Job Scheduling with Deep Reinforcement Learning: Environment Development and Evaluation.

Altaf Mohammed Aftab
Digital Engineering
Otto-von-Guericke University
Magdeburg, Germany
mohammed.altaf@st.ovgu.de

Devesh Singh
Data and Knowledge Engineering
Otto-von-Guericke University
Magdeburg, Germany
devesh.singh@ovgu.de

Poornima Venkatesha
Data and Knowledge Engineering
Otto-von-Guericke University
Magdeburg, Germany
poornima.venkatesha@st.ovgu.de

Syed Md Ismail
Data and Knowledge Engineering
Otto-von-Guericke University
Magdeburg, Germany
syedmd1@st.ovgu.de

Suhas Shantharam
Digital Engineering
Otto-von-Guericke University
Magdeburg, Germany
suhas.shantharam@st.ovgu.de

Mohammed khaleel ur Rahman
Digital Engineering
Otto-von-Guericke University
Magdeburg, Germany
khaleel.mohammed@st.ovgu.de

***Abstract*—The task of scheduling incoming jobs into a set of known machines, so as to optimize the machine utilization and the time of execution, is a complex undertaking that has many real-world applications. As the number of jobs and machines increases, solutions based on exhaustive search and exact optimization become infeasible. As an alternative, heuristics of time-consuming meta-heuristics can be applied, but they cannot guarantee the optima nor can they learn from experience. Hence in recent times there has been an interest in applying deep reinforcement learning (DRL) as a novel approach to overcome limitations of alternatives. In this paper we study the applicability of such approach to optimise job scheduling in the production of a printed circuit board factory. We start by analysing the existing factory setup: the different stages of production through which the different variants of chip needs to pass from, the distribution of machines in each stage and a few other parameters. Based on this we design, implement and validate an environment that replicates the dynamics of the problem. Previously the factory schedule for manufacturing was based on some heuristics. They considered production deadline for each variant of the job and simultaneously tried to reduce the idle time of each machine, thus improving overall production. So, we have some production statistics from the existing approach which will act as ground truth for our work. Based on such data we can evaluate our implementation of two DRL models, a standard and memory-enhanced one, comparing against alternative approaches and finding promising results.**

***Index Terms*—Flow shop Scheduling, Machine learning, Deep Reinforcement Learning, R2D2, DQN**

I. Introduction

Job scheduling is an important aspect in any manufacturing plant wherein different jobs are assigned to different machines in different stages to reduce the overall job completion time. For example let us consider a chip manufacturing factory, which has m machines, s stages and n jobs. The need for job scheduling arises as these m machines can be configured for different variants of jobs. So we have a situation where multiple jobs may contend for processing on a machine in certain stage while machines in some other stage may be sitting idle. Also not all jobs take the same processing time in each stage. So job scheduling is needed such that this conflict between the jobs for any stage and machine is reduced while simultaneously reducing the time that any machine sits idle.
In order to solve this problem with an increasing number of machines or jobs, exhaustive enumeration and consideration of solutions is not possible. A practical alternatives is the use of heuristics and meta heuristics [1]. But such alternatives have limitations, such as difficulties for maintenance, no guarantee of the optima and their inability to improve from experience, correcting past deviations from the optima. One approach that is proposed to overcome these shortcomings is deep reinforcement learning (DRL). However, it is not clear in practical cases what is the role of environment design or model classes in helping the solution achieve the optima. Therefore in this research we seek to understand exactly these concerns by using data from a current printed board factory.
We proceed with the objective to minimize total tardiness and an overall make-span. The production environment consists of 4 sequential production stages. Each stage had several identical parallel machines. The initial three stages have 5 machines each, while the 4th stage has 2 machines. We have 30 test cases from their approach, and in each test case we have to schedule 160 jobs. Each job has a priority and is associated with some family type, and also has a different processing time in each stage. Not all jobs pass through all the stages of production, as there can be stage skipping. The jobs are associated with a deadline which is governed by the factory's business requirement and is the feature which decides the

tardiness for a job.

In this paper first we study the practical task of creating an environment which mimics the factory setup with its constraints. Following the environment design and implementation, we can validate it by measuring how well it achieves a similar makespan and tardiness as the existing heuristic approach does, when replying such approach. We also introduce DRL models for our environment where the agent will train itself and attempt to reduce the overall makespan and tardiness even further. Hence, our contributions can be listed as follows:

- We design and implement an environment which mimics a Hybrid flow shop, taking into consideration the various constraints of a chip manufacturing plant.
- We validate the environment implemented, in two variations, by experimentally mimicking/replaying over them the existing Hybrid approach.
- We offer an early evaluation of DRL models for this task, which a focus on a Traditional (DQN) vs a Memory enhanced (R2D2) models, and differences in environment characteristics.

The remainder of the paper is structured as follows: First, the motivation and core concepts pertaining to our problem statement are discussed in the Sec. II. Next, the relevant literature in Sec. II. The characteristics of the scheduling problems analysis & hybrid flow shop in consideration are presented in Sec. IV, and our environment design and implementation is documented in Sec. V. We follow presenting and analyzing our results in Sec. VI. The paper is concluded in Sec. VII.

II. Background

Lets discuss the alternatives of job scheduling Deterministic approaches(DA) and heuristic approaches their flaws and our motivation. There are several reasons which motivates the production facilities today to go for non-deterministic optimisation algorithms. Designing job scheduling for a real world complex environment involves high implementation effort and run time [2]. With a deterministic approach we should know our production rates in advance. We would need to assume a fixed system and a fixed job sequence. Deterministic approach is not flexible with abrupt changes in plan. If any machine implements a deterministic approach, it won't be able to reassess the situation unless a human intervenes. They are also not good with multi-objective optimisation. The biggest drawback with DA is its exponential time complexity. They employ a brute force approach to find the best possible solution, and since scheduling is an NP hard problem, this way of solving it does not scale well [3].

To avoid the above mentioned problems, the PCB factory employs a hybrid heuristic approach to achieve job scheduling. In a given scenario they take decisions based on what worked best for them in the past. It's not necessary that their past experience was optimal [4]. Heuristics give them the advantage of tackling multi-objective optimisation. They also have a better time complexity than deterministic approach.

Year	Models	Papers	Authors
1997	LSTM	Long short-term memory	Sepp Hochreiter (TU Munich), Jürgen Schmidhuber (IDSIA Switzerland)
2015	Deep Q-learning	Human-level Control Through Deep Reinforcement Learning	Mnih et al.
2017	Rainbow	Rainbow: Combining improvements in Deep Reinforcement Learning	Hessel et al.
2018	Ape-X	Distributed Prioritized Experience Replay	Horgan et al.
2019	R2D2	Recurrent Experience Replay In Distributed Reinforcement Learning	Kapturowski et al.

TABLE I
DISTRIBUTED & MEMORY ENHANCED DRL MODELS

In this paper, we focus on presenting a better approach for flow shop job scheduling using DRL.

