Task Scheduling with Deep Q-Networks

Seth Harlaar
School of Engineering, Computer Engineering
University of Guelph
Guelph, Canada
sharlaar@uoguelph.ca

Jacob Buczakowski
School of Engineering, Computer Engineering
University of Guelph
Guelph, Canada
jbuczako@uoguelph.ca

***Abstract*—Traditional CPU scheduling algorithms are well developed and are effective at scheduling CPU tasks to optimize processing time. These methods, however, have reached their limit and offer little room for improvement in performance. The adaptive nature of Neural Networks makes them an excellent method to use in scheduling. This paper outlines the benefits provided by AI when used to implement a CPU Task Scheduler. The information is used to develop and test an AI model capable of CPU task scheduling using a Deep Q-Learning Neural Network. With the ability to evolve and produce more effective results with each iteration, this report shows the improvement of a well-trained Neural Network. We discuss the results of training over 100k steps and further modifications that would yield greater results.**

***Index Terms*—CPU, Machine Learning, Scheduling, Deep Q-Learning Network, PyTorch**

I. Introduction

This report presents our work in developing an Artificial Intelligence (AI) model capable of scheduling a CPU's tasks. Given the rise in AI usage across many industries, we are interested in taking the developments and benefits of this technology and applying it to CPU scheduling. There are many algorithms that have been developed to handle CPU scheduling and given the great amount of development put into them, they do so optimally. The goal of our research is to provide a model that will serve as the next step in CPU scheduling, and help optimize efficiency of scheduling, past the limitations of current algorithms, thus increasing throughput.

II. Background Information

Our research will be compared to current algorithms and methods of CPU scheduling, which include:
First Come First Serve (FCFS): Allows for all processes to be completed with same priority, however shorter tasks are delayed by long tasks.
Shortest Job First (SJF): Short tasks get to jump to the beginning of the line; however long important tasks may be pushed back as a result.

These algorithms work well, and allow for efficient scheduling, each with their own benefits and drawbacks. Since all these algorithms are deterministic, their strengths and drawbacks are constant, leaving room for improvement in the way of scheduling efficiency.

We will be utilizing a Neural Network in order to determine the most optimal CPU task schedule. Neural Networks (NN) are computer AI models loosely based on the neural connections found in human brains. A NN consists of various nodes that are organized in layers. The nodes are then connected to various other nodes in other layers of the network. Each connection of a node is assigned a weight, which acts as multiplier when the values going through each node are added during calculations. When data is input to the NN, the NN selects the most appropriate traversal through the intertwined nodes, and then outputs the calculated result. By developing a neural network to manage process scheduling, benefits of the above algorithms can be achieved by leaving out the drawbacks. The NN also provides the ability to evolve and adapt to the use cases it is provided with, so that it dynamically alters its performance based on the inputs it is being given.

We will be using the Deep Q Learning method to implement our NN. This method takes the Q – learning technique and uses a NN to help make approximations that help narrow down millions of solutions. The Q-learning technique creates a table that provides the desired "Q-value" for a given state/action. The Q value is obtained using the formula shown below:

$$Q(s,a) = r(s,a) + \gamma * maxQ(s',a) \tag{1}$$

This formula determines the q value by adding the immediate reward to the highest q value possible for the next state, multiplied by the discount factor gamma. Given the 1 to 1 mapping of the values in the table, large problems with thousands of states and actions end up creating tens of thousands of q values, which leads to problems that become too complicated to solve. By feeding the states and actions as inputs into the NN, the Q values can be approximated, and they can be used to select the next action. Fig. 1 shows visual representations of a Q-value table and a Deep Q – learning implementation using a NN.

