AI in Congestion Control

Ubeyd Ali Muhamud Abukar

Congestion control algorithms is a field that has been studied for many years. As a result many different protocols have been developed for the purpose of ensuring reliable end to end communication, that ensures that lost data is retransmitted, and packets arriving out of order will be reordered. However in recent years AI methods such as reinforcement learning have been proposed in the context of congestion control algorithm. This project investigates, how Q-learning can be implemented in TCP, and how the reward function will affect the behavior of the agent. Different versions of reward functions have been designed, implemented and tested in various network setups. The agents are even trained and tested against rule-based TCP protocols such as TCP Vegas and TCP New Reno. Hyper-parameter tuning has also been performed on the agent, and the results showed that hyper-parameter tuning was the deciding factor in regards to agents reaching their goal.

June 04, 2025

Theme:
AI based TCP Congestion control

Project Period:
Spring Semester 2025

Project Group:
Group 1049A

Supervisors:
Tatiana Kozlova Madsen, Mathias Drekjær Thorsager

Contents

Chapter 1. Introduction

Transmission Control Protocols have been extensively researched for decades. As a result numerous protocols have been designed, with the shared purpose of successfully transmitting data across the internet and ensuring that data arrive at its destination. Should there be packet loss along the way, TCP will ensure retransmission. And should packets arrive out of order at the destination, the protocol will perform reordering.

Each TCP protocol tackle the problem of congestion control in a different manner, with focus put on different performance metrics. While some protocols tackle this problem using throughput and packet loss as congestion indicators, other protocols use fluctuations in latency as congestion indicators. One protocol in particular even models the links, and tries to estimate optimal sending rates based on link capacity.

However these protocols follow a set of predefined rules that do not adapt well to the dynamic nature of computer networks. It is hard to capture sudden increase or decrease in available bandwidth capacity and effectively responding by only observing packet loss, or fluctuations in latency. In recent years a great effort has been put into researching how Artificial Intelligence can be used in the context of congestion control algorithms. reinforcement learning has been found to be a good candidate to tackle the problem of congestion control. The goal is to teach an agent how to successfully share limited resources with other traffic flows. By training the agent in environments where traditional rule-based congestion control algorithms struggle to adapt, it may be possible for the agent to learn and overcome the shortcomings of traditional Congestion control algorithms.

Chapter 2. Analysis

This chapter will present an analysis of the limitations of traditional congestion control algorithms, then present theory on reinforcement learning. This will be followed by a presentation of the state of the art in the field of reinforcement learning based congestion control algorithms.

2.1. Challenges of Congestion Control

There exist many different kinds of congestion control algorithms, and they all try to solve the same problem. That is to ensure that data arrive at its destination, by means of re-transmitting lost data, reordering packets that arrive out of order, and trying to avoid creating congestion along the way. However TCP is a rule-based approach to control congestion. Measures to avoid congestion are defined beforehand, making them inherently rigid and reactive. This makes them poorly suited to handle sudden burst of congestion or rapid shifts in traffic patterns. The paper [1] exposes a limitations of TCP's inherent rigid congestion control logic, that is the inability to distinguish between non-congestion loss and congestion induced loss. Non-congestion loss can happen when there is a handover between cell towers. In [1] they emulated non-congestion loss by ensuring that 1% of the transmitted data would be chosen at random and lost. They tested this on TCP cubic and found that the protocol would respond to every single occurrence of data loss by halving the sending rate. The paper also highlights that TCP is unable to efficiently adapt to varying network conditions. In an experiment where the network capacity is switching between 20 and 40 mbps every 5 seconds, TCP cubic is unable to fully utilize the windows when the capacity increases to 40 mbps. [1]

In [2], the focus is on TCP BBR. Although it is a model based approach, the parameters given to model are statically adjusted. A network can dynamically change with respect to elements such as traffic patterns, link failures, path changes. The lack of parameters that can be adjusted dynamically in response to the changing network conditions leads to inefficient behavior [2]

2.2. Reinforcement learning Theory

Reinforcement learning is a form of machine learning, that involves teaching an *agent* how to interact with an *environment* to achieve a specific goal. The agent has a number of different *actions* it can take, however it does not know which actions will help it reach its goal. Every action the agent takes will influence the environment causing it to change. Each instance of the environment is defined as a state, in other words each action the agent takes, causes a transition from one state to a new state. The environment will then respond to each action with a transition in state and a numerical *reward*. The reward can either be positive or negative, and the objective for the agent is to maximize the total accumulated reward over time. The reward signal can be defined in many different ways, however the numerical value that the agent receive is depended on the current action the agent takes and the current state the agent is in. As the agent continues to learn, it improves its ability to determine the probability of selecting each possible action for a given state. The mapping from state to a action is called a *policy*. [3]

Figure 1 shows how an agent interact with its environment in discrete time steps. At time *t*, the agent takes Action A_t, then the environment transitions from state S_t to $S_{\mathrm{t}+1}$ and reward $R_{\mathrm{t}+1}$ is given to the agent.

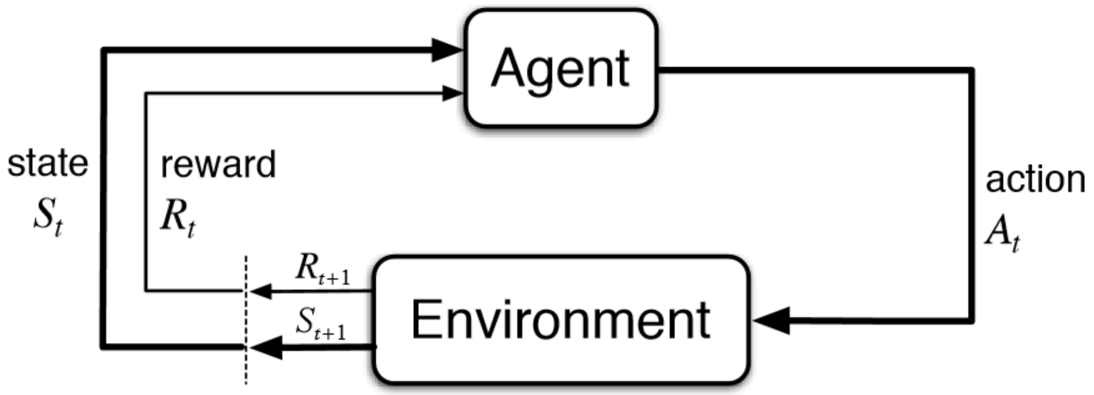


Figure 1: An agent interacting with its environment [3]

The environment is a key component in a reinforcement learning problem. Depending on the problem, it is possible to model the behavior of the environment, which is helpful for the agent and its learning process. With a model it is possible to predict how the environment will behave with with regards to state transition and possible future rewards. Reinforcement learning solutions that uses model are called *model-based* methods. There are also reinforcement learning solution that do not use a model to predict the behavior of the environment, these are called *model-free* method. These types of solutions are simpler and the agents in these methods learn by trial and error. [3]

2.2.1. Exploration vs exploitation.

In reinforcement learning, the objective of the learning agent is to maximize the total accumulated reward over time. This would require for the agent to take actions, that it knows yields high reward, meaning that it exploits known actions. However the agent will never figure out which actions yield the highest rewards without experiment and exploring all the possible actions it can take. If the agent commits to only exploitation, or if it only commit to exploration, then it will inevitable fail its task.

ε-greedy policy is one way to tackle the exploration vs exploitation tradeoff. The idea is to have the agent take action that exploits past experience most of the time, and once in a while based on a low probability, take a random action to explore new actions and learn.

