A Standardised Environment for the Application of AI in Production Scheduling Research

CALLUM DREW MCGOWAN

A Standardised Environment for the Application of AI in Production Scheduling Research

Callum McGowan

Supervisors: Dr Oliver Vogt & Dr Stefano Giani

Department of Engineering

Durham University

A Thesis presented for the degree of
Master of Science by Research

Submitted: March 11, 2024

Acknowledgements

I would like to thank my supervisors Oliver Vogt and Stefano Giani for their invaluable advice and guidance throughout this project.

Abstract

Production scheduling contains a wide variety of nondeterministic polynomial time hard problems that are differentiated by machine setups, constraints and optimisation targets. Traditionally, scheduling is run overnight for the next day and cannot adapt to a dynamic situation. Newer research methods such as deep reinforcement learning aim to address this problem, but require robust environments which can be generalised to these different setups in order to be examined properly. Currently, researchers must create their own environments when examining production scheduling problems due to the often proprietary nature of the research undertaken. This process both takes time and means that comparisons between methods are difficult. This work introduces a suitable environment which can be applied to different setups in order to reduce this wasted time and allow easier comparisons between current research methods.

Contents

Nomenclature

Reinforcement Learning

γ	discount parameter
$\mathcal{A}$	action space
$\mathcal{R}$	reward function
$\mathcal{S}$	state space
$\mathcal{T}$	state transition function
π	policy
a	action
$A(s, a)$	advantage value
E	expected discount return
e_t	experience
L_t	loss function
$Q(s, a)$	Q value associated with taking action a when in state s
r	reward
s	state
$V^{\pi}(s)$	value of being in state s using policy π
y_t	temporal difference target

Scheduling

α	machine setup
β	additional constraints
γ	optimisation target
$brkdwns$	resource breakdowns
C_j	completion time of a job
C_{max}	makespan, the final completion time of all jobs
d_j	deadline for a job
E_j	job earliness

FJc flexible job shop

Fm flow shop

Jm job shop

L_j job lateness

M_i^o matrix of an operation's viable resource routing options

o_{ji} set of operations

Om open shops

p_{ji}^s stochastic processing time for an operation

p_{ji} production time for an operation

Pm parallel machines

Qm heterogeneous machines

r_j release date for a job

s_{ji} Start time of an operation with job index j and operation index i

S_{jk} operation setup times

T_j job tardiness

Tpt_{avg}^j average throughput

Utl_i resource utilisation

WIP work in process

1. Introduction

1.1. Overview

The rise of Industry 4.0 has led to the increased interest in harnessing digital technology to increase productivity [1]. Industry 4.0 utilises cyber-physical systems such as sensors to collect vast amounts of data alongside the Internet of Things (IoT) in order to produce insights which can be acted upon to deliver superior cost efficiencies and better quality goods and services.

Production scheduling is one such application within Industry 4.0 which can help to drive increases in productivity and profitability. Production scheduling is the allocation of tasks to a set of resources in order to complete them within a factory environment. These resources are usually limited meaning that tasks are effectively competing for resource usage.

Good schedules lead to decreased manufacturing costs, reduced lateness of deliveries to customers, and increased machine utilisation [2]. Modern industrial factories face several challenges that makes the need to schedule effectively paramount. These challenges include increased product variety, shortening delivery fates to customers, reducing production lot sizes and an increase in customer orders but of smaller size. These are indicative of a transition from mass manufacturing to mass customisation in the global economy. These challenges can only be met with optimised schedules for the hundreds, if not thousands, of production orders present on a busy shop floor [3].

Production scheduling in complex, dynamic environments, such as within low volume, high variance factories (e.g. a job shop) is known to be non-deterministic polynomial-time hard ($\mathcal{NP}$-hard), meaning that as complexity increases, the time to produce optimal solutions grows to the point of being impossible to solve in polynomial time for large production instances [4]. This is true of many other similar combinatorial optimisation problems, yet scheduling problems seem to be especially hard to solve with traditional techniques. For comparison, the well known Capacitated Vehicle Routing Problem (CVRP) was solved optimally for an instance of 135 clients and 7 vehicles at the same time as a job shop scheduling problem was solved for just 10 jobs on 10 machines.

For many instances of these problems, the models provide an incomplete overview of the problem by leaving out many real world constraints. Additionally, there is a huge amount of variety within real world factories, as can be imagined when comparing semiconductor factories to car factories to job shop situations. Unavoidable changes to the environment, such as emergencies and an unknown influx of tasks form further constraints that lead to the need for a dynamic schedule due to the complexity they add, and also further reduce the ability to produce optimal schedules.

For real world situations, optimal schedules will likely not be achieved due to the reasons highlighted above. Nevertheless, improvements in production scheduling techniques can be made by advancing the approximation techniques used to produce schedules.

1.2. Motivation

Depending on product, factories can take on a wide variety of different general forms which can lead to added complexity in the scheduling task. These forms theoretically range from single machine setups to flexible job shops, which have many different types of machines and can produce a variety of different products, each requiring the completion of multiple different tasks [2].

There are then further added complexities introduced by additional constraints and the need to optimise for different objectives. These objectives and constraints can include considerations such as stochasticity, the elimination of late completion of tasks, reducing work in process (WIP), and reducing setup times and costs, among many other side constraints which can be found in real world factory setups and can change over time.

To try and provide schedules for these complicated situations, many different techniques are researched and deployed, including more traditional heuristic methods, expert systems and algorithms. Recent advances in computing power and machine learning (ML) techniques provide a new avenue to explore the problem. These techniques have shown the potential to be able to provide schedules which can adapt to dynamic environments once trained. They can also be trained for the specific environment required and therefore are of great interest for production scheduling research.

Current machine learning based production scheduling research is often limited to either less complex environments, such as parallel machine setups, or spans a limited time period of jobs rather than being continuous. Part of this limitation is due to the closed off nature of the reinforcement learning environments used, often due to their proprietary nature, which leads to research time being used to create and define the factory environment and the way the reinforcement learning agent interacts with it. The lack of standardisation also means that different methods cannot easily be compared to determine which is the most efficient, as well as meaning that benchmark optimisation algorithms and heuristics are not easily applied to the environments for further comparison. By extension, verification is more difficult which can lead to slower progress in the area.

This work introduces a factory environment which provides a robust generalisable framework which can produce intuitive representations for both human operators and reinforcement learning agents, whilst also being powerful enough to encapsulate the many different possible behaviours and constraints, or at least allow them to be added easily. The proposed environment can mimic highly complex and varied factory scenarios from the real world, with the ability to easily apply different algorithms and benchmarks to it, which will help to reduce research time spent repeating this step and allow more time to be spent refining the agents that interact with the environment.

The remaining sections of this report are organised as follows: Section 2 examines the different forms of production scheduling problems in more detail, as well as the current literature on the topic. Sections 3 and 4 presents the proposed environment and demonstrates how it can be used to replicate the different forms of production scheduling problems discussed in section 2. Section 5 then shows how different prediction models, including deep reinforcement learn-

ing, can be applied to the environments. Finally, the conclusions of this report are presented in section 6.

2. Production Scheduling

2.1. Classification of Production Scheduling Problems

Production scheduling is the family of problems that concerns assigning start times $s_{ji} > 0$ to a set of operations o_{ji} onto one of m production resources, where j is the job index and i the operation index. Each operation has an associated production time p_{ji} and all operations are grouped into n jobs. Release dates r_j and deadlines d_j define the times when job processing can be started and must be completed by, respectively [4]. The completion time of a job is then defined as C_j and the final completion time of all jobs, the makespan, is C_{max}.

Specification of production scheduling problems is done using $\alpha|\beta|\gamma$ notation, where α is the machine setup, β is the set of additional constraints, which can be empty, and γ is the optimisation target [2].

2.1.1. Machine Setups

When each job contains only a single operation, complexity is increased from a single machine problem by adding parallel identical machines (leading to Pm - parallel machines setup), which can then each process the jobs at different rates (leading to Qm - heterogeneous machines).

