RL-based Algorithm for Input-Queued Switches in Heavy-Traffic: An Empirical Study

Lakshya J Prakirt Jhunjhunwala Siva Theja Maguluri

October 9, 2022

1 Introduction

This report outlines the work done in $n \times n$ switch simulations during my project work at GT under Prof. Siva Theja Maguluri with Prakirt Jhunjhunwala's guidance.

1.1 Related work

The paper detailing Efficient Randomized Algorithms For Input-Queued Switch Scheduling mention the Max Weight Matching (MWM) method and iSLIP method. They also detail Random I, Random II and Random III which we also implement to serve high traffic switches. The paper Reinforcement Learning for Optimal Control of Queueing Systems by Qiaomin et al. proposes a new algorithm, called Piecewise Decaying ϵ-Greedy Reinforcement Learning (PDGRL), which applies model-based RL methods over a finite subset of the state space. They show that the average queue backlog under PDGRL with an appropriately constructed subset can be arbitrarily close to the optimal result.

1.2 Outline

In this work we aim to study the effectiveness of Q Learning algorithms in serving queues in a heavy traffic situation. We run our algorithms on a switch of size 3×3 and obtain corresponding results. There has been an increase in interest in the application of Q Learning algorithms in the sphere of optimization and efficient scheduling.
In this paper we observe the effects of 4 different Q Learning algorithms. Based on the observations made and some previous literature, we propose a modified L2-norm Greedy Q Learning algorithm to deal with the problem. To add to this, we also propose a Deep Q Learning Network model for predicting the best servicing matrices for the given configuration of the switch.

2 Model

We can model the scheduling problem as a matching problem in a bipartite graph with n input nodes and n output nodes. The edge between input i and output j is present if $q_{ij}(t)$ is nonempty; we give it weight w_{ij}, which equals the length of $q_{ij}(t)$. This vector, represented by $\mathbf{q}_t$ gives the current state of the queue. Given the transfer constraints in the switching fabric, a matching for this bipartite graph is a valid servicing schedule, which we will refer to as the service matrix in the report, denoted by $\mathbf{s}_t$. Throughout this paper, the letters in bold denotes vectors in $\mathbb{R}^{n \times n}$.
At any time t, $a_{ij}(t)$ ($\mathbf{a}_t$ in matrix form) denotes the number of packets that arrive at the input port i to be delivered to output port j. The term matrix and vector are used interchangeably throughout the paper. The mean arrival rate vector is denoted by $\mathbb{E}[\mathbf{a}_t] = \lambda$ and variance $Var(\mathbf{a}_t) = \sigma^2$.
The weight of the schedule is the sum of the queue lengths that are being served in the given time slot. A scheduling algorithm or policy picks the schedule $\mathbf{s}_t$ in every time slot. Using these notations, we arrive at

the expression for finding the current length of the queue. The servicing matrix $\mathbf{s}_t$ we refer to in this report is a unitary doubly stochastic matrix.

$$\mathbf{q}_{t+1} = [\mathbf{q}_t + \mathbf{a}_t - \mathbf{s}_t]^+$$

where $[x]^+ = \max(0, x)$. This is to make sure in cases where a queue which has no packets in line is served, the number of packets does not become less than zero. While such a service is futile, it is possible to be encountered when servicing the queue in an optimal fashion. All the queues in the current switch matrix are truncated at a value of c. Since we are maintaining a Q-Table of finite size, managing memory for larger values of truncation becomes difficult.

3 Simulations of Existing Algorithms

We start the experiments by performing simulations on a heavily loaded $n \times n$ switch, where $n = 3$. The arrival of packets is a Poisson Distribution. By heavily loaded, we mean the row and column sums of the arrival matrix distribution sums to 1. In case of a 3×3 switch, the matrix would look something like

$$\boldsymbol{\lambda} = \begin{bmatrix} 0.33 & 0.33 & 0.33 \\ 0.33 & 0.33 & 0.33 \\ 0.33 & 0.33 & 0.33 \end{bmatrix}.$$

We perform these simulations on 4 different standard service algorithms namely - MaxWeight, Random, Power of 'd', Pick and Compare with a truncation value of 5 for the switch-queue lengths.
These algorithms will be detailed elaborately in a section later on in the report. MaxWeight performs the best as expected with Pick and Compare coming next, though the difference between the two is not that huge.

3.1 Standard Service Algorithms

- **MaxWeight** - The maximum-weight matching (MWM) algorithm delivers a throughput of up to 100 percent and provides low delays by keeping queue sizes small. The double stochastic permutation matrix which serves the maximum number of packets in the queue is chosen to service the switch
- **Random** - A double stochastic permutation matrix is randomly generated to service the switch.
- **Power of d** - d random double stochastic permuatation matrices are randomly generated. The same matrix may be generated twice or more for $d \geq 2$. The matrix which services the highest number of packets is then chosen as the service matrix.
- **Pick and Compare** - The previous service matrix used is compared with the current randomly generated service matrix. Whichever service matrix serves the highest number of packets is used to serve the switch.