2.2.2. Q-learning

Q-learning is a model-free reinforcement learning method. A central element of this reinforcement learning problem is maintaining and updating a *Q-table*. The Q-table is a matrix, and its dimensions is determined by the number of states there are in the environment and the number of actions the agent can take. Each Q-table entry contains the expected value (Q-value) of taking a specific action in a given state [3]. As the agent is learning, it will update these Q-values, using a Q-function:

$$Q^{\text{new}}(S_t, A_t) = (1-\alpha) \cdot Q(S_t, A_t) + \alpha \cdot \Big(R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a)\Big) \tag{1}$$

$Q(S_t, A_t)$, is the Q-value of the state-action pair S_t and A_t. $R_{\text{t+1}}$ is the reward received by choosing action A_t in state S_t. The learning rate is denoted as α, and γ is the discount factor.[3] The Q-learning algorithm can be seen in figure Algorithm 1

Algorithm 1: Q-learning [4]

```
1 Initialize Q-table with 0 in all entries.
2 Set first action $a$ using $\varepsilon$-greedy
3 Loop for each update:
4     Take action $a$
5     Observe reward ($R_{t+1}$) and state ($S_{t+1}$)
6     Update Q-table by Equation 1
7     Choose next action ($a'$) using $\varepsilon$-greedy
8     $a \leftarrow a'$
9     $s \leftarrow s'$
```

2.3. State of the art

The focus of [1] is investigating reinforcement learning in the context of congestion control. Their Framework is a RL-based congestion control protocol named Aurora. It is an extension of the performance oriented congestion control. Aurora uses deep reinforcement learning, where it generates a policy that maps network statistics to choices of sending rates. The actions the agents takes is changing the sending rate. However it does so periodically. They divide time into intervals, named monitoring intervals. At the start of each interval changes to the sending rate is made. The rest of the interval is used to gather network statistics to asses the impact of changing the sending rate with respect to a handful of network statistics. These statistics are namely, latency gradient, the derivative of latency with respect to time. latency ratio namely the ratio of the current monitoring interval's mean latency and the minimum observed latency of any monitoring interval. Lastly the sending ratio, which is the ratio between the packets that have been send and the once that has been acknowledged. They pass the gathered network statistics to a neural network, and their reward function is a linear function of throughput, latency and loss. Aurora uses an emulated environment when training. Using a relativly simple neural network, their agent is able to perform well in network scenarios different than the once it was trained on. Aurora is very robust. Furthermore Aurora outperforms state of the art congestion control algorithms such as BBR. However one of the remaining challenges of aurora is fairness. Given the results presented in the paper, they have not trained it in an environment with other traffic flows. It may be that it will not play fair when other traffic flows are present. [1]

The focus of [5] is tackling the challenge of deploying AI models on network devices. They present a solution to the problem, which is a computationally light solution based on Recently emerging reinforcement learning based Congestion control algorithms. Promising reinforcement learning based congestion control algorithms has been developed. Protocols such as the one used in [1], uses a neural network in their solutions. However neural networks are quite computationally heavy, if they are to be used in this context. Deploying a neural network on a Network Interface Card (NIC), is infeasible. The neural network cannot finish its computations while meeting latency requirements that are under 10 μsec [5]. Furthermore NIC cannot meet the memory requirements of a neural network, and the instruction set of the NIC does not support the necessary mathematical libraries needed to implement a neural network [5]. The work presented in [5] is focused on how to overcome these challenges. They have successfully developed a method that compresses a complex reinforcement learning based congestion control algorithm, by distilling its neural network into decision trees. By first training the neural network, and then using a method called gradient boosted tree, they successfully managed to reduce the inference time by 500-fold, from 450 μsec and thereby enabling real-time decision-making within the strict latency requirements, with minimal impact in performance [5]. Their solution outperform current state of the art congestion control algorithms for data centers such as DCQCN and SWIFT.

The work in [2] is focused on developing a multi agent deep reinforcement learning framework that will work along side a rule based TCP protocol. The The agents are not directly involved in the congestion control loop, but are focused on dynamically changing parameters. They use BBR as a case study, where the goal of the framework is to dynamically adjust the parameters such as bottleneck bandwidth and Round trip propagation delay. [2]. They compare the performance of their framework with TCP BBR, where the network parameters are not dynamically adjusted. They found that by using the framework, the tuned BBR was more adaptable to changing network conditions as compared to traditional BBR. The tuned BBR was able to estimate round trip propagation more accurately in their dynamically changing network. Additionally they were able to lower Peaks in Round Trip Time (RTT) values [2]

2.4. Problem Definition

The nature of a computer network, in regards to dynamic traffic, unpredictable congestion, and potential packet loss, makes it hard to model. A model-free reinforcement learning solution such as Q-learning would be an ideal candidate to base a reinforcement learning agent on. Furthermore it would also be interesting to investigate how the reward function affects the performance of such an agent. The goal of an agent is to maximize its cumulative reward, and there are many different ways to define a reward function. Investigating what metrics should be included in the reward, and the importance of each metric relative to each other is highly interesting. The reward function can be the a deciding factor between wether an agent succeeds or fails in reaching its goal. This leads to the following problem definition

How can Q-learning be used in TCP congestion control, and how does the reward function affect the performance?

Chapter 3.
Design For QTCP

This chapter will provide description of the reinforcement learning based TCP protocol. The description will include state space representation, and design consideration for the reward function the agent uses to navigate its environment and a description of the action space.

3.1. State space

The representation of the state space is a key shortcoming of Q-learning. Although accurately representing the state space often suggests using numerous metrics, this approach leads to a high-dimensional state space, which in turn increase the time it takes for the agent to reach a stable policy. Additionally it causes the state space to grow exponentially [6]. With this shortcoming in mind the state space is limited to only three network metrics. Namely the mean values of RTT, time interval between packets being transmitted, and time interval between acknowledgements received. These three metrics are inspired by [6]. Using both the mean value of the sending rate along with the rate at which acknowledgements are received, allows the agent to decern instances where congestion is perceived on the network. If the average sending interval is much smaller than the mean interval between received acknowledgements, then ideally the agent would notice that it is flooding the network with too much traffic and as a result causing congestion.[6]

3.2. Reward Function

In the context of congestion control, the goal is to transmit as much data as possible as quickly as possible, without causing congestion on the network. If an agent is to achieve this goal, then it needs an appropriate reward function that can guide it. The reward function must be designed such that it will reward the agent for being in states where the RTT is low while maintaining a high throughput.

The most straight forward reward function would be:

$$\mathrm{R} = \mathrm{TP} - \mathrm{RTT} \tag{2}$$

where TP and RTT denotes throughput and Round Trip Time respectively. In this case a relatively high reward would be attained if the throughput is maximized while the RTT is minimized. However given the nature of these two network metrics, one of them may dominate the reward signal. It is common to measure RTT in millisecond, and throughput as megabits per second. If these values are used then throughput will naturally dominate the reward function. Changes in RTT would be almost inconsequential in the reward signal.

One approach would be to normalize the elements in the reward signal, such that their effect on the reward signal is balanced. The modified reward function would then be:

$$R = TP - C_{norm} * RTT \tag{3}$$

where $C_{norm} = \frac{TP}{RTT}$. By introducing the constant C, the two reward element, TP and RTT, will be in the same scale. However a new problem arises by introducing C. Which values should be used in the normalization constant? Should the normalization constant be a dynamic value that changes throughout training? Should it be a static value that is set at the start of training phase?

