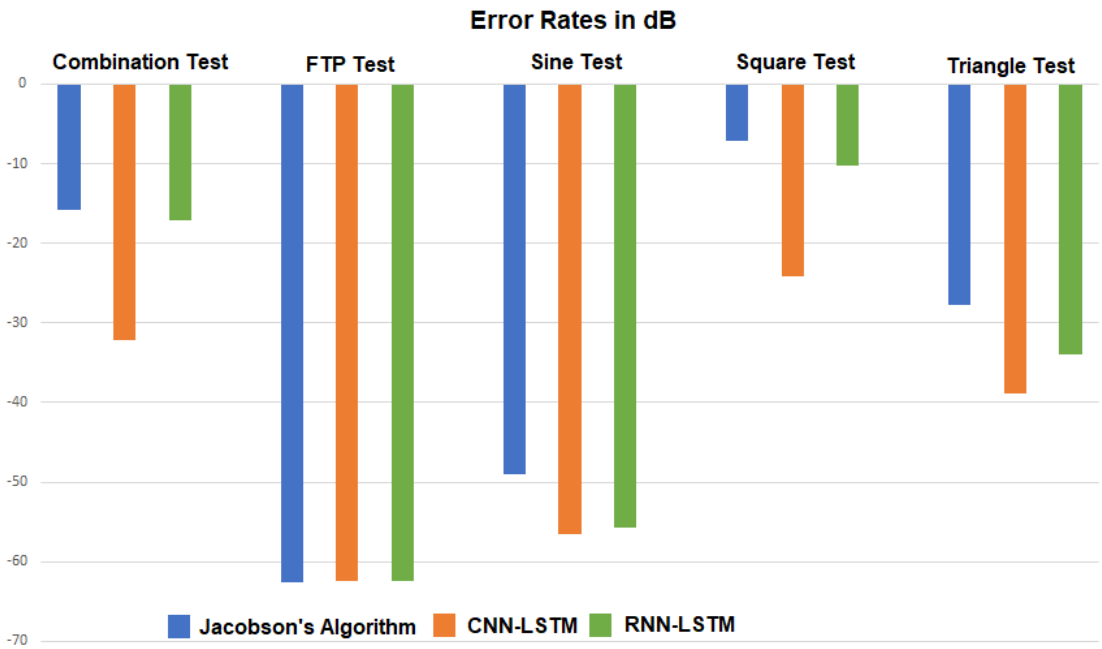

Figure 6.15: Error Rates in dB

Conclusion

Accurate prediction of RTT using neural networks can optimize network communication and better control congestion in connection-oriented network

Table 6.4: Error Calculations for Sine Dataset (Lower is Better)

	Sine Test		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-48.941833626	-56.546262694	-55.684754158
RMSE (dB)	-24.470906167	-28.272625401	-27.842392762
Weighted MSE (dB)	-24.158710411	-27.760938407	-26.753313796

Table 6.5: Error Calculations for Square Wave Dataset (Lower is Better)

	Square Test		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-7.1115897861	-24.194572805	-10.164030366
RMSE (dB)	-6.5564380177	-12.097286182	-5.0820151777
Weighted MSE (dB)	-3.9791015738	-10.623853553	-3.8379228942

protocols. By reducing the number of segment retransmissions caused by inaccurate prediction models, congestion can be greatly reduced. Jacobson's Algorithm uses the Chebyshev Bound to predict RTO times by predicting mean and standard deviation of previous RTT times. However, the technique used does not store historic values and thus cannot retain long term trends; therefore, it is not useful for finding long term patterns in RTT trends. This often leads to insufficient RTO predictions that causes a higher number of RTO expirations and thus retransmissions. RTT data for different communication patterns were collected from various applications using TCP and used to train an RNN-LSTM and a CNN-LSTM to forecast RTT times. The RNN-LSTM was able to estimate peak RTT values for a latency spike best when the communication pattern stayed constant. The CNN-LSTM was able to isolate patterns in the frequency of latency spikes as well as the peak RTT values. Jacobson's Algorithm was modified to use these prediction methods instead of a linear classifier. Armed with a more accurate predictor, RTO values were set accordingly and greatly decreased the number of retransmissions.

