A Deep Reinforcement Learning approach for an adaptive Job Shop Scheduling

A Thesis Presented
by

Sahil Belsare

to

The Department of Mechanical and Industrial Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Industrial Engineering

Northeastern University
Boston, Massachusetts

December 2021

Acknowledgments

This work was done during my time as a Master student at the Northeastern University, Boston. Many people have influenced and motivated me in this process and it is nearly impossible to list all of them. However, I would like to mention a few people to whom I am particularly indebted. First of all, I would like to thank my advisor, Dr. Mohammad Dehghani for his continued support and providing me with the exciting opportunities at every stage that has nurtured, inspired, and shaped my outlook towards Industrial research. I would also like to thank all my former colleagues, for a harmonious working atmosphere and always providing critical feedback during the process. Last but not the least, I owe many thanks to my parents and friends for supporting me in countless ways throughout the years.

Contents

List of Figures

List of Acronyms

AHB Advanced High-performance Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access, bursts, split and retry operations.

ATLM Arbitrated Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires. In addition to what is provided by the TLM, it models arbitration on a bus transaction level.

TLM Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.

JSS Job Shop Scheduling.

RL Reinforcement Learning.

DQN Deep Q-network.

Abstract of the Thesis

A Deep Reinforcement Learning approach for an adaptive

Job Shop Scheduling

by

Sahil Belsare

Master of Science in Industrial Engineering

Northeastern University, December 2021

Dr. Mohammad Dehghani, Advisor

Dispatching rules fundamentally dictate the performance of a job-shop problem. Heuristic approaches to solving Job Shop Scheduling (JSS) assumes JSS as a static environment and fails to incorporate the dynamic changes taking place in real time. Such changes caused by in-coming jobs often alters the characteristics of an entire job queue, making static solution approaches unable to meet the real-life requirements. In this study, we structured JSS problem as a sequential decision-making process wherein a Reinforcement Learning (RL) agent outputs the policy of dispatching rule by adapting to the dynamic nature of incoming jobs. We modeled and tested an on-line dispatching rule framework that contains a Job shop simulator and a Deep Q-network (DQN) RL agent. A single machine environment was simulated to investigate the application of DQN for a multi-objective job shop scheduling problem. Three practical objectives including, total tardiness, total make span, and percentage of jobs completed in time were evaluated. The experimental analysis indicates that the dispatching rule produced by RL agent achieves considerable gains when benchmarked with traditional heuristic approaches. The findings from this research are expected to be the basis for further investigations into applying reinforcement learning to more complex job shop environments in the future.

Chapter 1

Introduction

Production scheduling is one of the key research areas in manufacturing systems that has been a subject of interest for a long time. The goal is that all decisions combined will result in a well performing manufacturing system, with shorter cycle times, less tardiness, and increased throughput [1]. The dynamic job shop scheduling problem (DJSSP) represents nearly the actual production environment when compared to classical JSSP. Compared to the classical JSSP which has been proved to be NPhard, the dynamic JSSP is more intractable due the dynamic and ever-changing nature of the system. [2]

The Static JSSP assumes knowledge of all the jobs for processing wherein the information of shopfloor is completely known in advance. And the schedule planned remains constant. Hence outputting a deterministic scheduling scheme without any modification during the entire working process. An actual production process, continues to have variations, such as fluctuating processing overtime, change in job priority, deviation in processing parameters estimates, etc. These disturbances distract the realized execution of a static schedule far from its expected outcome and deteriorate the production efficiency seriously [2]. Therefore, it is of remarkable importance to develop an on-line scheduling method for the DJSSP to handle uncertain events in real time.

DJSSP is modeled base on the stochastic nature of the in coming jobs and the whole idea is to adapt to the changes in the actual conditions by constantly adjusting the scheduling plan, which is called dynamic scheduling. Dynamic scheduling has been intensively researched over the past decades. And various methods have been presented to address them. The most widely used ones among which are changing Dispatching Rules and Metaheuristics solution approach [4]. The dispatching rule approach enables real time reaction to changing job queue and thus achieving the time efficiency. Meanwhile, since different rules are suitable for different scenarios, it is hard for the decision maker to select the best rule manually at a specific time point. Metaheuristic algorithms decomposes dynamic scheduling problems into a series of static sub-problems and solve them separately achieving acquire higher solution quality but may be time-consuming and infeasible for real-time scheduling. Though various classical approaches can be shown to provide optimal solutions to various scheduling problem variants, they typically do not scale with problem size, suffering from an exponential increase in computation time and lack to realize dynamic changes in the system. On this account, if one can choose the most appropriate dispatching rule at each rescheduling point, then both the timeliness and shop performance can be guaranteed, which serves as the primary motivation of this study.

1.1 The Reinforcement Learning approach

In this study, we structured JSSP as a sequential decision-making process wherein a Reinforcement Learning (RL) agent outputs the policy of dispatching rule by adapting to the dynamic nature of incoming jobs. An RL agent determines the action, i.e., which dispatching rule to choose, and subsequently selects an appropriate job from the queue to be processed. The RL algorithm is not told which decisions to take but instead must discover which decisions yield the most reward by trying them or by experience from the past trials[2]. The RL scheduling agent generates the dispatching rule policy when trained with numerous trial and error with the environment. The agent receives positive or negative feedback from the environment which is essentially derived from the Key Performance parameters after each trial. After that learning phase, agent will have obtained a purposive, reactive behavior for the respective environment. Then, during the application phase an agent can make its scheduling decisions quickly by utilizing its reactive behavior. This approach overcomes the shortcoming of previously discussed methods, that are either too myopic (dispatching rules) or time-consuming (metaheuristics). Lots of research studies utilizes reinforcement learning to learn scheduling knowledge and, afterward, make decisions according to that knowledge. Aydin et al. (2000) use reinforcement learning to train agents so the they can select the most appropriate priority rule according to the shop conditions in real-time and have demonstrated better performance than common heuristic dispatching rules. Aydin and Öztemel [10] developed an improved Q-learning based algorithm named as Q-III to train an agent to select the most appropriate dispatching rule in real time for a dynamic job-shop with new job insertions. Wang and Usher [11] used the Q-learning to train a single machine agent which selected the optimal dispatching rule among three given rules so as to minimize mean tardiness. However, a lot of research still needs to go in the applications of reinforcement learning in online scheduling. As it still lack theoretical support to quantify the states and actions, how to define the reward function and etc. It also needs to be benchmarked with current heuristic, metaheuristic and machine learning methods. The study should be able to identify shortcomings of RL approach as well.

Furthermore, a large number of unknown possible states in the JSSP model makes it more difficult to solve the MDP using the direct RL approaches than using a simulation-based RL approach (Shitole et al. 2019). Use of simulation to create multiple scenarios and optimize the process gave importance to the field of simulation-optimization. Computer simulations prove to be an impressive alternative to overcome the challenges for RL modeling to optimizing a variety of optimization problems. Simulation proves to be a great resource to:

- Build a virtual environment when training data set is impossible or infeasible to obtain from the real world.
- Facilitate an agent to interact with the environment in a digital world and gain its intelligence while saving time and money.
- Allow an RL agent to map various stochastic states, otherwise unable to predict - as is the case with most industrial optimization problems

1.2 Contribution

With the motivations above, we modeled and tested an on-line dispatching rule framework that contains a Job shop simulator and a deep Q network RL agent. A single machine environment was simulated to investigate the application of Deep Q-Learning algorithm for a multi-objective job shop scheduling problem. Three practical objectives including, ‘total tardiness’, ‘total make span’, and ‘percentage of jobs completed in time’ were evaluated. The advantage of the single-machine environment is that the process of benchmarking RL solution against heuristic approaches is much simpler. Thus, the algorithms can always be compared with the heuristic rules, and it will be known how much room is left for algorithmic improvements. Since it is difficult to connect to a real system, a simulation model of the single-machine environment is built. The learning procedure interacts with the simulation and learns the scheduling knowledge. The experimental analysis indicates that the dispatching rule produced by RL agent achieves considerable gains when benchmarked with traditional heuristic approaches.

