Improving Network Congestion Control using Reinforcement Learning

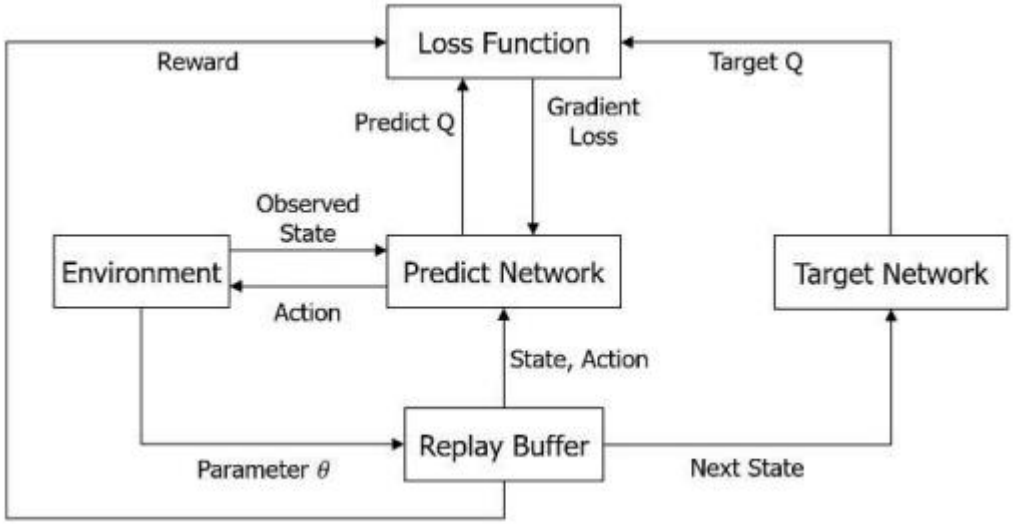
Problem Statement

Congestion Control plays a significant role in improving network resource utilization to achieve better performance. Although traditional Congestion Control algorithms like Slow Start, AIMD etc. exist, they are static in nature, i.e., they don't adapt dynamically to the changing nature of network in real time. Hence, it is desirable to design Congestion Control algorithms using Reinforcement Learning.

Objective

1. To Model and Design a Reinforcement Learning based Congestion Control Algorithm to improve efficiency.
2. To evaluate the model and compare it with the traditional Congestion Control Algorithms.

System Architecture



Modules

1. Simulated network environment

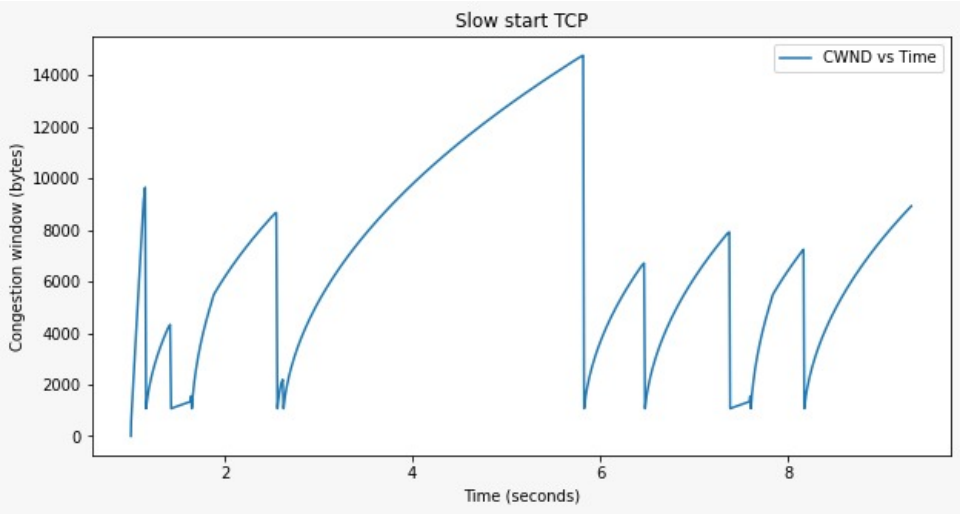
- The Network Simulator, NS3 is used to simulate the real-world network environment with congestion.
- INPUT: The network topology with parameters such as Bandwidth and delay.
- OUTPUT: Simulation output including the details of the packets transmitted.

