Improving TCP CUBIC Congestion Control with Machine Learning (2023)

James Knott

***Abstract*— Transmission Control Protocol (TCP) is commonly used for reliable internet data transfers. However, TCP can experience packet loss due to network congestion. Packet loss happens when data doesn't reach its destination for various reasons. In recent years, there has been a growing inclination towards adopting novel, clean-slate learning-based designs as alternatives to traditional congestion control mechanisms for the Internet. However, we posit that integrating machine learning techniques with the current congestion control schemes can achieve comparable, if not superior outcomes. This project endeavoured to address this gap and implement a system that can utilised with TCP CUBIC. Our method looked to enhance the efficiency of the TCP CUBIC congestion control by incorporating machine learning techniques. TCP CUBIC, the default congestion control variant in the current Linux Kernel, modifies the congestion window size based on a loss-based algorithm, thereby influencing the rate of data transmission. TCP CUBIC uses a parameter, beta, to modify the rate at which the congestion window grows. Our approach involves employing a model-free reinforcement learning algorithm, specifically a Q-learning algorithm to optimize the TCP CUBIC beta parameter, targeting an increase in throughput for TCP CUBIC connections. Through extensive testing performed in various simulated network conditions we demonstrate the performance and adaptability of the Q-Learning algorithm. Furthermore, this report details the various development decisions undertaken and their driving influences. It also provides an insight into the project's results, expanding on the existing system design, and elaborates on the potential for future work in this area.**

***Index Terms*—TCP, CUBIC, Q-Learning, Reinforcement Learning, Machine Learning**

I. INTRODUCTION

Transmission Control Protocol (TCP) is a protocol that is frequently used by internet users for reliable data transfers. However, TCP can suffer from packet loss through network congestion. Packet loss is the loss of data during network transmission. It occurs when one or more packets fail to reach their destination, which can happen due to a variety of reasons. When packets are lost the receiving node may not receive all the data it requires to properly reconstruct the data. The implications of packet loss are considerable, leading to network performance degradation in terms of packet delays, a decrease in throughput, and reduced application performance. As a response to these challenges, various TCP alternatives equipped with congestion control algorithms have been developed to curb such adverse effects. TCP CUBIC (CUBIC) is a notable example of these alternatives and has been universally adopted across standard operating systems, including Windows, Linux, and Mac [1]. CUBIC controls congestion window growth using a cubic function [1].

This project looked to reduce packet loss and increase throughput of TCP connections using machine learning. The proposed solution looked to develop a machine learning algorithm that modifies CUBIC parameters to improve throughput and packet loss by 15%. We did not look to redesign CUBIC but to create an algorithm that can work alongside its current implementation. Furthermore, in the current literature, there is a noticeable gap in machine learning algorithms used with the already existing CUBIC. Most studies look to create their own protocol that works in conjunction with machine learning [3][4][5][6][7][8].

This project looked to address this gap and implement a system that can utilised with CUBIC. Using a Q-learning algorithm to optimise CUBIC's congestion control parameters. Specifically, CUBIC's β (beta) parameter. CUBIC follows a cubic algorithm and has parameters that affect the growth of the congestion window. After every loss event, the new max congestion window is calculated by performing a multiplicative decrease of the congestion window by a factor of beta where beta is a window decrease constant (beta) [1]. During evaluation, results show that during transmissions experiencing packet loss our Q-learning algorithm can increase throughput by up to 13% and reduce packet loss by up to 8%. This signifies not only the efficacy of our approach but also its potential to enhance the overall performance of TCP connections, especially where packet loss is prevalent.

II. RELATED WORK

Machine learning in conjunction with congestion control is currently a topic being researched however, it has not been implemented in current systems where TCP CUBIC is the favoured variant of TCP. There are three main classes of learning algorithms supervised learning, unsupervised learning, and reinforcement learning (RL). In the current literature, reinforcement learning, and the deep variant of reinforcement learning (DRL) are the favoured choices [2][3][4][5].

Abbasloo et al, *Jay et al and Xu et al* all opt for Deep reinforcement learning [2][3][4]. Orca uses the underlying TCP to handle the connections [3]. They tested Orca over varying connections within USA and intercontinental connections. Outperforming Aurora within the USA and Aurora providing better throughput intercontinental experiments [2][3].

This project was supervised by Winston Seah (primary), Alvin Valera.

Aurora and *Xu et al* used purely machine learning based protocols opting to design their own [3][4]. While *Kong et al* used basic Reinforcement learning [5]. *Dong et al* use what they call Performance-oriented Congestion Control (PCC) [6].

While these papers are all successful at reinventing congestion control and in all cases outperform standard implementations of TCP such as CUBIC and NewReno.

Afonin et al describe a Q-Learning solution in conjunction with TCP [7] and is the most alike to this project. The study provides a rather surface-level exploration and does not delve deep into the intricacies of how their Q-Learning algorithm interacts with the congestion window. While it offers an overview, it lacks a detailed examination of the specific mechanisms and dynamics at play between the Q-Learning process and the congestion window adjustments. They note a reduction in packet delay of 15%.

There is a clear theme, among the current literature as each study looks to either rewrite existing or write their own congestion control protocol. even if an algorithm is technically sound, getting it adopted widely in the industry is a significant hurdle. Established protocols have the advantage of being tried and tested over time, and convincing network operators, device manufacturers, and software developers to switch to a new algorithm can be a daunting task. This project looks to address that gap and demonstrate a Q-Learning algorithm that works in conjunction with TCP CUBIC in the state it is currently.

III. DESIGN

In this project, we are not embarking on a full-scale redesign of TCP CUBIC. Instead, we are focused on exploring the potential enhancements that can be applied to the existing TCP CUBIC framework. Our objective is to fine-tune TCP CUBIC's congestion control parameters without necessitating a complete rewrite of the protocol or its congestion control algorithm. The primary aim of this model is to optimize TCP CUBIC's performance by improving latency and reducing packet loss.