Reinforcement learning (RL) is a trial and error process where an agent takes a number of actions in a given environment, for which it may or may not receive a reward. Each state depends only on the previous states. The goal is to maximize the reward by finding an optimal path. The problem with RL is that we need to exhaustively compute Q values, which is the return an agent would get starting from a state and performing an action following an optimal path, for all state-action pairs. This is expensive and hence a function approximator like a neural network can be used to approximate Q values for all actions in that particular state of the environment. This led to the development of Deep Reinforcement Learning, which was first implemented as DQN [5].

Several advances were made in deep reinforcement learning in the recent years. Table I shows a list of distributed and memory enhanced models proposed over time. Reinforcement learning have previously been successful in cases where useful features are handcrafted or in scenarios where low dimensional, fully observable state spaces are available. Mnih et al. [5] first developed a novel artificial agent called Deep Q-network (DQN) which was combination of neural network and reinforcement learning. This agent learnt policies efficiently from high dimensional data compared to traditional approach. The two key ideas presented to overcome the instabilities of using a function approximator - 'a neural network' are: 1) experience replay which stores the agents experience sequentially in a replay buffer. Mini-batches of experiences are sampled randomly to overcome the correlation among experiences that are stored sequentially and is used to train the network. 2) Presence of two separate networks- a main network and a target network. action-values are iteratively updated by main network towards the target. The target network is frozen for several timesteps and the values of main network are copied to the target network.

DQN can be overoptimistic due to the presence of the

max/greedy strategy in the target value computation. Such overestimation of Q-values are far more common and severe in practice. To get rid of overestimation Van Hasselt et al. [6] present Double Q-learning in which the overestimation of the Q value is reduced by modifying the target value computation such that there are two Q functions; one for action selection and the other one for Q value computation.

In DQN, experiences are drawn at random regardless of their significance from the replay buffer. Schaul et al. [7] propose an approach to sample important experiences based on the priority and learn more efficiently.

Wang et al. [8] present a new architecture for model-free Reinforcement learning. The last layer of the neural network was modified to have two separate estimators, one for state value function and another one for action advantage function. This enabled better policy evaluation in the presence of large number of actions with similar values.

Hausknecht et al. [9] addressed the shortcomings of DQN and its extensions of having limited memory. The impact of adding recurrences was examined by replacing first layer of neural network with LSTMs. The resulting Deep Recurrent Neural Network could just see a single frame at each time-step but was able to effectively integrate past information. This agent performed well not only on standard Atari games, but also on partially observable environment like flickering game screens.

Hessel et al. [10] exploited all the improvements made to the DQN algorithm and demonstrated that the improvements can be potentially integrated into a single learning algorithm. Six DQN extensions (Double Q-learning, Prioritized replay, Dueling networks, Multi-Step learning, Distributional RL and Noisy Nets) were combined and evaluated on Atari 2600. The result showed improvements in terms of data efficiency and final performance.

Horgan et al. [11] proposed a new architecture called Ape X that decouples acting from learning. The architecture consists of multiple actors and a single learner. Each actor interacts with its own environment and adds its experiences in a shared replay buffer and a single learner is trained with random samples drawn from the shared buffer. This agent performed better in terms of wall clock time.

Kapturowski et al. [12] examine memory-based RL agents and propose a new agent called Recurrent experience replay in distributed reinforcement learning (R2D2) to study interplay between distributed training, experience replay and recurrent state. They introduced two new strategies to train RNN from randomly sampled experiences: 1. stored state that additionally stores hidden state in the replay buffer to initialize the network, 2. Burn-in where a start state is produced only after the network is unrolled by using a small portion of experience.

Our work focuses on using these Deep Reinforcement improvements to solve job scheduling problem. The main problems our model need to address are: 1) Credit Assignment: ”How might a job chosen to be processed at any given point of time affect the whole job scheduling process in the future?” and 2) Memory: ”How tightly is scheduling the next job instance dependent on past scheduling after a number of jobs have already been scheduled?”. B-suite is a collection of experiments which is designed to profile agents performance on seven key aspects. B-suite profile of DQN and R2D2, is shown in : Fig.1. Research shows R2D2 to be good at learning tasks where memory poses a challenge, and DQN to be good at problems that are basic or include credit assignment challenges. This motivates us to select these agents for our study.

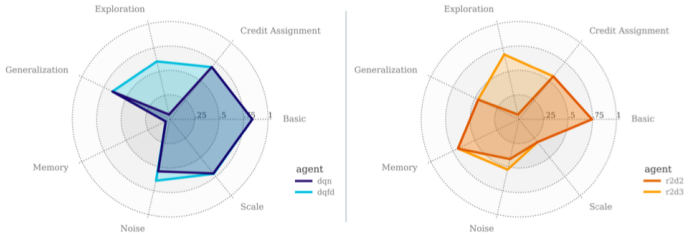

Fig. 1. Comparing aggregate performance of DQN and R2D2 on b-suite [13]

III. Related Work

In the previous sections we established in a general way existing approaches to the scheduling problem, and we introduced DRL models, motivating the selection of models for our study. In this section we review related work applying DRL to scheduling. This provides context to our research.

One representative existing solution to the flow shop job scheduling problem is provided as an adaptive scheduling framework [1]. It is a combination of both deterministic and heuristic logic making it a hybrid approach. The major advantage of this approach is in its results which shows lesser setup times, lesser makespan, lesser tardiness consequently leading to lesser penalties. The usage of DRL approach for this real world problem is to further reduce the makespan times and other evaluation parameters along with being to able to detect whether a machine has broken down and subsequently carry forward the job using a work-around approach without having to provide a logic or any try catch scenarios if and when such a situation is encountered.

Schirin Baer et al. [14] proposes Multi-Agent RL (MARL) based approach which uses many agents at a time step for job scheduling to process the input and to reduce resource allocation in a flexible manufacturing system (FMS). Each agent controls a different aspect of a product or an entirely new variant of it thereby reducing the overall design complexity and action space. It also provides ease of training sub-policies with distinct optimization goals. The complete overview of the factory setup is provided to the agents at the beginning for it to understand what resource needs to be allocated where and also to choose among the many alternatives. A graphical representation of the FMS was implemented using petri net to model plant architecture and the flow of the product. The agent uses the transition of the petri net to choose product to be assigned to the machine. Training was divided into

many stages from using single agent to learn the basic rules to fulfil a particular job to using multiple agents to learn meaningful integration and transition. The practicality of the training stages both first and second were proved by imposing it on a near enough portrayal of the plant, and the results obtained were encouraging.

Chien-liang liu et al. [15] concentrates on dynamic/real time job scheduling as opposed to the static environment. In order to address this, a Deep reinforcement learning model with Actor Critic network agent & Deep deterministic policy gradient(DDPG) was used. Actor network learn to take action under different situations with the help of a critic network which analyzes the state of the environment and returns it to the actor network. The model was evaluated on OR-library, a benchmark problem library on some instances which resulted in the equilibrium of makespan and execution time in dynamic environment.