2. Data Gatherer/Capturer
    - INPUT: Simulation output of the NS3 module.
    - OUTPUT: Set of features based on the captured packets
3. Learner
    - Learning from the data captured and performs an action also considering previous mistakes.
    - It uses DQN function approximation
4. Loss Predictor/Actuator
    - Uses the learned information and takes action according to the state of the network
    - Balances between exploration and exploitation using decaying epsilon greedy algorithm

Implementation

- The topology consists of the environment and the agent.
- The environment has observation and action spaces.
- The agent takes action given the observation which is used to give rewards and update the observation.

Slow Start Algorithm's Congestion Window Plot



The congestion window increases until there is a packet loss. When there is a packet loss, the congestion window is decreased to a very low value and starts all over again.

CODE

Topology:

```
/* Topology:
 *
 *   Right Leafs (Clients)                    Left Leafs (Sinks)
 *           |            \                  /        |
 *           |             \    bottleneck   /         |
 *           |              R0--------------R1         |
 *           |             /                 \         |
 *           |   access   /                   \ access |
 *          N -----------                     --------N
 */
```

Topology Setup:

The topology setup consists of a single bottleneck link that exists between N clients and N sinks. This implementation follows one on each side. There is also an access link that exists between a client/sink and the bottlenecks. Bottleneck bandwidth is 2Mbps with a delay of 0.01 milliseconds and the access bandwidth is 10Mbps with a delay of 20ms.

```
int main (int argc, char *argv[])
{
  uint32_t openGymPort = 5555;
  double tcpEnvTimeStep = 0.1;

  uint32_t nLeaf = 1;
  std::string transport_prot = "TcpRl";

  double error_p = 0.0;
  std::string bottleneck_bandwidth = "2Mbps";
  std::string bottleneck_delay = "0.01ms";
  std::string access_bandwidth = "10Mbps";
  std::string access_delay = "20ms";
  std::string prefix_file_name = "TcpVariantsComparison";
  uint64_t data_mbytes = 0;
  uint32_t mtu_bytes = 400;
  double duration = 10.0;
  uint32_t run = 0;
  bool flow_monitor = false;
  bool sack = true;
  std::string queue_disc_type = "ns3::PfifoFastQueueDisc";
  std::string recovery = "ns3::TcpClassicRecovery";
```

The RL-Agent model is a fully connected neural network with one hidden layer. The number of hidden layer neurons is the average of input and output neurons. Activation function used is ReLU.

```
get_agent.tcpAgents = {}
get_agent.ob_space = ob_space
get_agent.ac_space = ac_space

def modeler(input_size, output_size):
    """
    Designs a fully connected neural network.
    """
    model = tf.keras.Sequential()

    # input layer
    model.add(tf.keras.layers.Dense((input_size + output_size) // 2, input_shape=(input_size,), activation='relu'))

    # output layer
    # maps previous layer of input_size units to output_size units
    # this is a classifier network
    model.add(tf.keras.layers.Dense(output_size, activation='softmax'))

    return model

state_size = ob_space.shape[0] - 4 # ignoring 4 env attributes

action_size = 3
action_mapping = {} # dict faster than list
action_mapping[0] = 0
action_mapping[1] = 600
action_mapping[2] = -150

# build model
model = modeler(state_size, action_size)
model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
model.summary()
```

Model Summary:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 7)                 91

 dense_1 (Dense)             (None, 3)                 24

=================================================================
Total params: 115
Trainable params: 115
Non-trainable params: 0
_________________________________________________________________
```

In epsilon-greedy action selection, the agent uses both exploitations to take advantage of prior knowledge and exploration to look for new options: The epsilon-greedy approach selects the action. Going beyond, in Decaying Epsilon Greedy algorithm, initially the agent mostly performs

exploration to cover a wide variety of situations. The aim is to have a balance between exploration and exploitation.

```
# Epsilon-greedy selection
if step == 0 or np.random.rand(1) < epsilon:
    # explore new situation
    action_index = np.random.randint(0, action_size)
    print("\t[*] Random exploration. Selected action: {}".format(action_index), file=w_file)
else:
    # exploit gained knowledge
    action_index = np.argmax(model.predict(state)[0])
    print("\t[*] Exploiting gained knowledge. Selected action: {}".format(action_index), file=w_file)

# Calculate action
calc_cWnd = cWnd + action_mapping[action_index]
```

After configuring the action, the action is taken on the current environment state, and the reward and next state are obtained. The different configurations are listed in a later section.