A. Network Topology

In evaluating the impact of tuning TCP CUBIC's beta parameter, it is essential to isolate as many external variables as possible to get clear, unequivocal results. Utilising a simplified network topology comprising solely of a sending node and a receiver node offers an optimal setup for this kind of experimentation. This simple model ensures that external factors, often introduced with the inclusion of multiple nodes, don't skew or muddy the results. When dealing with more complex multi-node environments, the network inherently becomes susceptible to a myriad of additional variables, such as node-to-node interference, varying path delays, and potential bottlenecks in intermediate nodes. In contrast, a direct sender-receiver topology facilitates a controlled environment, allowing for precise evaluation and tuning of the beta parameter without extraneous influences. This approach guarantees that any observed effects or changes in performance are almost exclusively attributable to modifications made to the TCP CUBIC's beta parameter.

In current implementations of TCP CUBIC in Linux, Windows or Mac, these parameters are unable to be modified. Direct manipulation of TCP CUBIC's beta parameter is inherently constrained due to the hard-coded nature of certain kernel parameters. The kernel's implementations have been optimised for general use-cases, and allowing users to easily modify such parameters could inadvertently compromise the stability and predictability of network performance, as well as introduce potential security vulnerabilities. Thus, ensuring the robustness and security of the system often takes precedence over providing granular customization options for users.

Furthermore, once the transmission is underway, these parameters are locked in, ensuring consistency in the transmission behaviour and avoiding the complexities that could arise from altering parameters during an active session. This means CUBIC's parameters will need to be modified prior to every transmission and will stay consistent throughout the transmission. After each transmission is complete the effects of the parameter modifications can be observed.

Given the constraints in directly tweaking the TCP CUBIC's beta parameter within the operating systems, an alternative approach is to employ network simulation. Utilising a network simulator allows for the creation of a controlled virtual environment where parameters, like the beta value, can be modified freely without the risk of destabilising an actual network. This not only ensures a safe playground to test and observe the effects of such modifications, but it also provides the flexibility to replicate a variety of network conditions and scenarios, thereby offering comprehensive insights into the potential real-world implications of tuning the parameter.

B. TCP CUBIC

TCP CUBIC employs a unique congestion control strategy. After experiencing a loss event, which is an unacknowledged packet, it marks the window size at which the loss occurred as W_{max} [1]. It then decreases the congestion window by a constant factor β (beta), which is the window decrease constant, while continuing with the regular fast recovery and retransmission mechanisms of TCP.

As CUBIC transitions from fast recovery to congestion avoidance, it initiates an increase in the window size using a concave profile of the cubic function. This function is designed to have a plateau at W_{max}, which means that the concave growth persists until the window size matches W_{max}. In a graphical representation, a concave function appears as a curve where any line segment drawn between two points on the function lies below or on the function itself. This type of growth is often seen in scenarios where initial gains are substantial, but as time or the quantity of the variable increases, the rate of growth starts to slow down, leading to smaller incremental gains. Beyond that point, the cubic function shifts into a convex profile, initiating convex window growth. This growth pattern

is observed in situations where initial advancements might be modest or gradual, but as time progresses or the quantity of the variable increases, the growth rate escalates, resulting in larger incremental gains.

This approach to window adjustment, moving from concave to convex growth, is intended to enhance protocol and network stability while maintaining high network utilization. The plateau around W_{max} is where network utilization is considered to be at its peak as this is where the last loss occurred. Under steady-state conditions, most window size samples in CUBIC are close to W_{max}, promoting both high network utilization and protocol stability.

It's worth noting that protocols with convex growth functions tend to have their most significant window increment around the saturation point, which can introduce a substantial burst of packet losses.

The window growth function of CUBIC follows this formula:

$$W(t) = C(t - K)^3 + W_{max}$$

Where *C* is a CUBIC parameter, *t* is the time elapsed since the last window reduction, and *K* is the time it takes for the function to increase W to W_{max} in the absence of further loss events.

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}}$$

The parameter β (beta) within the TCP CUBIC algorithm plays a pivotal role in influencing the protocol's reaction to network congestion. It serves as a crucial factor in determining the rate at which the congestion window is adjusted following congestion events, such as packet loss. Specifically, β serves as the constant that controls the multiplicative decrease in the congestion window, effecting a reduction in its size. The choice of β is central to the algorithm's responsiveness, influencing its behaviour during congestion episodes. Smaller values of β result in more aggressive window reductions, facilitating a swift response to network congestion. Conversely, larger β values lead to more gradual reductions, allowing for a smoother and less abrupt response to network congestion.

Due to *β's* significant influence on TCP CUBIC's behaviour during loss events, it emerges as a pivotal parameter to fine-tune with the purpose of mitigating packet loss. The β parameter directly dictates the rate at which the congestion window decreases when congestion is detected, such as during packet loss scenarios. By modifying β, it becomes possible to strike a balance between a more aggressive or more gradual reaction to congestion, thus optimizing the protocol's response. A well-tuned β value can facilitate a more measured, controlled reduction of the congestion window during congestion events, consequently reducing the likelihood of packet loss. In this context, the fine-tuning of β represents a targeted and effective strategy to enhance network performance and bolster the reliability of data transmission in TCP CUBIC.

C. Reinforcement Learning

Sutton et al state, "Reinforcement learning problems involve learning what to do—how to map situations to actions—so as to maximize a numerical reward signal" unlike many forms of machine learning where the learner is instructed on which actions to take, here the learner must determine the most rewarding actions through experimentation and discovery [8]. Reinforcement learning entails a reciprocal process between an agent actively making decisions and its environment, where the agent is striving to reach a target despite the uncertainty of its environment. *Abbasloo et al* motivations for choosing reinforcement learning were based on its sequential and far-sighted nature which they believe complements the sequential decision-making process of congestion control [3].

Thus, reinforcement learning is a great fit for the project. Reinforcement learning looks to maximise the reward signal, in this project, the reward should be a function of the change in throughput and packet loss. Actions can be performed on the CUBIC parameter beta to affect the congestion window. These actions would be to increase, decrease or keep the same beta. The situations that *Sutton* refers to are also characterized as states of the environment that can be mapped to the current value of beta, the packet loss and the throughput of each transmission. Where the agent is the sending node which takes actions by modifying the beta value.