III. Previous Work

Agarwal and Jariwala [2] introduce the concept of AI in the context of CPU scheduling. It provides constructive feedback of various types of scheduling algorithms that use Neural Networks. In the report, they highlight the fact that CPU scheduling becomes a very complex problem without a single optimal solution. As a result, this type of problem caters to Neural Networks, as they excel at narrowing large problem sets and producing optimal answers. This report concludes that

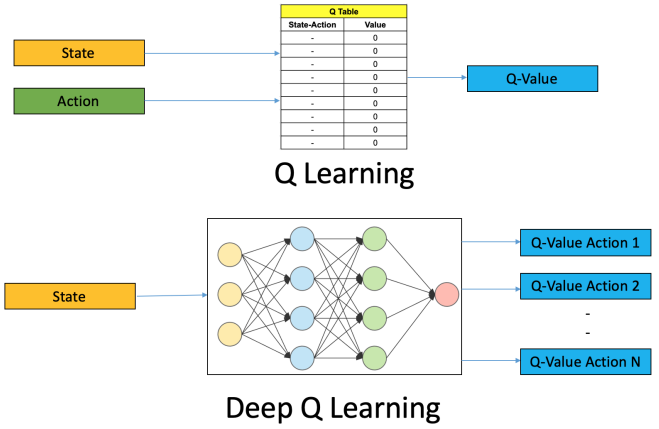

Fig. 1. Diagrams Showing Components of Different Q-learning Techniques

Genetic Algorithms are the best choice when implementing a Neural Network to schedule CPU tasks.

Nemirovsky et al. [3] highlight the need for AI schedulers in heterogenous systems. Due to the complexity of heterogenous systems, traditional scheduling algorithms become increasingly difficult to implement. Neural Networks are better suited to handle the large number of resources found in heterogenous systems. The report proposes a Machine Learning model that provides a 25-31% performance increase over traditional algorithms.

Sethi [4] introduces a fuzzy CPU scheduling algorithm for Real Time Operating Systems (RTOS). This implementation was then compared to a traditional priority algorithm, and an improved fuzzy algorithm. The sample sizes were quite small, at only 7 tasks, leading to a relatively weak analysis of these schedulers. As a result, this study would benefit from a larger sample to better analyze the effect of the fuzzy CPU scheduler.

Butt and Akram [5] continue the analysis of the previous study with a more in-depth look. In doing so, this report also introduced a fuzzy- round robin decision making system algorithm to further develop the idea of AI CPU scheduling.

IV. Experimental Procedure

A. Programs

To develop a Deep Q-Network Scheduler (DQNS) algorithm we needed an environment to simulate the selection, execution, and results of the scheduler so that we could train the scheduler with the achieved metrics and measure its performance. As such, we developed a simulation environment using Python as it is simple to program in and has a large community surrounding AI projects. To implement this environment, we developed a CPU program, a simulator program, a reports program, and a DQN scheduler program, as well as a schedule environment program for use by the DQN scheduler.

The CPU program accepts a task list file that follows the comma separated values format (.csv) which contains a set of tasks. The columns in the task list files are task ID, arrival time, burst time, and priority. The CPU program starts by parsing the data in the task list file and generating a list of task objects. The task object list is then passed to the simulator. The simulator accepts the task object list and runs the selected algorithm on the task object list to determine the order that the tasks should be executed in. The simulator generates a step list as a return value, which describes the execution and activity of the CPU while executing the tasks in the order determined by the algorithm.

Once the step list is completed, the step data is sent to the report program which calculates the average priority-weighted turnaround time (TT), average priority-weighted waiting time (WT), and completion time (CT). This data can then be used for performance analysis, comparison, and training.

B. Algorithms

The FCFS and SJF algorithm implementations were very straightforward. The FCFS algorithm simply sorts the program according to the arrival time. If multiple tasks have the same arrival time, then order is determined by input order of the task set. The final execution order for the FCFS algorithm is achieved once the sorting is completed.

To implement SJF, a queue containing all the ready tasks at the starting time is used. Then from the queue the task with the shortest burst time is selected. The observation time is then incremented to the completion time of the selected task and the queue is refreshed to include the new ready tasks. The program repeats this process until the scheduling is complete.

The DQNS program describes the DQN used for scheduling and the process used for training the network. It was programmed using the PyTorch library along with some custom code and has several parts. First some hyperparameters are set such as batch size, gamma, epsilon start and end, epsilon decay, tau and learning rate. Second, the neural network is described, from which an online and target network are instantiated. The supporting architecture for the DQN is then described, including the action selection function, the replay memory buffer, and the model optimization function. Finally, the training loop. The training loop iterates over task sets in a predefined folder. For each file, an environment is initialized, and the state is reset. While the task set is not fully scheduled a task is selected via the DQN. The selected task is sent to the schedule environment which returns a reward, an observation and a Boolean describing if the scheduling is complete. The current state, selected action, next state, and reward are all recorded in the replay buffer which is later used to train the network. After all the task sets in the task set folder have been completed, the training data is displayed.