Shop-scheduling problems are when the n jobs contain multiple operations which can each only be processed by resources of corresponding types. Flow shop (Fm) setups are when each job has an identical operational order. Distinct sequences of operations lead to a job-shop environment (Jm). Finally, for completeness, open-shops (Om) require no operational order constraints.

When there is replication of resources into groups, then operations can be processed on any of the resources within the group (e.g. with two laser cutters in a shop, laser cutting operations can be routed through either). This leads to flexible shop setups, for example the flexible job shop (FJc). These resource groups then have similar constraints as in Pm and Qm setups. Jobs therefore require a routing choice in addition to the operation sequencing.

The hierarchy of increasing complexities and the differences that lead to them is shown in figure 1.

2.1.2. Additional Constraints

The addition or changing of existing constraints β add further complexity to scheduling problems. One of the most common constraints is the addition of a setup time. Setup times can be added between different families, or between any jobs (S_{jk}). These are additionally machine dependent when the parameter is S_{jki}.

Another important constraint is the addition of stochasticity to parameters, such as the processing time p_{ji} being replaced with p^s_{ji}. This makes the problem model more applicable to real world scheduling scenarios due to the completion of jobs rarely being deterministic. Similar parameters add a stochastic effect to the release dates (r^s_j) of the jobs, affecting when the job is first seen by the controlling agent, and breakdowns ($brkdwns^s$), which represent unexpected machine outages.

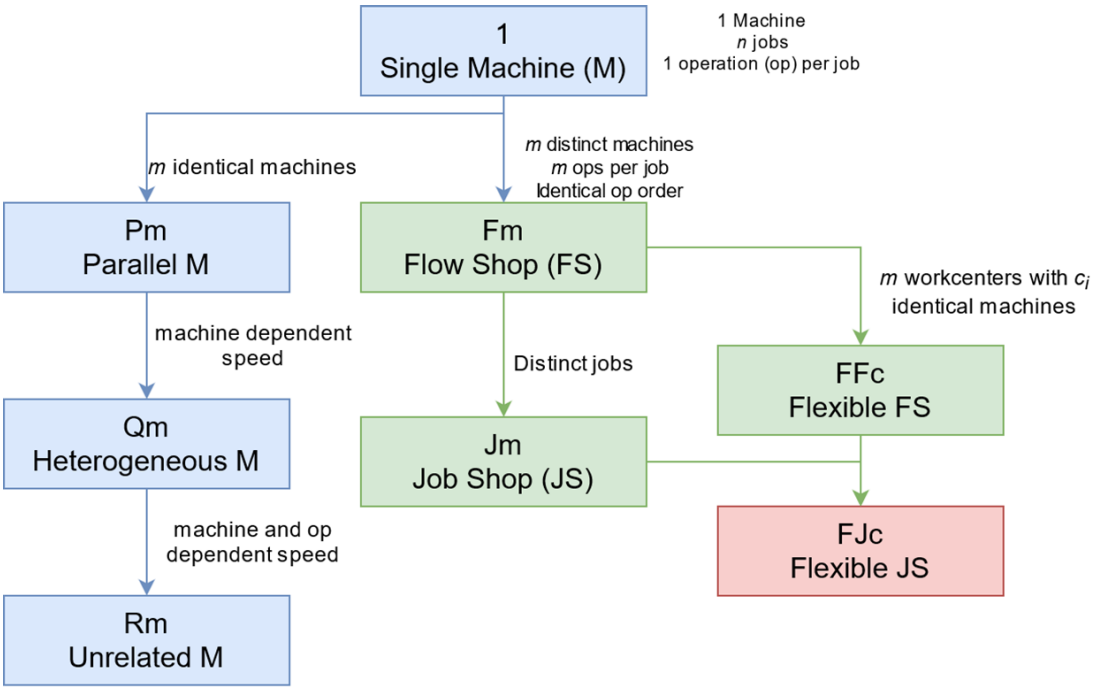

Figure 1: Hierarchy of different machine setups with increasing complexity [2]

Operations can go to different resources but also some resources can process different types of operations, for example a welder can perform different distinct types of welding which are different operations within the process and require different materials. This is denoted M_i^o and leads to a matrix of eligibility for each operation and its viable resource choices. A further extension of this is that different resources process the same operation at different speeds. For example, newer equipment is often quicker than older equipment, and a more experienced operator can often process an operation faster than someone who is less experienced.

2.1.3. Optimisation Targets

There are many different objectives in production scheduling which can be optimised. Some of the most common metrics related to scheduling problems are related to one or more of throughput, timeliness, and utilisation.

Throughput is the dynamic environment version of makespan, and measures the how many units can be produced or jobs can be completed in a given amount of time. The average job throughput, Tpt^j_{avg}, at time t is represented by

$$Tpt^j_{avg} := \frac{1}{t} \sum 1_{\{C_j \leq t\}}. \tag{1}$$

This can then be further expanded to operation throughput which can then be measured per machine or resource group [4].

Objectives related to timeliness include whether a job is completed late or early. As such, the lateness, L_j is defined using completion times and due dates as

$$L_j := C_j - d_j, \tag{2}$$

positive and negative lateness values therefore indicate a job was completed late or early respectively.

A metric describing all jobs completed late is described by tardiness, T_j,

$$T_j := max\{0, L_j\}, \tag{3}$$

and similarly jobs which finish early are described by earliness, E_j,

$$E_j := min|L_j, 0|. \tag{4}$$

Tardiness is the most obviously important of these two, as finishing a job late can lead to contractual fines, however finishing a job too early can also be detrimental due to, for example, storage restrictions [5].

A further timeliness consideration is work in process (WIP), which in a factory environment relates to jobs which have been started but have not yet finished. As such, individual job WIP is given by the equation

$$WIP_j = \begin{cases} 1 & \text{if } s_{j0} \leq t < s_{j0} + C_j \\ 0 & \text{otherwise} \end{cases}, \tag{5}$$

and total WIP is the sum of all job WIP values, multiplied by some factor scaling with the importance of the job.

Finally, the utilisation, Utl_i, of any given resource is given by

$$Utl_i = \frac{1}{t} \sum p_{ji} * 1_{\{s_{ji}+p_{ji}<t\}}, \tag{6}$$

which is the amount of time the machine was working over the total time t.

2.2. Literature review

Production scheduling problems are $\mathcal{NP}$-hard, which means that optimal solutions are impossible to find in large production instances as the number of possible solutions increases rapidly. For example, according to Fox in [6] when sequencing 10 orders, each with 5 operations, with a single machine for each different operation type, there are $10!^5$ or roughly 10^{32} possible schedules. It is obvious that a more complex situation in an actual manufacturing facility is more complex, as each of the number of orders, operations, and resources are usually substantially greater.

Traditionally, scheduling is run overnight for the next day and is too time consuming to be run again in the middle of the day. Complications can be added by disruptions, which can render these schedules obsolete and make the problem more difficult. Machines can breakdown and workers can be ill, therefore losing expected resources. Dynamic schedules which can produce new solutions quickly are required to adapt to these situations. Factory dispatchers are often accustomed to handling these disruptions, but their decisions can miss the bigger picture and therefore their impact as they are often crisis-oriented.

Thus computer-aided methods for scheduling must be run often and fast in order to deal with these fast changing situations. These requirements often lead to simplistic models which trade a decreased solution quality and lower realism of the model for a decreased solution time.

Traditionally, production scheduling problems are approached using either simulation models, analytical models, heuristics, or a combination of these methods.

Newer methods involve using artificial neural networks and deep reinforcement learning to make production schedules. These methods are difficult to train but in theory they can make changes quickly if required when trained.

This section aims to analyse the methods and current applications of the most common of the wide variety of approaches to solving production scheduling problems and the situation in which they are used.