D. Q-Learning

One of the major downsides of using a model-based approach is training the model. Training requires a large set of quality data as it looks to learn a model of the environment. Training a model to accurately represent the environment can be a challenging task, especially when the environment is complex and dynamic. It can require a large amount of high-quality data, which may be difficult or time-consuming to collect. On the other hand, with a model-free approach, this takes this out of the equation. Instead, Q-learning and Deep-Q Networks do not require a model as such and do not attempt to learn or use a model of the environment. Instead, they learn a value function directly from interaction with the environment. The benefit of model-free methods is that they can be simpler to implement and may require less data, as they do not need to learn a model of the environment. However, they may be less sample-efficient than model-based methods, which can use a learned model to simulate many different scenarios without requiring real interactions. This can simplify the learning process and potentially reduce the amount of data required. Model-free approaches may require more interactions with the environment to learn an optimal policy, as they cannot use a model to simulate future states. Q-learning and DQN utilise Bellman's equation to maximise future rewards [4][5]. For a given state *s* and action *a*, the Q function *Q(s,a)* can be defined in terms of the expected reward for taking action *a* in state *s*,

plus the expected value of the subsequent state *s*. Mathematically, Bellman's equation for the value function is given by:

$$Q(s,a) \leftarrow (1-a)Q(s,a) + \alpha(r + \gamma\, max_{a'}\, Q(s',a')$$

$Q(s,a)$ is the current estimated value of taking action *a* in state *s*.

r is the immediate reward received after taking action *a* in state *s* and transitioning to state s'.

γ is the discount factor, which models the agent's consideration for future rewards. A value of 0 makes the agent myopic (only caring about immediate rewards), while a value of 1 makes it fully consider all future rewards.

$max_{a'}\, Q(s',a'$ is the maximum estimated future reward when starting from state s' and considering all possible actions a'.

α is the learning rate. This parameter dictates how much of the new Q-value estimate we adopt to update our current Q-value.

A Q-table is a fundamental component in Q-Learning which stores these calculated Q values. It's a lookup table where we store the information about what action the agent should take under different circumstances. In other words, it guides the decision-making process of an agent. Specifically, the Q-table stores Q-values for every possible combination of states and actions in the environment. Each Q-value represents the expected future reward that an agent expects to receive if it takes a certain action in a certain state, considering both the immediate reward and the possible future rewards. The cell at the intersection of a specific state row and action column contains the Q-value associated with taking that action in that state.

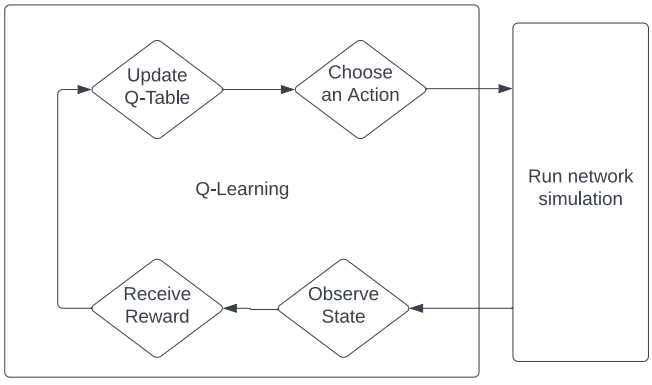

Fig. 1. Q-Learning System Architecture.

E. System Architecture

Tying all these components together provides a system architecture described in Fig. 1 Where the Q-Learning algorithm triggers a network simulation transmission for each step of the algorithm. The beta value, packet loss and throughput are observed after the simulation and become the new state. A new reward is calculated as a function of the packet loss and throughput. The reward is then updated in the Q-table. Then a new action is chosen based on the policy which modifies beta.

F. Sustainability

In an exploration of the societal benefits that may arise from the proposed improvements in latency and packet loss, several key sectors stand out. User Experience: Enhancements in these areas can fundamentally transform the end-user experience. Specifically, the acceleration in webpage loading times, the fluidity of video streams, and a comprehensive enhancement in internet service quality can be achieved. Remote Work and Distance Learning: With the rise of remote work and virtual education, these technical improvements could greatly enhance these areas, leading to increased remote work efficiency and smoother online interactions between educators and students.

This improved efficiency not only aligns with several United Nations Sustainable Development Goals (SDGs) but also aids in their achievement [9]. Firstly, it can contribute to SDG 7, as more efficient data transmission can lead to energy savings in data centres and promote wider access to digital energy services. Secondly, it supports SDG 8 by enhancing the productivity of businesses relying on digital services and fostering economic growth. Lastly, in line with SDG 9, the use of machine learning to enhance internet infrastructure exemplifies technological innovation, which is key in building resilient infrastructure and promoting sustainable industrialization. However, the actual impacts will depend on factors such as the extent of these improvements' implementation and the specific context of network infrastructure across regions.

IV. IMPLEMENTATION

A. Network Simulation

In this project, we utilised the ns-3 simulation framework to examine the performance dynamics of the TCP Cubic congestion control algorithm under varying network conditions. The code for our ns-3 script is located in the file tcp_cubic.cc.

Two nodes were established in a point-to-point setup with configurable parameters. The receiver operates with a PacketSinkHelper which simply receives packets and sends back acknowledgements. While the sender uses a custom application, designed to offer flexibility in data transmission settings. In particular, the sending rate of packets (Mbps) and the CUBIC beta parameter.

In our network simulation, we configured the environment to send a total of 1000 packets, each with a size of 1,500 bytes. This size was deliberately chosen as it corresponds to the Maximum Transmission Unit (MTU) typically observed in many Ethernet-based networks. Such a choice ensures that the packets aren't fragmented during transmission, offering a more streamlined and consistent evaluation. By focusing on this standardized packet size, our simulation endeavours to replicate

real-world scenarios, thereby providing a more accurate assessment of TCP Cubic's behaviour under common network conditions.

The focal point of the study is the modulation of TCP CUBIC's Beta parameter, whose effects on congestion window adjustments and packet acknowledgements are logged. Comprehensive statistics, encompassing throughput, packet loss rate, and overall bytes received, are captured, and archived for subsequent analysis by the Q-Learning model.

B. Software and Libraries

The implementation of the Q-Learning Model is located in the Jupyter Notebook model.ipynb is located in our GitLab. A Jupyter notebook. We opted to implement our Q-learning algorithm within a Jupyter Notebook rather than a traditional Python script for several compelling reasons. Firstly, Jupyter Notebooks excel in their ability to maintain variable states between runs. This means that if we need to rerun specific portions of our algorithm, we can do so without having to execute the entire script from the beginning, leading to quicker iterations and improved efficiency in our development process. Furthermore, Jupyter Notebook's rich display system is unparalleled. It offers us the ability to visualize data, display tables, and create charts in an interactive environment, enhancing our understanding of the algorithm's performance and behaviour. The combination of these features, along with the seamless interplay between code, outputs, and descriptive narratives, made Jupyter Notebook an ideal choice for our project, facilitating both development and subsequent analysis.