4 Q Learning Algorithms

After the initial simulations of serving the switches with known algorithms, we implement Q Learning algorithms to gauge the performance of learning algorithms on switch servicing optimization problems.

Reinforcement Learning is a simulation-based method where the optimal value function is approximated using simulations. An Reinforcement Learning model consists of four elements, which are namely an environment, a learning agent with its knowledge base, a set of actions that the agent can choose from and the response from environment to the different actions in different states. The knowledge base is made up of the so called Q-factors for each state–action pair.

Q-learning essentially solves the Bellman equation iteratively in an asynchronous style to obtain the optimal value function.

We use Off-Policy TD control for obtaining an optimal average queue length. Temporal difference (TD) learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function.

In all these algorithms we have chosen a truncation value of $c = 5$ since the size of the Q Table generated increases exponentially with c, hence maintaining it would require memory in orders of almost PetaBytes. The Q Learning algorithms I implemented are as follows:

- Average-Cost Q-Learning
- Differential Q-learning
- Relative-value-iteration (RVI) Q-learning
- Discounted-cost Q-learning

All four algorithms are simulated with 4 different behavioural policies for generating the service matrix. The Q Table for the given state-action pair is updated according to the Q Learning algorithms used. The reward in each step is the total number of packets present in all the n queues. The goal of the Q Learning algorithm is to minimise the queue length.
The following subsections outline the algorithms of the Q Learning techniques implemented. In all the algorithm expressions used below, $Q(\mathbf{s}_t, \mathbf{a}_t)$ denotes the Q value for the state $\mathbf{s}_t$ and its associated action $\mathbf{a}_t$. The associated action in case of our model is the servicing matrix $\mathbf{s}_t$. The update of the Q Table is detailed for each of the Q Learning algorithms used. μ_t denotes the average running reward obtained over t time steps. The reward at each time step is defined as the sum of the elements present in the current state $\mathbf{s}_t$, which is nothing but the total number of packets present in the switch required to be serves. The state $\mathbf{s}_t$ in the Q Learning algorithm corresponds to $\mathbf{q}_t$ defined earlier in the model description.

4.1 Average Cost Q Learning

Average Cost Q Learning algorithm averages the reward sum obtained over the t time steps that the algorithm runs. $\frac{1}{t}$ is used instead of a fixed α in our algorithm. Here $\mathbf{p}_t$ denotes the packet arrival matrix at time step t.

Algorithm 1 Average Cost Q Learning Update

$r_t \leftarrow \sum \mathbf{s}_{t-1}$
$\mu_t \leftarrow (1 - \frac{1}{t}) * \mu_{t-1} + \frac{1}{t} * r_t$
$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + \mathbf{p}_t - \mathbf{a}_t$
$Q(\mathbf{s}_{t-1}, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_{t-1}, \mathbf{a}_t) + \frac{1}{t} * (r_t - \mu_t + \min\{Q(\mathbf{s}_{t-1}, \mathbf{a})\} - Q(\mathbf{s}_{t-1}, \mathbf{a}_t))$

4.2 Differential Q Learning

Differential Q Learning algorithm obtains μ_t by using the differential update used in the Q table update weighed by the hyperparameter η. The hyperparameter α is used both the Q-Table and μ_t update. The step sizes of these hyperparameters are something we need to still research about, since we do not observe a smooth graph when varying the these hyperparameters.

Algorithm 2 Differential Q Learning Update

$r_t \leftarrow \sum \mathbf{s}_{t-1}$
$\mu_t \leftarrow \mu_t + \eta * \alpha * (r_t - \mu_{t-1} + \min\{Q(\mathbf{s}_{t-1}, \mathbf{a})\} - Q(\mathbf{s}_{t-1}, \mathbf{a}_t))$
$Q(\mathbf{s}_{t-1}, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_{t-1}, \mathbf{a}_t) + \alpha * (r_t - \mu_t + \min\{Q(\mathbf{s}_{t-1}, \mathbf{a})\} - Q(\mathbf{s}_{t-1}, \mathbf{a}_t))$

4.3 Relative Value Iteration Q Learning

Relative Value Iteration (RVI) Q Learning algorithm uses a special function $f(Q)$ to obtain an average estimate of the reward. The truncation value c comes into play here by playing a role in approximating the function $f(.)$ value for the given Q Table. While the calculations of finding $f(Q)$ may seem computationally intensive, we devised an efficient way to store the running sum of the Q Table for faster calculations.