Bao-an Han et al. [16] Propose a framework that uses Deep RL that directly learns behaviour strategies based on manufacturing state input. The process of scheduling is seen as multi-stage sequential decision making problem and the state action values are approximated using a deep neural network. Manufacturing states are input to the neural network as multi-channel images. The action space consists of several heuristic rules. By incorporating DRL improvements in the existing network (dueling double Deep Q-network, prioritized experience replay),a continuous interaction occurs between the agent and the environment and best policy is learned by the agent through trial and error process. static experiments were performed on instances from OR library and the results proved that optional solutions were obtained for small scale problem and for varied experiment on instances with random initialisation, adaptively better solutions were obtained.

In sum, shortcomings of meta-heuristics provide a motivation for applying DRL. Existing work covers multi-agent or common DRL algorithms such as actor critic. Furthermore existing work also considers indirect approaches, where the DRL agent is not in charge of scheduling itself but in charge of selecting a heuristic for a time period instead.

IV. Problem Overview

In this section we describe in detail the scheduling problem as specified for our study.

A. Overview

To schedule jobs in a hybrid flow shop [1], we need to implement an environment that mimics the factory layout and provides with possible options of actions to choose from. The action will be taken by the DRL agents. Fig. 2 represents the exact factory layout for our problem statement. In this section we present our research questions and describe in detail the scheduling problem that we study.

B. Research Questions

1) Developing environments to mimic the dynamics of the problem under study, how accurate are the metrics produced in our environment when replaying over them current scheduling strategies?
2) How well do DRL models, Traditional (DQN) vs Memory enhanced (R2D2), perform for solving a problem instance in our dataset?.
3) Do characteristics of the environment, such as strictness concerning stopping criteria, play a role in the difficulty of learning?

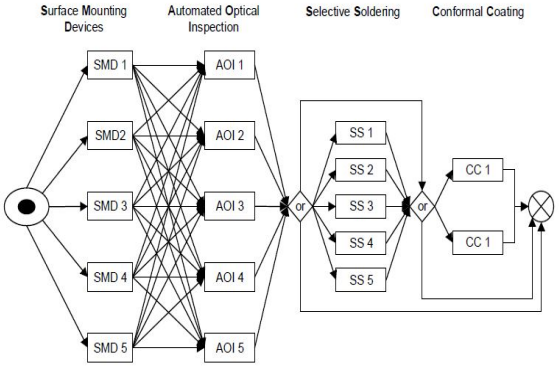

Fig. 2. The investigated production environment. [1]

C. Problem Attributes

The scheduling problem for our study consists of different problem instances. A problem instance is a collection of job orders with each of them having their own specifications. In turn the scheduling problem consists of different machines grouped into stages. The stages, as shown in are as follows:

- Stage 1: Surface Mounting Devices
- Stage 2: Automated optical inspection
- Stage 3: Selective soldering
- Stage 4: Conformal coating

The specification of jobs comprises of the following:

- *Order ID:* An unique identifier assigned to each job.
- *Priority:* The importance/urgency with which it needs to be scheduled.
- *Due Date:* The overall time by which job has to be completed.
- *Family Type:* The version of the job that is to be processed. In stages 1 and 4, the setup times of the machines are influenced by the family type of the previous job on it. In contrast in stages 2 and 3, the setup times of the machines are not influenced by the family type of the previous job on it.
- *Working time for each stage:* The time spent by each job in that stage. It is intertwined with family type of the job which again leads to machine setup times (major & minor).
  - *Major setup time:* If the family type of the previous job processed in that machine is different from the current then a major setup time is provided as follows: SMD processing stage is 65 min; CC

processing stage is 120 min; AOI processing stage : Not applicable; SS processing stage : Not applicable
- *Minor setup time;* This setup time is independent of the family type is applicable to all stages. The minor setup time is provided as follows: SMD processing stage is 20 min; AOI processing stage is 15 min; SS processing stage is 25 min; CC processing stage is 15 min.

V. Prototype Design and Implementation

A. Environment Design

In an environment the agent takes all the possible actions and collect the rewards. An environment should Assign/unassign jobs on machines for a timestep and return a new state along with reward for the previously completed jobs. The agent repeats this process till all the jobs are exhausted.

1) Environment 1 (Initial): Our initial environment mainly consists of main file, environment file and few other initialization files. Openai Gym toolkit is used to create and render the environment. The environment file requires of mainly three functions : Init(), Step() and Reset ().
Init():The init function reads the input from a workbook or excel file, initializes variables calls the reset function.
Reset (): Reset function initializes the following:

- timestep=0
- Job state: tracker for Job, checks if the job is started or ended. Also tracks the stages which the job completed. for timestep=0 all the none of the jobs have neither started nor ended.
- Machine state: tracks the machine in which job is being processed. for timestep=0 all the machines are free.
- contingency table: tracks valid & invalid jobs for a timestep. For timestep=0 all the jobs in their initial stage are valid.

Step(): It takes in an action at a *TimeStep* and assigns the job which internally updates the allocation of machine state and job state (when it started & when it will end). It also invalidates the possible actions on that machine & job for the next *TimeStep* until the job has been completed in that stage. The Algorithm 1 represents the broader picture of the function.

Function *Step(Action) :*
  $JobState, MachineState, ActionMask \leftarrow AssignJob(Action)$ Updates Job state, machine state and invalidates the action & machine for next TimeStep until it gets completed in that stage
  $reward \leftarrow getReward(TimeStep)$
  $newState \leftarrow generateNewState(TimeStep)$
  $updateQueue(removeActionTimeStep)$
  $isDone \leftarrow CheckComplete()$
  $info \leftarrow JobState, MachineState, ActionMask$
  return $intnewState, reward, isDone, info$;
end

Algorithm 1: Step Function

To reiterate our idea, we want to build an environment which imitates the baseline. Since the results of the baseline model is based on real world data it not possible to exactly imitate it. The assumption is that all the machines perform equally in terms of power, capacity etc., and without any breakdowns, thus it should be possible to get results too close to the baseline. Later a DRL agent is applied on top of it to improve the results further. So with the first model we were able to imitate the baseline based on the assumption. Due to unsupported data structure, imitation was possible but not flexible enough to integrate DRL model. This led to the creation of environment 2, which is not exact but close enough to baseline along with a DRL on top of it.

2) Environment 2 (Enhanced):: This environment utilizes the same logic as was in the initial version but presents the key parameters like reward, job state, machine state, action mask through a singular data structure that can be easily absorbed by the agent leading to a better integration with the DRL model. Because of the new data structure it also has better run time.