Assume that the capacity of the network is unknown, in this scenario the normalization constant must be a dynamic value that is set based on the minimum RTT measured and the maximum Throughput measured. However a constantly changing normalization constant may have an unintentional and unpredictable affect on the training of the agent. the unstable element in the reward function will under certain conditions yield different reward values for a being in a particular state. Meaning that depending on when in the agent finds itself in a particular state, the reward it will receive will not always be the same. Unless there is a method for ensuring that a consistent reward signal is distributed for any given state regardless of when it is visited, a dynamically changing normalization constant is not a viable solution. Besides, if the values used in C for RTT and TP are constantly changed, then the reward will always be 0.

Assume that the normalization constant is static. Additionally, Assume that the capacity of every link on the network and the network topology is known, and that the the propagation delay is also known. One possible option is to use the bottleneck link capacity as a reference for TP, and the propagation delay multiplied by two as a reference for RTT in the constant C.

$$C_{norm} = \frac{BtlBw}{RTprop} \tag{4}$$

Using these values would balance the reward function, and as a result give RTT and TP a relatively equal influence on the reward signal. However, this reward function will only yield negative reward values, unless the agent is transmitting data with a throughput equal to BtlBw while maintaining a RTT value equal to RTprop, in which case the reward signal will be 0. However such a reward function does successfully balance the influence of RTT and TP.

Whether or not those two metrics should have a balanced influence on the reward function is a separate matter. Depending on the situation under which the congestion control algorithm is applied to, there might be certain requirements for the latency or the throughput. There are certain scenario where latency is not a crucial factor, but emphasis is laid on achieving high throughput for example in satellite communication. There are other scenarios where the primary concern is to achieve low latency values, and throughput is only a secondary concern such as in online gaming. To accommodate these concerns weights can be introduced to the reward function. Weights that can be adjusted to the needs of the application. Such a reward function would be:

$$R = C_\alpha * \mathrm{TP} - C_\beta * C_{\mathrm{norm}} * \mathrm{RTT} \tag{5}$$

The weight C_α and C_β can take any value, and be set to accommodate the requirements of the scenario

Until this point, the reward functions that has been considered were based on the difference between TP and RTT. Reward functions can be designed in many different ways, and given that the goal is to maximize throughput while minimizing Latency, a different approach for the reward function would be:

$$R = \frac{\mathrm{TP}}{\mathrm{RTT}} \tag{6}$$

This reward function also accomplishes the goal of yielding high values when RTT is kept low while TP is kept high. It also possible to add weights to this reward function

3.3. Actions space

The action space of the agent will be kept simple. The agents only actions are to either increase the congestion window, decrease the congestion window or maintain it. The agent will follow an additive increase additive decrease scheme.

Chapter 4.
Evaluation Methodology

This focus of this chapter is to present the implementation of the Q-learning based Transmission Control Protocol, and present the network topology.

4.1. Network Simulator.

The goal of this project is to design, implement and evaluate a reinforcement learning based congestion control algorithm. To meet this goal a network simulator will be used namely NS-3. NS-3 is a discrete event network simulator, and it is primarily used for research purposes. A network identical to the one presented in figure Figure 3 will be implemented in NS-3. [7]

Furthermore, NS-3 Allows developers to use a number of different versions of TCP, such as TCP New Reno, TCP Vegas, and many more in their research. Furthermore, NS-3 also allows developers to design and implement their own Congestion control algorithm [7]. Implementing a new congestion control algorithm in NS-3 comes down to defining a Class that inherits from the NS-3 class *TCPCongestionOps* [8]. This class contains all the methods and operations essential for a Transmission Control Protocol.

4.2. Network Topology

The following section will present the network topology of the 9th semester project and the new updated network topology.

Figure Figure 2 shows the network topology used for both development and testing of QTCP in previous semester. The topology is simple, with one node dedicated to generating traffic, and one node dedicated to receiving traffic. However this network topology does not capture the necessary behavior and complexities of networking.

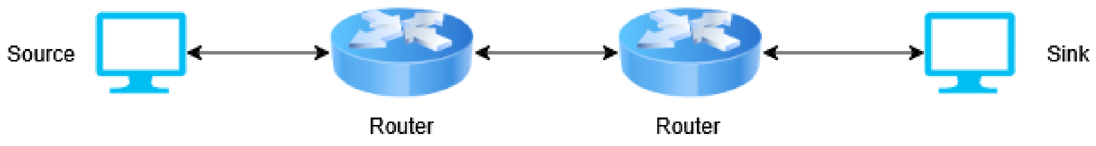


Figure 2: Previous network topology

Given that there is only one node generating traffic, the aspect of having link shared between multiple nodes is not captured with this topology. This motivated the design of a slightly more complex network topology that has a bottleneck link. Such a network topology is shown in figure Figure 3

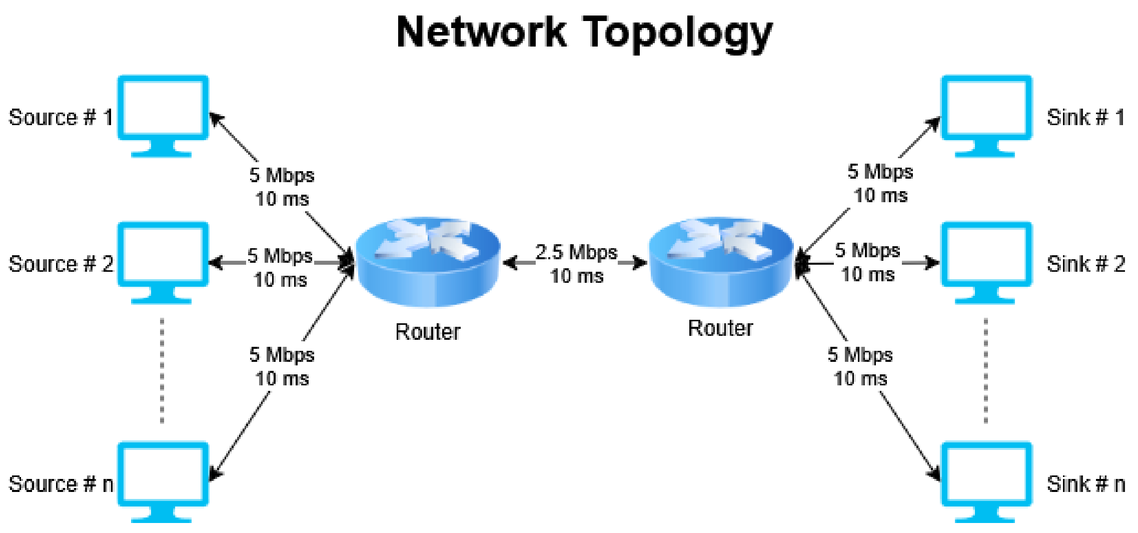


Figure 3: Dumbbell topology

In figure Figure 3, there are up to *n* different source nodes generating traffic, and *n* different sink nodes all receiving traffic. All source nodes transmit their data to the left router, thereby forcing all the traffic through the single link between the two routers before the traffic arrives at its designated destination. Written on each link on between devices in figure Figure 3 are the capacity of the individual link. Every single link except for the link between the two routers, have a bandwidth of 5 Mbps and a propagation delay of 10 ms. The bottleneck link only has a bandwidth of 2.5 Mbps. Setting the capacity of the link between the two routers to half of the every other link on the network, ensures that this particular link becomes a bottleneck in the network. In this particular network topology, the minimum RTT achieveable assuming no processing delay would be 60 ms.