2.2.1. Simple Heuristics

Heuristics are a common traditional method whereby a simple set of rules are used to produce schedules. Examples of simple heuristics include using a first in first out processing order for all eligible tasks for a given machine, or scheduling the next eligible task with the shortest or longest processing time [3].

The earliest due date rule introduced in [7] schedules the task with the earliest due date into the resource. This is optimal for single machine tardiness/lateness situations, as discussed in [3], and has been shown to be stronger when used online than some offline scheduling methods in [8].

Simple heuristics are advantageous as they are easy to understand and implement, which is a quality especially valued by human schedulers so that they can easily see the logic behind the decisions being made. These methods are also quick to run, due to their fast computational complexities of $\mathcal{O}(n \log n)$ where n is the number of tasks to be scheduled. This is due to them effectively being sorting problems and therefore having the same lower bound for their computational complexity. Their main disadvantage is their lack of foresight and production of sub-par results, especially when used to provide schedules for more complicated systems.

These two main factors need to be weighed up against each other when deciding whether or not to use simple heuristics. In some simple situations, such as the m identical parallel machines problem with a makespan optimisation target ($P//C_{max}$), which is still $\mathcal{NP}$-hard for two machines, the longest processing time heuristic has a worst case performance of $\frac{4}{3} - \frac{1}{3m}$ [9].

An example from the real world is the exam scheduling problem of Pearson VUE, where the goal is to process all exams and students in each of their given test centres whilst minimising the number of opening hours. The seats in each exam centre form a parallel machine setup with the jobs to be processed being represented by the different exams with varying lengths. The goal to minimise the number of hours each exam centre is open is equivalent to a minimisation of the makespan. Over the course of 2012 there were effectively 11261 different parallel machines problems accross the different centres. CPLEX 12.5, an optimisation studio produced by IBM, produced schedules which were on average 0.57% from being optimal. By comparison, the longest processing time heuristic resulted in a gap of 0.91%. However, the total CPU time to

produce the CPLEX solutions was 80 hours on a 240 core cluster, compared to less than 18 seconds on a single computer for the longest processing time solution [3].

Overall, it can be seen that these simple heuristics can produce powerful results for real, albeit less complex problems. However, different heuristics are suited more towards different optimisation targets and so in the real world, where targets often change, would need to be swapped between to be effective. These heuristics also do not take into account more than what is in front of them, which is to say that they have no consideration for other resources within a system's environment.

2.2.2. Advanced Heuristics

More elaborate dispatching heuristics also exist, such as the apparent tardiness cost dispatching rule proposed in [10]. This rule calculates an index value I_j for every job each time a machine is free, which in its more complicated form is given by the equation

$$I_j(t) = \frac{w_j}{p_j} \exp\left\{-\frac{max(d_j - p_j - t, 0)}{K_1 \bar{p}}\right\}, \tag{7}$$

where w_j is the weight of the job, which effectively acts as a priority factor by denoting the importance of job j relative to other jobs, the K values are parameters to be fixed and $\bar{p}$ is the average processing time of the remaining jobs. This rule was designed to be used for weighted tardiness problems but can be applied to other problems too.

The apparent tardiness cost rule has a computational complexity of $\mathcal{O}(n^2)$ due to how many times the index is computed. The calculations become slower than other rules due to the difficulty of computing the index coupled with the higher complexity. The rule is also harder to apply when compared to simpler heuristics in exchange for better performance.

Advanced heuristics which go beyond dispatching rules often have some problem specific knowledge about theoretical properties of scheduling problems. These properties are then exploited to produce production schedules. Other methods aim to derive better schedules starting from any previous schedules.

Many of these advanced techniques are built upon local search techniques, which requires a definition of neighbouring solutions, ways to explore them, and a way to evaluate which is better.

A strong example of an advanced heuristic is the NEH heuristic, which was first proposed for solving a flow shop with a an objective of minimising the makespan [11]. The steps to carry out the heuristic first requires the total processing time of all jobs in all machines to be calculated. These are then sorted in descending order. The first two jobs are then taken, and the order in which they minimise makespan is maintained. The next job in the descending sequence is then chosen and inserted in all possible sequences with the jobs already in the queue, and the minimum makespan order is again chosen. This process is repeated until all jobs are placed in the sequence via this insertion technique, building upon the previous best partial sequence. This has a computational complexity of $\mathcal{O}(n^3m)$. This was later improved to $\mathcal{O}(n^2m)$ with the NEHT technique [12].

NEH outperforms many other heuristics, with an average statistical deviation from the best known solutions in 120 different instances of just 3.3%. Quite often, NEH is used to seed metaheuristic techniques [3]. The faster NEHT gave performances of 2.24% in 10 large instances in an average of 0.077 seconds. There are many extensions and examinations of this technique in flow shop environments with makespan objectives [13; 14].

Further advancements are in the form of metaheuristics, which attempt to be general templates with increased performance over regular heuristics, with the downside of often being more computationally demanding. Metaheuristics commonly use several similar features. Examples of these common features include a representation of the solution, which is an arrangement of the variables involved, an initial solution or set of solutions from which to start the process, a diversification mechanism such as local search, with the goal of finding a better solution, a set of comparison criteria for judge new solutions against older ones, and a termination condition to limit the computational time used, as optimal solutions are not guaranteed due to the nature of scheduling problems. Example termination conditions include the number of iterations, the number of iterations without improvements, or CPU time taken [3].

An example use of a metaheuristic is the iterated greedy method, which was created to apply to a permutation flow shop problem with a makespan objective. An initial solution is presented, and then the method performs a destruction phase where some jobs are randomly removed. The incomplete solution is not reconstructed by reinserting the jobs using the NEH method, which is to say that each job is placed back in to maximise the objective function. Optionally, a local search can take place to examine adjacent solutions at this point, and then an acceptance criterion is used to determine which solution to take. This has some random element to it to encourage exploration of the solution space whereby a worse solution can be accepted with a probability that decreases with new solution quality. Algorithm 1 shows this process [15].

Iterated greedy has been shown to deviate from the 120 solutions mentioned alongside NEH by just 0.44%, but the solutions on average took just under a minute, which is much longer than for NEH [15].

2.2.3. Expert Systems

Expert systems contain organised bodies of knowledge which try to emulate the problem solving skills of an expert within its domain. As defined by Welbank,

> An expert system is a program which has a wide base of knowledge in a restricted domain, and uses complex inferential reasoning to perform tasks which a human expert could do [16].

Expert systems consist of three main parts, the knowledge base, the inference engine and the user interface. A diagram of this architecture is shown in figure 2.

The knowledge base is an essential part of the system as the knowledge it contains is used for solving the problems that the system encounters. Facts, heuristics and relationships known by domain experts are some of the forms the knowledge within the knowledge base can take. The most common representation technique for this knowledge is the usage production rules, which take the form of If-Then statements.

Algorithm 1 Iterated Greedy

Require:

$x :=$ NEH heuristic
$x :=$ Insertion Local Search(x)
$x_b := x$
procedure ITERATED GREEDY
  while (termination condition not satisfied) **do**
    $x' := x$
    for $i := 1 \rightarrow$destruct **do** ▷ Destruction phase
      $x' :=$ randomly extract a hob of x' and insert it into x'_R
    end for
    for $i := 1 \rightarrow$destruct **do** ▷ Reconstruction phase
      $x' :=$ best permutation after inserting $x_R(i)$ into all possible positions of x'
    end for
    $x'' :=$ Insertion Local Search(x') ▷ Optional local search
    if $C_{max}(x'') < C_{max}(x)$ **then** ▷ Acceptance criterion
      $x := x''$
      if $C_{max}(x) < C_{max}(x_b)$ **then**
        $x_b := x$
      end if
    else if $random \leq \exp\left\{-\frac{C_{max}(x'')-C_{max}(x)}{\text{Temperature}}\right\}$ **then**
      $x := x''$
    end if
  end while
  return x_b

The inference engine handles the knowledge base's content and examines its status. It determines the order in which inferences are made using various different inference methods.