The DQNS algorithm makes use of the schedule environment program. The schedule environment program keeps track of the list of tasks in the set it is currently being trained on, the schedule list containing the order of scheduling, and the step list corresponding to the execution and CPU activity of the current scheduling order. The schedule environment is also responsible for distributing rewards to the DQN and returning the next state, which is made up of up to 20 tasks to choose from and the current time.

C. Deep Q-Network

To setup our DQN for simulation we had to decide on some specifications we were going to use. Since we were using the Reinforcement learning tutorial from PyTorch [1] we decided to start with the values provided in the tutorial. We began with a batch size of 128, a discount rate of 0.99, a learning rate of 1e-4, and an update rate of 0.005.

Some other hyperparameters that were used but were not changed throughout the training process were epsilon start, with a value of 0.9, epsilon end, with a value of 0.05, and epsilon decay, with a value of 1000, which were all used for the epsilon-decay method of selecting an action. The replay buffer size was set to 1000.

We also needed a neural network to train and to generate actions from. We decided to go with a simple option of one hidden layer in between the input and output layers. The input layer has a size of 61 nodes. The neural network will be fed 20 tasks to select from, and will receive the burst time, arrival time, and priority of each task. The tasks themselves will therefore account for 60 inputs. The final input will be the current time, totalling 61. The output layer is a size of 20 nodes, where each node corresponds to a task selected to schedule. We used 45 nodes in the hidden layer which was selected based on a loose rule of half the input size plus the output size.

D. training

We also needed to make task sets that the DQNS could be trained on. To do so we used a Python script to generate hundreds of tasks sets, each with 100 tasks each. Each task was given a random value for the arrival time, burst time, and priority time. The arrival times and priorities were selected using a Poisson distribution with a mew value of 3 and 2, respectively, while the burst times were selected using a uniform distribution.

Since the FCFS and SJF algorithms are deterministic all the tests were completed on both algorithms before starting up the DQN. Then the DQNS program was run on the same task sets and the result could be compared.

The overall goal of this experiment was to lower the TT of the task set execution. We did this because we believed this accurately reflects the user experience. Some tasks that the computer performs are more important to the user experience than others. The more important tasks are given a higher priority and the less important ones a lower priority. The average priority-weighted turnaround time reflects this because it is a measure that reflects not only how long a task takes to complete, but also how important it is.

V. Presentation of Results

During testing we tweaked many of the different parameters involved in describing a DQN. The most challenging part of the DQNS algorithm was training the network itself. Over many of our tests, there was little to no improvement shown in the TT after tweaking the parameters.

A large portion of our testing involved changing the reward function used by the DQN to reward a good choice and punish a bad choice. For use in the reward functions and as measures of performance we calculated the TT, WT, and CT for each task set. The TT is calculated by taking the average of the turnaround time for each task in the task set multiplied by its priority. The WT is calculated by taking the average of the waiting time for each task in the task set multiplied by its priority. The CT time is the time that the last scheduled task in the task set is completed. We started by using a reward function that gave out rewards based on all three metrics measured. The reward function would divide a corresponding weight by each metric and add that value to the total reward. The results of these tests were minimal.

Over several iterations of the reward function, we came up with two variants that worked well, shown in equations (2) and (3).

$$round\frac{100000}{(1+currentTime-arrivalTime)} * \frac{1}{burstTime} \tag{2}$$

$$round\frac{100000 * priority}{(1+currentTime-arrivalTime)} * \frac{1}{burstTime} \tag{3}$$

The basic reward value given is the reciprocal of the turnaround time of the task selected. This would correspond to punishing the DQN for scheduling tasks that have been waiting for a long time and would encourage it to schedule tasks quickly. It is then divided by the burst time of the task, which would encourage the DQN to schedule the short ones to get them out of the way faster. The reward value is then multiplied by 100,000 so that the reward can scale properly with other negative rewards given. The reward values are rounded so that they are compatible with the PyTorch library. Equation (3) has an extra factor, the priority of the task, which the reward value is multiplied by, so that the DQN can be encouraged to schedule tasks that are more important.

Along with this reward function, we found that using a very low gamma value, of about 0.05 produced good results.