4.3. Existing Protocol used for comparison.

Extensive research has been conducted in the field of congestion control algorithms and as large number of congestion control algorithms has been developed and deployed for various use cases. Therefore there are a large number of different protocols to compare the reinforcement learning based protocol to. It is not feasible to compared the agent to all TCP variants. It has therefore been decided to compare the RL-based TCP protocol to three interesting TCP variants. These are TCP Vegas, TCP New Reno and TCP BBR. TCP New Reno can be described as a very aggressive protocol that emphasizes throughput. This protocol tries to maximize its throughput until packet loss occurs.

On the other hand TCP Vegas is a more conservative protocol that focuses on achieving low latency. TCP Vegas regulates the congestion window based on changes in measured RTT. TCP BBR on the other hand uses both throughput as well as RTT to regulate the congestion window. The goal of BBR is to find and maintain Kleinrock’s optimal operating point.

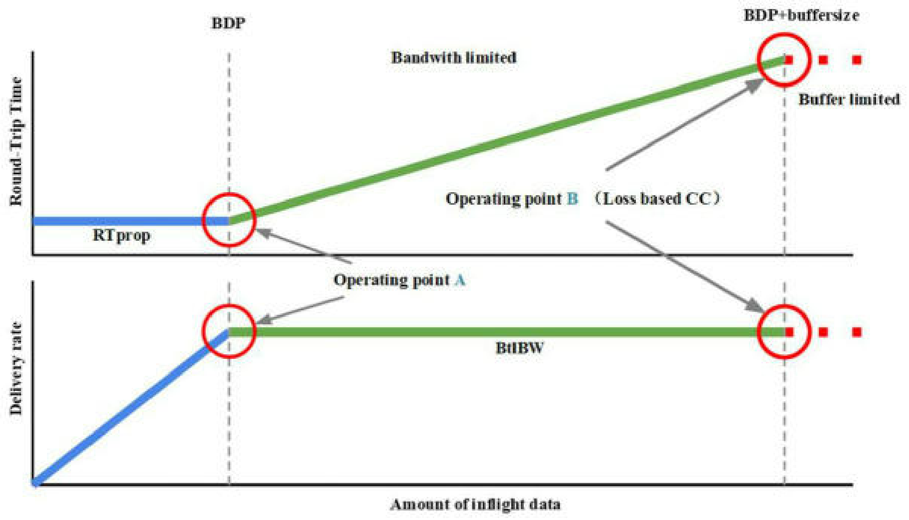


Figure 4: Kleinrocks optimal operating point [9]

In other words the objective of the protocol is to maximize throughput while minimizing throughput. To achieve this goal BBR avoids queuing delay, by estimating the bottleneck link's bandwidth and keeping track of the minimum observed RTT.

It would be interesting to see how the RL-based TCP protocol would perform in a network with other traffic flows following TCP Vegas or TCP New Reno.

Feature	TCP New Reno	TCP Vegas	TCP BBR
Congestion Indicator	Packet Loss	Fluctuation in RTT	Bandwidth and RTT
Primary Focus	Throughput	Latency	Bandwidth efficiency and low latency
Reaction style	reactive (after loss)	proactive (before loss)	proactive

Chapter 5.
Model Development.

In this section the development of the model used for this version of QTCP will be presented. The focus of this section is will be directed toward finding an optimal reward function. A number of different reward functions will be experimented with. The initial reward function used in previous project showed unexpected results with the new network topology.

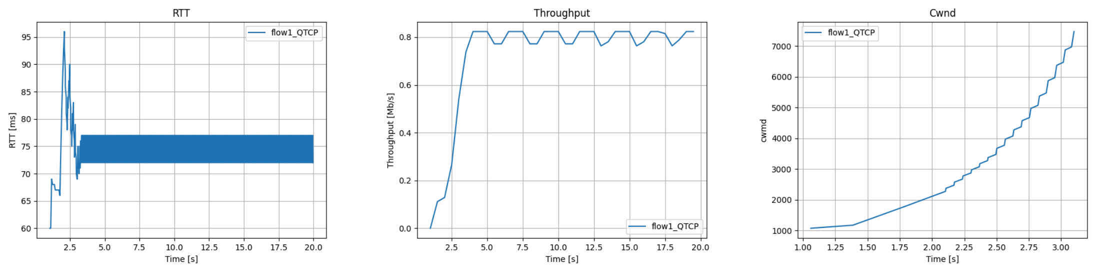
Multiple reward functions will be presented in this chapter. Each reward function will mainly be evaluated based on throughput and round trip time in various network setup.

5.1. Reward Function Version 1.

The reward function used in previous project is presented in this section. It is defined as:

$$\text{reward} = \text{TP} - \text{RTT} \tag{7}$$

The performance of the agent is shown in Figure 5. The agent successfully minimized the RTT, however given that there is only one traffic flow on the network, the agent should be able to fully utilize the capacity of the bottleneck link which is set to 2.5Mbps. In Figure 5 B, we see that the agent is only able to reach 1.5Mbps throughput the entire simulation.



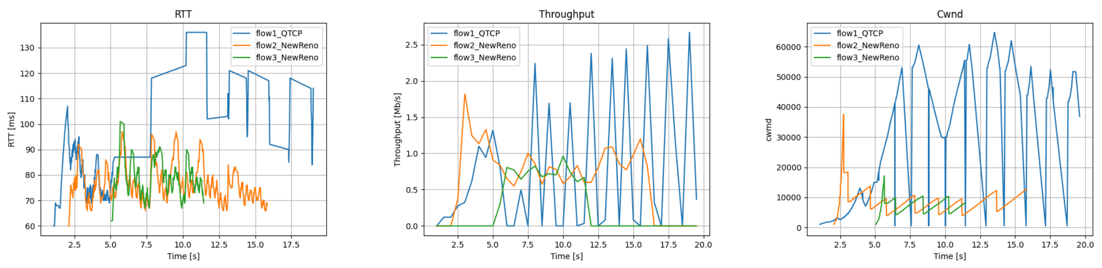
a) RTT b) Throughput c) CWND

Figure 5: Reward Function version: 1. The one used in previous project

Given the reward function, it appears that the agent learns to prioritize minimizing RTT while neglecting throughput. The agent's inability to fully utilize the capacity of a network where it is the only traffic flow, is rather peculiar. Although these results are not optimal, it is interesting to investigates how well this model performs in a network with multiple other traffic flows. The plot over congestion window only shows data until 3 seconds into the simulation. That is because of the mechanism behind how this metric is monitored. It is only when there is a change in the congestion window, that it is noted down. Since there are no more data for congestion window after three seconds, that means that the congestion window remains static after this point and until the simulation finishes

5.1.1. Reward Function Version 1 vs TCP New Reno

Figure Figure 6 shows how well the agent performs in a network where it has to competing for bandwidth with against two other TCP New Reno traffic flows.