The user interface allows the user to interact with the system, usually through a screen display and with an optionally explanation component. External interfaces are usually also included which allow the expert system to communicate with databases and other external sources of information [17].

The advantages of expert systems are the ability to encapsulate the knowledge of several experts, and to always be available and consistent with that knowledge.

However, since all of the decisions are made based on the rules within the inference engine, the output is only as good as the rules allow it to be. This means that sometimes expert systems can lack common sense and won't innovate in unusual circumstances. The difficulty in creating these inference rules is also high, as the knowledge of experts needs to be codified and sometimes it is difficult to explain the logic and reasoning behind decisions. The cost of creating or implementing an expert system is also expensive, in part due to the hard to codify logic and the requirement of domain experts in their creation. The creation process also takes a large amount of time.

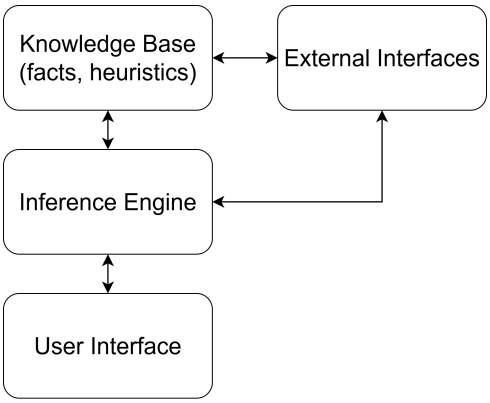

Figure 2: Architecture of a standard expert system.

The intelligent scheduling and information system introduced in [6] uses hierarchical planning to turn complicated problems into smaller pieces and was the first application of expert systems to job shop scheduling. Building on this other schedulers were built such as the opportunistic scheduler in [18].

Whilst these techniques are used somewhat successfully it should be noted that the problem spaces were small. In general, expert systems have been most successful when the number of possibilities at each step of the problem are small, which is often not the case for production scheduling problems [19; 20].

2.2.4. Approximation Algorithms

Approximation algorithms are another common approach to produce 'good enough' solutions quickly. There are a wide variety of different approaches taken to applying approximation algorithms within the field of production scheduling.

A branch and bound algorithm, which views the set of solutions as a tree which it searches along, is used in a stochastic single machine setup in [5]. However it took nearly an hour to solve a problem with around 20 jobs. A similar algorithm is examined in [21] for a two and three machine flow shop problem with six jobs, and finds that it took less than five seconds, although the number of combinations is obviously much lower.

Ant colony optimisation is used in [22] with a single machine setup and it is found to be reasonable, with expectations that it could be applied to larger production instances and more complicated flow shop setups. It was found to be fast compared to semi-exhaustive procedures, producing a solution for a 20 job two machine problem in less than three seconds compared to an expected 40 for more the more complicated method, with a 3.5% gap to that method's results.

There are many more examples of approximation algorithms such as particle swarm optimisation [23], and genetic algorithms [24]. In general, these methods benefit from being incredibly fast but often fall into local optima [25].

2.2.5. Machine Learning

Machine learning techniques aim to solve problems by learning from a set of training data and then building a mathematical model from that data in order to make predictions. These methods have the advantage of requiring less prior knowledge when compared to the rules based systems discussed previously [26].

Artificial neural networks (ANNs) aim to mimic a human brain's ability to learn from experience and generalise in order to solve complex problems using large amounts of data. They are able to determine non-linear relationships by training on available data in order to make their predictions. There are many different architectures for ANNs, each with variations designed to improve their performances based on the specific task they are being used for.

The most basic neural network structure is a multi-layer perceptron (MLP), which consists of at least three interconnected layers of nodes: an input layer, one or more hidden layers, and an output layer. An example diagram of a typical MLP with one hidden layer can be seen in figure 3. During the training phase, each input is passed into the neural network and then goes through the hidden layers in order to produce a prediction for the output. Each predicted output is then compared to the correct value which the network was aiming to produce. The network then back-propagates to change the weights of the weights of the connections between the nodes, slightly changing the model and its associated parameter weights. This changing of the weights during this phase is based on an initially selected optimisation function, for example stochastic gradient descent. This change in weights should make future predictions more accurate [27].

An MLP has been used in a case of six jobs over five machines to produce a production schedule deemed to be reasonable in [28]. This system was used by the manufacturing firm that they were working with to improve profitability.

Convolutional neural networks (CNNs) are regularised MLPs. They assemble simple patterns to regularise the data in order to model more complex patterns. This solves a common issue with MLPs have of overfitting the data due to being fully connected networks [27]. Convolutional neural networks are most commonly used for image recognition tasks, with the state-of-the-art architectures sometimes using over a hundred hidden layers to achieve above a 95% accuracy rate in a large scale image recognition task. Quite often convolutional neural networks are used alongside deep reinforcement learning, discussed in section 2.2.7.

Recurrent neural networks (RNNs) loop information from previous steps into the current step alongside the new inputs. This means that they can remember sequences in order to evaluate whole time series, so that important information from previous iterations is not forgotten [27]. There are many different types of RNN architectures, such as attention based RNNs, which form matrices of weights for each input prior to evaluating the data, and long short-term memory (LSTM) RNNs, which have layers where the nodes are replaced with special LSTM cells. RNNs in general are traditionally used for times series forecasting.

An RNN which tries to minimise the energy of the state of the network is applied to a cyclical job shop scheduling problem in [29]. It is coupled with a lagrangian relaxation method in order to reduce the complexity of the problem. The method was able to cope with larger sizes of problems with simple constraints but struggled when the problem got very large and

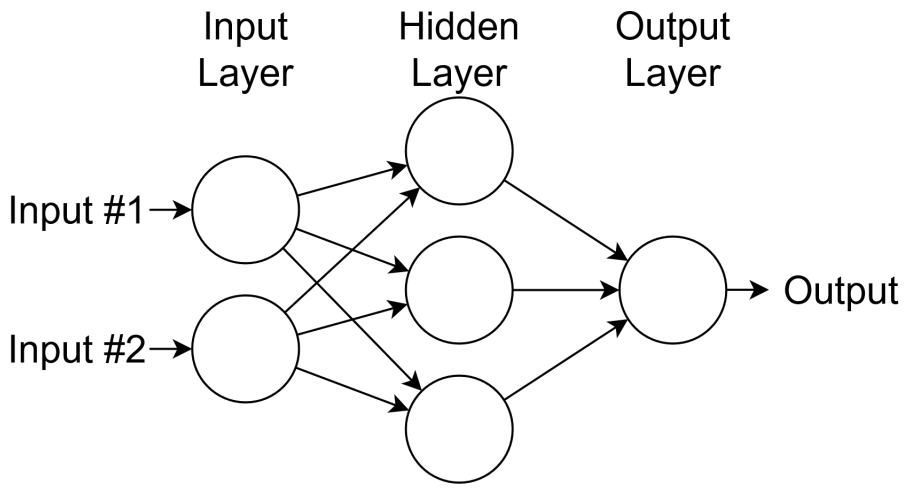

Figure 3: An example diagram of a basic multi-layer perceptron artificial neural network, consisting of four inputs, one hidden layer, and one output.

with more complicated constraints.

An LSTM RNN is combined with convolutional layers to create an auto encoder in [30]. However, rather than being used for producing production schedules themselves it is used to predict machine speeds, and so is more of an auxiliary method to actual scheduling methods to improve real world processing time predictions.

Supervised machine learning fails due to requiring labelled datasets in order to train to form predictions accurately. Within production scheduling the problems lead to increasingly large amounts of different scheduling possibilities which can't be accounted for simply as a regression between different dependent and independent variables.

2.2.6. Reinforcement Learning

Reinforcement learning concerns a variety of algorithms and techniques whereby an agent interacts with an environment through perception and action. Each step of the interaction between the agent and the environment involves the agent receiving an input, which contains a representation of the current state, s, of the environment, and then choosing an action, a. This action leads to a change in state of the environment which has an associated value, which is fed to the agent by means of a reinforcement signal r which acts as a reward. This basic feedback loop is shown in figure 4. The goal is for the agent's actions to tend towards the long term increasing sum of the reward, which it learns to do gradually through trial and error, guided by the algorithm used [31].