3) Rewards:: Agents decide on an action based on a policy, which seeks to maximize the cumulative rewards. So the agent is rewarded on successful completion of the job. If the job takes higher processing time than the Due Date then we reward it with negative squared error of time difference and if it takes less time we reward it with positive difference of time.

4) Stopping Conditions:: Stopping conditions depend on the environment configuration (which can be strict & non-strict)

- *Strict Environment:* Game ends as soon as any invalid action is taken. Stopping conditions:
  - if action is invalid.
  - if all the jobs are processed.
  - if the agent selected to wait for a job completion, but there was no job to be completed.
  - if the episode exceeds a pre-defined number of timesteps.
- *Non-Strict Environment:* Game ends only if all jobs are completed. Stopping conditions:
  - if all the jobs are processed.
  - if the episode exceeds a pre-defined number of timesteps.

5) Action Space: Number of Jobs(160) * Number of machines(17)= 2720 possible actions. We also include an additional action with the meaning of waiting for the next job to be completed (effectively making the time clock of the environment advance to the next time where a possible decision can be made). The action space dynamically changes based on *action mask* for a given *timestep*.

6) Observation space: This refers to how the environment is represented to a DRL agent.

- Jobs as rows (160 rows)
- Machine allocation (17 columns)
- Priority as one hot encoding (20 columns)
- Family type (SMD) as one hot encoding (40 columns)
- Family type (CC) as one hot encoding (3 columns)

- processing time for stages (4 columns)
- Difference of duedate & processing times (1 column)
- completed stages representation (4 column)

B. Model Implementation

Deep Q-Network (DQN) and Recurrent Replay Distributed DQN (R2D2) agents were implemented using Deepmind's open source library called Acme. It builds agents which are compatible with OpenAI Gym environments. These agents were built and tested using Google Colab.

1) Action Mask: As mentioned earlier, each problem instance has a total of 160 jobs that needs to be scheduled. And there are 17 machines in total on which a job can be scheduled. Hence the action space have $160 \times 17 = 2720 + 1$ actions. But when a job is scheduled on a machine , then all the other actions related to that machine, or that job (until it finishes) would become invalid. Thus at any given moment there will be more invalid actions present than valid actions. This necessitates a action-mask, which would allow agents to only choose from valid actions.

When a job scheduled on a machine from stage i is completed but all the machines from stage $i + 1$ are still occupied with other jobs, then in a real-world factory the job would have to wait for a machine from stage $i + 1$ to become available before it could be scheduled. This behaviour is captured using a wait action. The wait action was also introduced to the action space.

2) DQN Agent: The network passed to the DQN agent is Acme's AtariTorso network which consists three layers of 2D convolution layers, each with a ReLU activation function. After convolution layers a flatten operation is performed, and then there is a single linear layer. The details of the convolution layers inside AtariTorso network is as following:

- Layer 1: Output channels - 32, kernel size - 8x8, stride - 4x4
- Layer 2: Output channels - 64, kernel size - 4x4, stride - 2x2
- Layer 3: Output channels - 64, kernel size - 3x3, stride - 1x1

The DQN agent is a single-process agent. This is a simple Q-learning algorithm that periodically updates its policy by using prioritised action-replay. This agent has a variable ϵ indicating the probability of taking a random action. We set $\epsilon = 0.05$. The agent implements a dual DQN, where the target network is updated after every 500 steps.

3) R2D2 Agent: The recurrent neural network (RNN) given to R2D2 agent is similar to the AtariTorso nework described above but has one addition. After the same set of three convolutional layers with ReLU activation and flattening their output, the output is given to a LSTM layer with 20 hidden cells. This layer's output is then passed though a linear layer. The LSTM layers inside this network gives the R2D2 agent memory.

The R2D2 agent is also a single-process. This is a Q-learning algorithm that generates data via a (epsilon-greedy) behavior policy, inserts trajectories into a replay buffer, and periodically updates the policy (and as a result the behavior) by sampling from this buffer.

This R2D2 agent has a burn-in length of 2 steps and a trace length of 6 steps, these are features that ease the agent state initialization when learning. The R2D2 agent also stores the LSTM hidden states. Gradient clipping (to a value or 10) was used to prevent exploding gradients.

4) Optimisation: Adam optimization which is a stochastic gradient descent method that does adaptive estimation of first-order and second-order moments was used to learn the networks for both of above mentioned agents. The ADAM's learning rate was set to 1e-3.

C. Psuedo Code

Algorithm[3] describe the flow of the Jobs between agent & the environment.
Algorithm[2] describes the environment and its function.

```
Class Environment:
    Function init(Input) :
        call the reset function
        reset() :
    end
    Function Step(Action) :
        take the action from agent perform the action
        Return the new state along with reward
    end
    Function reset() :
        initialize the timestep and setup times
        initialize the action mask
        initialize the job and machine states
    end
    Function CalculateReward(timeStep) :
        positive reward for job completed before due
        date
        negative reward for job completed after due
        date
    end
    Function CheckComplete() :
        checks whether all jobs have executed
    end
    Function generateNewState(timeStep) :
        generate and return new state after action is
        completed
    end
end
```

Algorithm 2: Environment Class

VI. Evaluation

After presenting our research questions, the environment design and the models adopted, we can now present our experiment results. Generally we have 2 sets of experiments, first about environment validation, and then about the DRL models themselves learning on a single problem instance.

```
Xls ← pd.Excelfile(input.xlsx)
for sheetName in Xls.sheetNames do
    readData ← pd.readExcel(input, sheetName = sheetName);
    env ← gym.make(Stokowski, input, machine, sheetName);
    isComplete ← env.isComplete;
    while isComplete is not True do
        availMachines ← env.getAvailMachines();
        for i in availMachines do
            someAction ← OurSolver(availMachines);
            if someAction.shape[0] < 0 then
                job, machine, reward, isComplete, contingency, state ← env.step(someAction);
            end
        end
        if len(availMachines) ≤ 0 then
            env.checkComplete()
            isComplete ← env.isComplete ;
            if isComplete is not True: then
                count+ = 1  env.updateTime()
            end
        end
        else
            count+ = 1 env.updateTime()
            isComplete ← env.isComplete
        end
    end
    print("GameOver") ;
    output.xlsx ← writetoExcel(results);
end
```

Algorithm 3: Main.py

A. Environment Validation

In order to assess the faithfulness of the environment with the baseline, we compare the following metrics for the existing solution provided to us, against the metrics resulting from re-running the existing solution in our environments:

- *Makespan:* Total active time of machine for processing all jobs (Fig. VI-A).
- *Penalty:* If a job is completed before its due date, then the penalty is 0 else 1. At the end, how many jobs overshoots the due date is calculated as overall penalty (Fig.

Below are the results for all the three metrics on our 2 environments, compared to the original results provided for 30 problem instances.