The study aims to investigate and provide contribute to the following research questions aiming to bridge the gap between RL benchmarking problems for Manufacturing applications. A series of experimental analysis are carried out to:

- Provide extensive review of need of simulation based RL environment for industrial applications
- Demonstrate feasibility of modeling a simulation environment for training an RL agent
- Identify MDP model for a dynamic single machine job shop environment
- Analyze behavior of the proposed framework with different reward functions
- Benchmarking RL's performance with known heuristic solution approaches
- Suggesting improvements and future works for further research to be conducted based on the findings in this study

1.3 Structure of thesis

Reinforcement learning and dynamic job-shop scheduling with dispatching rules depict the two central concepts covered in this study. Accordingly, in Section 2 we discus basics of Reinforcement Learning and Markov Decision Process (MDP) which forms the underlying structure of RL environment. We then move on to put forward the necessity of integrating simulation with RL environment for training purposes. In section 3, provide an extensive literature review increased applicability of DES + RL environment for industrial applications. In section 4, we introduce functioning of online job-shop scheduling problem used in this study. We further elaborate on the job parameters and key performance index used for calculating the performance on scheduling rule. This leads us to talking about various heuristic solution approaches and their algorithms are written out along with their methods of implementation in Python, in section 5. We also discuss MDP formulation of the online job shop problem used in the study. In section 6, we thoroughly describe the experimental setup and analysis of experiments in the study are defined. The results are reported, benchmarked, and analyzed in order to determine which algorithms perform the best when compared to the exact solution provided. In section 7 discusses future works and the direction which further research can move in following the results of this thesis including improving the work by adding more complexities and expanding on the application of RL within the library. Finally, Section 7 concludes the work.

Chapter 2

An Overview of Reinforcement Learning

An emerging period for RL was in the 1990's with its foundation on Markov Decision Process (introduced in the 1950's). Later, RL grew to prominence in 2016 when it was credited with solving the game of Go using AlphaGo (Powell 2019). As seen in the Figure AA, an RL model consists of an agent, environment, its states, sets of possible actions, a scalar reward, and transition probabilities (Powell 2019).

- State - the set of predefined dimensions that convey information about the agents status within the environment
- Action - the list of available decisions that an agent can make at each time step
- Rewards - the reward an agent receives from the environment given its actions and states visited
- Transition Function - describes the movement of the agent from state to state

MDPs can be further classified as Fully observable MDPs, Partially observable MDPs, Semi-MDPs (Nian et al. 2020). These categories give a broad classification for RL models – Model-based RL and Model-Free RL. In Model-based RL we are aware of the transition probabilities and sample them from the probability distribution. Whereas in Model-free RL, we estimate the transition probabilities by simply observing the state transitions (Powell 2019). In an RL model, an agent is a decision-maker and an environment is everything that influences the agent's decision. An RL agent tries to learn a near-optimal policy by interacting with the environment with improved trial-and-error episodes. This process repeats itself trying to maximize the overall reward value (Sutton and Barto 2018). In order to solve a problem using RL, we need to formulate it as a Markov Decision Process

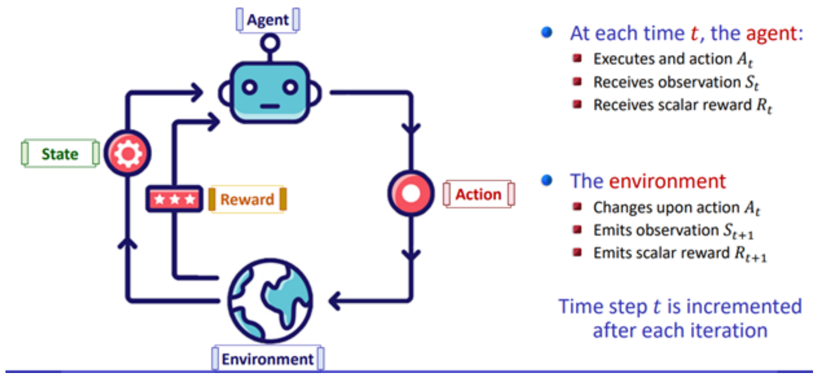

Agent
State
Reward
Action
Environment
At each time t, the agent:
Executes and action A_t
Receives observation S_t
Receives scalar reward R_t
The environment
Changes upon action A_t
Emits observation S_{t+1}
Emits scalar reward R_{t+1}
Time step t is incremented after each iteration

Figure 2.1: Structure of Reinforcement Learning

(MPD). MDPs are probabilistic mathematical frameworks developed by Richard Bellman (Sutton Barto, 2018). They are used to model interactions between a(n) agent(s) and the environment. In the context of reinforcement learning, MDPs are made up of four components: States, Actions, Transitions Functions, and Rewards.

2.1 Reinforcement Learning for Sequential Decision Making

Artificial Intelligence (AI), due to its proven benefits, is being utilized in various industries such as engineering, transportation, healthcare, energy, finance, and e-commerce (Powell 2019). AI's ability of 'self-learning algorithms' has become an attractive solution approach for industries to solve a vast range of sequential decision-making problems. Due to the diversity of applications, several communities have emerged to address the problem of making decisions over time to optimize some metrics (Powell 2019). One such influential approach in AI is Machine Learning (ML), which can be further branched into Supervised Learning, Unsupervised Learning, Semi-Supervised Learning, and Reinforcement Learning (RL). Recently, optimization theorists are focusing on RL due to its remarkable success in the operations management domain (Gosavi 2009). This is evident by the growing body of literature that is applying RL techniques to existing Operations Research (OR) problems. Among the numerous approaches to solve sequential decision-making problems, two that stand out are optimal control (which laid the foundation for stochastic optimal control), and Markov Decision Processes (MDP), which provided the analytical foundation for reinforcement learning (Powell 2019). There is a growing sense of belief that RL is a better alternative than classical Dynamic Programming (DP) to optimize MDPs and Semi-MDPs (S-MDP) (Sutton and Barto 1998).

This can be attributed to RL's prominent feature of optimizing stochastic systems by tackling the curse of dimensionality and the curse of modeling (Idrees et al. 2006). By pre-computing optimal policy, RL overcomes the issue of long online computation when compared with traditional mathematical control methods. Lately, RL has been applied in a variety of complex optimization problems in domains like supply chain, autonomous vehicles, drones, manufacturing, finance, and health science. A stochastic scheduling using dynamic RL policy outperforms various cyclic policies and meets the production constraints in a manufacturing industry. While on the other end, human-generated static policies proved less optimal when compared with RL developed dynamic response policies to minimize infectious disease outbreaks (Farhan et al. 2020). As for the gaming industry, remarkable results have been displayed to solve combinatorial optimization problem such as Rubik's cube. Deep Reinforcement Learning (DRL) has been applied successfully to scheduling problems, such as resource management or global production scheduling (Gros et al. 2020). Reinforcement learning has been widely applied to game-playing and surpassed the best human-level performance in many domains, yet there are few use-cases in industrial or commercial settings. Current RL libraries, such as OpenAI Gym, have many interesting problems, but they are not directly relevant to industrial use (Hubbs et al. 2020). A notable effort to solve classical optimization problems like VRP, TSP, Knapsack, and much more, by RL methods, has been published by Hubbs et al. (2020). OR-Gym is an open-source Python based library of reinforcement learning environments consisting of classic operations research and optimization problems. Hubbs et al. (2020) showed that RL approach outperforms the benchmark in many of the more complex environments where uncertainty plays a significant role. The paper also suggests that such platforms would ensure accessibility of RL to solve industrial optimization problems where models do not exist or are too expensive to compute online.