In general, the agent's actions determine both its reward and also the next state of the environment. This means that the agent must take into account both the immediate reward and the expected reward from the next state and therefore all future states when picking from its available actions. The agent may take a sequence of low reward actions in order to finally arrive at a state with a high reward and therefore must be able to learn which actions are optimal based on delayed rewards which can be arbitrarily far in the future.

There are four key components used for attempting to solve reinforcement learning problems: a model of the environment, a policy, a value function and a reward function. Models

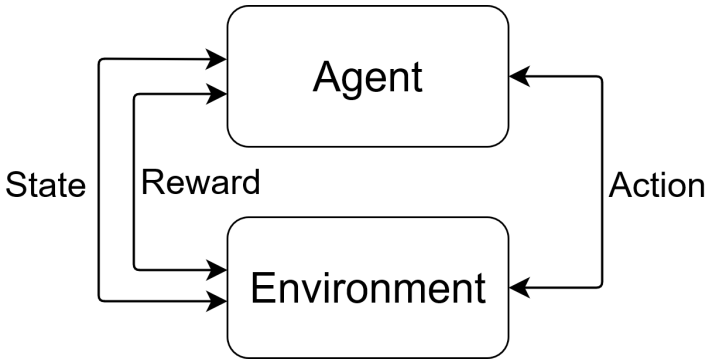

Figure 4: Basic reinforcement learning model where the agent receives a state signal and a reward feedback from the previous action and then performs a new action based on this feedback.

of the environment help to predict the next state based on the current state and proposed action. Policies define agent behaviours in a state. Reward functions effectively guide the agent towards learning how to achieve the goal of the experiment. Finally, value functions aim to evaluate states or state-action pairs for how good they are in the long run [32].

An important assumption in reinforcement learning is that each state contains all important information in order for the agent to make its decision. That is to say that all information pertaining to possible future states is contained within the current state representation. This is known as the Markov property. Such problems are well represented as Markov decision processes (MDPs). MDP problems contain a set of states, $\mathcal{S}$, a set of actions $\mathcal{A}$, a reward function $\mathcal{R}$ whereby

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathfrak{R}, \tag{8}$$

and finally a state transition function $\mathcal{T}$, whereby

$$\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S}), \tag{9}$$

where $\Pi(\mathcal{S})$ is a probability distribution for set $\mathcal{S}$ which in essence maps states to probabilities. Alternatively, $\mathcal{T}(s, a, s')$ represents the probability of action a leading to a transition from state s to state s'. Similarly, the probability for this transition P_a happening at time t is given by

$$P_a(s, s') = p(s_{t+1} = s' | s_t = s, a_t = a). \tag{10}$$

These probabilities are required due to the presence of stochasticity in the environment and therefore uncertainty in the state transitions [33; 34; 35; 36]. It should be noted that the state and action spaces must be finite for this to be valid, but countably infinite spaces can be reduced using techniques such as masking.

A common objective for agents looking to maximise their long term returns is to aim to maximise the expected discount return, E, given by

$$E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots\}, \tag{11}$$

where γ is a discount parameter. This discount parameter is between zero and one which leads to a higher valuation by the agent of immediate rewards over later ones.

In order to maximise this long term return, reinforcement learning methods aim to improve the agent's policy π over time, therefore improving its behaviour. More formally, policies map states to actions, or in stochastic cases map states to probability distributions of actions. The value of being in a state when using policy π is given by

$$V^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t = s\}, \tag{12}$$

which is the expected discounted return when following the specified policy when starting in state s. Policy π is then said to be better than policy π' if and only if for all states $V^{\pi}(s) V^{\pi'}$. In the case of finite MDPs, such as those examined in production scheduling problems, there are always one or more optimal policies which share the same value function and are at least better than or equal to all other value functions [32].

A common attempt in reinforcement learning is to apply algorithms which change the policy in real time. Q-learning is an example of this sort of algorithm, which applies directly to the agent's experience of interacting with the environment. In Q-learning, the experience of each state transition updates one element of a table containing Q-values, which map state-action pairs. Each Q-value entry $Q(s, a)$ has a value for every possible state action pair and effectively represents the expected reward for the given action in the specified state. The initial assigned value for each $Q(s, a)$ is arbitrary. Upon the state transition from s_t to s_{t+1} having taken action a_t and receiving reward r_{t+1}, the update

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{13}$$

where α is the learning rate, is applied [37]. The learning rate is present to allow convergence in non-deterministic systems. This is because the new Q-value after the update is dependent on the previous value, and so with the different rewards received in non-deterministic systems the Q function would bounce around and not converge. The learning rate means that new knowledge is only accepted in part. Usually the learning rate is set high initially and is decreased as training progresses, which has the dual effect of making the initial arbitrary values of Q less influential and then helping the values to converge later [38]. The algorithm for Q-learning can be seen in algorithm 2 [37].

The effect of Q-learning is that the agent does not know state transition probabilities or rewards unlike when using value functions. In stochastic settings, the agent learns about the transitions through experience of how the states move between each other depending on the action taken. This is important for creating agents that can adapt to new situations efficiently. Q-learning can also be beneficial in large state spaces as the agent explores without initial knowledge of all the available states. Instead, optimal actions can be selected without knowing anything about the environment's dynamics.

In any given finite MDP, Q-learning can identify the best action selection policy if initiated with a partly random selection policy, so that exploration can occur, and infinite exploration time [39].

Q-learning is used to select between which of a selection of simple heuristics to use at different times in a single machine setup with different criteria in [40]. It finds that Q-learning

Algorithm 2 Q-learning

Require:

States $\mathcal{S} = \{1, \ldots, n_s\}$

Actions $\mathcal{A} = \{1, \ldots, n_a\}$ ▷ $\mathcal{A} : \mathcal{S} \Rightarrow \mathcal{A}$

Reward Function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathfrak{R}$

Probabilistic Transition Function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$

Learning Rate $\alpha \in [0, 1]$

Discount Factor $\gamma \in [0, 1]$

procedure Q-LEARNING($\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \alpha, \gamma$)

Initialise $Q_0(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$

while Q has not converged **do**

Start in state $s \in \mathcal{S}$

while s is not a terminal state **do**

Calculate π accoring to Q and exploration strategy

$a_t \leftarrow \pi(s_t)$

$r_t \leftarrow \mathcal{R}(s_t, a_t)$ ▷ Receive new reward

$s_{t+1} \leftarrow \mathcal{T}(s_t, a_t)$ ▷ Receive new state

$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

$s_t \leftarrow s_{t+1}$

return Q

selects the most effective rule more often than not for the different objectives.

A similar single machine dispatching rule selection problem is examined using Q-learning in [41], but with a higher number of different rules, some of which are more complex. The research finds that Q-learning outperformed each of the individual dispatching rules for reducing WIP over the equivalent of a month of single shift days in a factory.

Traditional reinforcement learning fails in situations where the state and action spaces become too large, as the Q table effectively becomes too large to visit every state-action pair often enough to converge successfully. Real world scenarios often have sufficiently large state-action spaces that it is impractical to deploy these methods. This leads to the combination of deep neural networks with reinforcement learning techniques to create deep reinforcement learning.

2.2.7. Deep Reinforcement Learning

Deep reinforcement learning combines the usage of artificial neural networks with reinforcement learning.

The two main types of deep reinforcement learning are model-based reinforcement learning and model-free reinforcement learning.

In model-based reinforcement learning, the agent has access to or attempts to learn a function which predicts the state transitions and rewards of the environment. This allows the agent to create plans by seeing what would happen for a range of options and choosing between these choices explicitly. A famous example of this is Google DeepMind's AlphaZero which beat

state-of-the-art programs specialised for chess, shogi and go, as well as human masters of these games [42].

