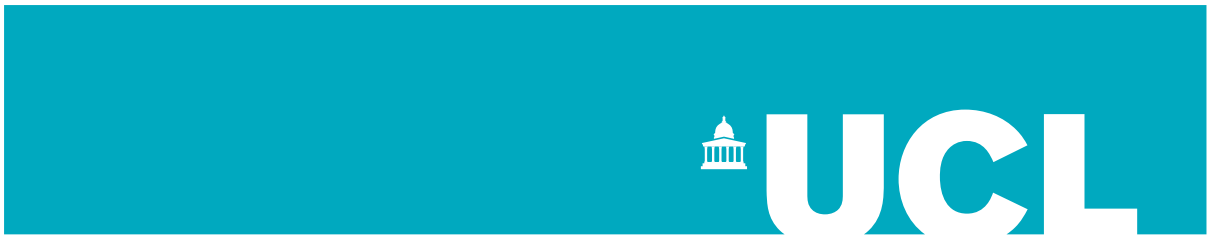

University College London (UCL)

ELEC0036

Dynamic Internet Congestion Control using Deep Reinforcement Learning

Author
ATIJIT (ATOM)
ANUCHITANUKUL

Student Number
17009878

Supervisor
Prof M Rio

Second Supervisor
Dr. R Grammenos

September 8, 2020
A BEng Project Final Report

Contents

Abstract

Rapid increase in the internet usage has proliferated the demand for data transmission over the networks around the world. In other words, the traffic over internet has become significantly more congested in recent years. As a consequence, the network latency and the number of lost or dropped packets of data have risen above the acceptable levels. Current Transmission Control Protocol (TCP) algorithms have been performing reasonably well in regulating the flow of data across networks to reduce congestion but lack the dynamic aspect to deal with today's fast-changing network environments. This research project seeks to develop a dynamic congestion control algorithm by developing a Deep Reinforcement Learning agent with the Advantage Actor-Critic (A2C) structure. This article summarises the all the accomplished developments that contribute to the main intention of the project.

1 Introduction

Over the last decade, the internet usage has grown significantly in size. New applications of the internet have emerged, each requiring a high performance of the network to transmit large amount of data. For a network with limited capacity, congestion would inevitably occur as it is overwhelmed with huge amount of data waiting to be transmitted.

Despite the state-of-art advancements of the internet, the same Transmission Control Protocol (TCP) has been employed for over three decades. TCP is a pre-configured rule-based algorithm designed to regulate how data of an application should be broken into small data packets and sent through the network, partially alleviating the problem of congestion. It is also responsible of handling lost or garbled packets by data re-transmission. Additionally, TCP algorithms have different pre-set goals, with some trying to maximise the throughput and some trying to minimise the queuing delay of the data transmission. This is due to the trade-off between high throughput and low latency. The inefficacy of implementing such a rule-based TCP is its inability to perform well in handling congestion for diverse network usage scenarios.

In this research project, the concept of creating a dynamic congestion control algorithm is introduced. The algorithm will be design such that it oversees the sender side of the network. The ideal congestion control design should be able to perceive the network structure and the level of congestion. Then, it should adjust the transmission rate for each application dynamically so that fairness between applications is ensured. The algorithm should be refined to the optimal point where suitable throughput is maintained, and latency is minimised to an acceptable value. To create such a sophisticated design, the concept of Deep Reinforcement Learning (DRL) is applied.

The main target of this research project is to create an agent with a dynamic congestion control algorithm that could outperform existing TCP algorithms used today. There are mainly two performance measures, system throughput and latency. Once trained, the performance of the agent should be compared against the performance of existing TCPs such as NewReno, XCP and Vegas. This comparison can be done by deploying these agents, one at a time, in a uniform simulated network. The throughput, latency and any other measurements can then be obtained from the simulation.

There are several related researches that have been conducted prior to the development of this project. The RemyCC algorithm, developed in [1], has a different, but interesting,

approach compared to this project's approach. The RemyCC algorithm is designed to observe the current structure of the network environment and would select the best rule-based Transmission Control Protocol (TCP) algorithm that would be able to regulate the internet traffic most effectively. Studying this algorithm provided the knowledge of TCP selection and environment observation that can be applied to this project.

The research done in [2] introduces another approach of using just Reinforcement Learning to develop a Q-Learning TCP algorithm. The research deals with the continuous state space using Kanerva Coding. Although the paper's approach only focuses on using reinforcement learning, the problem definition included at the beginning of the article is outlined in great detail. It identifies nearly all the main problems in the current rule-based TCP algorithms. Furthermore, understanding the methods applied in this research gives intuition into how Deep Reinforcement Learning can be applied.

Another related study in [3] introduces the approach of using Deep Reinforcement Learning to develop a congestion control algorithm, which can be applied to this project in the project design stage. As Deep Reinforcement Learning is the paper's approach, it shares a lot of similarities to this project's approach. However, the paper's approach is to create an end-to-end Deep Q-learning algorithm while this project's approach is to create a centralised A2C algorithm. The publication introduces several different reward functions that are effective in training the agent. The code published on GitHub (link provided on the paper) provided a good demonstration of how to create a simulated network environment in the OpenAI gym format from scratch.

In this report, Chapter 2 addresses the main goals and objectives of this project. The in-depth explanation of the concepts and theories required for this project is provided in Chapter 3. Chapter 4 demonstrates the steps required to implement both the agent and its environment. The report ends by showcasing the results of this project in Chapter 5 and the conclusion in Chapter 6.

2 Goals and Objectives

This project is classified as a theoretical and simulation project. In other words, this project intends to perform a rigorous study on the theory of Deep Reinforcement Learning (DRL) and apply the state-of-the-art DRL method of the Advantage Actor-Critic (A2C) to create a learning agent. For the simulation aspect, the project intends to create a simulation environment that resembles an actual network environment topology with a single bottleneck link between multiple senders and receivers. Hence, besides the primary goal of this project to create a dynamic internet congestion control algorithm, here are the essential aspect-specific objectives that contribute to the primary goal.