2.2 Need of Simulation for Reinforcement Learning

The characteristic of RL is learning through continuous interaction with the environment. For a gigantic model, training an agent offline may need a lot of steps to achieve a near optimal policy. Thus making it infeasible for live operations. To overcome this issue, the agent may be first trained in a simulated environment to obtain general knowledge of the process. Industrial systems like supply chain, manufacturing resource allocation, and industrial robotics are good examples of large problems with limited visibility of all possible states. This makes it extremely difficult, if not impossible, to write a program that could effectively manage every possible combination of circumstances occurring in real-life scenarios. For practical industrial optimization, the list of uncertainty is countless and hence the transition probabilities. Such complex models with large states can be augmented with RL in a simulated environment provided we know the distribution. Another hindrance of an industrial optimization problem is finding a readily available training data set. Sometimes a publicly available dataset cannot exactly recreate the training scenarios of the desired model. Furthermore, obtaining real world data can be inconvenient considering the time spent to acquire such a dataset by actually performing an experiment. In many industrial optimization problems, in the field of robotics, drones, manufacturing or healthcare obtaining a real-world data can be expensive considering the cost of operation just for the training purposes. In response to these challenges, researchers have harnessed the power of simulation tools to generate limitless training data set. These tools have become essential in the development of algorithms, particularly in the fields of Robotics and Deep Reinforcement Learning. Simulation enables rapid prototyping by easily customizing new tasks or scenarios. It also provides a low-risk benefit in training costly application like robotics or a million dollar industrial setup. There are primarily three types of system simulation methods: Discrete-Event Simulation (DES), Agent Based Modeling (ABM), and System Dynamics (SD). DES is a powerful tool to perform 'What If analysis' of a system, an essential characteristic to optimize sequential decision-making processes (Arulkumaran et al. 2017). Limited research is available regarding the combination of RL with DES. However, there exists a growing body of literature that is applying RL techniques to existing OR problems. Simulating an optimization problem on a Discrete Event System with different control parameter settings to see how the states and the objective unfolds is like interacting with the environment using different policies to optimize the return. Thus, the goal of learning the best settings of these control variables (best policies) to optimize the efficiency (objective) of the operation by simulating it fits into the Reinforcement Learning framework over an MDP (Shitole et al. 2019).

Chapter 3

A review Related Work

3.1 Reinforcement Learning for Manufacturing Applications

For industrial optimization, it can be noted that majority of the research has been published on applications of RL with DES in the domain of manufacturing, followed by the supply-chain industry. This popularity arises since both industries are highly dependent on the policy in-effect. Thus, it makes sense that optimization researchers tend to apply RL in these domains. The oldest paper in this survey dates to 2002. Creighton and Nahavandi (2002) have studied the behavior of an RL agent interacting with the DES model for training and developing a robust policy. The DES model consists of a stochastic serial line production facility with breakdowns that could be repaired online or offline. The interaction between a MATLAB RL agent and simulation software was facilitated by 'Quest', a visual basic server. The whole architecture was collectively termed as 'Reinforcement Agent Simulation Environment' (RASE). RASE was successful in training an agent to achieve an average lower cost of action and effectively identify an optimal operating policies of real production facilities. The RASE architecture allowed interfacing between the RL agents and commercial simulation products - a popular approach of integrating RL with DES.

3.2 Reinforcement Learning for Order release application

Classic operations research 'Order Release Problem' was tested RL by Schneckenreither and Haeussler (2018). The author begins with elaborating limitations of traditional order release mechanisms such as 'Parameterized Backward Infinite Loading (BIL) techniques. The prominent of which is the inability to consider the nonlinear relationship between resource utilization and flow times. For an Order Release Problem, the policies are highly dependent on the properties of incoming entities. Meaning, the queue characteristics like lead time, due dates, and other KPIs, highly influence the policy in-effect. RL's ability to learn based on the current state an agent makes it a suitable optimization technique. On the similar lines, Schneckenreither and Haeussler (2018) proposed an adaptive order release mechanism based on Deep Q Reinforcement Learning (DQRL) with a simulation environment. For every instance, an agent chooses the action that generates a reward while navigating to the next period by simulating the production system. The system performance was tested on a multi-product, two-stage hypothetical flow shop to measure cost-profit, lead times, and related measures. The proposed novel architecture was found to outperform the benchmark set by the traditional method by yielding lower total costs, less mean and standard deviation of tardiness, and a shorter shop floor throughput time (SFTT).

3.3 Reinforcement Learning for Job Shop application

On narrowing down to a specific application in manufacturing domain, the next popular problem for RL+ OR community seems to be job shop scheduling. The Job shop problem operates in a highly uncertain environment and adapting improved policies with time is crucial. Idrees et al. (2006) worked on scheduling arriving jobs to a single machine with an objective to minimize the mean tardiness. A DES model using JAVA programming was used to generate multiple scenarios of arriving jobs. The RL agent was trained to choose actions from a combination of the dispatching rules and number of workers. This paper displays a unique framework of optimizing traditional mathematical methods by effectively employing them based on the current state of model. The results obtained showed significant savings in the overall system's cost and was able to find optimal policy of hiring and dispatching rules once trained using the integrated environment. Taking a step further by including integrated process planning, Lang et al. (2020) experimented with Deep Q-Network (DQN) and DES for solving a flexible job shop problem. The authors integrated two DQN agents with a DES model wherein one agent is responsible for the selection of operation sequences, while the other allocates jobs to machines. Instead of using off-the shelf simulation software, the simulation training environment was implemented in Python library Salabim. The authors perform a thorough investigation on DQN's performance with a focus on its ability to transfer learning, a highly researched RL characteristic. A couple of notable outcomes from the paper are DQN's ability to always find a better solution than the GRASP algorithm for every problem, once trained. The other being, efficient transfer of knowledge as the prediction and evaluation of newly introduced production schedules requires less than 0.2 seconds. Several other papers have displayed positive results using DES model to train DQN RL agents. Zhang et al. (2018) showed that the algorithm interacts with the simulation and learns the batching knowledge gradually. The experimental results validate that this approach performs better than conventional decision rules for real-time batching job shops. These papers have successfully highlighted the combined use of DES and RL to solve a job-shop scheduling problems while integrating dispatching techniques.

Chapter 4

Problem Definition

4.1 Single machine Dynamic job shop problem

In a Job Shop scheduling problem, there exists n jobs J1, J2, . . . , Jn, each has o number of operations O1, O2, O3. . . On, with each operation having varying processing times P1, P2,. . . P3. These jobs need to be scheduled on m machines. The goal of scheduling is to allocate a specified number of jobs to a limited number resources to meet specific objectives that are to be optimized. Before each operation of a job is processed, some preparation needs to be done on the corresponding machine in order to make the machine ready to process that particular job. This period is usually called as setup time. This will in turn affect the total makespan of the jobs in queue. In a typical dynamic flexible job shop, the arrival times for these n jobs to the workshop are uncertain and are modeled according to a known distribution. In this study we consider a non-delay single-machine-Dynamic job shop scheduling problem. Where in all the jobs in a queue are to be processed on single machine. Jobs arrive at an inter arrival rate and are immediately added to the queue. Details on job characteristics and experimental assumptions are explained in section 4.2.