```
new_ssThresh = int(cWnd/2)
actions = [new_ssThresh, new_cWnd]
```

```
# Take action step on environment and get feedback
next_state, reward, done, _ = env.step(actions)

rewardsum += reward

next_state = next_state[4:]
cWnd = next_state[1]
rtt = next_state[7]
throughput = next_state[11]
```

The RL agent uses a DQN approach for training, as specified below.

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, ... till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

Chasing a nonstationary target!

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2 \right] \Big|_{\theta=\theta_k}$$

$$s \leftarrow s'$$

Updates are correlated within a trajectory!

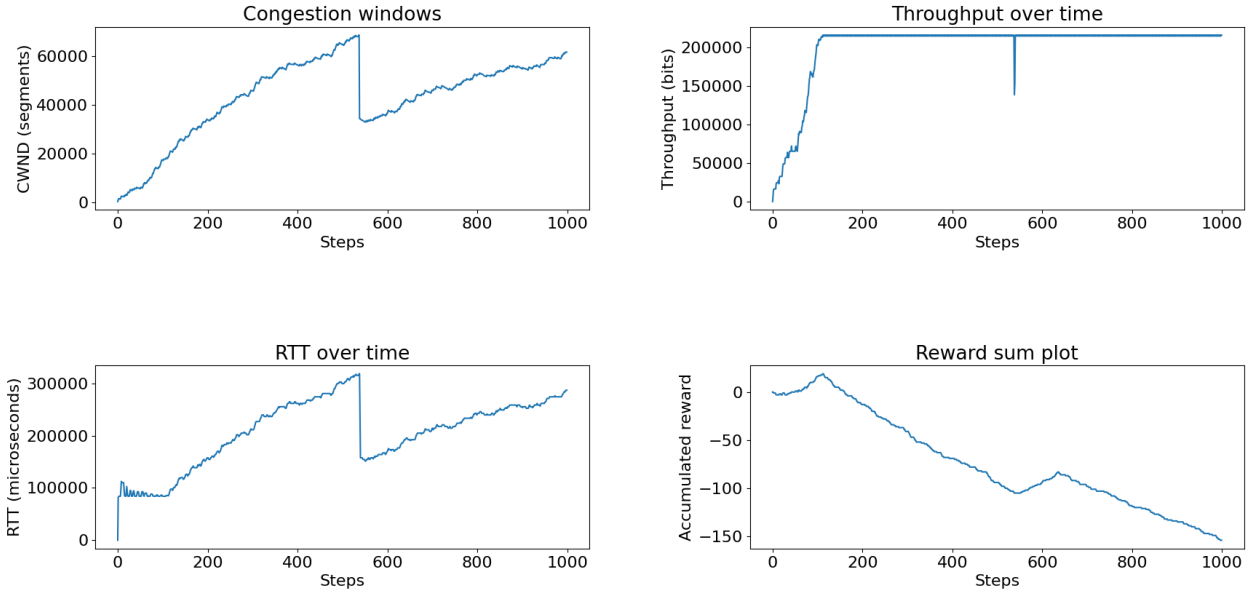
```
# Train incrementally
# DQN - function approximation using neural networks
target = reward
if not done:
    target = (reward + discount_factor * np.amax(model.predict(next_state)[0]))
target_f = model.predict(state)
target_f[0][action_index] = target
model.fit(state, target_f, epochs=1, verbose=0)

# Update state
state = next_state

if done:
    print("[X] Stopping: step: {}, reward sum: {}, epsilon: {:.2}"
          .format(step+1, rewardsum, epsilon),
          file=w_file)
    break

if epsilon > min_epsilon:
    epsilon *= epsilon_decay
```

Plot:

Config 1:

This implementation does not impose a cap/limit on the congestion window, which would cause repeated congestion/packet loss over time as shown below. Moreover, the agent's behavior would be erratic.