First of all, there are two objectives for the development of the agent. The first objective requires the agent to have the capability of altering some aspects of its environment by choosing an action at each step based on its perception of the current situation (state) of the environment. The second objective requires the agent to have the ability to improve its action selection capability by learning from the effectiveness of the chosen action. Once the learning (training) stage is complete, the agent is expected to perform better than the benchmark rule-based TCP NewReno algorithm by achieving higher throughput and low latency.

Secondly, there are three objectives for the creation of the simulation environment. Besides creating it to resemble an actual network environment, the simulation should be flexible and adjustable. That is, the simulation parameters, such as the bottleneck link bandwidth, the number of data senders and receivers, and the maximum size of data to be sent, can easily be adjusted to simulate different scenarios of the network environment. In addition to the flexibility of the simulation, the simulation environment should be able to provide sufficient information that represents the state of the environment to the agent. The simulation should also be capable of executing the action selected by the agent to change the environment state.

Last but not least, in terms of the programming aspect, the algorithms created in this project should be structured properly. Also, the code should contain sufficient comment lines to explain each part of the algorithm, making it easy for software developers to comprehend and further develop the code in the future. As this project will be developed using Python and C++, the structure of all the Python scripts in the entire program must follow the PEP 8 set of rules [4] while all the C++ scripts must follow the C++ Coding Standards [5].

3 Theory and Analytical Bases for the Work

This project aims to develop a dynamic internet congestion control algorithm that is capable of operating in different network environments. This algorithm is designed based on the concept of Deep Reinforcement Learning with the main goal of maximising the throughput and minimising the latency of data transmission across the network.

There are two crucial parts developed in this project: the agent and its environment. The agent needs to be capable of operating in the environment and learning to improve its future actions. The environment needs to be constructed to resemble an actual network environment the best way possible.

This chapter aims to provide thorough explanation of the fundamental concepts and theories essential to the development of this project.

3.1 Deep Reinforcement Learning

As mentioned previously, Deep Reinforcement Learning (DRL) is the core concept for the development of the agent. This concept is built upon two precursor concepts of Deep Learning (DL) and Reinforcement Learning (RL). The RL concept provides the underlying structure of the DRL agent while the DL concept extends the applicability of the agent. The explanation of this section is a succinct summary from various sources. It is based on Chapter 1-7, 13, 15.7 and 15.8 of [6], [7] and [8]

3.1.1 Reinforcement Learning

To begin with, let us understand the main properties of an agent. An agent interacts with its environment by making an observation of the environment. Depending on the nature of the environment, the observation that the agent makes could vary. An observation could be a set of measurements made from the agent's sensors (e.g. LiDAR sensor) if the agent were to autonomously drive itself to its assigned destination in a self-driving car scenario. Another possible observation could a real time image of the console screen if the agent were to play a game of Atari. These observations inform the agent of what situation or 'state'

it is currently in. Then, the agent would select an action that is best suited to its current state. To do so, the agent requires to have a policy that selects an action given the state. Let us denote s as the state, a as the action chosen and π as the policy. The expression $\pi(a|s)$ refers to the probability of the action a being selected given the current state s. The following expression shows that the sum of $\pi(a|s)$ for all possible actions in the state s is always 1.

$$\sum_{a} \pi(a|s) = 1$$

After the action is performed, the environment is changed to the next state, s'. Subsequently, the environment returns a reward, r, to the agent as an indicative signal of the effectiveness of the chosen action. In general, effective actions would yield high positive rewards while ineffective actions would yield low or negative rewards. The agent would then use this reward to update its policy to ensure that actions taken in the future will be more effective to receive higher rewards. A policy is considered optimal when it is capable of choosing actions that would yield the highest total reward at the end of the agent's operation.

The agent-environment interaction characteristic can be viewed as a sequential decision-making process where, at every step, the agent has to choose an action that is the most effective. It is also important to mention that the action taken at every step does not only affect the immediate reward, r, but also future rewards as it influences the next states that the agent will be in. This decision-making problem can therefore be formulated as a Markov Decision Process (MDP). A finite MDP refers to a problem that has finite sets of possible states ($\mathcal{S}$), actions ($\mathcal{A}(s)$) and rewards ($\mathcal{R}$). Although the problem in the network system is not finite, we will explore the case of finite MDP for simplicity reason. The concept of finite MDP can easily be altered to support the continuous case of this project.

From the start to the end of the agent's operation, the agent performs a series of actions at multiple states and receive rewards accordingly. Any instance of this recurring state-action-reward process can be viewed as a trajectory below,

$$S_0, A_0, R_0, S_1, A_1, R_1 ..., S_{T-1}, A_{T-1}, R_{T-1}, S_T, A_T, R_T$$
$$S_t \in \mathcal{S}, A_t \in \mathcal{A}(s), R_t \in \mathcal{R}$$

where a subscript denotes the time step (t) that the agent is currently in, and T denotes the terminal time step.

In the finite MDP, each of the random variables, S_t and R_t, has a discrete probability distribution that is entirely dependent on the previous state ($s \in S_{t-1}$) and action ($a \in A_{t-1}$). The probability of a particular set of the next state ($s' \in S_t$) and the reward ($r \in R_t$) occurring in the time step t can be determined by the following expression.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$
$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1$$

To achieve the agent's objective of maximising the total reward, the agent should seek to maximise the *expected return*, G_t, which can be expressed as a function of the reward sequence of the agent's trajectory, shown in the expression below,

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T \\ &= \sum_{k=0}^{T} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

where γ is the *discount factor*, $0 \leq \gamma \leq 1$. If $\gamma = 0$, the agent's objective would be to maximise the immediate reward, R_{t+1}.

Using the expression for the *expected return*, we can determine how good it is for the agent to be in a particular state or to perform a certain action, given a policy π. To do so, we can compute the *state-value function* ($v_\pi(s)$) and the *action-value function* ($q_\pi(s,a)$) respectively.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k R_{t+k+1}|S_t = s] \\ q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \end{aligned}$$

The *state-value function* ($v_\pi(s)$) can alternatively be expressed as a function of the *state-value function* of the successor state ($v_\pi(s')$). The final form of the following expression is called the *Bellman Equation*.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma \mathbb{E}[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma v_\pi(s')] \end{aligned}$$

As different agent policies have different *state-value functions*, we can find an *optimal policy* (π_*) that has the best *state-value functions*, meaning that it can maximise the return. A policy π is considered to be better than another policy π' if and only if $v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S}$. Thus, for the policy π_*, its *state-value function* is called the *optimal state-value function* ($v_*(s)$), and can be expressed below.

$$v_*(s) = \max_\pi v_\pi(s)$$

Similarly, the policy π_* also has an *optimal action-value function* ($q_*(s,a)$) which can be expressed below.