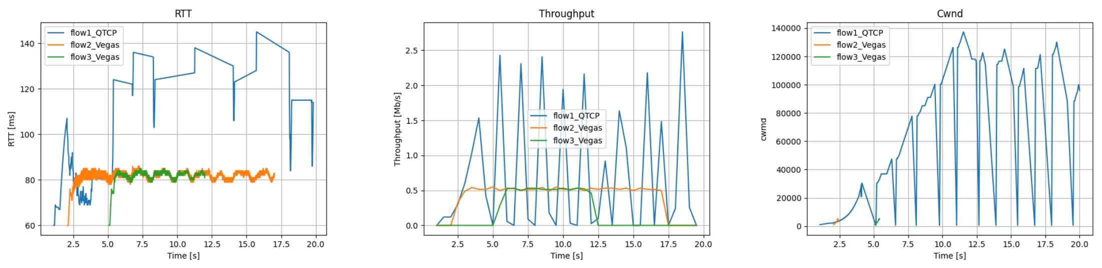
a) RTT b) Throughput c) CWND

Figure 6: Reward Function version: 1. Trained in a network with multiple TCP New Reno traffic flows

The agent is able to hold its ground against the other Traffic flows, and compete in a relatively fair manner against the two other New Reno traffic flows. However shortly after the third traffic flow begins its transmission, the agents performance becomes unstable.

5.1.2. Reward Function Version 1 vs TCP Vegas

This test evaluates how well the agent performs after it has been trained in a network with other traffic flows that uses TCP Vegas. The results of this test is presented in Figure 7



a) RTT b) Throughput c) CWND

Figure 7: Reward Function version: 1. Trained in a network with multiple TCP vegas traffic flows

TCP vegas is a passive protocol compared to TCP New Reno. This is reflected in the throughput and congestion window plots in figure Figure 7.

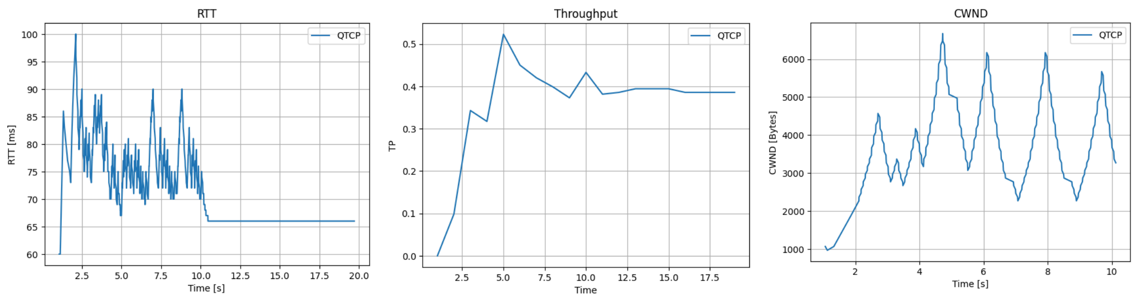
Compared to the reinforcement learning agent, the rule-based TCP protocol is only able to transmit data at a rate of 0.5 mbps. The agent on the other hand exhibits unstable behavior when the third traffic flow begins its transmission of data. The RTT plot in figure Figure 7 shows that the agent is able to keep low RTT values until the third traffic flow begins. There is a Drastic increase when the third traffic flow is initiated.

5.2. RTT Focused Reward Function .

In this iteration of QTCP, the reward function used to navigate its environment will be the Reciprocal value of RTT, ensuring that lower values of RTT will yield higher rewards. The reward function is defined this way, in order to investigate how isolated performance metrics as reward functions will affect the agent.

$$\text{Reward} = \frac{1}{\text{RTT}} \tag{8}$$

Figure 8 Shows the performance of the RTT focused agent. Although the agent converges to a very low RTT value, it does so needing more time when compared to the original Reward function shown prior to this. It takes the agent 10 seconds to reach a stable and low RTT value. Furthermore the throughput measured from this agent is critically low. The agent only manages to utilize a fraction of the available capacity. This behavior is to be expected, as the agent only concerns itself with minimizing RTT.



a) Round Trip Time b) Throughput c) Congestion window

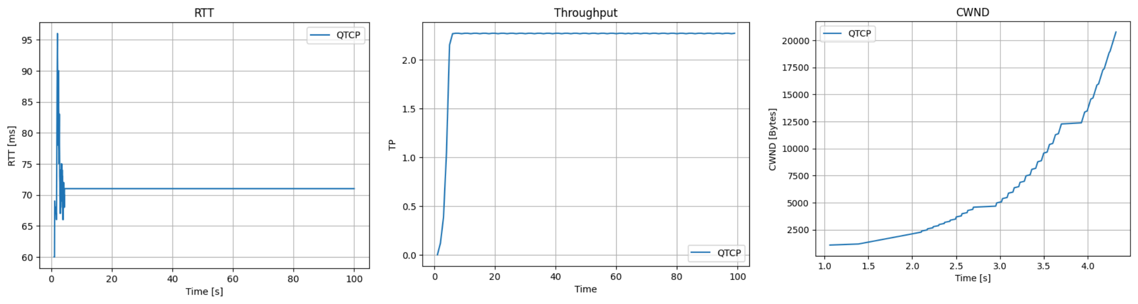
Figure 8: Reward Function: RTT focused

5.3. Throughput Focused Reward Function.

In this iteration of QTCP, the reward function used to navigate the environment will be throughput. This choice is made in order to investigate how Throughput as the isolated performance metric in the reward function will affect the behavior of the agent. The reward function is defined as:

$$\text{Reward} = \text{Throughput} \tag{9}$$

Given that the agent is only using throughput to evaluate the current network states, it is expected that it will solely focus on maximizing throughput as much as possible. It is also expected that RTT values measured by this agent will be relatively high. That is because RTT is not a metric that is considered in the reward function. The performance of this agent is shown in Figure 9



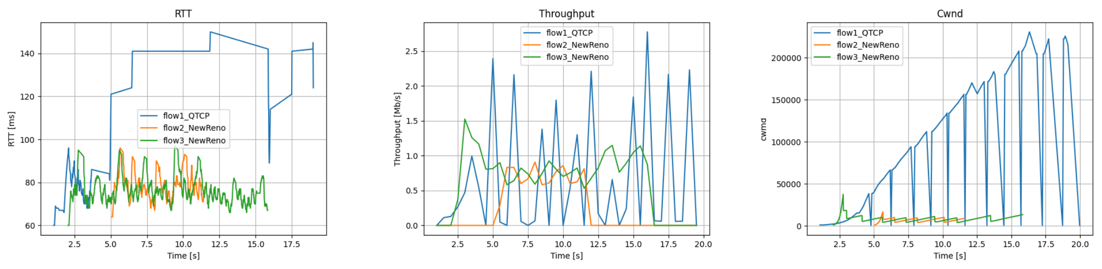
a) Round Trip Time b) Throughput c) Congestion window

Figure 9: Reward Function: Throughput focused

The Results shown in Figure 9 are not quite as expected. The agent successfully manages to maximize its throughput, as expected given the reward function. However the agent manages to stabilize the latency of its traffic flow. It is shown that the agent manages to maintain an RTT value of 71ms. Although higher than the prior iterations of the reward function, it only a minimal difference. The RTT value is still surprisingly low, despite now being considered in the reward function. However solely focusing on throughput might lead to problems when other traffic flows are present in the network. The agent might greedily use up all the available bandwidth and leave nothing for other traffic flows. This behavior may not be ideal, since it does not ensure fairness on the network.