Table 6.6: Error Calculations for Triangle Wave Dataset (Lower is Better)

	Triangle Test		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-27.800413245	-38.88273859	-33.927498405
RMSE (dB)	-13.900206372	-19.441376771	-16.963749592
Weighted MSE (dB)	-27.256336009	-07.2274250723	-33.861560648

Table 6.7: Error Calculations for MMORPG Dataset (Lower is Better)

	MMO Dataset		
	Jacobson	**CNN-LSTM**	**RNN-LSTM**
Mean Square Error (dB)	-24.580936608	-27.098865546	-33.206593345
RMSE (dB)	-12.290468242	-18.549432621	-16.603297572
Weighted MSE (dB)	-23.988264391	-6.801503947	-32.915621024

Future Work

The datasets used to train the neural networks may introduce unwanted bias. For example, in the congestion datasets, patterns were repetitive and inherently predictable but only in the context of the specific conditions of the experiment; every time the congestion experiment parameters (i.e. the size of the chunks, the size of the files, etc.) were changed, the intervals between high latency spikes changed accordingly. This meant that if the LSTM models were trained with congestion datasets with one set of parameters, it was able to predict RTT patterns with test sets with the same parameters; when tested with congestion datasets with different parameters, overfitting led to slower pattern recognition. Further exploration into diversifying artificially generated training sets to include RTT patterns from multiple parameters could lead to faster pattern recognition with unseen parameters.

In the current stage of this research, almost all of the experimentation has been done offline. Wireshark is used to collect RTT values into a time-series, the LSTM

models are used for time-series forecasting, and the performances of Jacobson's Algorithm and the modified Jacobson's Algorithm presented in this document have been compared. However, actual implementation of this algorithm onto hardware has not been explored by this research. The experimentation done for this document was done using an NVIDIA RTX 2080 graphics card, and after optimization was able to perform the necessary calculations in real time; however, it is hardly feasible due to power usage and cost to afford a top-of-the-line graphics card for network calculations. Looking at past trends can give insight to the future of this research: network controllers were once implemented as expansion cards due to the amount of processing and the application-specific electronic circuitry required to connect computers to computer networks. However, following the trend of integrating System-On-Chips (SOCs), Ethernet network cards have been implemented on Application Specific Integrated Circuits (ASICs) and absorbed onto the motherboard. One possible future for this research may be its implementation onto an ASIC and integrated onto the motherboard on future edge devices.

APPENDIX SECTION

APPENDIX A

Github page: https://github.com/BikramjitD/Thesis

Example RNN-LSTM Setup:

```
model = Sequential()

model.add(LSTM(100, input_shape=(1, look_back)))

model.add(Dense(20))

model.add(Dense(10))

model.add(Dense(6))

model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer='adam')

model.fit(trainX, trainY, epochs=10, batch_size=1, verbose=2)

# make predictions

trainPredict = model.predict(trainX)

testPredict = model.predict(testX)
```