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

From the expressions for $v_*(s)$ and $q_*(s,a)$, one can notice that there is a relationship between the two functions. In order for the agent to be in the states with the highest values, it has to take actions that also have the highest values. This relationship can be described

using the *Bellman optimality equation* below.

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \\
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\
&= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]
\end{aligned}$$

A general practice to find the *optimal policy* is done by performing repeating sequences of *policy evaluation* and *policy iteration*. The *policy evaluation* process updates the *value functions* for all states for a fixed policy. Then, the *policy iteration* process uses the updated *state-value functions* to fabricate a new and possibly better policy. As mentioned before, these two processes are repeated until both the *value functions* and the policy converge, meaning that the *optimal policy* is found.

If a Reinforcement Learning problem can be described by a finite deterministic Markov Decision Process, the technique of *Dynamic Programming* can be adopted to search for an *optimal policy*. The method of *Dynamic Programming* updates the *state-value function* by making a prediction of the *value* of the current state based on the estimation of the *values* of the successor states. This mechanism is called *bootstrapping* as it makes an estimation based on other estimates. As a result, the *Dynamic Programming* method is capable of improving the policy without having to wait until the termination of the agent's operation.

On the other hand, if the MDP is not completely known, the *Monte Carlo* method is a solution for this problem type. Unlike the *Dynamic Programming* method, the *Monte Carlo* method does not rely on the *bootstrapping* method. That is, the *Monte Carlo* method updates the *action-value function* based on the actual rewards obtained by generating multiple episodes. An episode is an instance of the agent's possible trajectory from the start to the end. As a result, the *Monte Carlo* method is an actual learning method, unlike *Dynamic Programming*, as it learns an optimal policy from actual interactions with the environment. Additionally, this method can be adopted to develop an agent to operate in a simulated environment as, in some cases, episodes can be generated easily in a simulation. The *state-value function* can be estimated using the expression,

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

where $V(S_t)$ is the estimate of the *state-value function*, and $\alpha \in (0,1]$ is the step-size parameter.

By considering the characteristics of the agent environment (the network system) of this project, it is obvious that neither the *Dynamic Programming* method nor the *Monte Carlo* method can alone be used to create an agent to perform well under this circumstance. This is because the network system environment can not be formulated as a finite Markov Decision Process as it has a continuous state and action spaces. Therefore, the agent designed for

this project has adopted the *Temporal-Difference Learning* method which is a combination of both the *Dynamic Programming* method and the *Monte Carlo* method. Another concept of *n-step Temporal-Difference Learning* is also used in the second version of the agent to compare the performance between the two learning methods.

The *Temporal-Difference Learning (TD(0))* method is similar to the *Dynamic Programming* method due to the fact that it could update the *Value Function* by *bootstrapping* without having to wait until the termination of the agent's operation. Nonetheless, this method also shares similarity to the *Monte Carlo* method as it can learn straight from the actual experience of operating in the environment. The *Temporal-Difference Learning* update rule for the *Value Function* ($V(s)$) can be outlined in the expression below.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

From the expression above, the target for the update is the expression, $R_{t+1} + \gamma V(S_{t+1})$, which is the sum of the immediate reward (R_{t+1}) and the estimated *Value Function* of the next state ($V(S_{t+1})$).

The *Temporal-Difference Learning* method, compared to the *Monte Carlo* method, is faster as it only needs to store and compute the immediate reward, whereas the *Monte Carlo* method stores all the rewards from the agent's trajectory to compute the return (G_t). Nevertheless, the *TD(0)* method could sometimes be biased and unstable because it only learns based on just the immediate reward and the value estimate of the next state. If the training process starts arbitrarily wrong, the agent could converge to a wrong solution.

To reduce the potential bias of the *TD(0)* method, the *n-step Temporal-Difference Learning (n-step TD)* method offers a solution that is in between the *TC(0)* and the *Monte Carlo* methods. The figure 3.1.1.1 below, taken from [6] shows the position of the *n-step TD* method with respect to the other two methods.

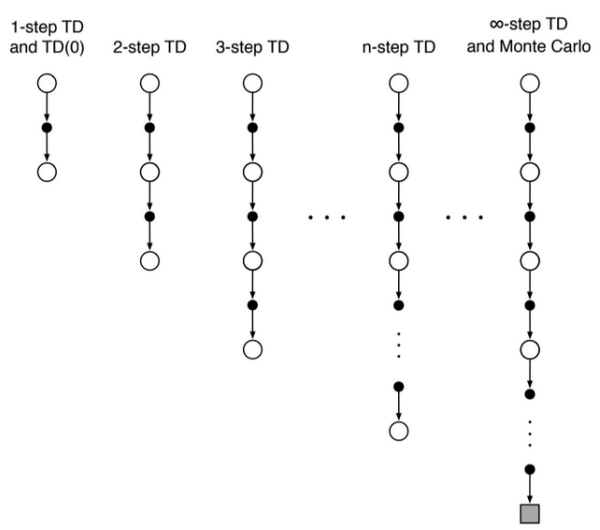

Figure 3.1.1.1: The backup diagram of the n-step Temporal Difference Learning *method ranging from* $n = 0$ *(TD(0)) to* $n = \inf$ *(Monte Carlo method)*

From 3.1.1.1, the white nodes indicate the states while the smaller black nodes denote the actions taken. The *TD(0)* method performs a one-step look-ahead to only the next state

while the *n-step TD* method performs a n-step look-ahead for updating the *Value Function*. Therefore, the update rule for the *n-step TD* method can be expressed as follows:

$$V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)]$$
$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...\gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

where $G_{t:t+n}$ denotes the n-step return as a sum of the discounted rewards and the *Value Function* of the n^{th} state.