The results of this reward function were promising and showed a decrease in the desired metric, TT. Tables I, II, III, and IV show the summary of the results of the DQNS using reward functions (2) and (3), which were used to complete two tests, numbered 13 and 14. To demonstrate the improvement of the DQNS algorithm, we compared it to the SJF and FCFS algorithms.

TABLE I
Performance of Individual Algorithms on Bundle 7 in Test 13

	All 500 Files in Bundle			**First 100 files in Bundle 7**		
Metric	***DQNS***	***FCFS***	***SJF***	***DQNS***	***FCFS***	***SJF***
TT	2051.9	2226.7	1511.6	2014.18	2270.61	1508.89
WT	7499.7	8240.9	5508.8	7359.6	8290.09	5506.05
CT	1495.8	1493.6	1495	1490.48	1492.65	1490.08

TABLE II
PERFORMANCE COMPARISON OF DQNS TO FCFS AND SJF IN TEST 13

Metric	All 500 Files in Bundle		First 100 files in Bundle 7	
	FCFS-DQNS	*SJF-DQNS*	*FCFS-DQNS*	*SJF-DQNS*
TT	210.88	-540.2	256.433	-505.29
WT	741.18	-1991	930.473	-1853.6
CT	-2.156	-0.774	2.17	-0.4

Tests 13 and 14 were both completed using bundle 7, a bundle of 500 tasks sets, each containing 100 tasks. The only difference between the DQNS in test 13 (DQNS-13) and the DQNS in 14 (DQNS-14) is the reward function used. Otherwise, all hyperparameters remained the same. The results that DQN-13 and DQN-14 produced are compared to in test 13 and 14 are the same results from running the FCFS and SJF algorithms on bundle 7.

A. Test 13

Table I shows the overall average of the performance of all three algorithms in test 13 on bundle 7. It shows the performance average over all 500 files in the bundle, and the performance average over the last 100 files in the bundle. Table II shows the average performance of the DQNS-13 algorithm as compared to the other two algorithms. When we look at the 1st and 3rd columns in table II we can see that the performance increases over time, shown as an increase in the difference of the average metric values of FCFS algorithm minus the DQNS algorithm. We can see the same results in column 2 and 4 of table II when observing the performance of the DQNS algorithm as compared to the SJF algorithm. When looking at the completion times in table I it is interesting to note that the completion time does not change significantly across the different algorithms. This is because if the CPU always has tasks waiting to be executed it will always be running and a difference in order of execution will not change the required amount of execution time. We also did not employ any multithreading or parallelization techniques in the CPU.

Fig. 2 shows the TT for each algorithm performed on each file in the data set. The trendline shows an increase in the performance of the DQNS over the data set, corresponding to a decrease in TT as x approaches 500.

Fig. 3 shows a moving average of the relative performance of the DQNS algorithm in test 13 over time as compared to the FCFS algorithm. The graph shows the difference between the FCFS and the DQNS algorithm for each file in the data set. An obvious increase in performance is visible. The numbers begin in the negative region and increase over the task set to show a maximum increase of about 400 in TT over the FCFS algorithm. The numbers end in the positive region showing that the DQNS algorithm outperforms FCFS.

Fig. 4 shows a moving average of the relative performance of the DQNS algorithm in test 13 over time as compared to the SJF algorithm. This plot shows the difference between the SJF algorithm and the DQNS algorithm for each file in the data set. Again, there is an obvious increase in the performance of the DQNS algorithm. When compared to the SJF algorithm however, difference in TT values is negative. This shows that the DQNS algorithm performs worse than the SJF algorithm and does not achieve equal results over time. Even at its peak, the DQNS algorithm still performs about 450 TT worse than the SJF.