Using OpenAI's Python library, gym, you can create reinforcement learning environments [10]. This allows you to establish an environment that provides a standard API for interacting with the environment. The API has methods for resetting the environment to its initial state (reset()), taking an action in the environment (step()), and other helpful utilities. As such, gym has been utilised to create the environment for this project defined in the TCPEnv class.

C. Actions

As mentioned in section Section II.A, a Q-table requires states and actions. As we are looking to find the optimal *beta* value to minimise the latency of a TCP CUBIC connection, beta is the value that actions will be performed on. In this case, *beta* may either be increased, decreased, or kept the same. This gives us three actions that can be performed. Thus,

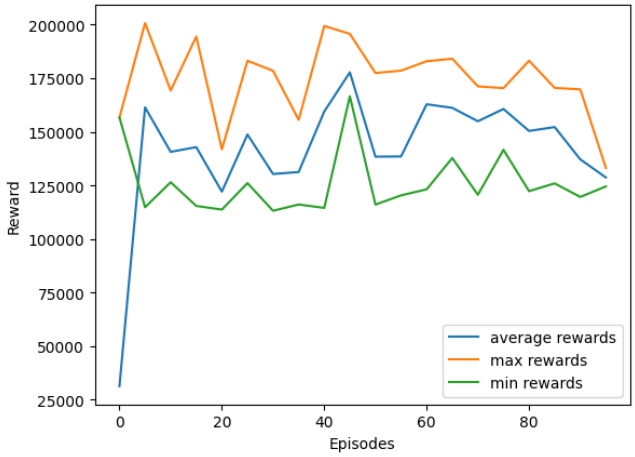

Fig. 2. Rewards using epsilon decay starting from episode 0.

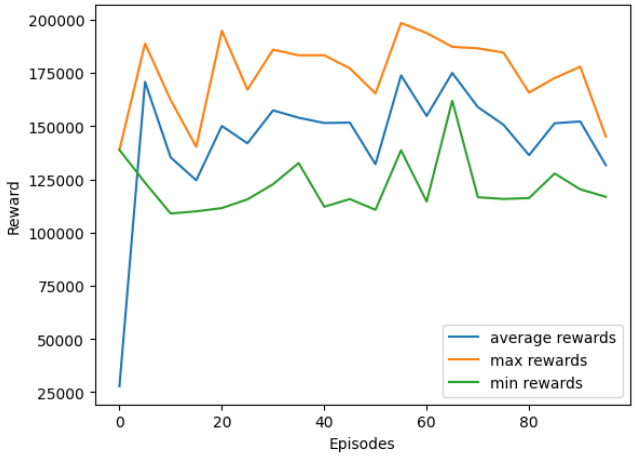

Fig. 3. Rewards using epsilon decay starting from episode 20.

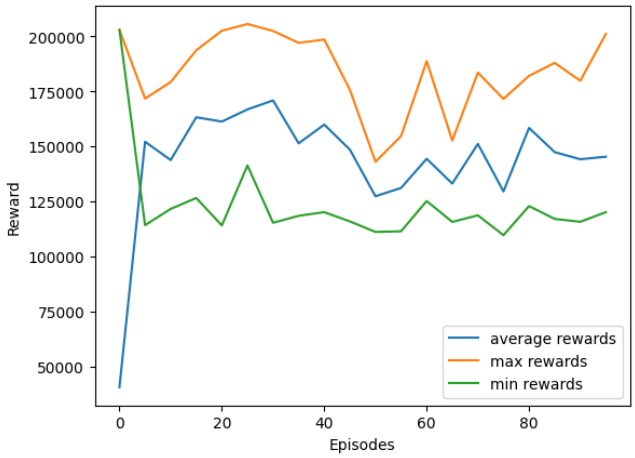

Fig. 4. Rewards using epsilon decay starting from episode 50.

every time the step function of our environment is called, one of these three actions will be performed. This will update beta with a new value or keep it the same.

The action is decided using an epsilon-greedy policy. In our Q-learning implementation, we employed the epsilon-greedy

policy to balance exploration and exploitation during the agent's decision-making process. Initially, the agent is fully in exploration mode, with a 100% chance of taking a random action, as *epsilon* is set to 1. This approach ensures a broad sampling of the action space early on. However, to gradually transition the agent towards exploiting its acquired knowledge, we introduced a decay mechanism for ε. Specifically, the decay starts after 100 episodes and continues until the 500th episode. During this period, *epsilon* diminishes incrementally, which is calculated by the formula:

$$\frac{epislon}{END_{EPSILON_{DECAYING}} - START_{EPSILON_{DECAYING}}}$$

Consequently, as episodes advance, the likelihood of the agent relying on its Q-values, rather than random exploration, increases. By the end of the 500th episode, the agent predominantly leans on its learned experiences. This epsilon-greedy strategy ensures that while the agent gains varied experiences initially, over time, it makes decisions more aligned with its accumulated knowledge.

Different epsilon decay policies were tested initially shown in Fig. 2., Fig. 3. and Fig. 4. The three reward graphs depict distinct learning trajectories of the agent. Fig. 2., demonstrates rapid initial gains, characterized by an early spike in rewards followed by stabilization. In contrast, Fig. 3., starts steadily but reveals a widening range in outcomes as the episodes progress, ending on an optimistic note with an upward trend. Fig. 4., while consistent in its reward differences, hints at a slight decline in the latter episodes, indicating potential challenges. Collectively, the graphs highlight varying rates of learning, exploration depth, and consistency in the agent's journey. However, Fig. 3. towards the end, the average reward trend is upward sloping, hinting at a potential convergence or improved learning efficiency. Thus, we decided to go with the decay starting after 20% of total episodes.