As already mentioned, in reality machines break down in factory. They might not perform as per their designed capacity. So it might be unjust to compare a real result with a simulated environment. But it has to be considered since it is the best ground truth available for our research. Our initial environment reports a lower makespan than the baseline in the majority of the test instances. The enhanced environment however has a higher makespan. We call our environment enhanced not because its performing better on the above three metric but because of its customization with DRL agents later.

Both the baseline and initial environment report no penalty over the problem instances, while the enhanced model suffered from high penalty in almost all the test instances.

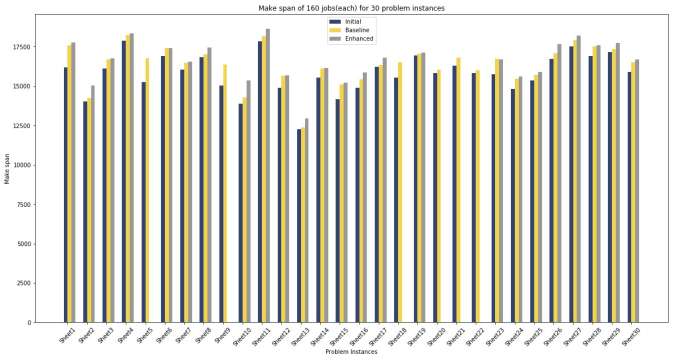

Fig. 3. Makespan of 3 approaches

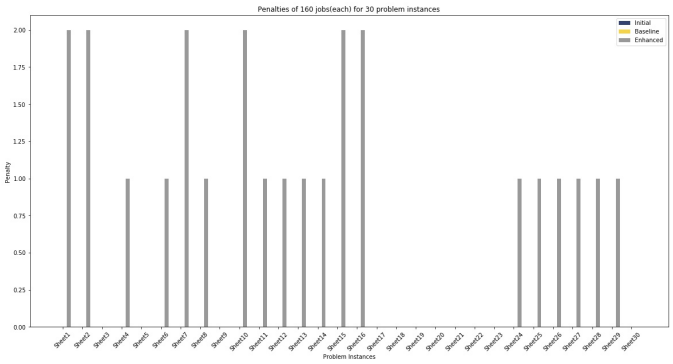

Fig. 4. Penalties of 3 approaches

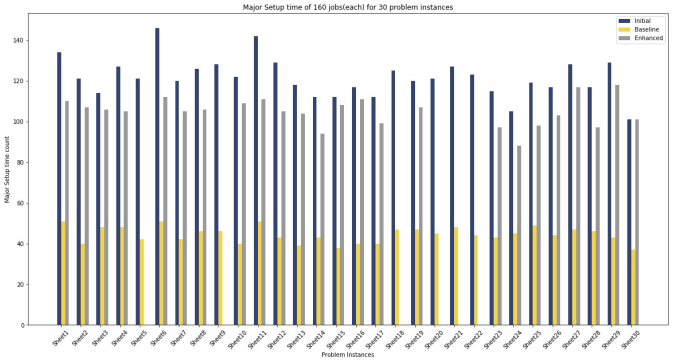

Fig. 5. Major Setup times of 3 approaches

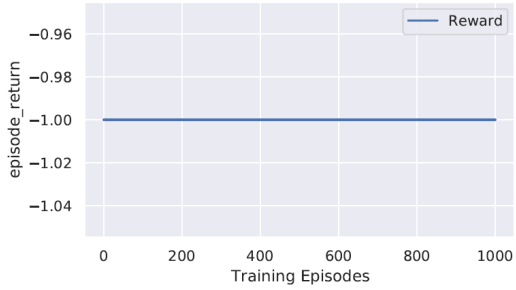

Fig. 6. Episode Returns of agent DQN for a strict environment

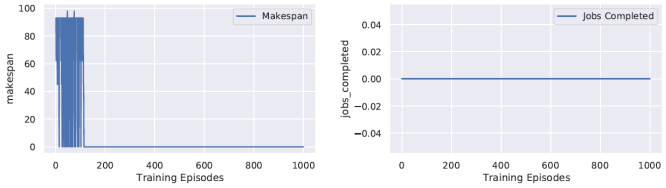

Fig. 7. Make-span and jobs completed of agent DQN for a strict environment

Compared to the baseline both the initial and enhanced environment models were shown to achieve a high major setup time.

To conclude, our experiments show the difficulties of fully replicating the original environment, and validate that our initial environment tracks closer the baseline when compared to our enhanced environment. Our results also show that a critically difficult to mimic metric is that of major setup times, which suggests us to specifically research this aspect in the future.

B. Deep Reinforcement Learning Agents

In this set of experiments we consider both the DQN and R2D2 agents on a single problem instance from those provided to us (instance #2). Following we will see the early results from integrating acme agents with the two environment settings. These two environments emerge from the fact the an agent can choose a random action with probability ϵ. A *strict environment* ends the training for an episode if an invalid action was chosen, while a *flexible environment* will continue training. These two environment settings are experimented with for a strict environment is assumed to give a more faithful simulation of real world setting, where it is impossible to taking an invalid action of scheduling a completed or in process job on a busy machine or a machine from wrong stage. While the flexible environment would allow an agent to learn even after an invalid action was taken, which is relevant mostly on training but not on testing time.

1) Strict Environment: As we could see in the Fig.7 that DQN agent was able to complete no job in any of the 1000 episodes. Hence the agent receive no rewards as seen in Fig. 6. The DQN training loss could be seen in the Fig. 8. Since the loss was fluctuating, rolling averages was

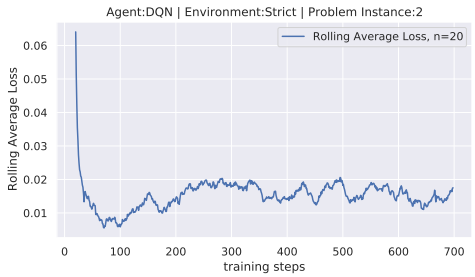

Fig. 8. Rolling average loss while training for agent DQN for a strict environment

Fig. 9. Episode Returns for agent R2D2 for a strict environment

applied over it. The DQN training loss has a downward trend to it, suggesting some extent of learning takes place over a stable environment.

As we could see in the Fig.10 that R2D2 agent was able to complete some jobs in all of its 1000 episodes of training. The R2D2 agent completed a maximum of 15 jobs in one episode. The episodes in which R2D2 agent complete a job, it receives a respective reward. This could be seen from the similar peaks in the episode returns, as seen in Fig. 9. The average reward observed across all the episodes was 1315 units.

The R2D2's training loss could be seen in the Fig. 11. With learning steps on x-axis and loss on y-axis. Again rolling averages was applied over the loss curve to dampen the fluctuations. The obtained loss curve has an unusual upward trend to it. This trends requires further study to determine if it is an error in the model or can be explained by some diversity in the episodes seen.