```
# Config 1: no cap
new_cWnd = calc_cWnd
```

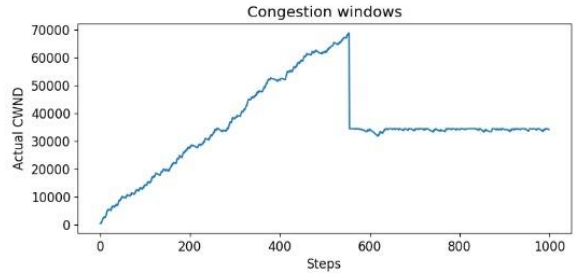


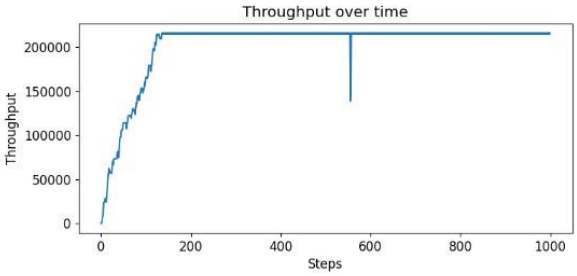
Config 2:

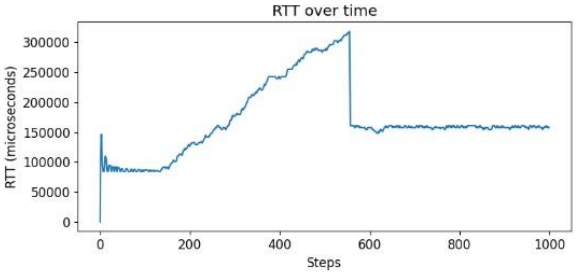
Once congestion occurs, the congestion window is decreased to half of the current value and capped at that value.

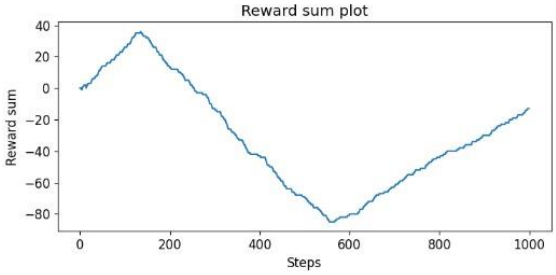
However, it would be desirable if we could avoid congestion altogether and cap the congestion window while throughput reaches maximum.

```
# Config 2: cap cWnd by half upon congestion
# ssThresh is set to half of cWnd when congestion occurs
# prevent new_cWnd from falling too low
ssThresh = state[0][0]
new_cWnd = max(init_cWnd, (min(ssThresh, calc_cWnd)))
```







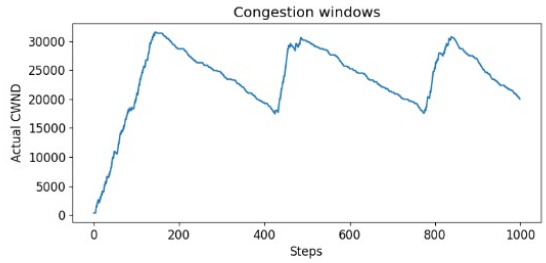


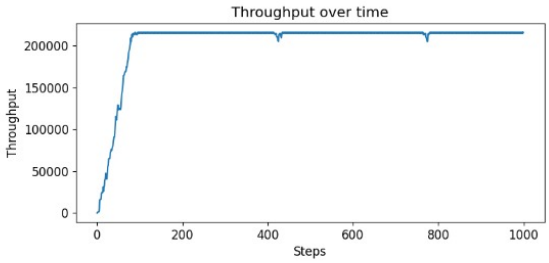
Config 3:

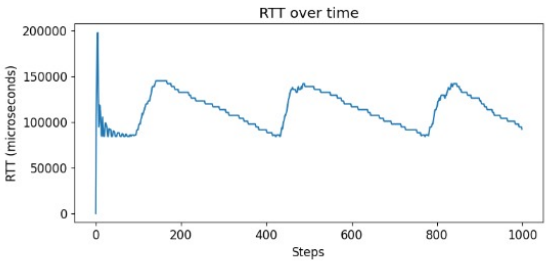
Varying throughput cap – When the variance in throughput falls below a certain value, this indicates that maximum throughput is reached, and hence the congestion window is capped. However, the agent is not stable in this case.

Fine tuning may offer better results.