5.3.1. Throughput Focused Reward Function vs TCP New Reno

This test is conducted for the purpose of verifying how well a throughput focused reward function will perform when the agent is put in an environment with other traffic flows. In this particular test, the other traffic flows follow New Reno. The results are shown in figure Figure 10. The figure shows 4 different plots, each showing the progression of RTT, throughput, congestion window over time, and the cumulated reward over episodes of training



a) RTT b) Throughput c) CWND

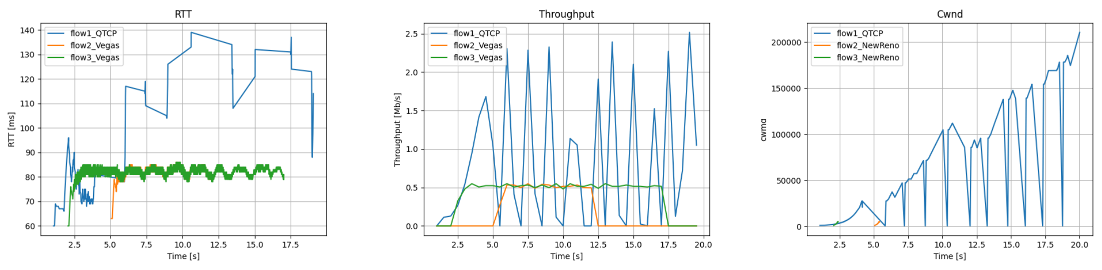
Figure 10: Throughput focused Reward function. Training in a network with multiple TCP New Reno traffic flows

As expected the agent performs poorly when it has to share capacity of links with other traffic flows. The agent gets off to a great start, however things take a turn for the worse as other traffic flows are introduced to the network. There is a slight spike in RTT when the first rule-based traffic flow is introduced to the network 2 seconds into the simulation. However there is a major spike in RTT observed on by the agent's traffic flow 5 seconds into the simulation, which is when the third traffic flow initiates its own transmission of data.

In regards to the throughput of the agent, the plots show that it is incredibly unstable. It spikes periodically and even reaches values over 2mbps at times, however it drastically falls down to 0 mbps shortly after each spike. The congestion window is also increasing at an extreme rate, however there we see that the congestion window falls down to 0 at exactly the same times as when the throughput reaches 0.

5.3.2. Throughput Focused Reward Function vs TCP Vegas

This test was conducted to confirm that an agent that follows a throughput focused reward function fails to perform adequately in a network where other traffic flows are present. The other traffic flows follow The TCP vegas protocol. The results of this test are shown in figure Figure 11



a) RTT b) Throughput c) CWND

Figure 11: Throughput focused Reward function. Training in a network with multiple TCP Vegas traffic flows

Just as we saw with the result in figure Figure 10, this version of the agent is unstable in a network with other traffic flows, even if the other traffic flows subscribe to TCP vegas protocol.

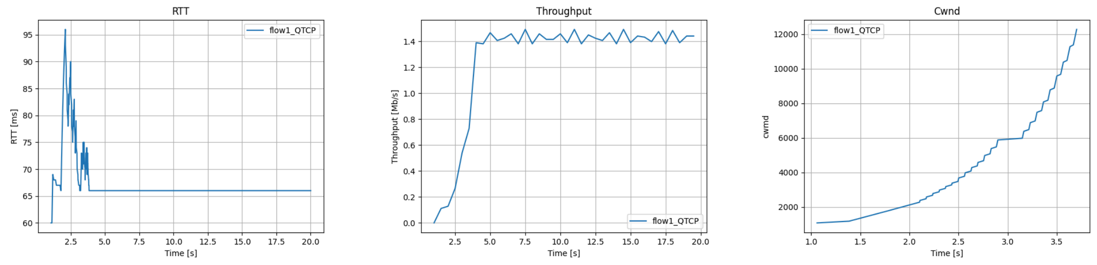
5.4. Reward Function Version 2.

In this iteration, the agent will use a reward function that includes both RTT and throughput, however a constant will be multiplied onto the RTT value.

$$\text{Reward} = \text{TP} - C * \text{RTT} \qquad (10)$$

where $C = C_\beta * C_{\text{norm}}$.

The performance of this iteration of the agent is shown in Figure 12.



a) RTT b) Throughput c) CWND

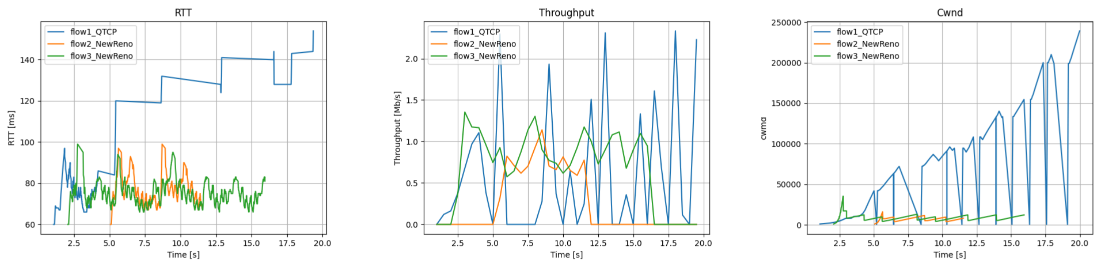
Figure 12: Reward function version 2. Trained and evaluated alone in a network

Different values for the constant C, has been evaluated. The best results were seen when $C = 3$. That means that the weight $C_{\beta} = 0.072$, and $C_{\text{norm}} = \frac{2500}{60}$.

In figure Figure 12, we see that the agent is able to maintain a really low RTT value, while using a 56% of the available capacity. These results are very similar to the once seen in Figure 5. There is only a slight difference in the throughput. While reward function version 1 has a throughput of 1.6 mbps, this reward function namely version to has a throughput of 1.4 mbps.

5.4.1. Reward Function Version 2 vs TCP New Reno.

In this test we evaluate how well version 2 performs in a network where it has to compete with TCP New Reno, the results are shown in figure Figure 13



a) RTT b) Throughput c) CWND

Figure 13: Reward function version 2. Trained and evaluated in a network with multiple TCP New Reno traffic flows

This iteration of the agent is unable to perform in a stable manner, when it is placed in a network with competing traffic flows. The RTT plot in figure Figure 13, shows the progression of RTT throughout the simulation. When all three traffic flows are transmitting simultaneously, we see that the agent experience a drastic increase in RTT. The throughput of the agent is stable until the third traffic flow initiates its own transmission. Even when the agent is the only one remaining in the network, the throughput remains unstable.

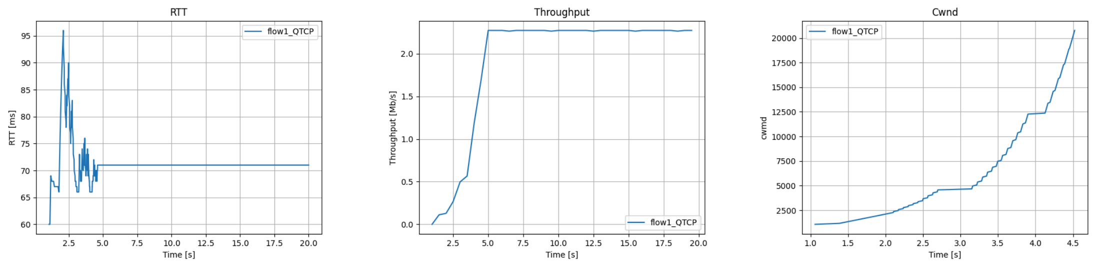
5.5. Reward Function Version 3.

In this iteration, the agent will also use a reward function that includes both RTT and throughput, however a weight will be applied to the Throughput-term of the reward function. The reward function will be defined as:

$$\text{Reward} = \text{TP/C - RTT} \tag{11}$$

where $C = C_\alpha * C_{\text{norm}}$. Multiple different values for C has been evaluated, and the best results were seen when $C = 0.5$, $C_\alpha = 0.012$ and $C_{\text{norm}} = \frac{2500}{60}$

The performance of this iteration of the agent is showed in Figure 12. And it is very promising.