The Gantt-Chart is a convenient way of visually representing the JSSP graphically. It shows all the information needed to visualize queue of jobs waiting to be processed. For example, job 1 can be started for processing earliest at time t=1 and is due at time t = 20. Similarly, job 5 can be processed earliest at time t= 5 and is due at time t=20. Let us assume we schedule the job is First In First Out manner then the Gantt chart would look like:

We can see that Job 1 and 2 are the only jobs getting processed within their mentioned time limit. While Job 3 started within the prescribed time frame but is not finished before it's due date. Job 4,5,6, and 7 were started late and finished late. The total make span of this schedule comes

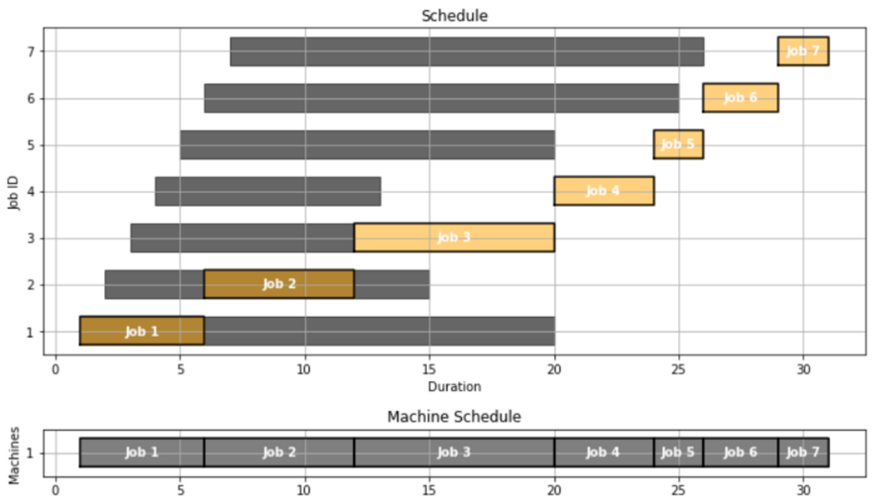


Figure 4.1: Gantt Chart

to 34 units, with 2 jobs on time and 5 jobs late.

4.2 Simulating a Dynamic Single Job shop

The dynamic job shop environment under consideration in this study was simulated in Python using SimPy package. Before going ahead with it let us understand what Discrete Event Simulation is. In simpler words, DES is a way to model events using statistical functions. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation time can directly jump to the occurrence time of the next event. In order to simulate this, we will need to decide on how to represent these different parameters of the system using probability distributions since the dynamic job shop problem is inherently stochastic. This can be done by understanding the system's behavior in following terms -

Event: an instantaneous occurrence that changes the state of the system. An event initiates a corresponding operation. In our system an event can be a time step when a job is allocated to the machine, or a time step when a job is done processing and exists the system.

Operation: A sequence of activities that changes the state of the system. In out model, an operation would be to simply hold the machine occupied for processing time of the job.

Process: method defining a sequence of events, operations, and activities in chronological order corresponding to the behavior of an entity in the real system.The assumptions made in our implementation include:

- No order constraints exist among jobs
- A machine can only process one job at a time
- Machine interruptions are not allowed
- Machines are available at the initial moment
- Machine cannot be kept idle
- No task for a job can be started until the previous task for that job is completed
- A task, once started, must run to completion.

In the model under consideration, when the simulation starts a job arrives according to the inter arrival rate having a triangular distribution. The jobs keep arriving at mentioned inter arrival rate and are added to the queue instantly. Each job has its own properties – Job ID, number in queue, Processing time, setup time and due at. The processing time, setup time and due at time follows triangular distribution as well. The jobs keep arriving and occupy the queue until the end of simulation. The performance of job shop schedule is evaluated based on the following key performance index.
Makespan – Total time taken to process a set of jobs (includes processing + setup time)
Late jobs – Number of jobs completed past their due time
Number on time – Number of jobs completed before their due time
Fraction on time – Percentage of total jobs completed on time
Maximum Lateness – Maximum time units a job was late
Total lateness – Sum of time units of all the late jobs.

Chapter 5

Modeling DJSSP as a Reinforcement Learning Problem

An RL model consists of an agent, environment, its states, sets of possible actions, a scalar reward, and transition probabilities. The paradigm of reinforcement learning handles learning in sequential decision-making problems. The elements of the reinforcement learning problem can be formalized using the Markov Decision Process (MDP) framework (Sutton and Barto,1998). An MDP is tuple of States, Actions, Reward, Transition Probability defined as:

- State - the set of predefined dimensions that convey information about the agent's status within the environment
- Action - the list of available decisions that an agent can make at each time step
- Transition Function - describes the movement of the agent from state to state
- Rewards - the reward an agent receives from the environment given its actions and states visited

States:

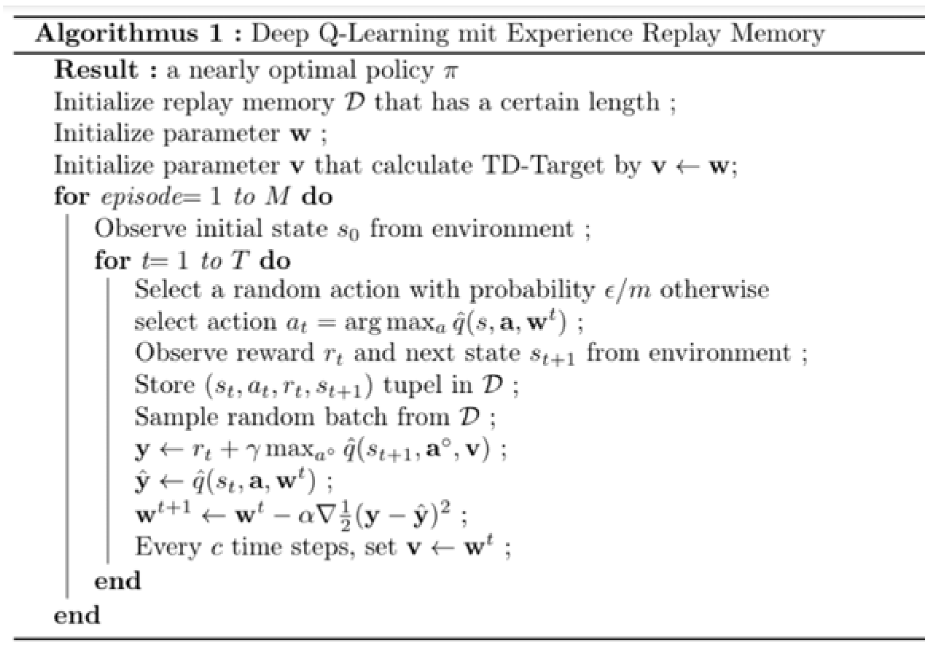
Since there are lots of ways to define the state and action, we will list the definitions for as many as possible in this section. The state of the environment is formed by the states of the machine and queue. Because the machine is always idle when making decisions, the state of the machine can be ignored here. The state of a queue can be some statistic attributes of the jobs in the queue: Release ID of jobs, Release time, Processing time for each job, Due date for each job, Remaining processing

time, Sum of remaining due dates, and Number of jobs in the queue.
Actions: The action an agent needs to take is the selecting the dispatching rule. The ultimate outcome of the agent's learning process is to output the policy of dispatching rule appropriate to the given job queue - FIFO: First In First Out, EDD: Earliest due date, and SPT: Shortest processing + Setup time.
Rewards: The reward function is feedback an agent receives from an environment in accordance to action taken. So, the reward function can be modeled by using the KPIs discussed in section 4.2, and details of modeling a reward function are explained in the section 6.