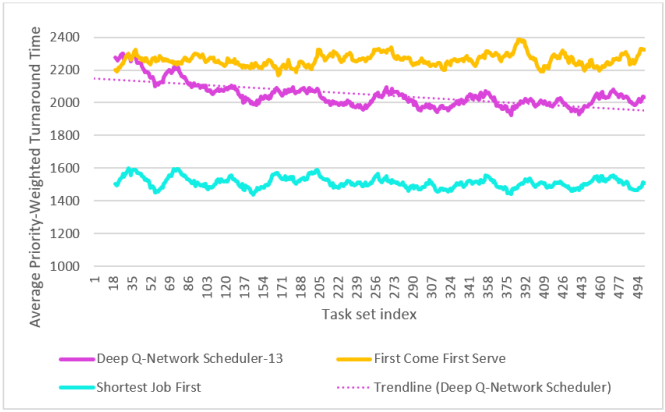

Fig. 2. Moving Average Plot of Performances of Different Algorithms on Bundle 7

B. Test 14

Table III shows the overall average of the performance of all three algorithms in test 14 on bundle 7. It shows the performance average over all 500 files in the bundle, and the performance average over the last 100 files in the bundle. Table IV shows the average performance of the DQNS-14 algorithm as compared to the other two algorithms. The results of test 14 were very similar to test 13. Table III demonstrates an increase in performance over the training period of the DQNS algorithm, seen by the decreasing of the TT. Table IV shows increases in relative performances of the DQNS-14 algorithm as compared to the FCFS and SJF algorithms. For the same reasons of the CPU always having a task to execute and no multithreading or parallelization techniques being employed,

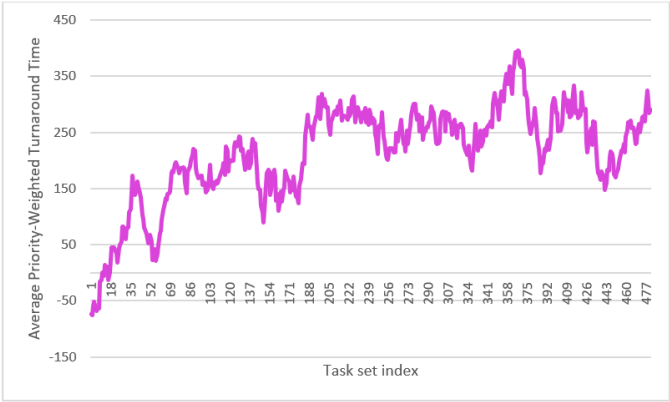

Fig. 3. Moving Average Plot of Performance of DQNS-13 as Compared to FCFS Algorithm

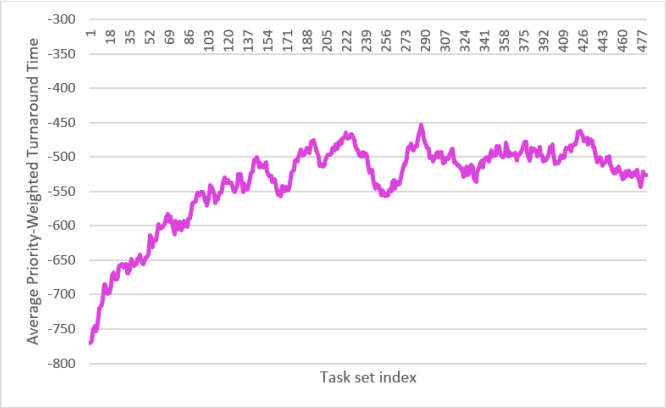

Fig. 4. Moving Average Plot of Performance of DQNS-13 as Compared to SJF Algorithm

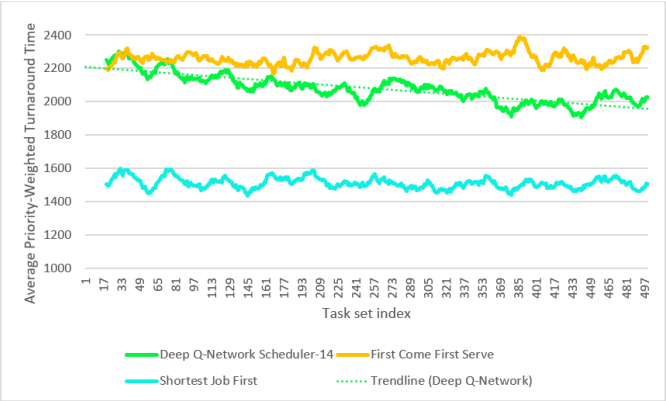

Fig. 5. Moving Average Plot of Performance of Different Algorithms on Bundle 7

we can see no major difference in completion time of the algorithms.