```
# Config 3: if throughput cap detected, fix cWnd
# detect cap by checking if recent variance less than 1% of current
thresh = state[0][0] # ssThresh
if step+1 > recency:
    tp_dev = math.sqrt(np.var(tp_history[(-recency):]))
    tp_1per = 0.01 * throughput
    if tp_dev < tp_1per:
        thresh = cWnd
new_cWnd = max(init_cWnd, (min(thresh, calc_cWnd)))
```



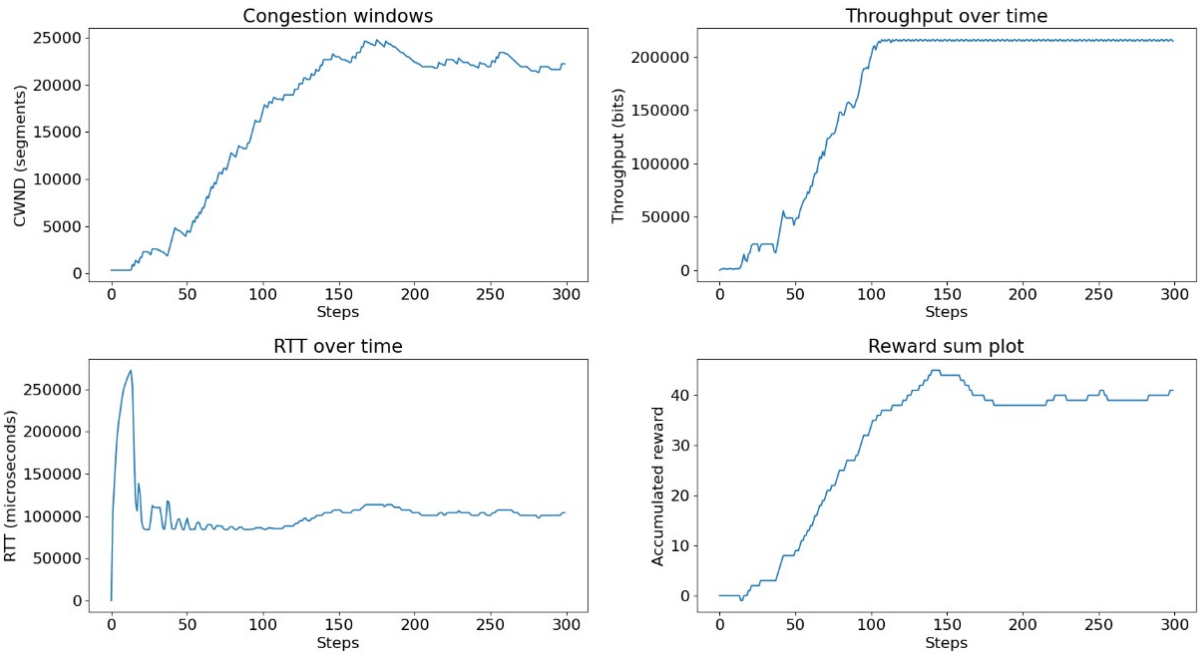




Config 4:

Fixed throughput cap – Once the throughput reaches an ideal maximum, a hard cap is set for the congestion window. This causes the agent to stabilize. However, the maximum value has to be manually chosen, which is dependent on experimental data and hence not practical.

```
# Config 4: detect throughput cap by checking against experimentally determined value
thresh = state[0][0] # ssThresh
if step+1 > recency:
    if throughput > 216000: # must be tuned based on bandwidth
        thresh = cWnd
new_cWnd = max(init_cWnd, (min(thresh, calc_cWnd)))
```



Performance Metrics

1. Round Trip Time (RTT) -> The duration it takes for a network request to go from a starting point to a destination and back again to the starting point.
2. Congestion Window -> It is a factor that determines the number of bytes that can be sent at any point in time.
3. Throughput → It is the amount of data transmitted successfully from the sender to the receiver in unit time. It is also known as "Effective Bandwidth" and is measured in Mbps.

It would be preferable if both the RTT and Congestion window are optimum and stable, and if the throughput value would be at a maximum value such that there would be full utilization of bandwidth capacity.

References

Author name (et al), year, Paper title, Publishing details

1. Yiming Kong, Hui Zang, and Xiaoli Ma. 2018. Improving TCP Congestion Control with Machine Intelligence. In Proceedings of the 2018 Workshop on Network Meets AI & ML (NetAI'18). Association for Computing Machinery, New York, NY, USA, 60–66. DOI:https://doi.org/10.1145/3229543.3229550
2. Huiling Jiang, Qing Li, Yong Jiang, Gengbiao Shen, Richard Sinnott, Chen Tian, Mingwei Xu. 2020. When Machine Learning Meets Congestion Control: A Survey and Comparison. *ArXiv* abs/2010.11397 (2021).