Using either the *TD(0)* or the *n-step TD* method, the Advantage Actor-Critic (A2C) agent can be constructed based on one of the methods. As the name suggests, the agent contains two parts: the actor and the critic. The actor simply performs actions in the environment based on what it observes while the critic criticizes on the actor's actions based on the reward it receives from the environment.

The concept of the A2C agent is categorised as an *On-Policy Policy Gradient* method. The term *On-Policy* refers to the concept of improving the policy during the agent's operation. The concept also introduces the idea of using the *weight vector* ($\mathbf{w}$) for the new form of the *Value Function* ($\hat{v}(s,\mathbf{w})$) which can be computed using an Artificial Neural Network (ANN). The term *Policy Gradient* refers to the method which directly learns a *parameterised policy* that has the parameter vector, $\boldsymbol{\theta}$. The probability of selecting action a in time step t is now denoted as $\pi(a|s,\boldsymbol{\theta})$, and can be expressed below.

$$\pi(a|s,\boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$$

The parameter vector, $\boldsymbol{\theta}$, can also be computed from an ANN. To the concept of *On-Policy Policy Gradient* method into perspective of the A2C method, the actor chooses an action by following the policy, π, while the critic helps updating the vectors, $\boldsymbol{\theta}$ and $\mathbf{w}$, by calculating the *TD Error* or the *Advantage value* (δ_t). For the *TD(0)* method, the *Advantage value* can be calculated using the following equation.

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$$

For the *n-step TD* method, the *Advantage value* can be calculated using the following equation.

$$\delta_t = G_{t:t+n} - \hat{v}(S_t,\mathbf{w})$$
$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...\gamma^{n-1} R_{t+n} + \gamma^n \hat{v}_{t+n-1}(S_{t+n})$$

Once the *Advantage value* is calculated, the vectors, $\boldsymbol{\theta}$ and $\mathbf{w}$, can be updated as follows:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha(\delta_t)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \\ &= \boldsymbol{\theta}_t + \alpha(\delta_t)\nabla\ln\pi(A_t|S_t,\boldsymbol{\theta}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha(\delta_t)\nabla\hat{v}(S_t,\mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha(\delta_t)^2\end{aligned}$$

where the symbol, ∇, denotes the gradient of a function. The *Value Function* ($\hat{v}(s,\mathbf{w})$) can be computed using the critic ANN.

For a scenario of continuous action space, the policy, π, can be defined as a Gaussian distribution so that only the statistics of the policy distribution needs to be learned. The term, $\pi(a|s,\boldsymbol{\theta})$, can be expressed as follows:

$$\pi(a|s,\boldsymbol{\theta}) = \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2})$$

where $\mu(s,\boldsymbol{\theta}) \in \mathbb{R}$ and $\sigma(s,\boldsymbol{\theta}) \in \mathbb{R}^+$ are the mean and the standard deviation of the Gaussian distribution. To ensure that the condition for $\sigma(s,\boldsymbol{\theta})$ is met, the value is exponentiated before the distribution is constructed. The two values can be computed using the actor ANN.

3.1.2 Deep Learning

Deep learning (DL) is a method of creating an Artificial Neural Network (ANN) that is capable of finding underlying patterns or structures in a system. Such neural networks can be trained by introducing it to a huge set of data as input. These data are essentially examples of scenarios in a system. For instance, if a system is to distinguish between cats and dogs, the input data set could be pictures of those animals. The learning process can be supervised, semi-supervised or unsupervised. Supervision in this context refers to the procedure of labelling the input data to the neural network as a guide, and vice versa for unsupervised learning. Semi-supervision refers to the technique of labelling only a small quantity of the data set.

In the case of this project, the Deep Learning method applied is unsupervised. As mentioned previously, the A2C agent requires a total of two ANNs: one for the actor and one for the critic. The actor ANN computes the mean and the standard deviation of the policy Gaussian distribution while the critic ANN computes the *Value Function* for a particular input state.

3.2 Network System

This section attempts to describe the nature of the network system. It also introduces area-specific terms which will be used in this article when explaining about the network environment. Additionally, the description of the TCP NewReno algorithm is provided here as it will be used as a benchmark algorithm for the A2C agent.

3.2.1 Network Environment

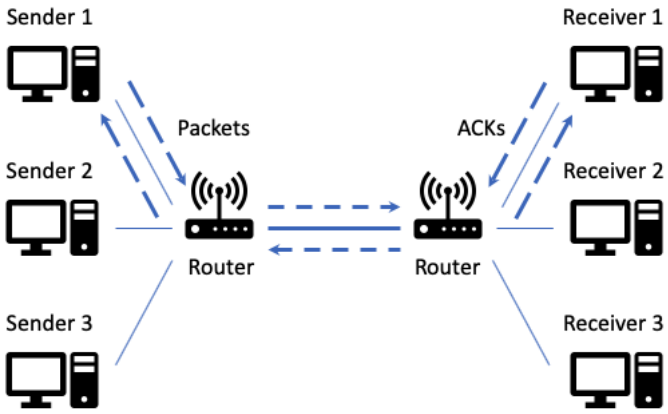

Figure 3.2.1.1: The Network Topology with multiple senders and receivers connected to a single bottleneck link

The figure 3.2.1.1 above shows a common network topology. In the topology, there are multiple connections sharing one data transmission channel or the bottleneck link. A connection refers to a flow of data from one data sender to one data receiver and back. The data sender transmits the data through multiple data packets with smaller size. Once a data packet is received at the receiver's end, the receiver transmitted a feedback signal through the bottleneck link back to the sender. This feedback signal is called packet acknowledgment (ACK) and is sent to inform the sender that the packet has successfully been received. As lost or dropped packets trigger the receiver to send the duplicate ACK signals instead, the data sender can therefore adjust its transmission rate or the congestion window to maximise the number of successfully sent packets over a unit of time (throughput). The sender can also monitor the time of transmitting a certain packet and the time of receiving the ACK signal of that packet. The difference between the two timestamps indicates the delay of packet transmission. This is described as the round-trip time (RTT) as it is the time interval between the start of the transmission and the arrival of the ACK signal.