In table II the performance metrics for both the agents are compared. Here the best episode is defined as an episode

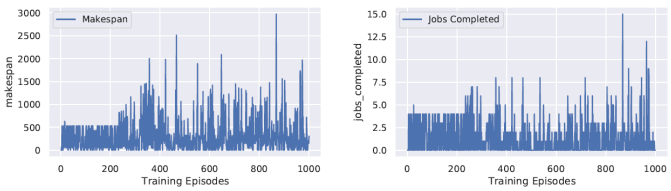

Fig. 10. Make-span and jobs completed for agent R2D2 for a strict environment

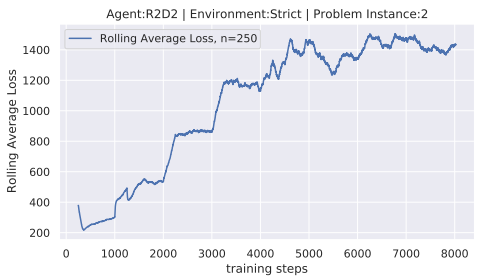

Fig. 11. Rolling average loss while training agent R2D2 for a strict environment

Performance Metric / Agents	DQN	R2D2
Max. Jobs Completed	0	15
Make-span (best episode)	-	2969 min.
Reward (best episode)	-1 unit	14999 units
Average reward	-1 unit	1315 units
Training Time (wall clock)	45 min.	1087 min.

TABLE II
PERFORMANCE METRIC OF DRL AGENTS FOR A STRICT ENVIRONMENT

in which an agent completed most jobs with least make-span. We could see that R2D2 agent completed more jobs and collected much more reward from the environment, than the DQN agent.

One interesting pattern noticed for the DQN agent was that even when in a episode no jobs were complete, the make-span was still non-zero. This points towards the fact that in these episodes the agent did start a job on some machine but could not complete it. This comes from the wait action which was introduced in the action space. The agent is erroneously choosing to wait even when machines in next stage are available.

In sum, it could be inferred that for a strict environment and the counted training steps evaluated, the training of the DQN agent failed while the R2D2 agent had moderate success in learning problem instance #2. Though the increasing loss curve for R2D2 agent requires further investigation to explain this behaviour.

2) Flexible Environment: In Fig.13 we can see that on a non-strict or flexible environment that DQN agent was able to complete all the 160 jobs in 40 out of the 100 episodes. Hence it receives reflective rewards as seen in Fig. 12. The agent collected an average reward of 86208 units across 100 episodes. The DQN training (rolling

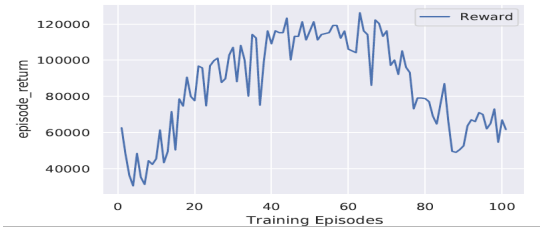

Fig. 12. Episode Returns of agent DQN for a flexible environment

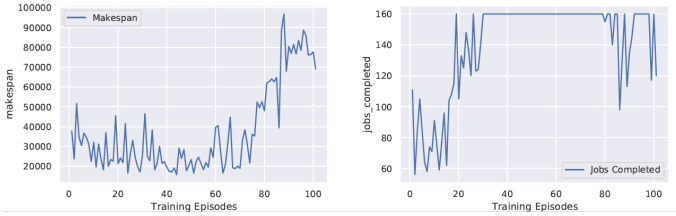

Fig. 13. Make-span and jobs completed of agent DQN for a flexible environment

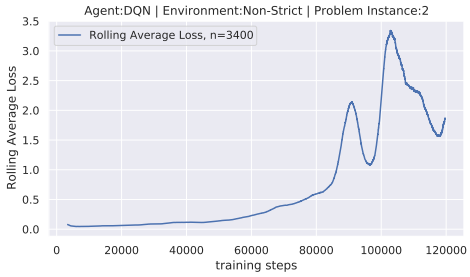

Fig. 14. Rolling average loss while training of agent DQN for a flexible environment

average) loss could be seen in the Fig. 14. The DQN training loss has a upward trend to it, similar as seen for the case of R2D2 on the strict environment.

As we could see in Fig.16 that R2D2 agent was able to complete almost all of 160 jobs in more than 90 its 100 training episodes. The make-span per episode curve had an downward trend, suggesting that with tuning it might still be improved. The rewards per training episode could be seen in Fig. 15. The average reward observed across all the episodes was 86172 units. The R2D2's training

Fig. 15. Episode Returns of agent R2D2 for a flexible environment

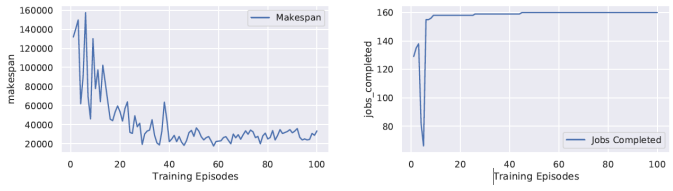

Fig. 16. Make-span and jobs completed of agent R2D2 for a flexible environment

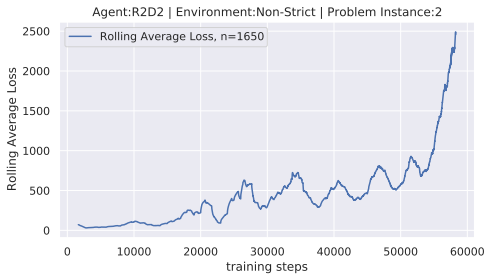

Fig. 17. Rolling average loss while training agent R2D2 for a flexible environment

Performance Metric / Agents	DQN	R2D2
Max. Jobs Completed	160	160
Make-span (best episode)	15733 min.	17562 min.
Reward (best episode)	126290 units	120232 units
Average reward	86208 unit	86172 units
Training Time (wall clock)	9696 min.	8010 min.

TABLE III
PERFORMANCE METRIC OF DRL AGENTS FOR A FLEXIBLE ENVIRONMENT

(rolling average) loss could be seen in Fig. 11. With learning steps on x-axis and loss on y-axis. The obtained loss curve has an upward trend to it, also showing a need for further research.

In table III the performance metrics for both the agents are compared. Here the best episode is defined as an episode in which an agent completed most jobs with least make-span and/or maximum reward.

With using a flexible environment, both agents learned to not use the wait action erroneously (i.e., when there is nothing to wait for). And both the DQN agent and R2D2 agent had good success in completing all the jobs of problem instance #2, coming up with competitive strategies. DQN agent had better values for performance metrics noted in Table III, though the R2D2 agent had comparable results. Interestingly in this one of DQN agent, the performance goes down after stabilising at peak while for R2D2 agent, no drop in performance was noted and the agent only got better with each training episode. Another pattern that points toward the fact that agents were successful in solving the problem instance was when the jobs completed per episode maxed out at 160 and stabilised for some training episodes, the make-span for those episodes kept on reducing. In easier terms, when the agents were able to schedule all of the 160 jobs in one episode, they kept on finding more optimal ways of schedule all jobs in next episodes and thereby decreasing make-spans.