D. States

Next, what each state consists of must be considered. In Q-learning, a type of reinforcement learning algorithm, the term "state" refers to a representation of the current condition of the environment that the agent is interacting with. It encapsulates all the relevant information that the agent needs to make an informed decision about the next action it should take. The state of the environment is represented by a 3-dimensional continuous space consisting of three parameters: the beta value and the packet loss and throughput. The *beta* value is a parameter for the TCP CUBIC congestion control algorithm, which controls how quickly the TCP protocol responds to network congestion. *Packet loss* acts as the percentage of packets not received by the receiving node. *Throughput* is the packets per second reaching the receiving node. This allows the environment to represent the current *beta* value and the subsequent *loss* and *throughput* of transmission.

E. Reward Function

We trained the Q-Learning function with a linear reward function designed to prioritize both the enhancement of throughput and the minimization of packet loss based on the reward function used in *N. Jay et al* [3].

$$throughput - (2 * packetloss)$$

Where *throughput* is measured in packets per second and *loss* is the percentage of all packets sent but not acknowledged. This reward function is constructed to guide the agent towards actions that both optimise the rate of data transfer (throughput) and significantly reduce occurrences of data packet loss. The dual emphasis ensures that the agent doesn't blindly chase high throughput at the expense of reliability and quality of the transmission.

F. Step Function

The step function in the *TCPEnv* class represents an integral part of the Q-learning process, functioning as the principal bridge for interactions between the Q-learning agent and the TCP environment. This function, in each iteration of the Q-learning procedure, processes an action selected by the agent based on the existing state-action value estimates (the Q-table). Upon reception of the selected action, the function subsequently modulates the environment's state and calculates the reward derived from the action's outcome.

The first phase of this function involves the application of the action, which modifies the beta parameter of the TCP CUBIC congestion control algorithm. An ns-3 network simulation ensues, with the associated packet loss and throughput being recorded. These updated parameters, beta, associated packet loss and throughput, collectively form the new environmental state.

Following the state transition, the function engages in reward calculation. These reward values are fundamental to the Q-learning process, guiding the update of the Q-table.

Concurrently, the function oversees the management of episodes. An episode concludes when the action count surpasses a predefined limit, set at 100 steps in the present context. The episode's termination triggers the activation of the done flag, marking the episode's conclusion.

The function concludes by returning the new state, the accrued reward and the episode's termination status. The step function is pivotal to Q-learning operations, defining the agent-environment interaction paradigm, determining state transition based on agent actions, and establishing the reward allocation scheme. Consequently, the agent exploits this information to refine its Q-table, thereby optimizing its action-selection policy.

G. Learning Rate

The difference in learning rates can have a profound impact on the learning dynamics of an algorithm like Q-learning. A learning rate determines the extent to which newly acquired information overrides old information. In Fig. 3. The *learning rate* is 0.1 and in Fig. 5 is 0.01. Fig. 5. exhibits a steadier.

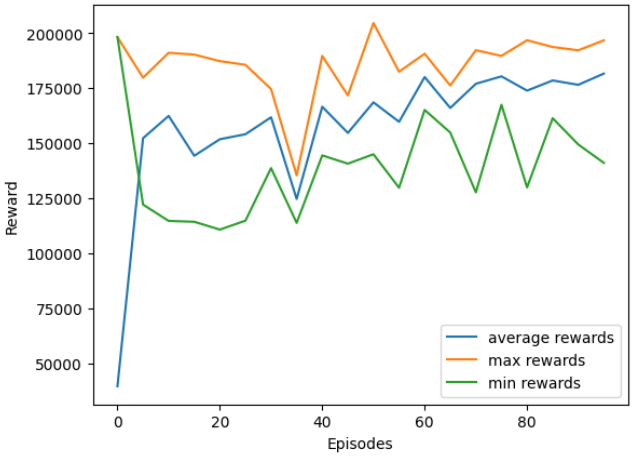

Fig. 5. Episode rewards learning rate = 0.01.

ascent, suggesting a possibly more consistent learning or performance improvement. In contrast, Fig. 3. shows more fluctuation in its *average rewards* line, which could indicate less consistent learning or performance. Thus, we opted to use a *learning rate* of 0.01 benefiting from a slower adaptation, allowing the model to better generalise from its experiences.

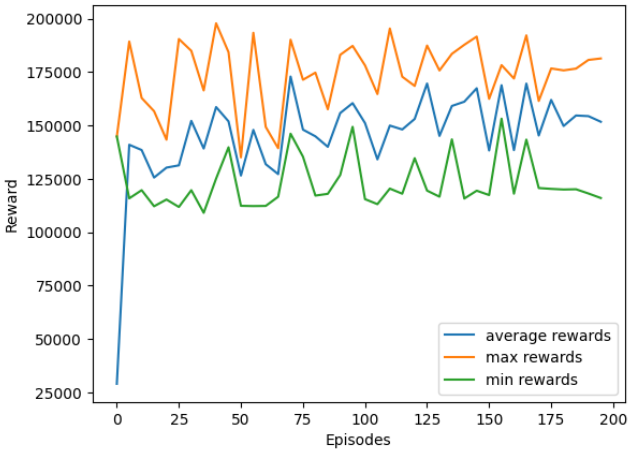

Fig. 6. Episode rewards discount factor = 0.90.

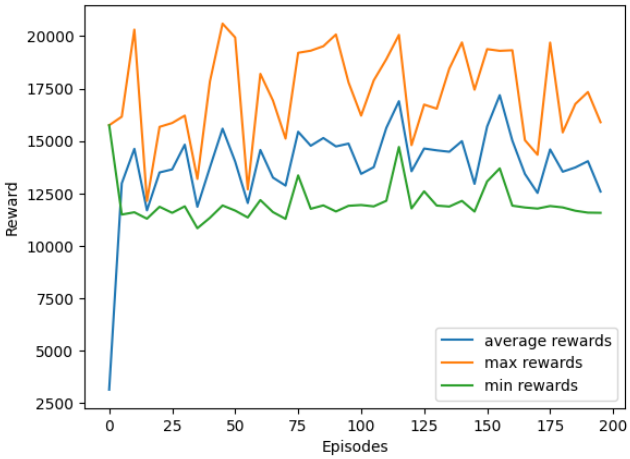

Fig. 7. Episode rewards discount factor = 0.95.

H. Discount Factor

The *discount factor* is a parameter used in reinforcement learning to balance immediate and future rewards. A value closer to 1 gives more importance to future rewards, promoting long-term planning, while a value closer to 0 prioritises immediate rewards, making the agent more short-sighted in its decisions.