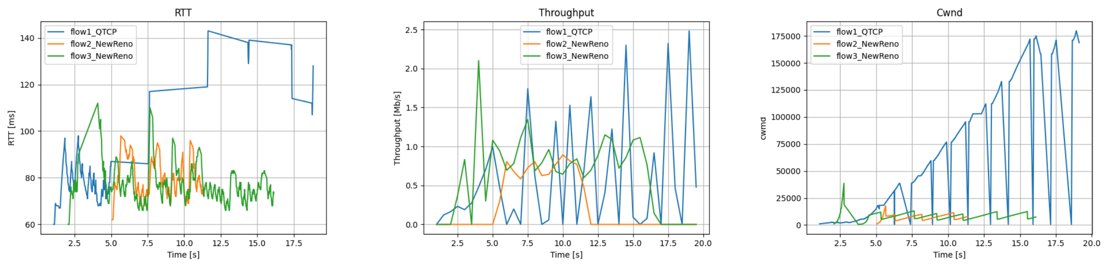
a) RTT b) Throughput c) CWND

Figure 14: Reward function version 3. Trained and evaluated alone

The agent is able to converge to a low RTT value relatively quickly and also ensure a high throughput throughout the simulation. Unlike version 1 and 2 of the reward function, this version is able to use the entire capacity of the bottleneck link when it is alone in the network

5.5.1. Reward Function Version 3 vs TCP New Reno.

Despite showing promising results when this version of the agent was tested in a network where it was the sole actor, it performs rather poorly when other traffic flows arrive to the network. Figure Figure 15 shows the results of this particular test.



a) RTT b) Throughput c) CWND

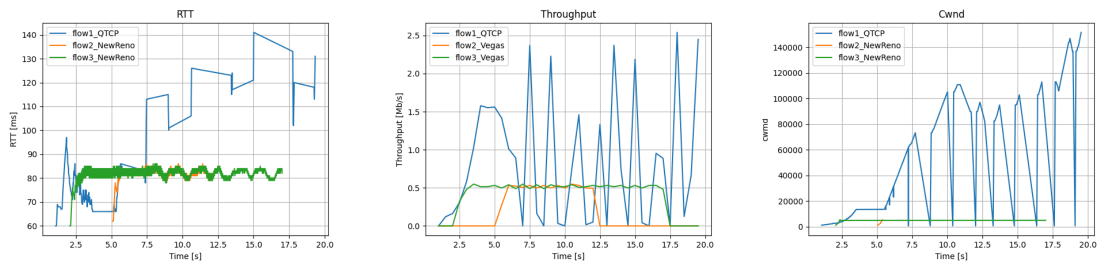
Figure 15: Reward function version 3. Trained and evaluated in a network with multiple TCP New Reno traffic flows

Although the RTT values of the agent experience a drastic increase just as we saw in version 2, it happens at a later point in time in the simulation. The drastic increase in

RTT is measured 7,5 seconds into the simulation. However the throughput of the agent is seemingly stable until the third traffic flow is initiated, 5 seconds into the simulation.

5.5.2. Reward Function Version 3 vs TCP Vegas.

This version of the reward function is rather stable when there are only two traffic flows on the network, however when a third traffic flows is introduced, the performance of the agent deteriorates. In figure Figure 16, we see that the agent performs very well in regards to throughput and RTT until 7.5 seconds into the simulation.



a) RTT b) Throughput c) CWND

Figure 16: Reward function version 3. Trained and evaluated in a network with multiple TCP Vegas traffic flows

At 7.5 seconds, the RTT values drastically increase to values much higher than the once measured for TCP-vegas, while the throughput begins to oscillate between two extreme values.

Chapter 6.
Tests

6.1. Hyper-parameter tuning

In the model development section, the performance of the different versions of the agents were shown. Every iteration of the learning agents fail when their are trained in network with other traffic flows. Choosing which of them to perform hyper-parameter tuning on will be an arbitrary choice. The network setup used for hyper-parameter tuning will be a network with a single reinforcement learning agent, and 2 TCP New Reno traffic flows.

6.1.1. Parameters.

The parameters that will be tuned are:

1. Learning Rate
2. Discount Factor
3. Decay Rate in the ε-greedy policy.

The default value used in model development for each of them are:

1. $\text{Learning Rate} = 0.7$
2. $\text{Discount Factor } = 0.2$
3. $\text{Decay Rate} = 0.005$

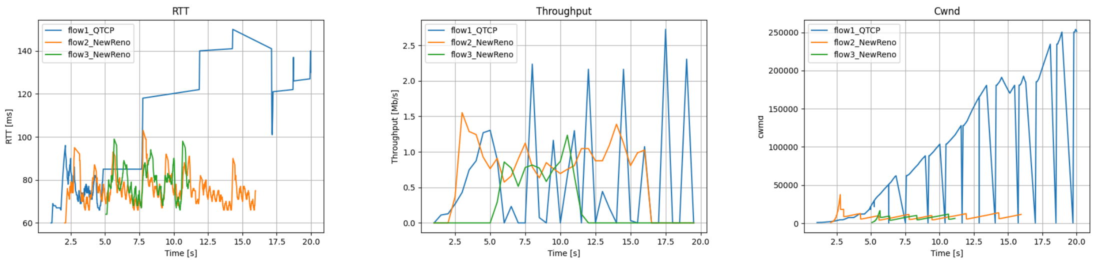
These three parameters will each have a predefined discrete value space, and then a value within this space will be randomly selected. The randomly selected combination of these three parameters, that has the highest accumulated reward will be used in further tests.

The value space for each of the hyper-parameters are:

1. Learning Rate: *{**0.1**, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7}*
2. Discount Factor: *{0.90, **0.95**, 0.98, 0.99, 0.995}*
3. Decay rate: *{0.005, **0.01**, 0.015, 0.02}*

The combination of hyper-parameters that yielded the highest score are marked in bold above. These values are used for their respective parameter in the following tests. The results can be seen in Figure 17 and in Figure 18

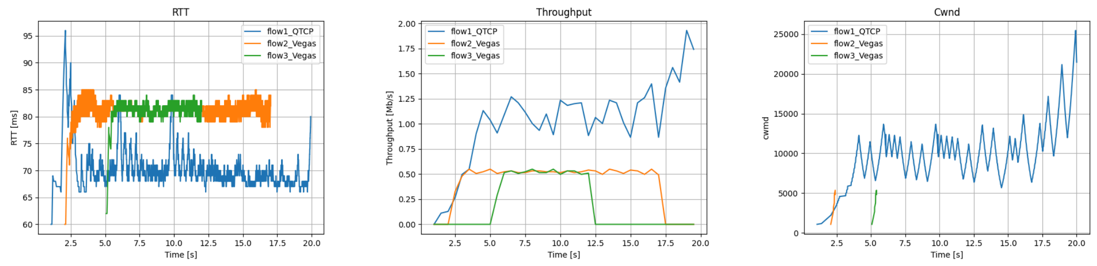
Despite the efforts of tuning the hyper-parameters, there is not a noteworthy change in the performance of the agent when it is trained in a network with multiple TCP New Reno traffic flows. Shortly after the third traffic flow is initiated, the agent sees a drastic increase in RTT. The throughput of the agent also becomes very unstable, fluctuating between 0 and 2 mbps.