Algorithm 3 RVI Q Learning Update

$f(Q) \leftarrow \frac{1}{(c+1)^3} * \sum Q(\mathbf{s}_{t-1}, \mathbf{a}_t)$
$r_t \leftarrow \sum \mathbf{s}_{t-1}$
$Q(\mathbf{s}_{t-1}, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_{t-1}, \mathbf{a}_t) + \alpha * (r_t - f(Q) + \min\{Q(\mathbf{s}_{t-1}, \mathbf{a})\} - Q(c))$

4.4 Discounted Cost Q Learning

Discounted Cost Q Learning algorithm uses the hyperparameter γ to discount the minimum Q value obtained for the current state over the possible actions from the Q Table. The discount factor γ essentially determines how much the reinforcement learning agents cares about rewards in the distant future relative to those in the immediate future.
When $\gamma = 0$, the agent is entirely myopic and only cares about the immediate reward. If $\gamma = 1$, the agent will evaluate each of its actions based on the sum total of its future rewards.

Algorithm 4 Discounted Cost Q Learning Update

$r_t \leftarrow \sum \mathbf{s}_{t-1}$
$Q(\mathbf{s}_{t-1}, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_{t-1}, \mathbf{a}_t) + \alpha * (r_t + \gamma * \min\{Q(\mathbf{s}_{t-1}, \mathbf{a})\} - Q(\mathbf{s}_{t-1}, \mathbf{a}_t))$

4.5 Behavioural Policies

In this report, behavioural policies refer to the different policies choose the matrix used to service the switch. The 4 policies we have used are as follows:

- ϵ Greedy
- Random
- MaxWeight
- ϵ MaxWeight or Random

In the following subsections I will be explaining the above mentioned policies one by one.

4.5.1 ϵ Greedy

Algorithm 5 ϵ Greedy

$\epsilon \leftarrow Uniform\ Random\ Number \in [0, 1]$
if $\epsilon < epsilon$ **then**
    service_matrix = $\arg\max\{Q(\vec{s}_{t-1}, \vec{a})\}$
else
    service_matrix = RandomServiceMatrix
end if

4.5.2 ϵ MaxWeight Random

Algorithm 6 ϵ-MaxWeight Random

$\epsilon \leftarrow Uniform\ Random\ Number \in [0, 1]$
if $\epsilon < epsilon$ **then**
    service_matrix = MaxWeightServiceMatrix($\vec{s}_{t-1}$)
else
    service_matrix = RandomServiceMatrix
end if

5 Simulations Results

The following section outlines the simulation results obtained for the 4 Q Learning Algorithms using the 4 Behavioural policies. All these simulations are for a training episode of length 1 million and testing of length 2 million. The simulations are performed for a 3×3 switch for a truncation value of 5.

5.1 Average Cost Q Learning

Average cost Q Learning performs the worst out of all the algorithms. Since the Q value update for the average cost algorithm used does not have any hyper-parameters, we won't be plotting graphs.

5.2 Differential Q Learning

In the following slide I will show the outputs obtained from different graphs for the different behavioural policies implemented in Differential Q Learning. The table below contains the best average queue lengths obtained from all the tested cases.

Behavioural Policy	**Average Queue Length**
ϵ Greedy	24.822387 η: 0.1 and α: 0.0005
Random	26.577637 η: 0.8 and α: 0.005
MaxWeight	30.242637 η: 0.2 and α: 0.7
ϵ-MaxWeightRandom	27.717741 η: 0.9, α: 0.01, ϵ: 0.1

Table 1: Best Simulation results for Differential Q Learning $n = 3$ and $c = 5$

5.3 Relative Value Iteration Q Learning

The table below contains the best average queue lengths obtained from all the tested cases.

Behavioural Policy	**Average Queue Length**
ϵ Greedy	24.742919 α: 0.0005
Random	26.710662 α: 0.0005
MaxWeight	30.242554 α: 0.9
ϵ-MaxWeightRandom	27.594404 α: 0.1 and ϵ: 0.1

Table 2: Best Simulation results for Relative Value Iteration Q Learning $n = 3$ and $c = 5$

5.4 Discounted Cost Q Learning

The table below contains the best average queue lengths obtained from all the tested cases.

Behavioural Policy	Average Queue Length
ϵ Greedy	25.230730 α: 0.1 and γ: 0.6
Random	26.926160 α: 0.01 and γ: 0.95
MaxWeight	30.242558 α: 0.7 and γ: 0.7
ϵ-MaxWeightRandom	27.523360 α: 0.6 , γ: 0.4 and ϵ: 0.1

Table 3: Best Simulation results for Discounted Cost Q Learning $n = 3$ and $c = 5$

5.5 Summary

Here I will be showing some of the visualizations of the above results to get better insights of the numbers obtained.
It can be seen that the best obtained queue lengths perform only slightly worser than MaxWeight. This tradeoff is not very huge considering the time complexity for using MaxWeight is $\mathcal{O}(n^{2.5})$ and is not practical for existing high traffic networks.