Although increasing the congestion window to the maximum possible might increase the throughput of the network, sending too many packets at a time would overwhelm the network as more data would be queued or dropped, causing an increase in the network latency or the RTT. The trade-off between high throughput and low latency is described as the *congestion control problem*. An effective TCP algorithm should be able to regulate the traffic by finding an optimal point within the trade-off.

3.2.2 TCP NewReno

The TCP NewReno algorithm adjusts the congestion window based on the Additive Increase Multiplicative Decrease (AIMD). The AIMD rule has three repetitive stages of adjusting the congestion window: the slow start stage, the congestion avoidance stage and the fast recovery stage. The slow start stage slowly increase (additively increase) the congestion window until the slow start threshold (ssThresh) is reached. After that, the congestion avoidance stage increases the congestion window at a slower rate than the slow start stage until some packets are dropped by the router. The information regarding congestion over the network is fed back to the sender by the duplicate ACKs from the dropped packets. To reduce the congestion, the sender performs a multiplicative decrease of the congestion window by halving it in the fast recovery stage. The trend of the congestion window when following the AIMD rule is shown on figure 3.2.2.1 taken from [9].

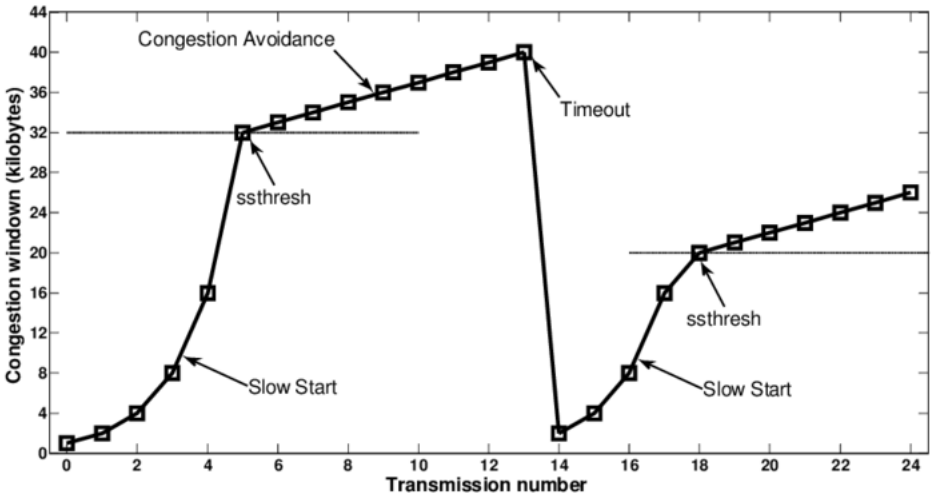

Figure 3.2.2.1: The congestion window trend of the TCP NewReno algorithm

The three stages of the AIMD rule use the following expressions to alter the congestion window (CWND).

- Slow Start: CWND = CWND + 1 for one ACK received
- Congestion Advoidance: CWND + 1/CWND for one ACK received
- Fast Recovery: CWND = CWND/2

4 Technical Method

This chapter will start by outlining the implementation of the Advantage Actor-Critic (A2C) agent and end by describing the development of the simulation of the network environment.

4.1 Advantage Actor-Critic (A2C) Agent

4.1.1 Action Space and Observation Space

At the start of the agent development, the action space and the observation space were defined. Both spaces are continuous spaces. For the action space, the action is the congestion

window (*CWND*) of each data sender. For the observation space, the agent observes a total of five features from the environment: the average bytes in flight (*bytesInFlightAvg*), the segment section acknowledgement sum (*segmentsAckedSum*), the average round-trip time (*avgRTT*), the average inter-transmission time (*avgInterTx*) and the average inter-arrival time (*avgInterRx*). These observation features of the agent were selected based on their correlations with the output parameter (*CWND*). These correlations are displayed on a heat map below. To reduce the input noise or the Curse of Dimensionality, if two input parameters are highly correlated to each other, only one is selected.

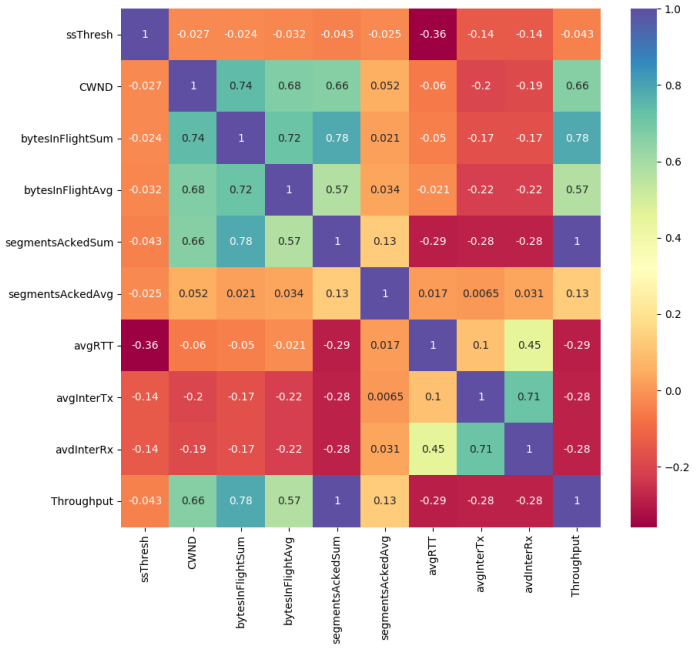

Figure 4.1.1.1: Heatmap of the Agent's Observation Features

4.1.2 Utility Function and Reward Function

To define the agent's goal, a Utility Function (U_t) is created based on the network throughput (*Throughput*) and the average round trip time (*avgRTT*). The Utility Function applies the logarithm function to ensure that the agent can converge to achieve fairness in bandwidth allocation among multiple senders. The Utility Function can be expressed below.

$$U_t = \alpha \log(throughput) - \beta \log(avgRTT)$$

where α and β are tunable hyper-parameters. Setting one parameter higher than another means that the agent would aim to maximise or minimise the parameter's product pair.