5.1 Reinforcement Learning Agents

5.1.1 Deep Q-Network (DQN)

The DQN algorithm uses a neural network instead of a traditional Q-table (Li, 2018; Mnih et al., 2015). Here, the neural network is used as a function approximator to map states in the environment to actions that the agent can take. The use of neural network comes into advantage in cases where state space in considerably large as and we do not have to store the entire environment. With this network in place, we can perform an update using a modified Bellman equation. However, with DQN, we also have to address issues which come with the addition of the neural network. Since we have to train the neural network on the agent's experience, we must store each step that the agent takes. This issue is tackled by creating a memory element to store agent's learning process called experience buffer. An experience buffer holds the state, action, reward, next state, and terminal indicator for the agent at each step. When we train the network, we sample this collection of experiences with and train the network with this batch of samples. Because DQN is still an off-policy estimation, we still act in an -greedy manner and when we choose a greedy action, we take the maximum of the estimated action-values when choosing an action to take. Within the DQN we calculate the target values for our neural network to predict against using the Bellman equation shown in Equation 4.5. These targets are used to train the neural network which is being used to map the state inputs to action outputs (and hopefully find the correct action to take in each given state). In this case, the same network is being used to both generate and evaluate the estimations of the action values. This causes the network to overestimate predictions which in turn inhibits the convergence of the algorithm. In order to combat this overestimation, Double DQN is introduced (Li, 2018).

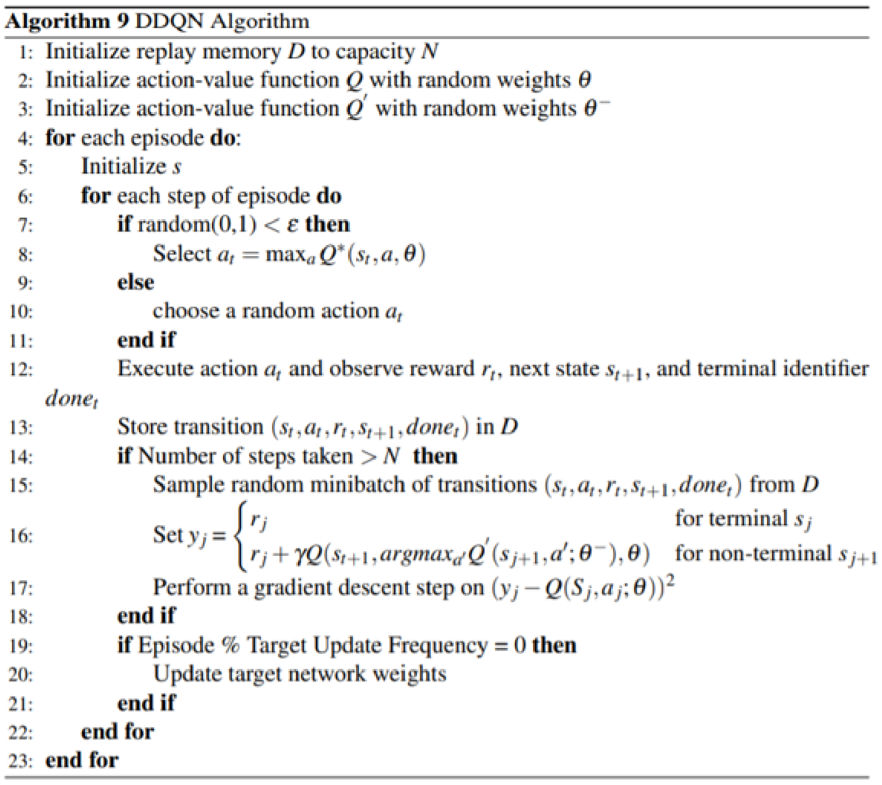


```
Algorithmus 1 : Deep Q-Learning mit Experience Replay Memory
  Result : a nearly optimal policy $\pi$
  Initialize replay memory $\mathcal{D}$ that has a certain length ;
  Initialize parameter $\mathbf{w}$ ;
  Initialize parameter $\mathbf{v}$ that calculate TD-Target by $\mathbf{v} \leftarrow \mathbf{w}$;
  for episode= 1 to M do
      Observe initial state $s_0$ from environment ;
      for t= 1 to T do
          Select a random action with probability $\epsilon/m$ otherwise
          select action $a_t = \arg\max_a \hat{q}(s, \mathbf{a}, \mathbf{w}^t)$ ;
          Observe reward $r_t$ and next state $s_{t+1}$ from environment ;
          Store $(s_t, a_t, r_t, s_{t+1})$ tupel in $\mathcal{D}$ ;
          Sample random batch from $\mathcal{D}$ ;
          $\mathbf{y} \leftarrow r_t + \gamma \max_{a^\circ} \hat{q}(s_{t+1}, \mathbf{a}^\circ, \mathbf{v})$ ;
          $\hat{\mathbf{y}} \leftarrow \hat{q}(s_t, \mathbf{a}, \mathbf{w}^t)$ ;
          $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \alpha\nabla\frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^2$ ;
          Every $c$ time steps, set $\mathbf{v} \leftarrow \mathbf{w}^t$ ;
      end
  end
```

Figure 5.1: DQN-Algorithm

5.1.2 Double DQN (DDQN)

DDQN addresses the overestimation problem in Vanilla DQN by creating a second neural network which is used to evaluate the predictions made by the original, local action value estimation network. This allows for more stable learning, accurate results, and a higher chance of convergence. The second neural network is a copy of the local one. When the first neural network makes a prediction on the best action to take based on the current state, the second target neural network is able to estimate the value of taking that action. Therefore, the action-value estimate is not simply based on the greedy best action but based on the prediction of the value of a given action. This can be seen in the update Equation 4.6 for DDQN. Here, the first network prediction represented by argmax Q(st+1,a;t) is fed into the second target network with weights . (Li, 2018). In addition, in order to update the weights of the target network and continue to improve it, the weights from the local network are copied to the target network at pre-set intervals. Below is the DDQN algorithm implemented (Mnih et al., 2015) and adapted to the Binary KP in Google Colab. Simialar to DQN, the network structure here was the same input layer, 2x32 hidden layer, and output layer setup. However, in this algorithm, two separate networks were created. They each had these same dimensions and were also kept constant throughout the experiments.