Fig. 5 shows the TT for each algorithm performed on each file in bundle 7. The trend line shows an increase in the performance of the DQNS over the course of training, corresponding to a decrease in TT as x approaches 500.

Fig. 6 shows a moving average of the relative performance of the DQNS-14 algorithm over time as compared to the FCFS algorithm. The graph shows the difference between the FCFS and the DQNS algorithm for each file in the data set. An obvious increase in performance is visible. The numbers begin in the negative region and increase over the task set to show a maximum increase of just over 400 in TT as compared to the FCFS algorithm. The numbers end in the positive region showing that the DQNS algorithm outperforms FCFS.

Fig. 7 shows a moving average of the relative performance of the DQNS-14 algorithm over time as compared to the SJF algorithm. This plot shows the difference between the SJF algorithm and the DQNS algorithm for each file in the data set. Again, there is an obvious increase in the performance of the DQNS algorithm. When compared to the SJF algorithm

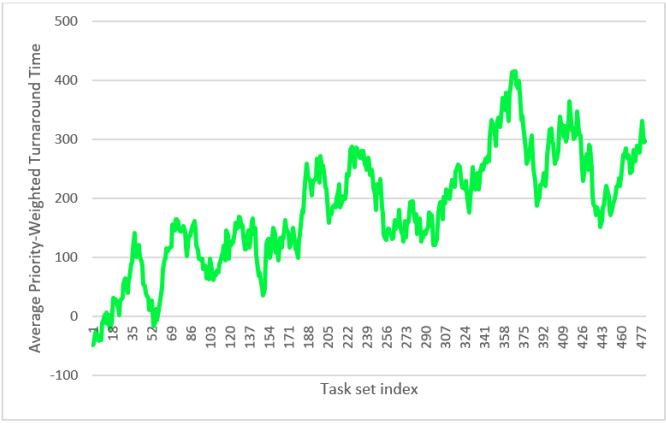

Fig. 6. Moving Average Plot of Performance of DQNS-14 as Compared to FCFS Algorithm

however, the difference in TT values is negative. This shows that the DQNS algorithm performs worse than the SJF algorithm and does not achieve equal results over time. Even at its peak, the DQNS algorithm still performs just under 450 TT worse than the SJF.

TABLE III
PERFORMANCE OF INDIVIDUAL ALGORITHMS ON BUNDLE 7 IN TEST 14

Metric	All 500 Files in Bundle			First 100 files in Bundle 7		
	DQNS	*FCFS*	*SJF*	*DQNS*	*FCFS*	*SJF*
TT	2081.3	2226.7	1511.6	2003.94	2270.61	1508.89
WT	7545.12	8240.9	5508.8	7208.45	8290.09	5506.05
CT	1496.47	1493.6	1495	1491	1492.65	1490.08

TABLE IV
PERFORMANCE COMPARISON OF DQNS TO FCFS AND SJF IN TEST 14

Metric	All 500 Files in Bundle		First 100 files in Bundle 7	
	FCFS-DQNS	*SJF-DQNS*	*FCFS-DQNS*	*SJF-DQNS*
TT	181.44	-569.68	266.67	-495.05
WT	695.77	-2036.4	1081.63	-1702.4
CT	-2.86	-1.48	1.65	-0.92

C. Comparison of Test 13 and 14

Fig. 8 shows a comparison between the results from test 13 and test 14. The plot line labelled Test 13 represents the DQNS in test 13 as compared to the FCFS algorithm and is the same plot line as displayed in Fig. 3. The plot line labelled Test 14 represents the DQNS in test 14 as compared to the FCFS algorithm and is the same plot line as displayed in Fig. 6. When observing both plot lines, it makes sense that both are fairly similar considering that all parameters of the DQNS, except the reward function, are the same.