To avoid misleading the agent, instead of using the Utility Function as the reward, a Reward Function (R_t) is defined based on the change in Utility Function value between two consecutive time steps. The Reward Function is defined by the following expression.

$$R_t = \begin{cases} a & if \quad U_t - U_{t-1} > \epsilon \\ b & if \quad U_t - U_{t-1} < \epsilon \\ c & otherwise \end{cases}$$

where a, b, c and ϵ are also tunable hyper-parameters. The parameter a is always positive as it would encourage the agent to increase the Utility Function, meaning to increase the throughput and decrease the network latency (*avgRTT*).

4.1.3 A2C Agent Code Implementation

The agent was implemented using two Python classes: *GeneralNetwork* and *Agent*. The *GeneralNetwork* class builds two the neural networks for the actor and the critic. The *Agent* class then uses the neural network from *GeneralNetwork* to create the actor and the critic for action-choosing and learning, respectively. The implementation of the agent is dependent of the package *PyTorch* (*torch*).

4.1.3.1 General Artificial Neural Neural Class (*GeneralNetwork*)

The GenericNetwork class outlines the neural network structure. It is a child class of *torch.nn.Module* Each part of the agent, the actor and the critic, contains a three-layer neural network with the feed-forward structure. With the feed-forward structure, the input would enter at the first layer. The output from the first layer is then passed to the next layer and so on.

The *GeneralNetwork* class initialises by taking in five input parameters: the learning rate, the neural network input dimension, dimension of the first layer, dimension of the second layer and the neural network output dimension. Then, the neural network layers are constructed by instantiating the *torch.nn.Linear* class. The *Adam* optimiser from *torch.optim.Adam* is used to optimise the neural network parameters, with the same learning rate as the network.

To compute the output of the neural network, a method *forward* is implemented. This method takes the observation features as the input and outputs depending on the whether the network is for the actor or the critic.

4.1.3.2 A2C Agent Class (*Agent*)

The *Agent* class initialises by taking a total of eight parameters: the actor learning rate, the critic learning rate, the number of observation features (the neural network input dimension), the discount factor (γ), the actor neural network output dimension, dimension of the first layer, dimension of the second layer and the number of action feature(s) (i.e. 1). Using these parameters, the class instantiates the *GeneralNetwork* class to construct the two neural network structures.

The actor neural network has two output parameters. That is, the mean (μ) and the standard deviation (σ) of the policy (action distribution). However, the critic neural network has only one output parameter which the *value function* of a particular state.

Using the actor neural network, the method *choose_action* takes in the current values of the observation features and passes them to the neural network. The network then yield the mean and the standard deviation which are used to construct a Gaussian distribution by instantiating the *torch.distributions.Normal*. The value of the standard deviation produced by the neural network is bounded to a range between -10 and 10 to ensure that the value is neither too small nor too large. The value is also exponentiated before passing as a parameter to construct the distribution to ensure that it is always positive.

Subsequently, by using the method *rsample* of the Gaussian distribution class object, the value of the action can be sampled from the distribution. As mentioned in the previous Chapter, the actions that are favourable have high probability of being sampled. After that, the logarithm of the probability of the chosen action (*log_probs*) is calculated for the critic to update the policy. The method *choose_action* ends by returning the value of the action for the environment to execute.

Finally, another method in the *Agent* class is the *learn* method. The method takes in four parameters as input: current state observation, reward, next state observation and the done flag. Depending on the learning method applied, the reward parameter could just be the immediate reward if the *Temporal-Difference Learning (TD(0))* method, or it could be the n-step return if the *n-step Temporal-Difference Learning* method is used. The done flag essentially informs the agent whether the episode has terminated or not. Afterwards, the method sets the gradient of each optimiser for the actor and the critic to zero. Using the method *forward* of the critic neural network, the *learn* method computes the *value functions* for the current state and the next state. Then, the *TD error* or the *Advantage value* is calculated based on the reward (or the return) and the *state-value functions*. At the end, the actor loss is calculated as the product of the negative *log_probs* and the *Advantage value* while the critic loss is calculated as the square of the *Advantage value*.

4.2 ns-3 Simulation of the Network Environment

For the development of the network simulation, a network topology was created by creating simulation scripts in C++ that uses tools provided by the network simulator package *ns-3*. The network topology contains multiple senders and receivers connected to a single bottleneck link. Using the framework *ns3-gym* [10], simulation scripts are represented in the Python *OpenAI Gym* format. The agent then interacts with the environment using the *Gym* Application Programming Interface (API). The figure 3.2.1.1 from the preceding chapter shows the network topology created in the environment.

There are five C++ scripts created for the network simulation. The main script, *sim.cc*, specifies all the necessary parameters for the construction of the network topology, such as the number of senders and receivers, the bandwidth of the bottleneck link and the access delay between a data sender and the router. After the parameters specification section, the script uses the function *AddValue* from the *CommandLine* class to parse the specified parameters as command-line arguments. This step removes the difficulty of having to manually insert those parameters every time when running the simulation.

Before the components of the network topology are created, *sim.cc* connects the script to the *ns3-gym* framework via the *OpenGymInterface* class. The *OpenGymInterface* class defines how the agent, implemented in Python, can interact with the environment. It is based on the structure outlined in the other four C++ scripts. The description of the four scripts will be explained later on.

After that, the script creates the bottleneck link (*bottleNeckLink*) by instantiating the *PointToPointHelper* class. The attributes of the bottleneck link, the bandwidth and the pre-set channel delay, have also been set here. Then, by instantiating the *PointToPointHelper* class again, the script creates the link (*pointToPointLeaf*) between a sender or receiver to a router. At the end of this code section, the helper class *PointToPointHelper* is instantiated to duplicate *pointToPointLeaf* equal to the total number of senders and receivers.

The final section of the *sim.cc* script creates the data senders and receivers. The senders are created using the *BulkSendHelper* class. This helper class helps install a *BulkSendApplication* which tries to create as much traffic as possible over the network by sending data through the network as fast as possible up to the specified limit. Each data receiver or the packets sink is created by using the *PacketSinkHelper*.