In Fig. 6, with a *discount factor* of 0.90, we observe a more consistent upward trajectory in average rewards over the 200 episodes. This behaviour suggests a balanced approach between exploration and exploitation. Conversely, Fig. 7, which has a *discount factor* of 0.95, presents a more fluctuating profile in average rewards. This indicates that the algorithm, with a higher emphasis on future rewards, may be undertaking a more explorative approach in the initial stages but struggles to maintain a consistent performance in later episodes. Therefore, for more consistent learning we chose a *discount factor* of 0.90.

V. Evaluation

To evaluate the Q-Learning algorithm, we compared the packet loss and throughput of a transmission with the default *beta* = 0.7 with our algorithm. The sending node data rate was kept constant at 8Mbps. However, the receiving node data rate was tested at 4 different values, 4Mbps, 6Mbps, 8Mbps and 12Mbps. These varying values allow us to test the performance of the Q-Learning algorithm in varying network conditions with different levels of packet loss and throughput. The lower receiving rates experience high packet loss as the receiver is unable to handle the sender's higher data rate and the higher receiver rates experience lower packet loss.

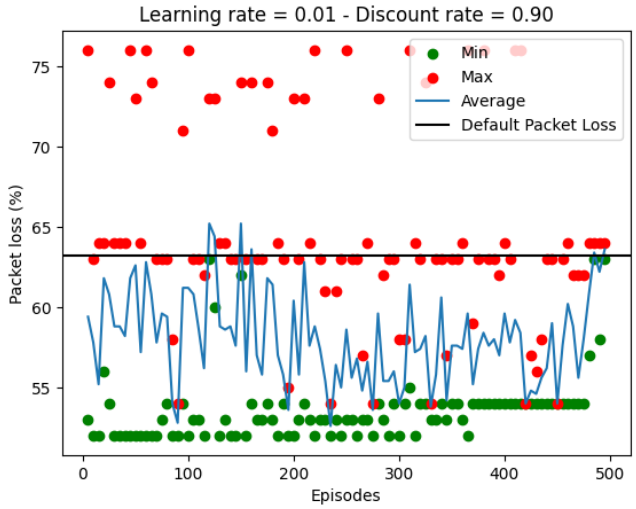

Fig. 8. Packet loss with a receiving data rate of 4Mbps.

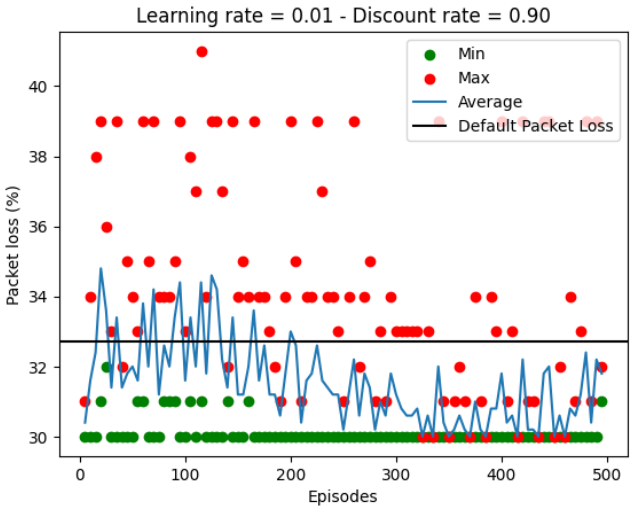

Fig. 9. Packet loss with a receiving data rate of 6Mbps.

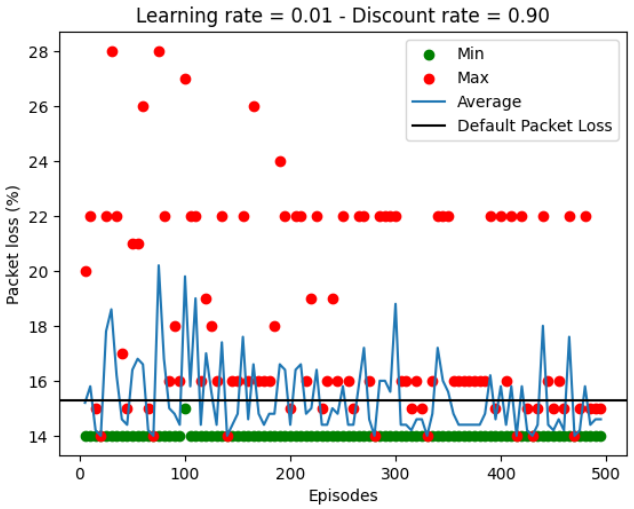

Fig. 10. Packet loss with a receiving data rate of 8Mbps.

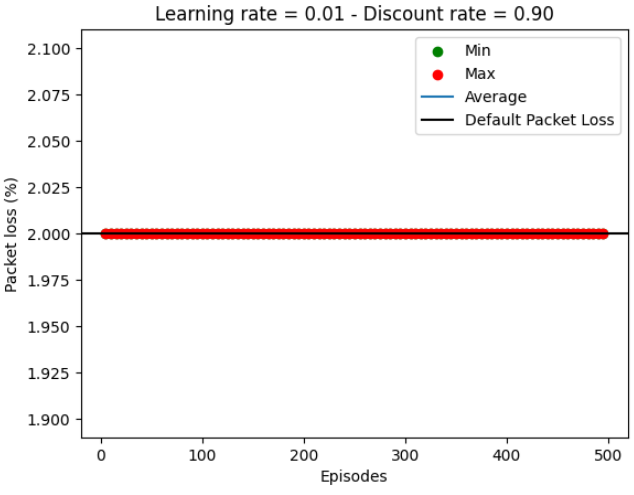

Fig. 11. Packet loss with a receiving data rate of 12Mbps.

A. Packet Loss

In each of the evaluations described in Fig. 8., Fig. 9., Fig. 10. And Fig. 11. the default packet loss from a transmission using the default *beta* of 0.7 is shown with the black line. At each step of each episode, the packet loss was logged. Each episode represents 100 steps and takes the average packet loss for each transmission.