We can observe though, that the results from test 13 are generally higher from index 1 to about 320. We can conclude that the DQNS-13 performs better than the DQNS from test 14 in this section. It is interesting to note that DQNS-13 outperforms DQNS-14 in the early stages, as it does not consider the priority of the tasks, which are very important

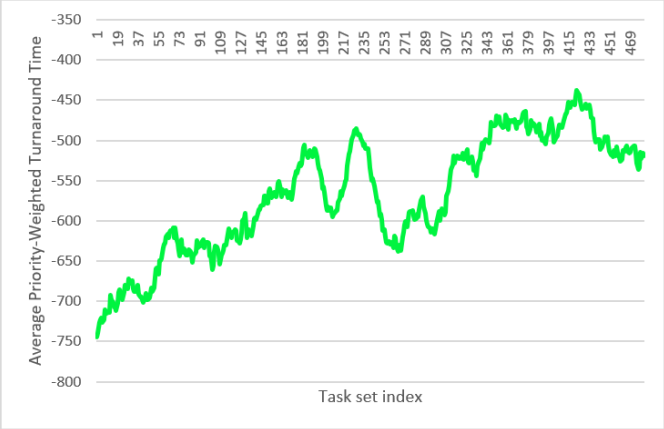

Fig. 7. Moving Average Plot of Performance of DQNS-14 as Compared to SJF Algorithm

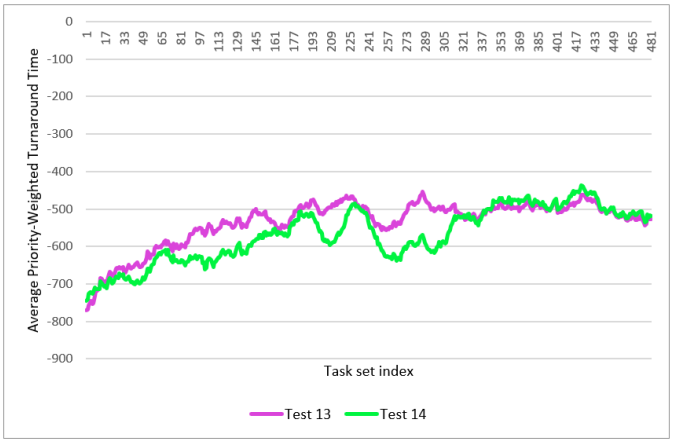

Fig. 9. Moving Average Plot of Performance of Test 13 and Test 14 Results as Compared to SJF Algorithm

in the calculation of performance. After about index 320, or around 64,000 steps, the DQNS-14 algorithm starts to outperform the DQNS-13 algorithm by a small amount, shown by the higher test 14 values on the graph. It is also interesting to note the switch in performance by the two algorithms. Perhaps this shows that DQNS-14 is a better algorithm but takes longer to train.

Fig. 9 shows another comparison between the results from test 13 and 14. The plot line labelled Test 13 represents DQNS-13 as compared to the SJF algorithm and is the same plot line displayed in Fig. 4. The plot line labelled Test 14 represents DQNS-14 as compared to the SJF algorithm and is the same plot line displayed in Fig. 7. Again, we can see that the results are fairly similar which makes sense when considering that all parameters, except for the reward function, are the same.

Looking at Fig. 9 we can see that DQNS-13 outperforms DQNS-14 at indices 1 to about 300. We can again note the peculiarity of this as DQNS-13 does not consider the priority of the tasks. After about index 300, or around 60,000 steps, we can see DQNS-14 overcome DQNS-13 again. This could possibly be attributed to the fact that DQNS-14 is the better algorithm of the two but may take longer to train.

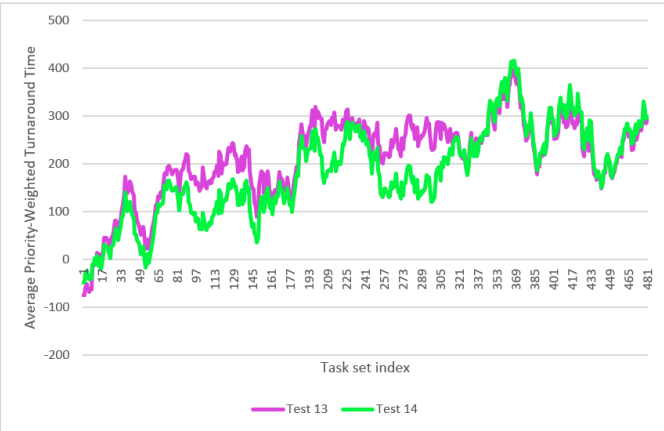

Fig. 8. Moving Average Plot of Performance of Test 13 and Test 14 Results as Compared to FCFS Algorithm