Algorithm 9 DDQN Algorithm

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: Initialize action-value function $Q'$ with random weights $\theta^-$
 4: for each episode do:
 5:     Initialize $s$
 6:     for each step of episode do
 7:         if random(0,1) $< \varepsilon$ then
 8:             Select $a_t = \max_a Q^*(s_t, a, \theta)$
 9:         else
10:             choose a random action $a_t$
11:         end if
12:         Execute action $a_t$ and observe reward $r_t$, next state $s_{t+1}$, and terminal identifier $done_t$
13:         Store transition $(s_t, a_t, r_t, s_{t+1}, done_t)$ in $D$
14:         if Number of steps taken $> N$ then
15:             Sample random minibatch of transitions $(s_t, a_t, r_t, s_{t+1}, done_t)$ from $D$
16:             Set $y_j = \begin{cases} r_j & \text{for terminal } s_j \\ r_j + \gamma Q(s_{t+1}, argmax_{a'} Q'(s_{j+1}, a'; \theta^-), \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
17:             Perform a gradient descent step on $(y_j - Q(S_j, a_j; \theta))^2$
18:         end if
19:         if Episode % Target Update Frequency = 0 then
20:             Update target network weights
21:         end if
22:     end for
23: end for
```

Figure 5.2: DDQN-Algorithm

Chapter 6

Experimental Analysis

6.1 Experimental Setup

In this section we go through the method in which we integrate simulation of DJSS modelled in SimPy, RL environment created in OpenAI package and training agent using RayLib. Below figure summarizes the same in a block diagram.

OpenAI is a Python based open source interface developed to provide structure of training RL algorithms. It is essentially a framework as an open-source library, which gives access to a standardized set of environments. It contains essential functions and code sets required to facilitate training of an RL agent. In our study, we modeled our simulation in SimPy and interfaced it with the OpenAI environment. That is, when the step function of the Gym-Environment is called, the provided action is propagated to the relevant block of the simulation model. RLlib is a library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. The below figure details the wrapping of SimPy codes with OpenAI gym and their interface with RayLib.

6.2 Experimental Analysis

In this section, we elaborate details of experimental analysis performed in order to build a stable DJJSP RL model. In each experiment we test the stability of MDP model formed, and benchmark it's performance against exact methods FIFO, EDD, SPT. Each experiment has its outcome which influenced design of the next experiment. The process eventually lead us to uncover characteristics behavior of DJJSP when solved using RL algorithms. After our initial findings on

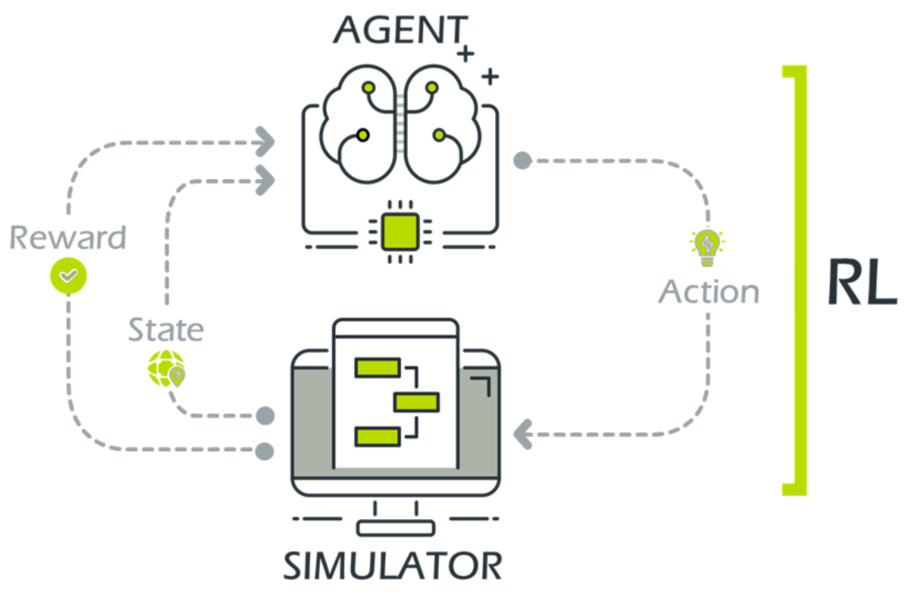


Figure 6.1: Experimental Setup

stable MDP function, we then generate 2 test cases – Job queue influenced by due date and job queue influenced by processing time. The RL agent is then tested to verify if it is able to map appropriate policy. It must be noted that the configuration of RL agent was kept constant through the series of experiment. RL agent's configuration can be found in the Appendix section. Each experiment analysis is explained in the following structure –

- Objective of Trial
- Description of Test job queue
- MDP model used
- Benchmarking and analysis of RL performance
- Learnings from the experiment

6.2.1 Pilot Testing of framework

The base model consisted of 7 jobs and 1 machine. The objective of this experiment was to conduct pilot test on MDP model of the DJSSP and benchmark RL's performance against heuristic approaches. Following simulation parameters were used to model the environment. Inter arrival rate of jobs followed Triangular (1,2,3). Each job had its processing time drawn from a Triangular (3,6,8) and setup time from triangular (0.8, 1, 3). The due at time of the job was drawn from Triangular (9,12,15). This was then added to the time instant at which the job entered the queue. Meaning, if a job enters the queue at time t = 3 and randomly samples the due at parameter 12 then the job is due at 3+12 = 15 time step. All parameters were modeled in same time unit.

State: Base MDP model – The idea behind base MDP model was to keep it simple. Meaning, simple pass in the job characteristics as state parameters. Meaning, we pass 7 set of jobs and their corresponding values at state values.

- Position in queue
- Processing time and Setup time Due time

Action: The action an agent needs to take is the selecting the dispatching rule. This will remain constant throughout the series of experiments.

- FIFO: First In First Out
- EDD: Earliest due date
- SPT: Shortest processing + Setup time

Reward: The reward is derived from the KPIs discussed in section 4.2 . It is essentially a weighted function of KPIs. A negative wight for a KPI indicates an agent should try to minimize the value of that parameters over the course of training. On the other hand, a positive weight for a KPI indicates the agent should try to maximize that reward. For this trial we keep the reward function simple to its objective.

$$Reward = (-2 \times Late\_Jobs) + (50 \times Fraction\_on\_Time)$$

After running the trial for 600 episodes a stable reward function was achieved as seen in the Figure 6.2. After initial trial and error, the agent was able to map an appropriate dispatching rule for the given job queue. That resulted in achieving high KPIs when compared to other solution approaches. As seen, by using dispatching rule obtained by RL agent maximum jobs were completed in time with lowest makespan as well. RL solution was able to produce an efficient solution, with better allocation, lower number of late jobs and maximum percentage of jobs completed in time.

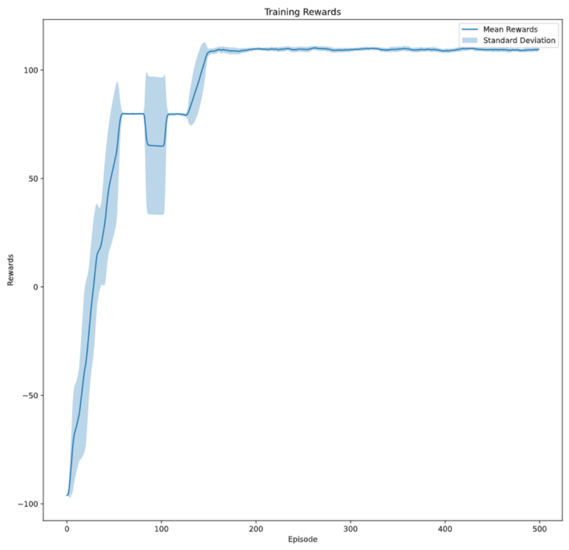