The most noticeable improvement in packet loss occurs with the 4Mbps receiving data rate described in Fig. 8. The average packet loss is consistently below the default transmission's packet loss only going above on 4 occasions. Comparing the average packet loss to the default packet loss of 63.6% our algorithm averages a packet loss over the 500 episodes of 58.2% which is an 8.6% improvement. While there is notably a large variance with the maximum packet loss consistently above 70% the majority of maximums sit around the default level. Overall, this shows that under conditions where there is a high level of loss, the algorithm performs well and reduces packet loss as intended.

With a receiving rate of 6Mbps described in Fig. 9. Average packet loss is also consistently below the default packet loss of 32.7%. The average packet loss for the 500 episodes was 31.3% which is a 4.3% decrease. During the purely exploratory stage, before epsilon decays at episode 100, the average packet loss fluctuates around the default value. However, as the algorithm reduces exploration average packet loss trends downwards. It appears to find a consistent policy around the 300-episode mark where the maximum packet loss is significantly reduced, and the maximum does not break 34%. Packet loss is not reduced as significantly as with the 4Mbps sender however, the consistency of the policy is much clearer in later episodes. Although the reduction in the change of packet loss with a 6Mbps sender the algorithm still manages to find a successful policy.

With the receiving rates of 8Mbps and 12 Mbps per second, the algorithm appears to have minimal effect. With the 8Mbps receiving rate as described in Fig.10. The average packet loss fluctuates around the average loss of 15.3%. With the 12Mbps receiving rate as described in Fig.11. The packet loss stays constant at the 2.7% rate. Although there are no improvements in packet loss at these rates, it does display the adaptability of the algorithm to handle lower rates of packet loss and not degrade performance.

B. Throughput

Throughput tells a similar story to packet loss. With 4Mbps and 6Mbps receiver rates described in Fig. 12. and Fig. 13., respectively. The average throughput with a receiver rate of 4Mbps is consistently above the default value of 242.9 packets per second. With an average of 275.1 packets per second, the throughput is increased by 13% on average. With the average only dropping below the default value during 4 episodes. This shows that in environments with high packet loss, the algorithm can reduce packet loss while keeping improving throughput.

With the 6Mbps receiver described in Fig. 13. Again, during the purely exploratory stage, the throughput is more variable than in later episodes. However, around the 250-episode mark

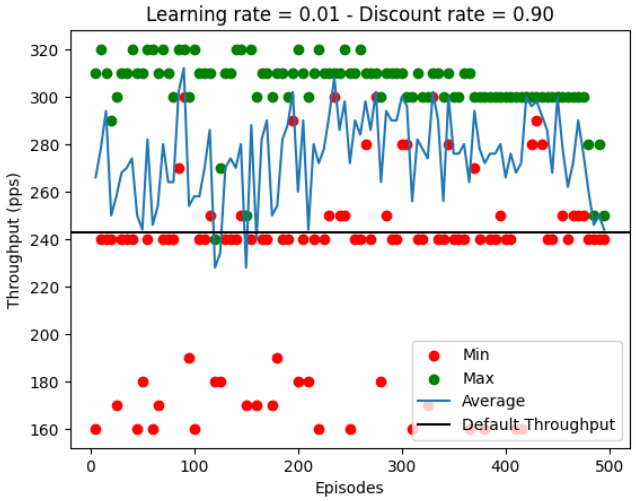

Fig. 12. Throughput with a receiving data rate of 4Mbps.

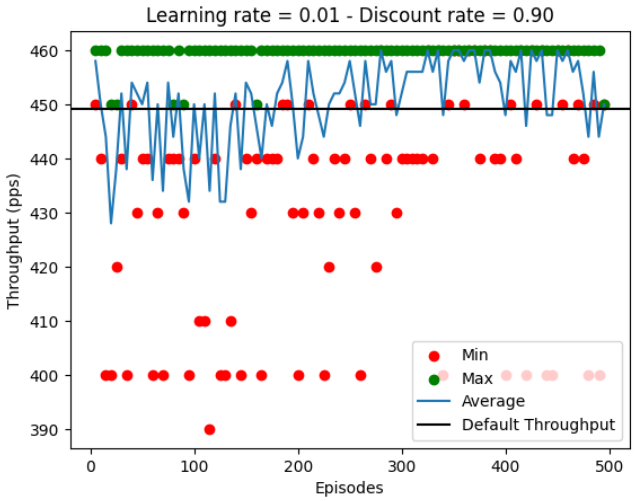

Fig. 13. Throughput with a receiving data rate of 6Mbps.

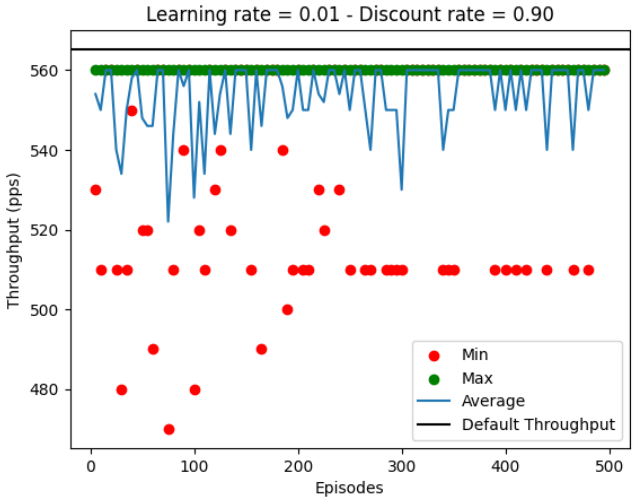

Fig. 14. Throughput with a receiving data rate of 8Mbps.

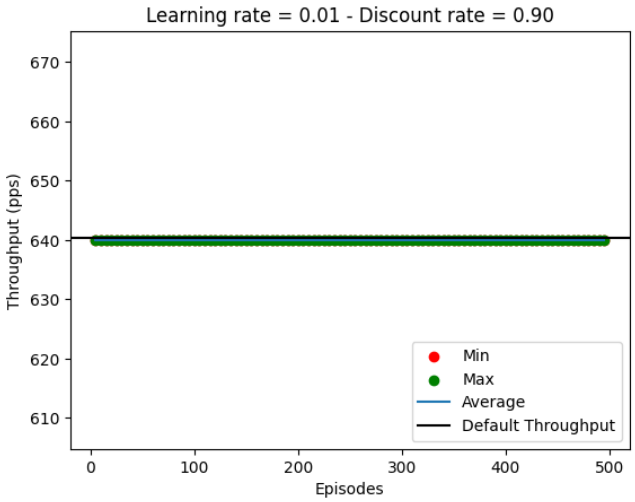

Fig. 15. Throughput with a receiving data rate of 12Mbps.

there is a consistent policy found, and the throughput bounces between 460 – 450 packets per second. Overall, the average increase is negligible at < 1%. However, with an improvement in packet loss at this data rate this shows that throughput is not being sacrificed at the expense of improving packet loss.