For the other four scripts, the header file *tcp-rl.h* and source file *tcp-rl.cc* implements the abstract base class for the TCP agent. They abstractly define necessary functions of the TCP agent: *IncreaseWindow*, *PktsAcked*, *CongestionStateSet* and *CwndEvent*. The function *IncreaseWindow* takes in the new value of the congestion window and applies it to the senders. The function *PktsAcked* calculates the total round-trip time. The functions *CongestionStateSet* and *CwndEvent* provides additional information of the congestion state and the data transmission event that has occurred as a result of the change in the congestion window.

The header file *tcp-rl-env.h* and source file *tcp-rl-env.cc* then implement the mentioned functions for the time based TCP agent. The source file *tcp-rl-env.cc* also defines the observation and action spaces of the agent. The two spaces are defined as continuous using the object *box*.

4.3 The Training Script

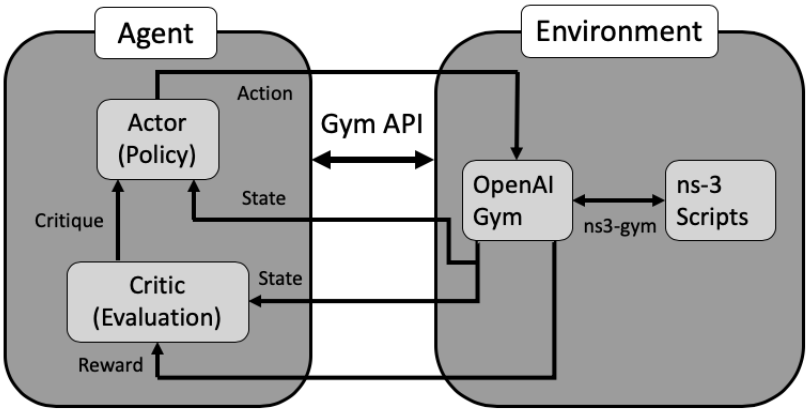

Figure 4.3.1: The Agent-Environment Interaction Structure

Figure 4.3.1 shows the structure implemented in the training script to combine the agent and the environment together. The script consists of a nested loop. The outer loop is a *for* loop that determines the number of episodes to train the agent. At each iteration of the outer loop, the environment is reset to its initial state. The inner loop is a *while* loop that loops forever until the episode is terminated (done flag set to True). Within the inner loop, the training script feeds in the current state observation to the *choose_action* method of the agent. The method then computes the action to be taken by the environment using the method *step* of the *Gym* API. The method *step* returns four output parameters: the next state observation, the immediate reward, the done flag and extra information about the environment (provided by the function *CongestionStateSet* and *CwndEvent* in the simulation scripts). Afterwards, if the *TD(0)* method is used, the training script feeds in the current state observation, the reward, the next state observation and the done flag to the *learn* method of the agent at every iteration of the inner loop. If the *n-step TD* method is applied, the training script calls the *learn* method n steps after each state and feeds in the n-step return instead of the immediate reward.

5 Results and Analysis

This section presents the performance of the two versions of the A2C agent compared to the performance of TCP NewReno operating it in the same network environment. The first version of the agent adopts the *Temporal-Difference Learning* method (*TD(0)*) while the second version adopts the *n-step Temporal-Difference Learning* method with the parameter n being set to 10. For the agent configuration, the learning rates of the actor and critic are *5.19e-5* and *2.42e-5* respectively. The discount rate (γ) is set to *0.99* so that the agent becomes farsighted. The parameters, α and β, of the Utility Function are set to 1 and 0.01 respectively. For the Reward Function, the parameters, a, b and c have been set to 100, -10 and 0, respectively.

For the network configuration, the bandwidth of the bottleneck link has been set to 2 Mbps. The network topology in this scenario contains two senders and receivers.

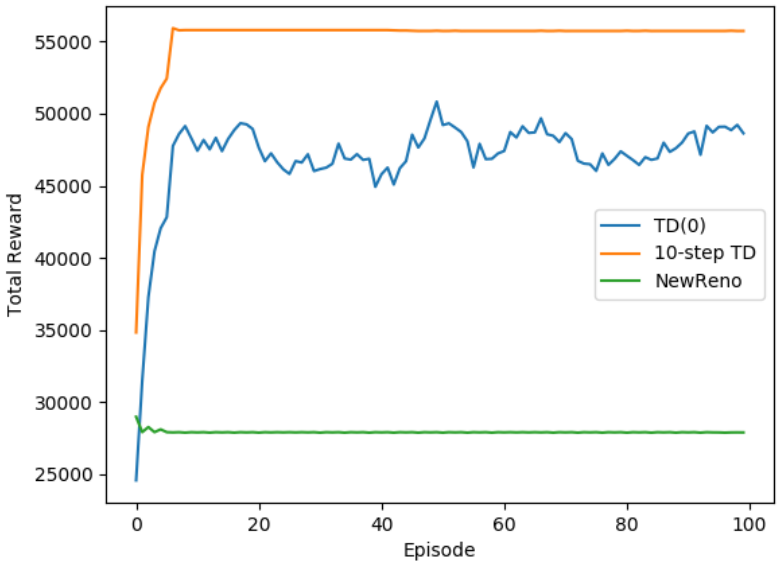

Figure 5.1: The total reward received by each agent at the end of each episode

As seen from figure 5.1, The two agents was able to perform better to receive more rewards as compared to the TCP NewReno algorithm. The agent with the *TD(0)* method acquired was more unstable than the other agent with the *10-step TD* method acquired. The instability of the *TD(0)* agent can be clearly observed on the learning curve as it could not converge to a single value.