The advantage to model-based methods over model-free methods, which do not attempt to learn the model of the environment, is a substantial increase in sample efficiency. The main downside to the applicability of model-based methods is that the true model of the environment is often not available to the agent. The agent therefore has to learn the model from experience, which leads to several potential downsides. The biggest drawback is that any bias which is present in the learned model will be exploited by the agent, leading to an agent that performs well within the simulated environment but is drastically overfitted and therefore is not optimal in the real environment. Model learning is also difficult, meaning that the model may not be learned even after large amounts of computational time is expended.

Model-free methods effectively give up the potential sample efficiency gains in order to be easier to implement and tweak. Model-free methods have been more extensively developed and tested and are far more popular within production scheduling research.

Within model-free reinforcement learning there are two different approaches for what the agent should train to learn. The first of these approaches is an extension of Q-learning, where the agent attempts to approximate Q-values. Typically these methods operate off-policy, meaning any update can use data from obtained at any point during the training phase, regardless of the agent's action.

An example of a Q-learning approach is Deep Q-Networks (DQNs), first introduced by Google DeepMind in [43], effectively launched the field of deep reinforcement learning. DQNs replace the Q table with a neural network which is trained to approximate the Q-function. The neural network is initiated with parameters θ and estimates Q-values such that $Q(s, a; \theta) \approx Q * (s, a)$ where $Q*$ is the optimal Q-function. This is done by minimising the loss function, $L_t(\theta_t)$ at each time step t, given by

$$L_t(\theta_t) = E_{s,a\sim\rho(.)}[(y_t - Q(s, a; \theta_t))^2], \tag{14}$$

where ρ is the behaviour distribution over the transitions $\{s, a, r, s'\}$ collected from the environment, and y_t is the temporal difference target, which is given by

$$y_t = E_{s'\sim\varepsilon}[r + \gamma \max_{a'} (Q(s', a'; \theta_{t-1}))|s, a]. \tag{15}$$

The temporal difference error is then given by $y_t - Q$. Note that when optimising the loss function the parameters from the previous iteration are fixed. In contrast to targets in supervised learning, the network parameter weights affect the targets used here. Differentiating the loss function with respect to the parameter weights leads to the gradient of each parameter weight, Δ_{θ_t} given by

$$\Delta_{\theta_t} L_t(\theta_t) = E_{s,a\sim\rho(.);s'\sim\varepsilon}[(r + \gamma \max_{a'} (Q(s', a'; \theta_{i-1})) - Q(s, a; \theta_t))\Delta_{\theta_t} Q(s, a; \theta_t)]. \tag{16}$$

These values are used to update the parameter weights during the gradient descent step. However, computing the full expectations of the gradient values is often computationally ex-

pensive. In this case, stochastic gradient descent can be used to optimise the loss function by sampling the expectation values rather than using the full dataset in each iteration.

Commonly, the behaviour distribution ρ is selected by an ϵ-greedy strategy that chooses a random action with probability ϵ and follows the greedy strategy otherwise with probability $1 - \epsilon$. This helps to encourage the agent to explore the environment [44]. ϵ is also usually decreased every cycle as the network converges until it reaches a set minimum value. Within production scheduling, exploring the environment helps the to agent learn about the factory environment it is in so that it is less likely to get stuck choosing the same action repeatedly, depending on setup.

DQNs also tend to use a feature called experience replay to increase stability during the fitting process whilst training. A single experience contains the information of the starting and final state, the action taken and the reward, i.e $e_t = (s_t, a_t, r_t, s_{t+1})$. A random sample of experiences are chosen from a replay buffer, which are then used to fit the network.

There are several advantages given by using experience replay. The first is that otherwise, experiences obtained by trial and error are used once and then discarded, but there are some rare yet important experiences, such as ones involving damage, which should be reused in order to be effective. Secondly, the learning from the rewards is sped up as experiences are replayed again and again, meaning the network usually converges more quickly. Finally, the order of the experiences being random helps the network to avoid overfitting due to processing sequences which often contain similar states and therefore similar actions. This helps to learn the probability distribution for actions by decorrelating the datasets. In order for experience replay to be useful, the environment must not change over time, as then past experiences can be irrelevant or harmful [45; 46].

Another common way to improve DQNs is to introduce a target network to further increase stability. DQNs are function approximators trying to emulate Q-functions, which are exact. Due to this, when the neural networks' predicted Q-values are updated to make $Q(s_t, a_t)$ converge closer to the desired value, nearby states, including $Q(s_{t+1}, a)$ for all a, also have their value indirectly altered. This introduces instability into the process. Target networks are a copy of the main neural network which are used to evaluate $Q(s_{t+1}, a_{t+1})$, which is then used to train the main Q-network during the backpropagation phase. Periodically, usually over a fixed number of steps, the target network's parameters are synchronised with those of the main network. This delay between Q updates affecting the target values makes oscillations and divergence much less likely. Algorithm 3 shows the process for applying a DQN with experience replay and a target network [43].

Experience replay and target networks are used in almost all DQN research, but there are also some other improvements that can be made on the architecture which further improve their performances, such as double dueling Q-networks [47]. It is worth noting the existence of such improvements, but as they are not as ubiquitous in usage within production scheduling research they will not be discussed further.

A modified deep Q-network is used in [48] to try and minimise the effects of bottlenecks in a flexible job shop environment by making decisions at the point when a machine is idle about what should be queued. It handled the dynamic action space but was limited by having

Algorithm 3 Deep Q-learning with Experience Replay and Target Network

Initialise network Q with random weights
Initialise target network $\hat{a}$
Initialise replay memory $\mathcal{D}$ to capacity N
for episode $= 1, M$ **do**
  for $t = 1, T$ **do**
    With probability $\epsilon select a random action \mathrm{a}_t$
    otherwise select $a_t = \underset{a}{max} Q(s_t, a_t; \theta_t)$
    execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition $(s_t, a_t, r_t, s_{t+1}, done)$ in replay memory $\mathcal{D}$
    if enough experiences in $\mathcal{D}$ **then**
      Sample random minibatch of transitions $(s_t, a_t, r_t, s_{t+1}, done)$ from $\mathcal{D}$
      Set $y_t = \begin{cases} r_t, & \text{if episode terminates at time t.} \\ y_t = r_t + \gamma \underset{a'}{\max} \hat{Q}(s_{t+1}, a_{t+1}; \theta_t), & \text{otherwise} \end{cases}$
      Perform gradient descent step on $(y_t - Q(s_t, a_t; \theta))^2$
      Every C steps set $\hat{Q} = Q$

deterministic processing times and a lack of deadlines within its constraints.

Another job shop environment, this time within the semiconductor industry, is examined in [49]. A DQN is used to minimise the makespan but it failed to beat heuristics despite handling a relatively small number of machines.

A graph neural network is employed alongside a DQN in [50] to a job shop environment and found that the method out competed heuristics. However this is only over a time period of thirty jobs total.

A more complex graph neural network approach is used in a search online, learn offline manner in [51]. However this method was only applied to a parallel machine setup within production scheduling. The results were promising for general combinatorial optimisation problems so perhaps this could be applied to more complex systems.

The alternative approach to Q-learning methods is policy optimisation, which is a family of methods that explicitly represent the policy $\pi_\theta(a|s)$. These methods attempt to optimise the parameters θ either by performing optimisation techniques such as gradient ascent directly on the performance objective $J(\pi_\theta)$, or in a more indirect manner by maximising local approximations of this objective. These optimisation techniques are on-policy as they only update using data collected whilst using the most recent version of the policy. This family of methods also often involves learning an approximation $V_\phi(s)$ of the on-policy value function $V^\pi(s)$ to help with how to go about updating the policy.

Policy optimisation methods in theory handle continuous and stochastic environments better and have a faster convergence, whereas Q-learning methods can be more steady and sample efficient. Actor critics aim to take advantage of both of these features in order to provide a new method for tackling reinforcement learning problems.