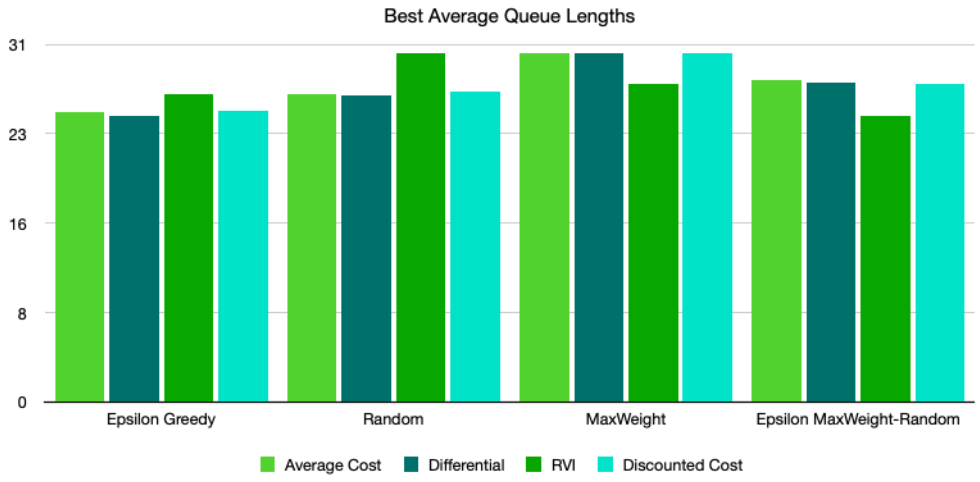

Figure 1: Comparison of the 4 behavioural policies and Q Learning Algorithms

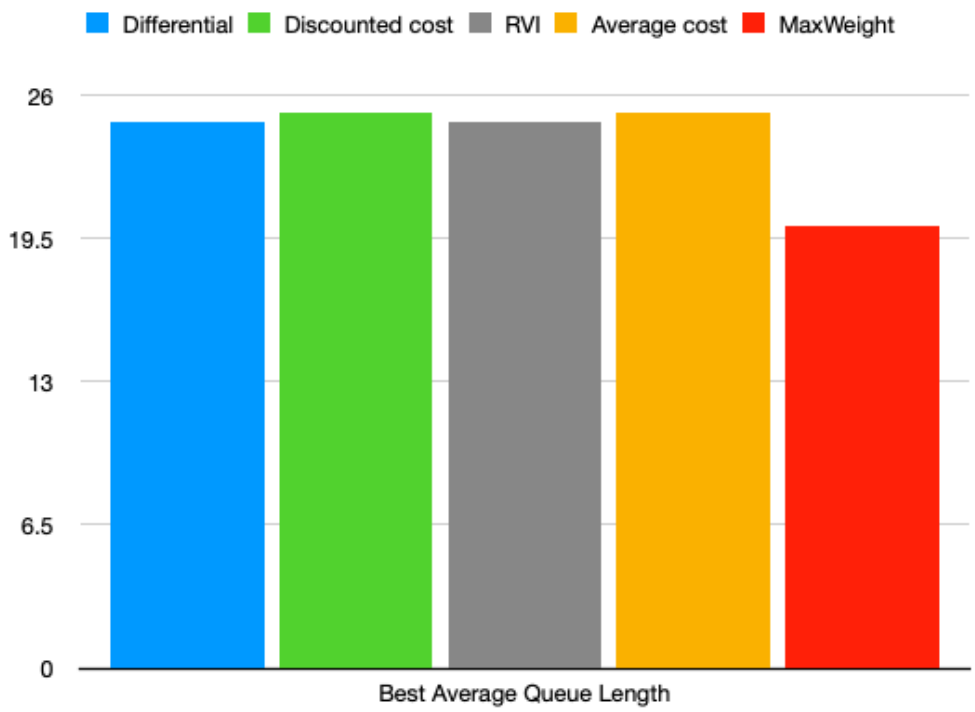

Figure 2: Comparison of the best Queue lengths obtained from the 4 models with MaxWeight

6 Modified L2 Norm RL

We have come up with a unique solution to tackle with the truncation of the Q Table, introducing the Modified L2 Norm Algorithm.

In this during training, we use the L2 norm of the current switch to figure out how the queue should be served. If it is lesser than an arbitrarily chosen value c, we serve the switch with a *randomly* generated service matrix, else we generate the service matrix using the MaxWeight strategy.

In the case where the maximum packet length is lesser than a certain truncation value c_1, we update the Q Table. c_1 is chosen as $\frac{c}{2}$ in such a manner so that the memory required to maintain the Q Table is not too huge.