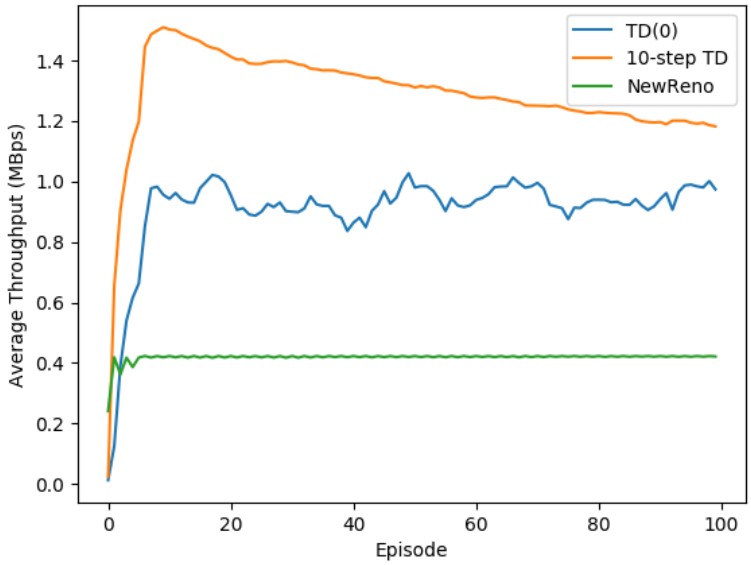

Figure 5.2: The average throughput achieved by each agent at the end of each episode

A similar trend to that of figure 5.1 can be observed in figure 5.2. The versions of the agent was able to achieve higher network throughput than the TCP NewReno algorithm. The agent using the *10-step TD* method was able to achieve higher throughput than the agent using the *TD(0)* method, although the achieved throughput dropped slightly in episodes towards the end of training.

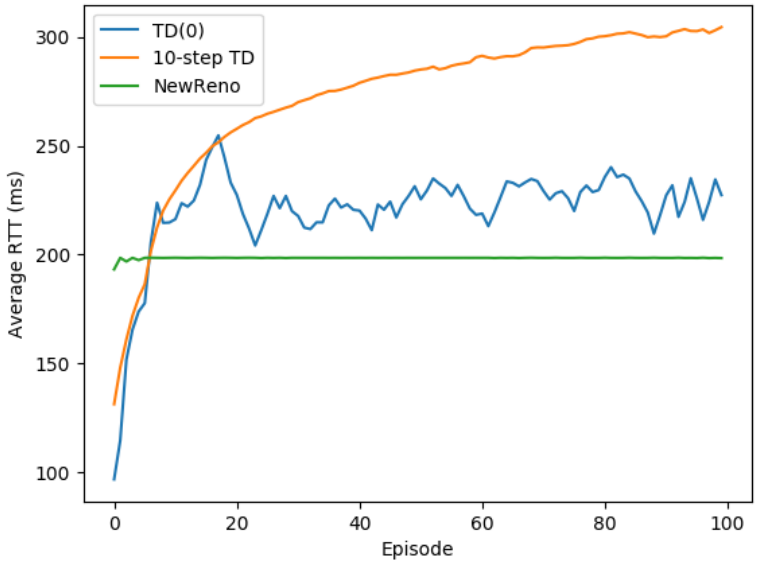

Figure 5.3: The average round-trip time (RTT) achieved by each agent at the end of each episode

Finally, figure 5.3 shows the average round-trip time (RTT) achieved by each agent. As expected, as the two agents were able to achieve higher network throughput than the TCP

NewReno agent, the average round-trip time (RTT) of both A2C agents were higher than that of TCP NewReno. The RTT of the agent using the *10-step TD* method was also higher than that of the agent using the *TD(0)* method.

6 Conclusion

In summary, this project offers a novel solution to the problem of internet congestion control. By conducting a rigorous study in the research fields related to this project, the concept of Deep Reinforcement Learning was chosen to create an agent with the Advantage Actor-Critic design capable of achieving its objective of maximising the throughput while trying to minimise the network latency. When compared to a rule-based TCP algorithm like TCP NewReno, the agent was able to outperform the TCP NewReno.

The performance of the agent can further be enhanced using the method of Asynchronous Advantage Actor-Critic (A3C). This method, by adding another 'A' to A2C method, introduces the concept of parallel processing of multiple identical agents operating in slightly different network environment setting. Each agent then contributes by updating a single shared network between agents. This solution could potentially be the most suitable solution of the internet congestion control problem.

A Code Appendix

The code developed in this project can be found in this URL:
https://github.com/AtomAnu/ML_A3C_internet_congestion_control

References

[1] K. Winstein and H. Balakrishnan, *TCP ex Machina: Computer-Generated Congestion Control*. MIT Computer Science and Artificial Intelligence Laboratory, 2013. [Online]. Available: https://web.mit.edu/remy/

[2] W. Li, F. Zhou, K. Chowdhury, and W. Meleis, *QTCP: Adaptive Congestion Control with Reinforcement Learning*. Department of Electrical and Computer Engineering, Northeartern University, Boston, 2018. [Online]. Available: https://ece.northeastern.edu/personal/meleis/tnse-18.pdf

[3] N. Jay, N. H. Rotman, P. B. Godfrey, M. Schapira, and A. Tamar, *A Deep Reinforcement Learning Perspective on Internet Congestion Control*. Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, 2019. [Online]. Available: https://arxiv.org/pdf/1810.03259.pdf

[4] G. van Rossum, B. Warsaw, and N. Coghlan, *PEP 8 – Style Guide for Python Code*. Python Software Foundation, 2013. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[5] T. C. Consortium, *C++ Coding Standards*, 2000. [Online]. Available: http://www.literateprogramming.com/cppstnd.pdf

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, second edition ed. Cambridge, Massachusetts: The MIT Press, 2014.

[7] L. Fridman, *MIT 6.S091: Introduction to Deep Reinforcement Learning (Deep RL)*, 2019. [Online]. Available: https://www.youtube.com/watch?v=zR11FLZ-O9M

[8] M. Lapan, *Deep Reinforcement Learning Hands-On*. Birmingham, United Kingdom: Packt, 2018.

[9] X. Zhong, Y. Qin, and L. Li, "Transport protocols in cognitive radio networks: A survey," *KSII Transactions on Internet and Information Systems*, vol. 8, 11 2014.

[10] P. Gawłowicz and A. Zubow, *ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research*, November 2019. [Online]. Available: http://www.tkn.tu-berlin.de/fileadmin/fg112/Papers/2019/gawlowicz19_mswim.pdf