Actors take an input of the state and output the best actions, essentially controlling the agent

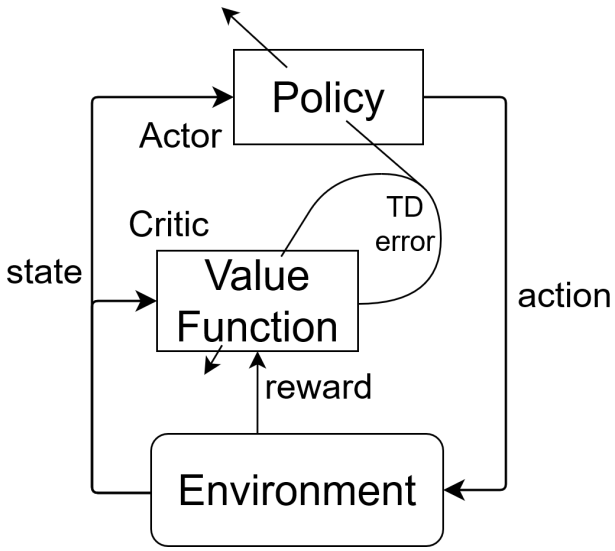

Figure 5: Architecture of an actor critic model, showing how the actor is the application of a policy that returns probabilistic actions based on the environment's state, and the critic is the value function that evaluates the actor's actions and provides an error used to perform policy correction. Recreated from [32].

via policy optimisation. The critic evaluates the action by computing the value function, making it value based. Over time both of these models get better at their roles and should learn to handle the environment better than the if the models were used separately. The architecture of an actor critic model is shown in figure 5.

Advantage actor critic (A2C) deconstructs Q values into the value function and advantage value $A(s, a)$, whereby the advantage value is then given by the equation

$$A(s, a) = r + \gamma V(s') - V(s). \tag{17}$$

Instead of learning Q values, the critic learns the advantage values so that it can evaluate how much better an action can be. This helps to stabilise the model by reducing the high variance of policy networks.

Asynchronous advantage actor critic (A3C) models consist of multiple independent networks with different weights who interact with different environment copies in parallel. This means that they can explore a much larger part of the state space in less time. The independent agents are trained individually and periodically update a global network which holds shared parameters. The agents then update their own parameters to match the global network's and continue their independent exploration of the environment. The algorithm for this process can be seen in algorithm 4 [52].

A two stage hybrid flow shop, effectively two parallel machine problems, is examined with multiple A2C agents in [53]. The optimisation objective is to minimise makespan and total tardiness and chooses between a set of dispatching rules to use when requested by an empty machine. It beats simple heuristics but doesn't examine the results on unseen randomly generated datasets.

An asynchronous actor critic model is used in a job shop environment with dynamic com-

Algorithm 4 Asynchronous advantage actor critic

Global shared parameter vectors θ and θ_v and global shared counter $T = 0$
Thread specific parameter vectors θ' and θ'_v
Initialise thread step counter $t \leftarrow 1$
repeat
  Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronise thread specific parameters $\theta' = \theta$ and $\theta'_v = \theta'$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  Set $y_t = \begin{cases} 0, & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non terminal } s_t \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ' : $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v : $d\theta_v \leftarrow d\theta_v + \frac{\partial (R - V(s_i;\theta'_v))^2}{\partial \theta'_v}$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$ =0

ponents involving breakdowns and additional orders in [54]. The model again selected between different dispatching rules at decision time. It achieved a makespan scheduling score of 91% in a static job shop and 80% in a dynamic job shop which outperforms simple heuristics in a range of environments up to 20 jobs on 15 machines.

A flexible job shop with more than 100 jobs and 100 machines is examined in [55] using an actor critic based method. Features of the input set are extracted to help find a policy in large datasets and it beats other benchmark methods, but it is noted that the random seed is set so the results are effectively cherry picked as this seed setting process will be required for any alterations to the scenario.

A semiconductor fabrication flow shop problem is examined using A3C methods in [56]. Limited uncertainties are considered, but the A3C algorithm is found to optimise the mean cycle time of the system and is superior to rules based methods.

2.3. Summary

Current deep reinforcement learning research, and by extension production scheduling research in general, often falls short of examining flexible job shops with continuous inputs with realistic

constraints. Every piece of research for production scheduling requires an environment to be produced in order to test the algorithm that is being used. This wastes time for every piece of research creating and refining the environment rather than the algorithm. This work introduces an environment that aims to reduce this wasted time.

Additionally, as each environment is bespoke, it can be difficult to compare techniques between papers which use different environments, or even if the same technique performs consistently well. By introducing a standardised environment, this problems will be reduced.

The introduced environment will also help to open up a wider range of production scheduling research constraints and optimisation targets. Currently, the main optimisation target is often makespan, when other metrics could be more desirable or examined in tandem. For example, minimising WIP is a desirable metric for small SMEs.

The environment proposed and discussed in the following sections aims to take a step towards addressing each of these problems, by allowing these features to be examined without having to spend time coding them each time a researcher wants to examine production scheduling problems, as well as making the ability to compare methods easier by attempting to create a standard environment.

3. Proposed Environment

An effective environment is required in order to interact with the various different techniques used in production scheduling research. The environment must be generalisable to the many different types of production scheduling problems which can encapsulate the different machine setups, constraints, and optimisation targets that are used in the research, as detailed in section 2.2. This section details the code structure used to produce the environments, including how the various classes interact with each other using their in-built methods. The full code for the environment can be found in appendix A.

The code is structured such that there are four main classes of object which interact with each other. These are the factory class, resource class, job class and operation class. The interaction between these classes is shown in figure 6.

The factory class regulates the flow of all information, and provides the interface through which any required commands can be run. In the case of reinforcement learning, the agent interacts solely with the factory class when creating its decisions. Similar to a real factory, this interface includes the ability to add and remove jobs and resources, and schedule individual operations into a resource's queue.

Resources process individual operations within the environment, but the handling of their queues is contained within the factory class, and therefore the main aim of the class is to facilitate this functionality.

Jobs are comprised of an ordered set of operations that must be completed. They also include appropriate job level information such as its deadline.

Finally, operations are the building blocks of jobs and are defined by the ability to complete them through a single action performed by an appropriate resource e.g. bending or laser cutting.

The rest of this section discusses the usage of each of these classes in more detail, as well as necessary features and considerations which allow them to accurately represent a factory.

3.1. Factory Class

The factory class is the main class through which the factory environment, with all its included resources and jobs, is simulated and controlled. It contains some of the validation logic for many of the checks required for a factory simulation to run successfully. Factory objects are initiated with a starting list of resource and job objects, as well as an empty list of completed jobs which is used for verification purposes. The UML diagram for the factory class, including its methods and properties, is shown in figure 7.

The properties for the factory object include getter and setter methods for the `resources`, `jobs` and `completed_jobs` within the factory. There are also property methods to retrieve the total `wip_value` of all jobs within the factory. Finally, the `resource_queues_are_not_empty` property indicates whether or not all resources within the factory have empty processing queues, which can be used to check if a simulation is finished if there are a finite number of jobs and operations.

All other workings of the factory are then included in the class methods. Adding new jobs or resources to the factory's respective properties can be achieved using `add_job` or `add_resource`.

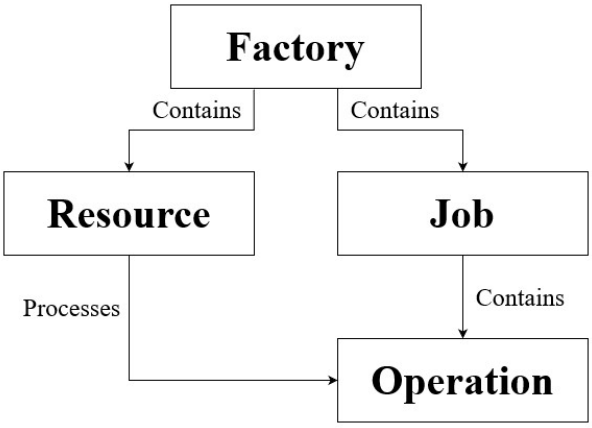

Figure 6: Interaction between the environment classes.