Algorithm 7 Modified L2 Norm

```
if L2.norm(q_switch_matrix) ≤ c then
    service_matrix = RandomServiceMatrix
else
    service_matrix = MaxWeightServiceMatrix(q_switch_matrix)
end if
if max(q_switch_matrix) ≤ c_1 then
    Update Q Table
end if
```

When testing, we use the Q Table to predict the service configuration whenever the maximum packet length in the switch is lesser than c_1. Otherwise, we service the queue using the MaxWeight Strategy.

A simple calculation shows that the size of the Q Table for a switch of size $n = 3$ is around 60 million. However, we have chosen a training period of size 13 million and a test period of size 10 million. The Q

Table built during the training is phase is used to obtain the best service matrix for the given state.

We simulate the above algorithm for all the 4 Q Learning algorithms (as they have different Q Table updates).

6.1 MaxWeight

To get a good baseline for the above algorithm, we test it against a full MaxWeight Strategy of serving the switch for 10 million iterations. MaxWeight gives an average queue length of 244.5977. We observe that for certain values of hyperparameters used in the simulations, out algorithm is able to perform slightly better.

6.2 Average Cost Q Learning

The Average Cost Q Learning algorithm does not require multiple simulations as it does not have any hyperparameter dependent on it. The average queue length obtained in this case is 251.7375.

6.3 Differential Q Learning

Differential Q Learning performs slightly better than the MaxWeight algorithm for certain values of α and η. The output graphs unfortunately do not follow a proper gradient, but is rather jagged and is difficult to interpret. One of the future works could be to possibly understand the implication of these step sizes.

Here we can observe that for $\eta = 0.1$ and $\alpha = 0.5$ we obtain the least average queue length of 231.80205. While the difference between the MaxWeight and Modified Algorithm using Differential Q Learning is not that huge, we do observe a positive improvement in performance.