VI. Conclusion

The best results we were able to achieve were when we lowered the value of the gamma hyperparameter. This gamma hyperparameter is used in the optimization function. It is a weight that is applied to the reward values of the next state of the observations in the replay buffer. The gamma hyperparameter determines how much the next state should be considered when the DQN selects actions. If the gamma hyperparameter has a high value, close to 1, then the next states will heavily influence the back propagation, else, when it is close to 0, the next states will provide little influence on the back propagation. When using a high gamma value, we found that the DQN was unable to train to achieve any sort of improvement and was extremely unstable. This may be due to having a high gamma value, which can sometimes harm learning performance [6]. By lowering the gamma value, we could see some actual performance increases. We can understand this by saying that it doesn't matter what tasks come next, or what tasks are left behind, the most important objective for the DQNS is to pick the optimal task out of the list given at the time.

When looking at all four plot lines in Fig. 8 and Fig. 9, we can see that the increase of performance against FCFS and SJF algorithms is not stable. We can see noise in performance at around indices 30, 80, and 220 in both DQNS-13 and DQNS-14. Perhaps this could be attributed to the hyperparameters of the DQNS algorithm. The Tau hyperparameter is used for adjusting the weights and biases of the network. If a high Tau value is used, then the network will adjust very quickly to the loss results. This is good because it increases learning speed, but it is also bad because the neural network can too quickly adjust to a small specific data set and become specialized. This will reduce its performance in the next iterations when the training data changes again. Employing more complex

methods such as NROWAN-DQN [7] could reduce these unstable actions.

Another issue with the DQN is that the results seem to plateau after some time. Fig. 3 and Fig. 4 from DQNS-13 show this especially well. Fig. 6 and Fig. 7 from DQNS-14 may not have enough data to show what is really happening. There is too much jittering in the data near the end to tell if results have plateaued or are still trending upward. Perhaps the DQN is reaching a local minimum and cannot break out of it. This could be caused by the gamma value being too low, which can decrease exploration and cause the DQN to get caught in a local optimum [6]. A potential fix for this is to use noise as priors, which can sometimes enable nueral networks to break out of plateaus [8].

Overall, we believe that our results are promising but more work is required. We do acknowledge that the algorithms we compared the DQNS against were relatively basic ones, and that there are better solutions out there which will outperform the DQNS by much farther, for example a priority orientated algorithm. We did demonstrate, however, that DQN's can be used for task scheduling. We believe that with more training and perhaps employing some advanced more techniques for stabilizing learning and pushing it further it can begin outperforming some algorithms that are applied in commercial settings.

REFERENCES

[1] 'Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.0.0+cu117 documentation'. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html (accessed Apr. 19, 2023).

[2] H. Agarwal and G. Jariwala, 'Analysis of Process Scheduling Using Neural Network in Operating System', in Inventive Communication and Computational Technologies, G. Ranganathan, J. Chen, and Á. Rocha, Eds., Singapore: Springer Singapore, 2020, pp. 1003–1014. doi: 10.1007/978-981-15-0146-3_97.

[3] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, 'A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs', in 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 2017, pp. 121–128. doi: 10.1109/SBAC-PAD.2017.23.

[4] M. A. Butt and M. Akram, 'A novel fuzzy decision-making system for CPU scheduling algorithm', Neural Comput & Applic, vol. 27, no. 7, pp. 1927–1939, Oct. 2016, doi: 10.1007/s00521-015-1987-8.

[5] Sethi, Manoj. Proposed Fuzzy CPU Scheduling Algorithm (PFCS) for Real Time Operating Systems. International Journal of Information Technology. 5. 583-588, 2019

[6] S. Han, W. Zhou, J. Lu, J. Liu, and S. Lü, 'NROWAN-DQN: A stable noisy network with noise reduction and online weight adjustment for exploration', Expert Systems with Applications, vol. 203, p. 117343, Oct. 2022, doi: 10.1016/j.eswa.2022.117343.

[7] V. François-Lavet, R. Fonteneau, and D. Ernst, 'How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies'. arXiv, Jan. 20, 2016. doi: 10.48550/arXiv.1512.02011.

[8] L. Meng, M. Goodwin, A. Yazidi, and P. Engelstad, 'improving the Diversity of Bootstrapped DQN by Replacing Priors With Noise', IEEE Trans. Games, pp. 1–10, 2022, doi: 10.1109/TG.2022.3185330.