Figure 6.2: Reward Function - Trial 1

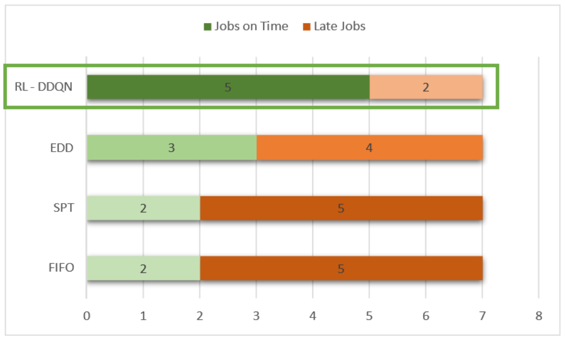


Figure 6.3: Job statistics - Trial 1

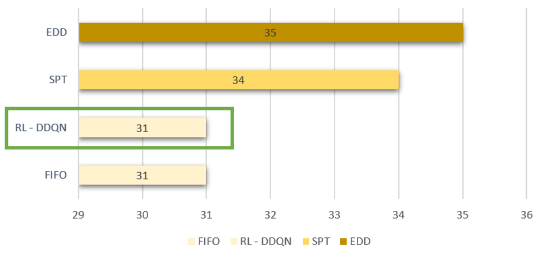


Figure 6.4: Makespan - Trial 1

6.2.2 Testing scalability of base model

Naturally, the next step was to increase the number of jobs in queue to model a realistic view of dynamic job shop scheduling problem. To do so, simulation time was increased and a set of 70 jobs in the queue were tested. The same MDP function as of base model was used to train an RL agent. When trained for 600 episodes the reward function appeared to unstable and was not converging in multiple trials. This resulted in RL agent producing fluctuating dispatching rule nearly randomly, this was a clear indication that the agent was not able to map relation between input state and actions to take. The MDP for base model was clearly insufficient to handle large scale scenarios.

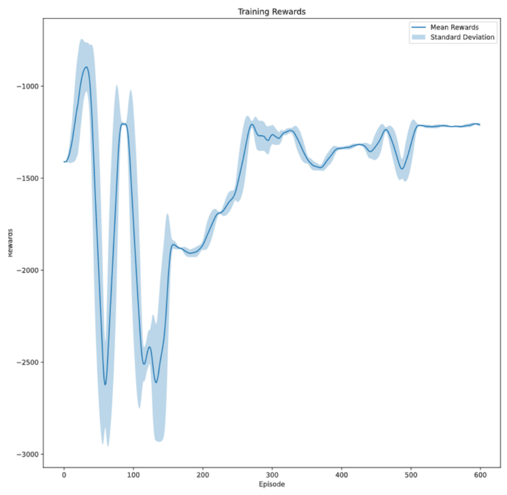


Figure 6.5: Unstable Reward Function - Trial 2

6.2.3 In search of stable and scalable MDP function

In the MDP function of base model state parameters were basically the job characteristics, as we increased the length of queue and simulation time the agent lost track of mapping relation. As a result, we decided to include queue statistics and job statistics as an additional parameter to the state function. The additional state inputs added were remaining processing time, sum of due dates of remaining job in queue, and instantaneous queue length. Remaining processing time and sum of due time remaining jobs in queue would help agent understand which factor is affecting the queue most and in turn objective function. Instantaneous queue length gave an idea of number of jobs waiting to be processed.

State:

- Position in queue
- Processing time and Setup time Due time
- Remaining processing time
- Sum of remaining due dates
- Queue length

Reward: Reward function: In addition to number of Late Jobs and percentage of jobs completed in time new KPIs were added to the reward function. Makespan and number of jobs completed in time. These two KPIs would help agent get feedback in balancing multi-objective dispatching schedule. The first two factors control in minimizing lateness and makespan and the next two factor in promoting number of jobs on time.

$$Reward = (-5 \times Late\_Jobs) + (-1 \times Makespan) + (20 \times Number\_on\_Time) + (100 \times Fraction\_on\_Time)$$

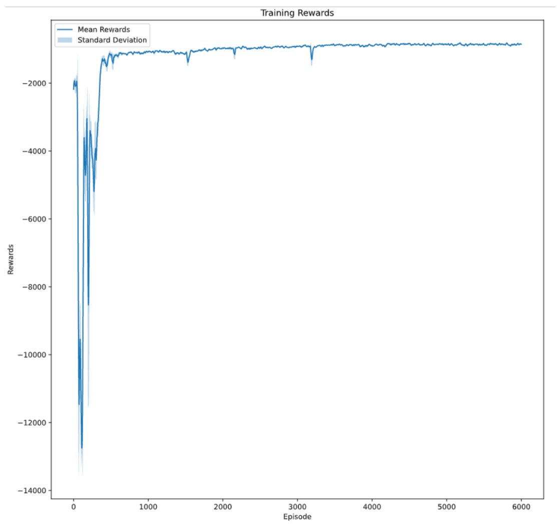


Figure 6.6: Reward Function - Trial 3

After running the trial for 6000 episodes multiple times a stable reward function was achieved as seen in the Figure 6.6. After initial trial and error, the agent was able to map an appropriate dispatching rule for the larger job queue. The result indicates that we have a stable reward function when ran for multiple iterations. The dispatching rule outputted by RL agent was however not the most efficient one. It can be seen that, simply following the EDD dispatching rule policy would yield best result. The performance obtained by following RL agent's dispatching rule was better than FIFO and SPT but failed to compete with EDD in terms of number of jobs completed in time. However, RL agent performed better than EDD when it came to minimizing total makespan. Upon closely observing the RL agent's dispatching policy it was noted that it mostly consisted batches of EDD and FIFO dispatching rule. This resulted in overall RL's performance settling between EDD and FIFO approach. That is somewhere in between. This behavior can be attributed to the generalized reward that was drafted during this trial. Remember, our reward function consisted of three parts – minimizing number of late jobs, minimizing make span, and increasing percentage of jobs completed on time. And rightfully so, the RL agent outputted the policy finding a sweet spot for balancing those rewards. This indicates that the MDP model, and in particular the states of our MDP model are well drafted to help the RL agent understand the nature of jobs queue. The behavior of RL agent is heavily influenced by the reward function as well. So, in the next trial we aim at drafting reward function that it specifically uses KPIs close to the objective of the trial.

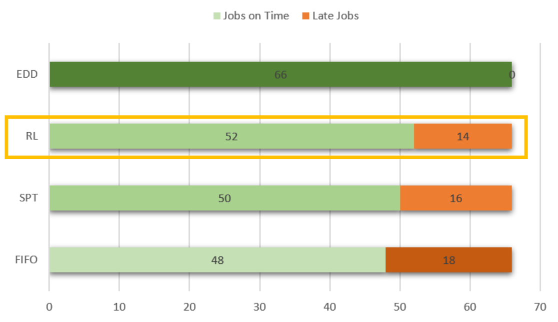


Figure 6.7: Job statistic - Trial 3

In the previous trial we observed that the best dispatching rule was produced by EDD policy. Meaning the queue characteristic was majorly dependent on it. Also, RL's policy was skewed because of the makespan parameter in the reward function. The magnitude of makespan being larger than other parameters shifts the focus of the RL agent away from the intended objective. This gave us the motivation to come up with objective oriented reward function than a generalized, all purpose reward function. Keeping all the parameters constant including job set, we modified reward function only. The change was replacing the KPI parameter 'makespan' by 'Sum of all late days'.

Reward =

$$(-5 \times Late\_Jobs) + (-1 \times Sum\_of\_Pastdue) + (20 \times Number\_on\_Time) + (100 \times Fraction\_on\_Time)$$

After running the training for 2000 episodes we found the that the change in reward function had considerably positive effect in RL's policy of dispatching rule. The policy outputted by RL agent contained smart combination for EDD and FIFO method due to which, it was able to complete all jobs on time and with make span lower than EDD approach. Ultimately providing best solution among all.

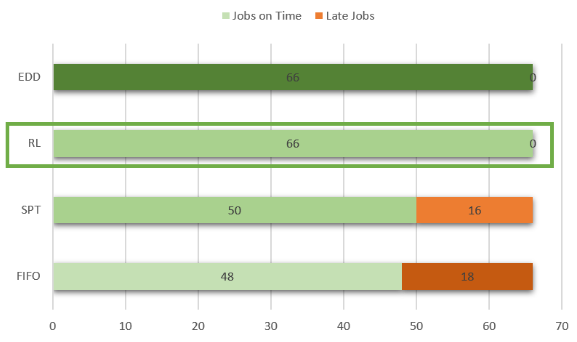


Figure 6.8: Job statistic - Trial 4

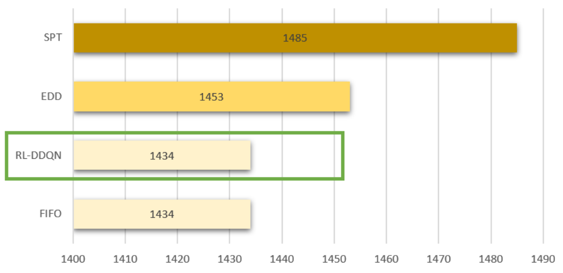


Figure 6.9: Makespan - Trial 4

6.2.4 Varying Test Conditions - Job queue influenced by due date