This shows that a flexible environment for action choice is a more suitable setup of DRL agent training, than a strict environment setup. Our results are also not conclusive on whether R2D2 or DQN provide definite different performances. Our experiments suggest room for model tuning that could achieve better results. Similarly our experiments show a concerning trend in loss increasing for some agent-environment configurations, demanding further research. Other evaluation metrics over all the environment and agent settings could be seen in the appendix

VII. CONCLUSION

In this paper we built environments for a job scheduling problems, faithfully representing its dynamics. We observed that reproducing the results from current practice is not immediately achievable, due to the fact that the data provided to us did not consider theoretical machines, but instead presented machines with varying speed that change the processing times and lead to different dynamics than those we could faithfully model. To quantify that difference we provide validation experiments signalling the degree to which our environment differs to the provided problem instances in terms of its statistics.

Moving from the environment implementation to the model evaluation, from early results we could conclude that R2D2 performed much better than DQN for the strict environment, while DQN performed better than R2D2 for the strict environment, though R2D2 had comparable evaluation metrics and hence we cannot make a definitive conclusion about the models compared to each other. R2D2 had considerably more stable training than DQN. This is partly in agreement with the hypothesis that memory enhanced agents like R2D2 would perform better for a job scheduling problem. The fluctuating loss and more importantly the growing loss for R2D2, point towards the fact that agents could achieve better training results through hyper-parameter tuning and experimentation with the network architectures given to agents. More extensive research could point towards which agent performs better for the given job scheduling problem. Similarly it becomes important to establish the ideal makespan and penalties on theoretical machines so our models can be evaluated better. Finally, going beyond one to many problem instances, with some set for training and others for testing, is quite important.

More data either real time or synthetic is needed. Due to hardware limitations, the model was not fine-tuned to the desired extent. So, it will be nice to evaluate the model again after fine -tuning it further. Along with the agents considered in this paper, it is also noteworthy to consider other deep reinforcement agents or multiple agent approach, also learning from demonstration scenarios as they might also lead to better results.

The next step for this study in future would be to train the DRL agents over more problem instances and also test their generalisability in a train/test setup. We would also be interested in evaluating the models with some interpretability or introspection techniques, to understand them better and provide more targeted tuning.

ACKNOWLEDGEMENT

We would like to thank Gabriel Campero Durand for his continuous support and encouragement from the beginning till the end of this research. We are grateful to him for keeping us motivated by his kindness and helping us in all kind of difficult situations. Without him it would be impossible for us to complete our research.
We would also like to thank Nahhas Abdulrahman & VLBA team for providing us with all the necessary data, clearing all our doubts and guiding us in the right direction whenever we deviated.

REFERENCES

[1] P. Aurich, A. Nahhas, T. Reggelin, and J. Tolujew, "Simulation-based optimization for solving a hybrid flow shop scheduling problem," in *2016 Winter Simulation Conference (WSC)*, pp. 2809–2819, IEEE, 2016.

[2] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2015.

[3] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of operations research*, vol. 1, no. 2, pp. 117–129, 1976.

[4] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2015.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[8] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[9] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *arXiv preprint arXiv:1507.06527*, 2015.

[10] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[11] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, "Distributed prioritized experience replay," *arXiv preprint arXiv:1803.00933*, 2018.

[12] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney, "Recurrent experience replay in distributed reinforcement learning," in *International conference on learning representations*, 2018.

[13] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, *et al.*, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020.

[14] S. Baer, J. Bakakeu, R. Meyes, and T. Meisen, "Multi-agent reinforcement learning for job shop scheduling in flexible manufacturing systems," in *2019 Second International Conference on Artificial Intelligence for Industries (AI4I)*, pp. 22–25, IEEE, 2019.

[15] C.-L. Liu, C.-C. Chang, and C.-J. Tseng, "Actor-critic deep reinforcement learning for solving job shop scheduling problems," *IEEE Access*, vol. 8, pp. 71752–71762, 2020.

[16] B.-A. Han and J.-J. Yang, "Research on adaptive job shop scheduling problems based on dueling double dqn," *IEEE Access*, vol. 8, pp. 186474–186495, 2020.

APPENDIX

Here we list the evaluation metrics for one run of both the DQN and R2D2 agents across the strict and flexible environment setting. The performance metrics for problem instance #2 is reported here.

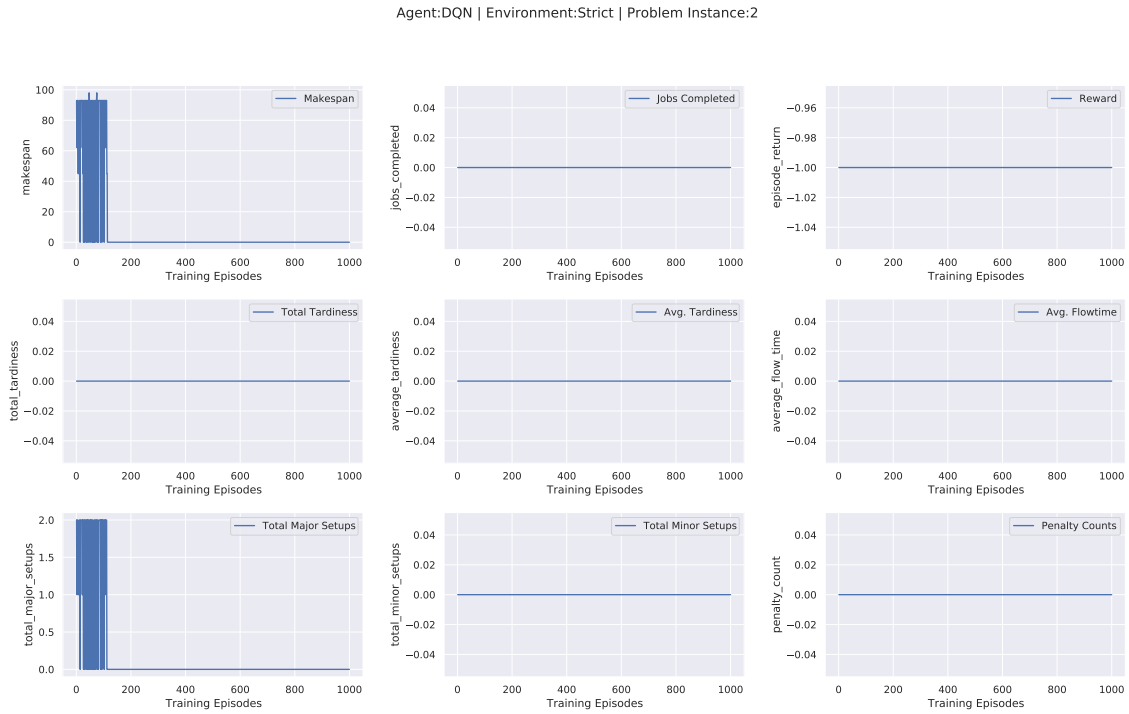

Fig. 18. Performance Metrics for the DQN agent in a strict environment.