Table 6.8: Example Subset of RTT Dataset

1st 25 entries	2nd 25 entries	3rd 25 entries	4th 25 entries	6th 25 entries
0.003221765	0.003500046	0.001843604	0.004661366	0.021
0.003440795	0.00328224	0.002031672	0.004527655	0.022
0.003654389	0.003060954	0.002231683	0.004375102	0.023
0.003859913	0.002838916	0.002441169	0.004205591	0.024
0.004054831	0.002618864	0.002657547	0.004021211	0.025
0.00423674	0.002403513	0.002878149	0.001	0.026
0.004403396	0.002195519	0.003100254	0.002	0.027
0.004552744	0.001997446	0.003321122	0.003	0.028
0.004682942	0.001811738	0.00353803	0.004	0.029
0.004792384	0.001640684	0.003748302	0.005	0.03
0.004879721	0.001486395	0.003949346	0.006	0.031
0.004943876	0.001350773	0.004138681	0.007	0.032
0.004984056	0.001235491	0.004313973	0.008	0.033
0.004999768	0.001141971	0.00447306	0.009	0.034
0.004990816	0.001071366	0.004613979	0.01	0.035
0.004957311	0.001024546	0.004734994	0.011	0.036
0.004899667	0.00100209	0.00483461	0.012	0.037
0.004818595	0.001004274	0.004911601	0.013	0.038
0.004715094	0.001031071	0.004965016	0.014	0.039
0.00459044	0.001082151	0.004994196	0.015	0.04
0.004446172	0.001156884	0.004998782	0.016	0.001
0.004284068	0.001254348	0.004978716	0.017	0.002
0.004106127	0.001373341	0.004934248	0.018	0.003
0.003914545	0.001512396	0.004865924	0.019	0.004
0.003711684	0.001669797	0.004774588	0.02	0.005

invert predictions

trainPredict = scaler.inverse_transform(trainPredict)

trainY = scaler.inverse_transform([trainY])

testPredict = scaler.inverse_transform(testPredict)

Example CNN-LSTM Setup:

```
raw_seq = rtt

n_steps = 100

X, y = split_sequence(raw_seq, n_steps)

n_seq = 10

n_steps = 10

X = X.reshape((X.shape[0], n_seq, n_steps, n_features))

model = Sequential()

model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
input_shape=(None, n_steps, n_features)))

model.add(TimeDistributed(MaxPooling1D(pool_size=2)))

model.add(TimeDistributed(Flatten()))

model.add(LSTM(50, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

model.fit(X, y, epochs=10, verbose=2)

yhat2=[]
```

REFERENCES

[1] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 314–329, Aug. 1988.

[2] E. G. Guerra, "Reducing sctp's (stream control transmission protocol) time-to-complete by manipulation of its retransmission time out minimum," Master's thesis, Texas State University, 601 University Dr. San Marcos, Texas, 78666, 8 2014.

[3] P. Geurts, I. El Khayat, and G. Leduc, "A machine learning approach to improve congestion control over wireless computer networks," in *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pp. 383–386, Nov 2004.

[4] G. P. El Khayat I. and L. G., "Enhancement of tcp over wired/wireless networks with packet loss classifiers inferred by supervised learning," *Wireless Networks*, vol. 16, pp. 273–290, Feb. 2010.

[5] S. McClellan and W. Peng, "Estimating retransmission timeouts in ip-based transport protocols," 04 2013.

[6] A. Papoulis, "Probability, random variables, and stochastic processes, 2nd ed.," pp. 149–151, 1984.

[7] Z. Yang and A. Srivastava, "Value-driven synthesis for neural network asics," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, (New York, NY, USA), pp. 1:1–1:6, ACM, 2018.

[8] G. Combs, "Wireshark." `https://www.wireshark.org/#download`, 1998.

[9] B. DasGupta, D. Valles, and S. McClellan, *A Comparison of MLA Techniques for Classification of Network Bandwidth Loss*, pp. 771–775. CSCI, 2018.

[10] N. F. Mir, *Computer and communication networks*. Prentice Hall, 2015.

[11] R. R. Stewart and Q. Xie, *Stream Control Transmission Protocol (SCTP): a reference guide*. Addison-Wesley, 2002.

[12] M. Bernico, *Deep learning quick reference: useful hacks for training and optimizing deep neural networks with TensorFlow and Keras*. Packt Publishing, Ltd., 2018.

[13] A. Tokgoz and G. Unal, "A rnn based time series approach for forecasting turkish electricity load," *26th Signal Processing and Communications Applications Conference (SIU)*, 2018.

[14] K. L. A. Sarah and H. Kim, "Lstm model to forecast time series for ec2 cloud price," *16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing*, pp. 1085–1088, 2018.