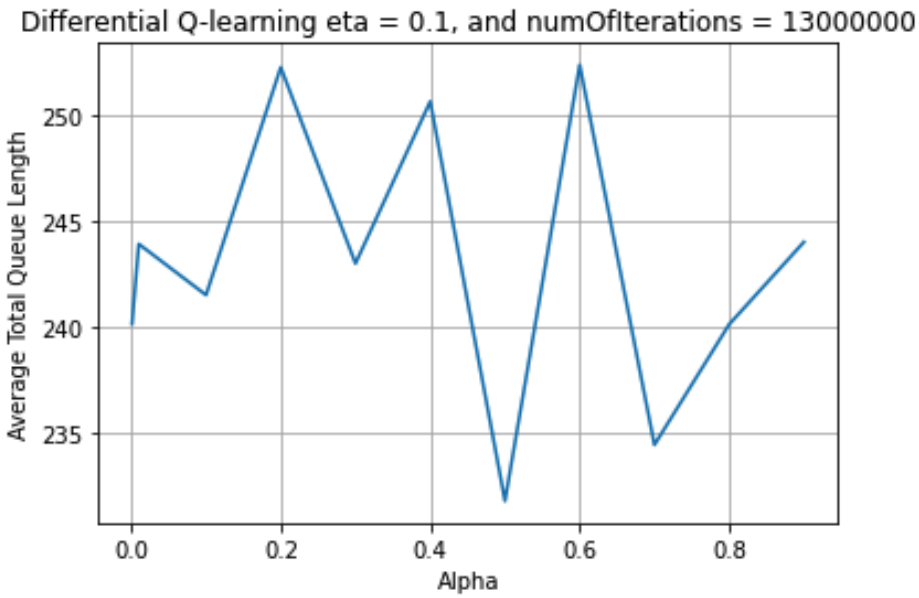

Figure 3: The Average Queue Length obtained for varying values of α

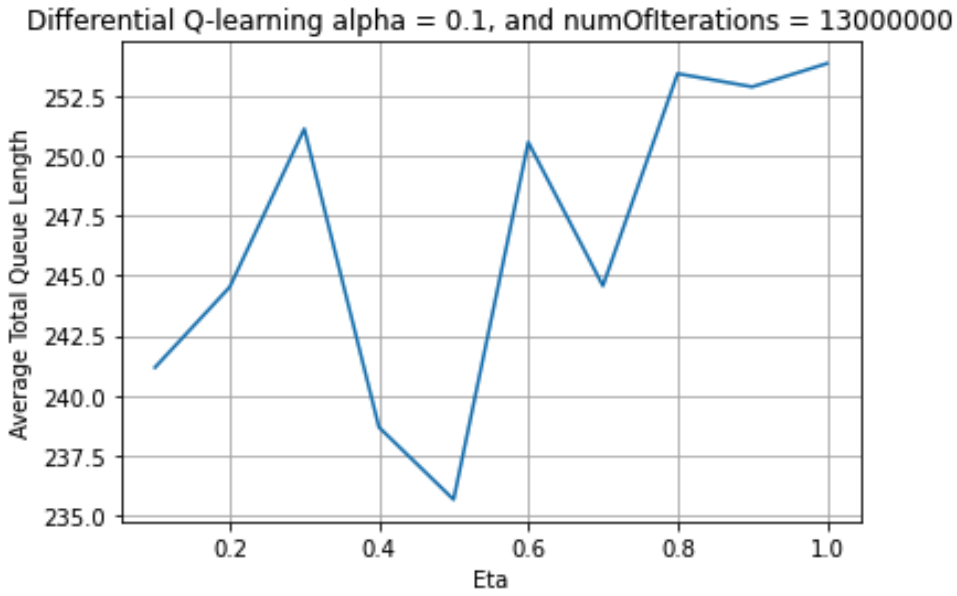

Figure 4: The Average Queue Length obtained for varying values of η

6.4 Relative Value Iteration Q Learning

The best average queue length obtained using Relative Value Iteration algorithm for the Q Table update is 232.1697. This value is obtained for $\alpha = 0.008$. Again, it is unclear why this value of alpha gives this value. In the figure given below I have plotted the variation of the average queue length with α.

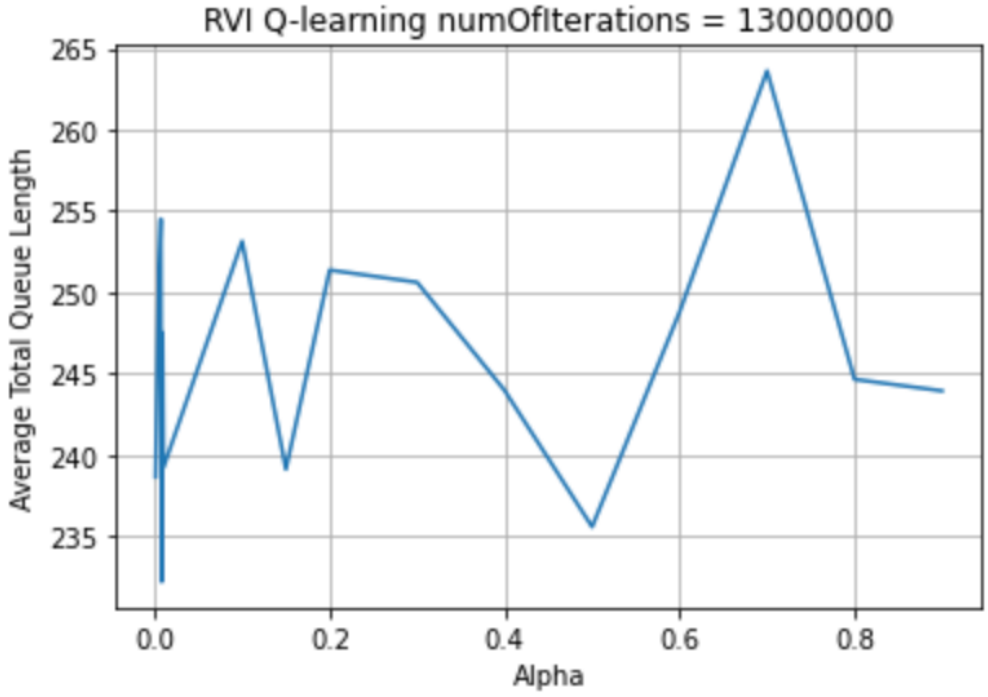

Figure 5: The Average Queue Length obtained for varying values of α

6.5 Discounted Cost Q Learning

Here we plot the variations of average queue length for varying values of γ and α hyperparameters. The minimum average queue length obtained is 233.1513, for $\alpha = 0.1$ and $\gamma = 0.7$. The graphs obtained for the same are as follows.