a) RTT b) Throughput c) CWND

Figure 17: Results after Hyperparameter tuning. Agent trained along side TCP New Reno traffic flows

However in Figure 18, we see a drastic improvement in performance, after tuning the hyper parameters. In Figure 7, we saw that the agent becomes unstable when the third traffic flow is initiated. However this is not the case in Figure 18. The agent is not only able to maintain an RTT that is lower than TCP Vegas throughout the simulation, it has a throughput that double that of the two TCP Vegas traffic flows. In this test, the agent is also able to stabilize it self, and increase its throughput, when the other two traffic flows seize their transmission. This is unlike any of the other evaluations.



a) RTT b) Throughput c) CWND

Figure 18: Results after Hyperparameter tuning. Agent trained along side TCP Vegas traffic flows

Chapter 7. Discussion

7.1. State-space representation

Q-learning has many benefits especially when applied to a computer networks. It is a simple solution that can work without a detailed model of the environment. Once the agent has learned, using the learned Q-values only requires a lookup in a table. The inference time of an agent that only needs to look up a table to make a decision will meet the computational and latency requirements of a Congestion control algorithm [5]. However this benefits is also one of the drawbacks of using Q-learning. If we want the agent to better sense its environment, then we must increase the state variables. However doing this will increase the size of the Q-table, since each state-action pair will need an entry in the table containing the Q-value of that given state-action pair. The proposed solution to this problem is to use function approximation. Papers such as [2], use a neural network in their solution. Instead of storing their q-values in a table, they provide the state variables to a neural network, which in turn will calculate the q-values associated with the action the agent can take. This solution allows the agent to better sense its environment, since state-variables are not a limiting factor anymore. Another benefit of this solution is its ability to generalize. If the agent should encounter a state, that is different from any of the states it has trained on, then it will be able to estimate reasonable q-values based on similar states it had visited in the past. This generalization ability is not possible with a solution that uses table look-ups. Congestion control algorithms have strict memory and latency requirements. According to [5], most congestion control algorithms have a latency requirements on their decision time which is $O(\mathrm{RTT})$. Depending on the size of the neural network, a substantial amount of memory might be required. It is not every Networks Interface Card that can meet the memory requirements of a neural network [5]

7.2. Action space

The actions space of the agent in this project is simple. The agent is controlling the congestion window by additive increase or additive decrease. Rule-based TCP traffic flows such as New Reno use an Additive increase multiplicative decrease scheme for their congestion window. It would have been interesting to see how the agent would have performed with a similar action space. In [5] they even investigated a solution that uses multiplicative increase multiplicative decrease. For further development, it would be interesting to investigate how a more complex actions space would affect the behavior of the reinforcement learning agent.

7.3. Reward function

In this paper, finding an optimal reward function has been the core objective. The metrics used in the developed reward functions are throughput and RTT. The results show that using these two alone in the reward function is not sufficient, in making the agent reach its goal. The agent was only able to achieve its goal while competing with other traffic flows, when the other traffic flows were TCP vegas. Including more performance metrics in the reward function might improve the performance of the agent. If packet loss was included in the metric, and the agent was heavily penalized when packet loss happened might have a positive affect on the performance. By including this metric in the reward function, could increase the agents ability to detect congestion.

The current reward function are only focused on maximizing throughput and minimizing RTT. There are no measures taken place to ensure that the agent will act in a fair manner on the network. In every iteration of the reward function, we see that the agent is acting in a greedy manner. Although unstable, it tries to send as much data trough the network while disregarding other traffic flows. Even in the last test the agent's sending rate is double that of the two other TCP vegas traffic flows. Given the nature of how the agent is implemented, it does not have information of how many traffic flows it is competing against. However, assuming that the agent has access to this information, then the most straightforward solution would be to have the agent measure the available capacity, divide it by the number of traffic flows, and have the agent transmit at a rate matching this fraction.

7.4. Hyper-parameters.

Hyper-parameter tuning has been a major success for the agent's performance. Figure 18 and Figure 7, shows two agents, both with the same reward function, state space and action space. Both agents are also competing against TCP vegas. The only difference is the hyper-parameters used by the agent. By adjusting the learning rate, discount factor and epsilon decay rate, we see incredible improvement in performance by the agent. However the hyper-parameters used in Figure 18, are actually the ones found to be optimal for an agent competing against two TCP New Reno traffic flows. Strangely enough these hyper-parameter values, did not significantly improve the behavior of the agent competing against two TCP New Reno traffic flows as shown in Figure 17.

Because of time constraints, it was not possible to perform hyper-parameter tuning on the other reward function. It would have been interesting to see how the agent with reward function version 3, would perform when it was given optimal hyper-parameters.

Chapter 8. Conclusion

The work this project was centered around the problem definition:

How can Q-learning be used in TCP congestion control, and how does the reward function affect the performance?

Q-learning can be used in TCP congestion control, by defining the the environment as the computer network, the agent as the congestion control algorithm, and representing the states as key performance metrics of the network. The network was simulated in NS-3, and the key performance metrics used as state variables are:

1. The mean RTT over an interval
2. The mean Time between successive transmission of packet.
3. The mean time between successive acknowledgements

The actions the agent could perform is to increase or decrease the congestion window following and Additive increase additive decrease scheme. The agent is also able to maintain the current congestion window.

Using the states and actions defined, the agent can calculate Q-values for each state-action pair using a Q-function.

As for how the reward function affects the performance of the agent. The results showed that the agents priorities and goals are depended on what metrics the reward function includes. If the agent is only using throughput as reward signal, it will naturally only focus on increasing throughput. Conversely if the agent only focuses on latency, it will work towards minimizing RTT, while disregarding throughput.

Chapter 8
Bibliography

[1] N. Jay, N. H. Rotman, P. B. Godfrey, M. Schapira, and A. Tamar, “Internet Congestion Control via Deep Reinforcement Learning,” *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1810.03259°

[2] S. Ketabi, H. Chen, H. Dong, and Y. Ganjali, “A Deep Reinforcement Learning Framework for Optimizing Congestion Control in Data Centers,” in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, IEEE, May 2023, pp. 1–7. doi: 10.1109/noms56928.2023.10154411°.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. in Adaptive Computation and Machine Learning series. Cambridge: MIT Press, 1998.

[4] U. A. M. Abukar, R. H. Davidsen, and C. Ejsing, “Reinforcement Learning Based CCA in TCP using ns-3.” Aalborg Universitet, 2024.

[5] B. Fuhrer *et al.*, “Implementing Reinforcement Learning Datacenter Congestion Control in NVIDIA NICs,” in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, May 2023, pp. 331–343. doi: 10.1109/ccgrid57682.2023.00039°.

[6] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, “QTCP: Adaptive Congestion Control with Reinforcement Learning,” *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2019, doi: 10.1109/TNSE.2018.2835758°.

[7] “TCP models in ns-3.” [Online]. Available: https://www.nsnam.org/docs/models/html/tcp.html#writing-a-new-congestion-control-algorithm°

[8] “ns-3: ns3::TcpCongestionOps Class Reference.” [Online]. Available: https://www.nsnam.org/docs/release/3.28/doxygen/classns3_1_1_tcp_congestion_ops.html#details°

[9] W. Zhao, Z. Li, Q. Yang, and Q. Wang, “Optimization of BBR Congestion Control Algorithm Based on Pacing Gain Model,” *Sensors*, vol. 23, no. 5, p. 2345, 2023, doi: 10.3390/s23052345°.

Chapter 8
Index of Figures