Previous experiment with objective oriented reward function proved efficient. The obvious next step was to test the newly drafted MDP function with even tardi job queue, one where no heuristic solution approach could find a better solution. A highly tardi job set was tested with the same MDP function and training parameters. It could be observed that RL agent was able to identify a mixed dispatching rule to provide an efficient output in all KPIs, when compared to other solution approaches.

The training process was ran for 2000 episodes, keeping all the parameters same as of previous experiments. Since due dates of all the jobs was nearly infinite the main deciding factor was makespan. Results showed that the RL agent was able to identify the job queue was dependent on processing time. The dispatching policy was a mix of FIFO and SPT, which resulted in RL performing better than other heuristic approaches.

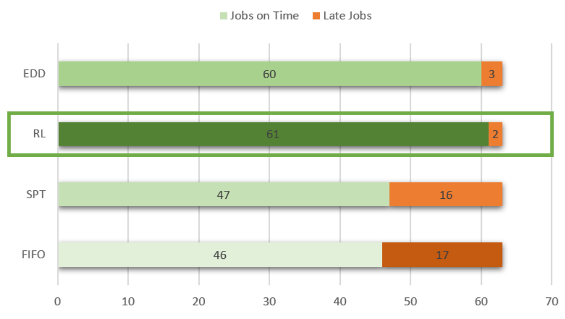


Figure 6.10: Job statistic - Trial 5

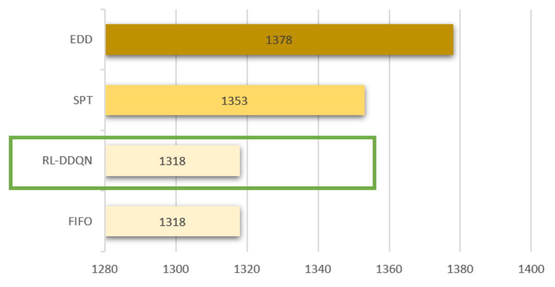


Figure 6.11: Makespan - Trial 5

6.2.5 Varying Test Conditions - Job queue influenced by processing time

Lastly, we test another scenario to model an objective oriented reward function for jobs with extremely loose due time. In this case, we keep the due time of jobs nearly infinite, and hence the KPIs are now dependent on the processing time of the job. Since the governing factor is processing time, it will have a considerable effect on the makespan of the job queue. Keeping this relation in mind we define a reward function for a job queue with such characteristic.

Reward =

$$(-5 \times Late\_Jobs) + (-100 \times Makespan) + (20 \times Number\_on\_Time) + (100 \times Fraction\_on\_Time)$$

After running the training for 2000 episodes we found the that the change in reward function had considerably positive effect in RL's policy of dispatching rule. The policy outputted by RL agent contained smart combination for EDD and FIFO method due to which, it was able to complete all jobs on time and with make span lower than EDD approach. Ultimately providing best solution among all.

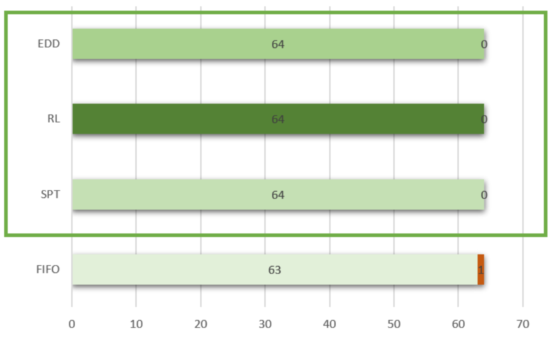


Figure 6.12: Job statistic - Trial 6

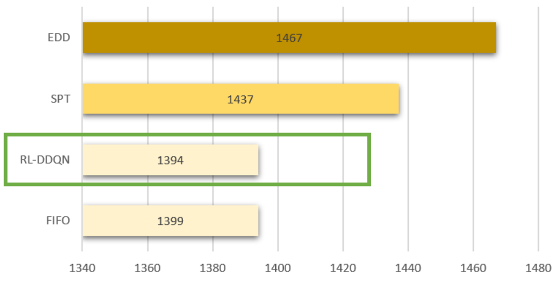


Figure 6.13: Makespan - Trial 6

To sum up the experimental analysis, we started with a naïve approach of basic DJSSP with basic MDP model only to realize that the model was unable to handle larger job queue. The subsequent experiments designed provided follow major insights-

- SimPy, OpenAI, and RayLib can be successfully integrated to train an RL agent
- There exists a method to translate DJSSP into an RL problem
- Relevant state inputs are much beneficial than drafting a generalized reward function in case of DJSSP
- A generalized reward function yields solution between two heuristic approaches

- Objective-related reward functions define better policy
- RL agent was able to identify dispatching rules for mentioned test cases

Chapter 7

Future Work

Several experiments were ran to draft a stable MDP function for various test case scenarios of a dynamic job shop scheduling problem. With the work done in this study, the goal is to begin a process of further research into the use of RL algorithms within the context of manufacturing applications. The work of identifying MDP function and benchmarking RL's performance with traditional dispatching rule provides basis of study for a vast research area. The aim of this study was to test initial feasibility, considering promising results, it only makes sense to perform additional experimentation to test different algorithms that will generate further insights into the effectiveness of RL. This study should be further carried forward by expanding work in the areas mentioned below. Developing more realistic simulation environments, testing different RL algorithms, benchmarking RL's performance with metaheuristic approaches, and develop an environment or MDP function which is able to handle all types of queue characteristics.

Chapter 8

Conclusion

The study was initiated with multiple objectives revolving around simulation, reinforcement learning, and dynamic job shop problem. It began with reviewing extensive material on integrating simulation with reinforcement learning to develop a framework for training an RL agent. We also highlighted previous work, method and structure in SimPy, OpenAI Gym, and RayLib can be integrated to achieve the same. The developed framework was then used to quantify MDP model in many ways and defined many reward functions to investigate their influence on the algorithm. After incremental experimentation it was realized that unnecessary inputs can only make the state space even larger and worsen the performance. Using relevant state space with a generalized reward function yields solution between heuristic approaches. The study showed promising results with objective oriented reward function. However this opens up a research topic of drafting an RL model which will be able to handle variety of queue characteristic. One solution in mind is to check the queue parameters and deciding which reward function to use instead of trying different reward functions and then selecting the best one. Overall, the work in this problem space is just getting started and the initial steps taken in this thesis can serves as insight into the potential challenges and solutions one can face when using RL. Along with highlighting the success and opportunities, the parallel goal of this paper is to encourage researchers to work on the challenges addressed. This needs to be accomplished by a combined effort from the simulation, reinforcement learning, and operations research community.

Bibliography

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

Appendix A

First Appendix Headline

Appendix B

Second Appendix Headline