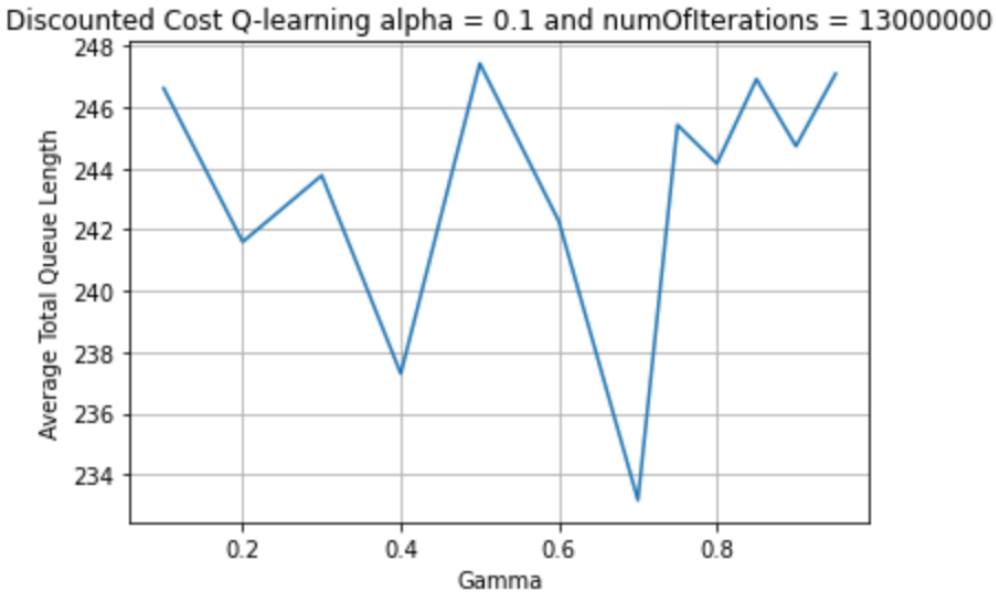

Figure 6: The Average Queue Length obtained for varying values of γ

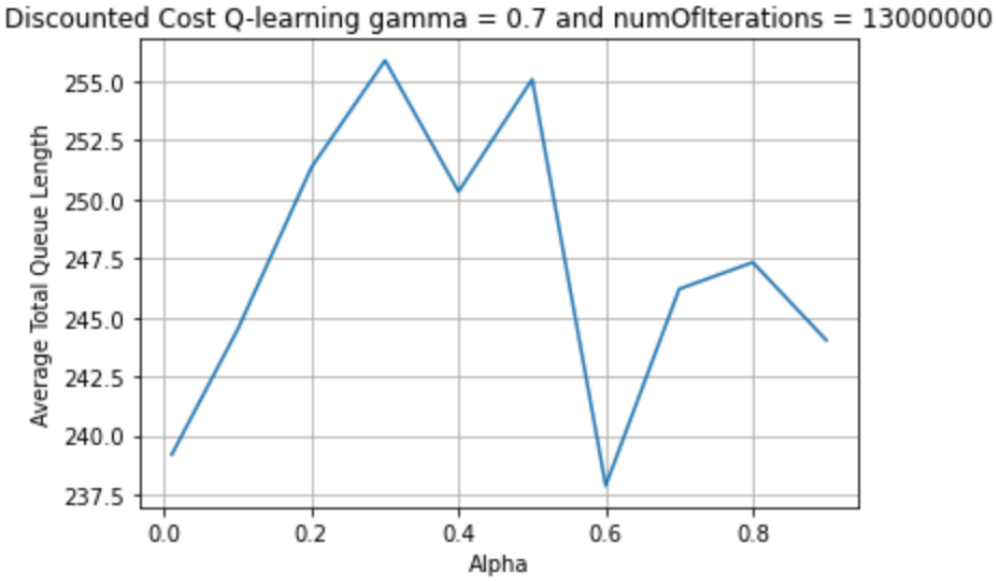

Figure 7: The Average Queue Length obtained for varying values of α

6.6 Summary

From the values observed for the tested hyperparameters, we can observe that Differential Q Learning updates perform the best, with Relative Value Iteration updates coming in at a close second. Average Cost

Q Learning performs the worst, with the average queue length higher than that of MaxWeight. We observe an improvement of 5.23% using the Differential Q-Learning approach over MaxWeight.
One possible way to improve performance could be using even larger training periods, as we are sure with the number of iterations we have used all possible state and action pairs have not been covered entirely.