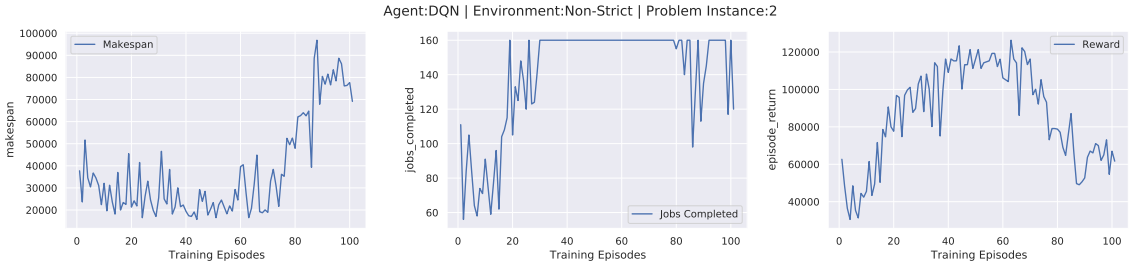

Fig. 19. Performance Metrics for the DQN agent in a flexible environment.

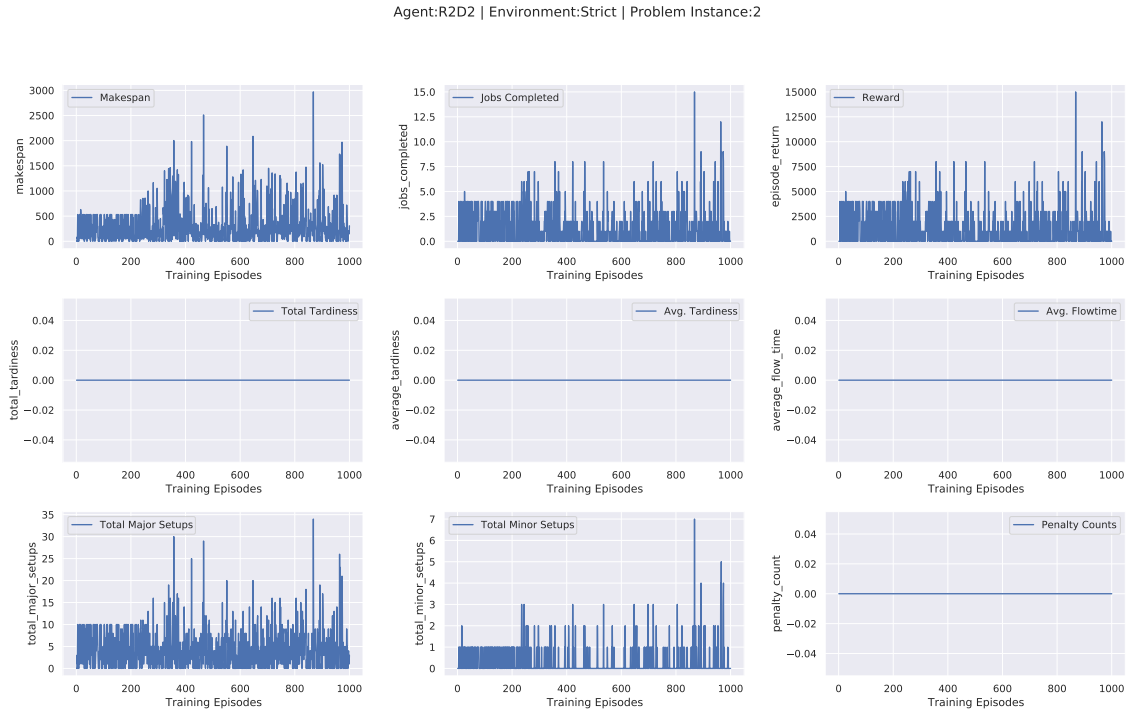

Fig. 20. Performance Metrics for the R2D2 agent in a strict environment.

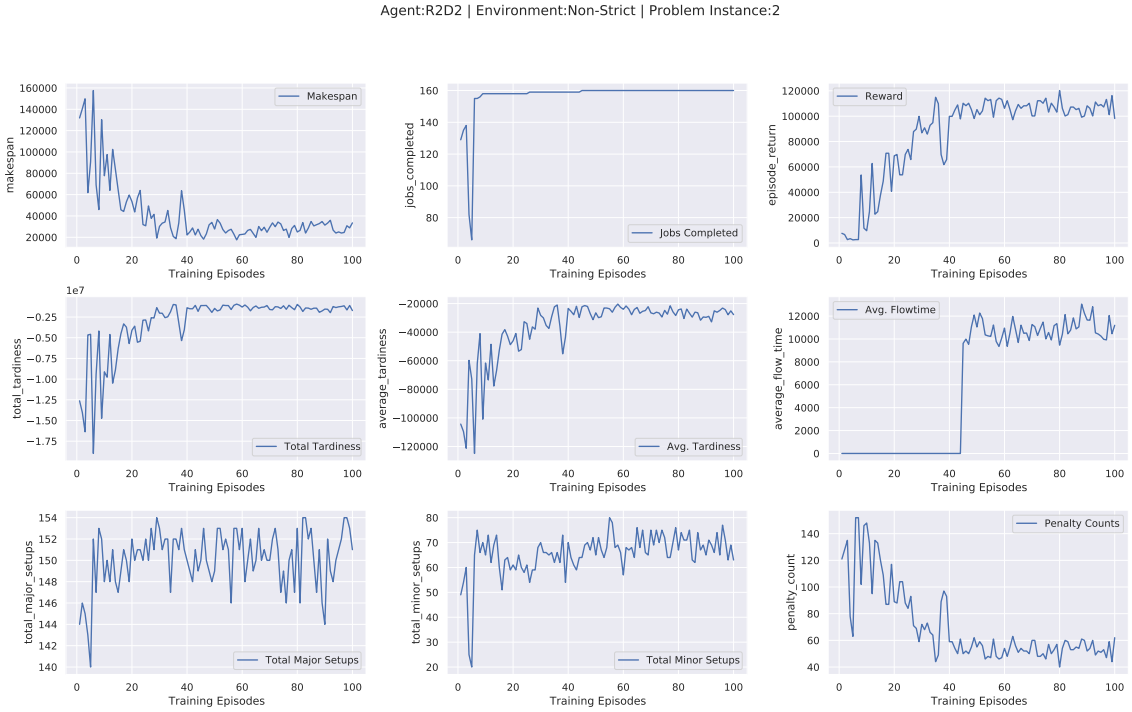

Fig. 21. Performance Metrics for the R2D2 agent in a flexible environment.