At a receiving data rate of 8Mbps per second described in Fig. 14. The average throughput is affected negatively with the default transmission's throughput of 565.2 packets per second and the average throughput over all episodes of 553.4 packets per second. This produces a 2.1% decrease in throughput.

Interestingly, with a receiver data rate of 12 Mbps described in Fig. 15. there is no change in throughput which is the same result with packet loss.

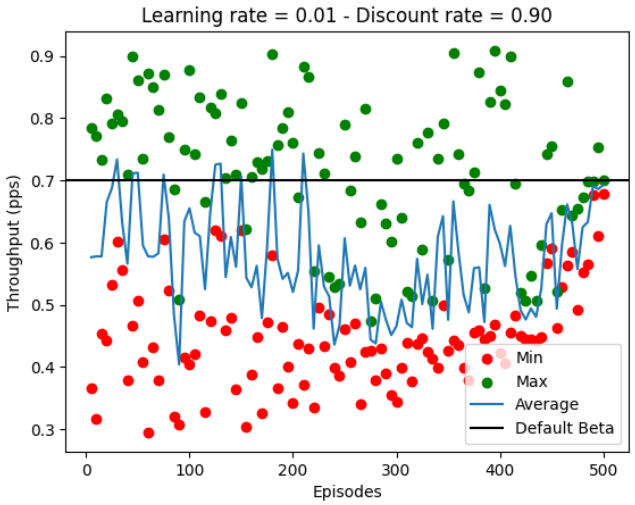

Fig. 16. Beta values during episodes with receiver data rate 4Mbps.

C. Beta

When looking at the changes in beta during episodes, the algorithm chose a policy of lowering beta as described in Fig. 16. Beta functions as the multiplicative decrease constant for TCP CUBIC. After loss events, the congestion window is

decreased by a factor of beta. Lower values of beta reduce the congestion window size more aggressively. Thus, when there are high levels of loss it makes sense that the algorithm would decrease the beta as this would allow for greater reductions in the congestion window size after each loss event.

D. Summary

The performance of the Q-learning algorithm developed in this project performs significantly better during transmissions of high levels of packet loss. The algorithm decreases beta which in turn causes a greater reduction in the congestion window size during loss events. With a packet loss decrease of 8.6% and an increase in throughput of 13%, this is a significant improvement. However, this does not satisfy the goal of 15%. Furthermore, at a receiving data rate of 6Mbps, this is further reduced and at 8Mbps and 12Mbps there was negligible change. As packet loss grows 15% becomes larger and thus makes it more difficult to achieve this goal. While at lower rates of packet loss, there are fewer loss events meaning beta has a reduced effect on the congestion window size.

Something to note is that as this algorithm can only modify TCP CUBIC parameters after each transmission this is something that is not ready to be implemented for general use. TCP is used for reliable data transfer and therefore consistency and practicality of a congestion control method is key. However, in situations where there is a need for repeated transmissions with similar data, this would simulate a set of conditions similar to our project. Our Q-Learning proves to be able to learn a policy which does decrease packet loss and increase throughput particularly.

VI. Conclusion

In this project, we looked to fine-tune TCP CUBIC's parameters to improve congestion control. However, the goal of a 15% improvement in packet loss and throughput was not achieved. This project has demonstrated that tuning TCP CUBIC's parameter beta can improve congestion control performance.

In future work, the impacts of the scaling constant *C* on congestion control could be investigated. As with beta predominantly affects the reduction congestion window during loss events. However, *C* affects the congestion window growth. Investigating the tuning of both C and beta simultaneously could find a balance of improving congestion control during loss events and during normal transmission. In particular, as *C* affects window growth this could reduce the loss events in the first time with a modified congestion window growth.

References

[1] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," ACM SIGOPS Operating Systems Review, vol. 42, no. 5, pp. 64-74, July 20081.

[2] S. Abbasloo, C. -Y. Yen, and H. J. Chao, "Classic Meets Modern: a Pragmatic Learning-Based Congestion Control for the Internet," in Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20), New York, NY, USA, 2020, pp. 632-647, doi: 10.1145/3387514.3405892.

[3] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A Deep Reinforcement Learning Perspective on Internet Congestion Control," in International Conference on Machine Learning, 2019, pp. 3050-3059.R. Fardel, M. Nagel, F. Nuesch, T. Lippert, and A. Wokaun, "Fabrication of organic light emitting diode pixels by laser-assisted forward transfer," *Appl. Phys. Lett.*, vol. 91, no. 6, Aug. 2007, Art. no. 061103.

[4] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-Driven Congestion Control: When Multi-Path TCP Meets Deep Reinforcement Learning," in IEEE Journal on Selected Areas in Communications, vol. 37, no. 6, pp. 1325-1336, June 2019. doi: 10.1109/JSAC.2019.2904358.

[5] Y. Kong, H. Zang, and X. Ma, "Improving TCP Congestion Control with Machine Intelligence," in Proceedings of the 2018 Workshop on Network Meets AI & ML (NetAI'18), New York, NY, USA, 2018, pp. 60-66. https://doi.org/10.1145/3229543.3229550.

[6] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: re-architecting congestion control for consistent high performance," in Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15), USA, 2015, pp. 395-408.

[7] I. L. Afonin, A. V. Gorelik, S. S. Muratchaev, A. S. Volkov, and E. K. Morozov, "Development of an adaptive TCP algorithm based on machine learning in telecommunication networks," in 2019 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO), Russia, 2019, pp. 1-5. doi: 10.1109/SYNCHROINFO.2019.8814023.

[8] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. MIT Press, 1998.

[9] "17 Goals for Sustainable Development", United Nations, https://sdgs.un.org/goals.

[10] "Gymnasium is a standard API for reinforcement Learning, and a diverse collection of reference environments" Gynasiuim.farama.org. [Online]. Available: https://gymnasium.farama.org/.