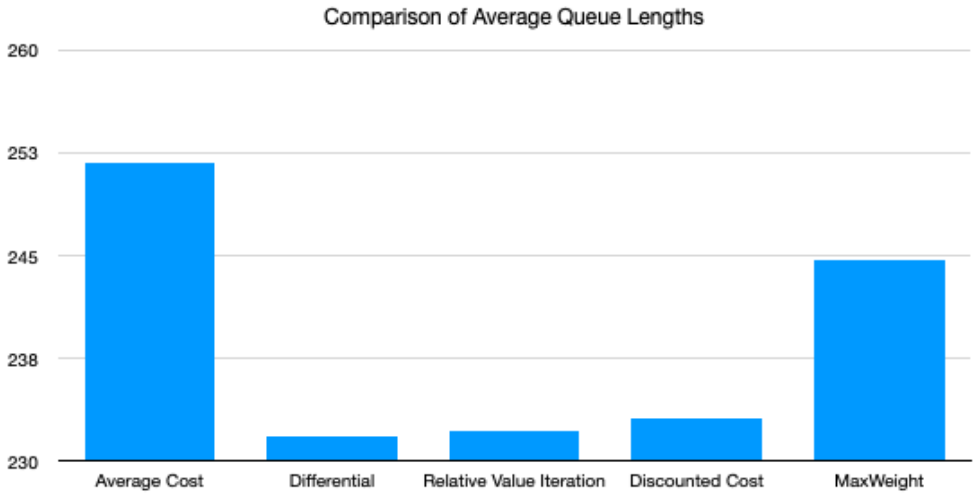

Figure 8: The Average Queue Length obtained for different Q Table updates

7 Function Approximation

Possible future work to solve the problem of maintaining large Q Tables is delving into using a Deep Neural Network to generate the service matrix using our Modified ϵ Greedy Reinforcement Learning Algorithm.
We have created a primitive algorithm for the same. We observe that MaxWeight performs bettter than DQNs, however I trained them over a small number of time steps, we may observe better results for larger training periods.
One of the problems of the DQN algorithm is that it overestimates the true rewards; the Q-values make the agent think is going to obtain a higher return than what it will obtain in reality. To fix this, we use the Double DQN (DDQN) algorithm, decoupling the action selection from the action evaluation. The target network is updated with the policy network weights every *target_update* iterations. The policy network's predicted actions is compared with that of the target network's expected Q value to update the weights according to the PyTorch's SmoothL1 loss function. We use RMSprop as the optimizer for the back-propagation of the loss.
I have trained the model over 10,000 timesteps for 50 episodes, starting from the previous state at every episode. For the update rule, if the Frobenius norm of the current queue matrix is lesser than a threshold, we choose to service it using the DDQN network using the ϵ-greedy behavioural policy, else we use the MaxWeight algorithm to service the switch.

Algorithm 8 Modified DDQN Algorithm

```
if frob.norm(q_switch_matrix) ≤ c then
    if rand.uniform ≤ ϵ then
        service_matrix = policy_net(q_switch_matrix)
    else
        service_matrix = RandomServiceMatrix
    end if
else
    service_matrix = MaxWeightServiceMatrix(q_switch_matrix)
end if
```

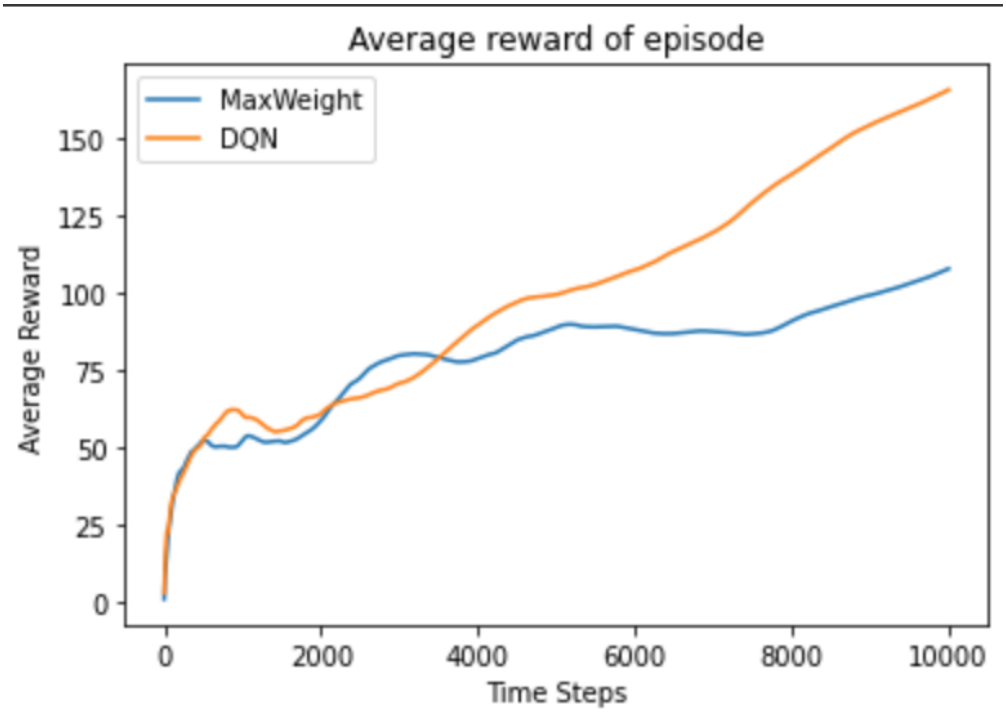

Figure 9: The Average Queue Length obtained for different Q Table updates

8 Conclusion

We can see that DDQN has potential and can offer promising results if longer simulations are performed. Futhermore, exploring DQN using other Q-Learing techniques is a possible direction which further can lead to.

9 Personal Remarks

I would like to thank Prakirt for guiding me regularly and Prof. Siva for giving me this opportunity and being a great support throughout the project. Every meeting I have had with both of them was thought-provoking and engaging. I learnt a lot through out this project thanks to their candid interactions and honest inputs.