Resources and jobs can be found within their respective property lists using their respective `find_` methods. Jobs can be found using their IDs with `find_job`. Individual resources can be founding using their name with `find_resource_by_name`, and all resources that can process a certain type of operation can be found with `find_resource_by_operation_type`.

The flow of operations and jobs within the factory are also handled with factory class methods. Operations are queued into resources using the factory object's `queue_operation` method, which takes as its parameters the `resource_name` that will process the operation, the `job_id` that the operation is part of, the `operation_job_index` which is the index in the job which the operation lies, and a `resource_queue_position`, which is the place in the queue the operation will be added. By default, this method can take just the `resource_name` and `job_id`, and will queue the first operation in that job's operation list at the end of the specified resource queue.

Operations can also be removed from a resource's queue using the `unqueue_operation` method. Using this in conjunction with the `queue_operation` method allows reordering of operations within resource queues.

The factory class also contains methods which handle the passing of time within the simulated environment. The `process_shortest_operation` method triggers the processing of all operations at the front of each resource queue by the amount of time found using the `find_minimum_resource_time` method. This accounts for the resources' `processing_speeds`. After this, all operations which have been completed are marked as such and removed from the front of the resource and job queues. All jobs have their deadline reduced by the amount of time passed. Finally, any jobs with no remaining operations are moved into the `completed_jobs` list, meaning their associated `wip_value` will now not count towards the factory's total. This method is partly based on the assumption that operations willnot be removed from a resource queue after they are in progress due to there being setup time penalties. Otherwise, in built methods in the resource class can handle separate operation processing.

3.2. Resource Class

The resource class represents a single resource within the factory, whether that is a machine or a worker, which can process a specific operation type. Resource objects contain a queue of

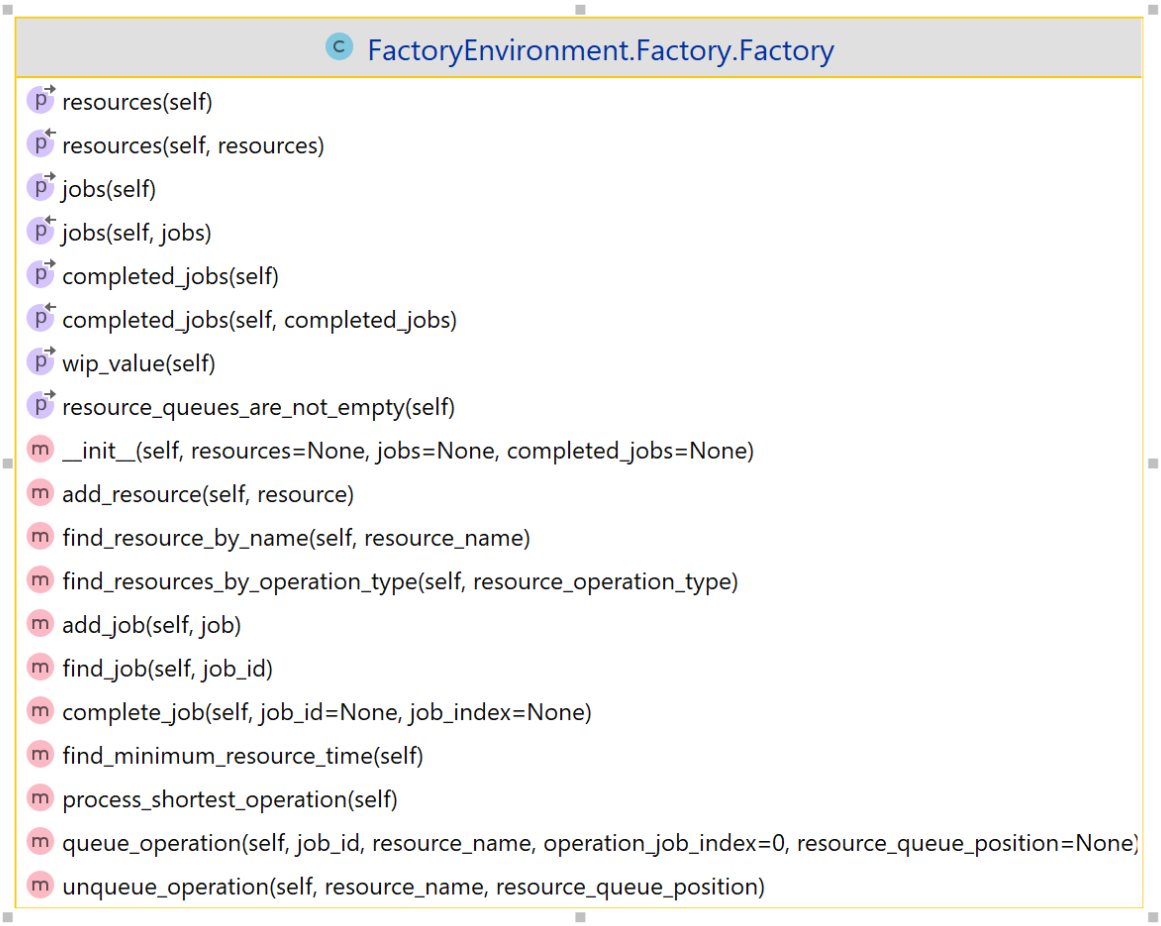

Figure 7: UML diagram for the factory class including class methods and properties.

operations that they are to process, in the order that they will process them. These operations are assigned through the factory classes `queue_operation` method. Resources in this environment setup can only process one type of operation, and can only process one operation at a time. The UML diagram for the resource class, including its class methods and properties, is shown in figure 8.

A resource object is initiated with a set of properties that includes its `name`, which identifies the resource, its `operation_type`, which defines which type of operation that the resource, its `queue`, which is initiated empty, as well as a `wip_rate`, which represents the comparative cost of using the resource, and a `processing_speed`, which is a comparative measure of how fast the resource processes operations compared to other resources of the same type.

Many of the resource based methods in the factory class call a method in the resource class that then produces the required functionality. The `process_first_in_queue` method is called by the factory class's `process_shortest_operation` method, and similarly the methods `add_operation_to_queue` and `remove_operation_from_queue` are called by the respective factory class methods.

There are some methods which provide information, such as `check_times_in_queue` and `check_cumulative_time_in_queue`, which returns a list of the time remaining for each operation in the resource's queue, list of the cumulative times to process the remaining operations in the queue, respectively. The `get_first_in_progress` method returns if the first operation in the queue is in progress, so that it can be checked if setup time penalties will occur if the operation is unqueued.

The `first_operation_out_of_order` method acts as a validation check for if the first operation in the queue is the first operation in its job's queue, meaning it is ready to be processed.

3.3. Job Class

The job class aims to mimic a job within a factory by holding an ordered list of operations which must be processed in the right order so that the job can be completed. The UML diagram for the job class, including its class methods and properties, is shown in figure 9.

Job objects are initiated with a set of `operations` that must be processed in order to be completed, an empty set of `completed_operations`, a unique `id`, and a `deadline`. There are also properties which represent the total `wip_value` for the job, which is the sum of the job's operations' and completed operations' *WIP* values, as well as whether the job is `in_process`, `completed` or `late`.

The `add_operation` method initiates a new operation object and adds it at the required index to the job's `operations` list, and the `complete_operation` method moves the first operation in the `operations` list into the `completed_operations` list. The `move_deadline` method is called by the factory class's `process_shortest_operation` method, and reduces the job's deadline by the specified amount of time.

There is also a `find_operation_index` method which returns the index of the operation in the `operations` list of the specified `operation_type` parameter. Finally, the method `assign_operational_order` assigns the index value for each operation in the `operations` list

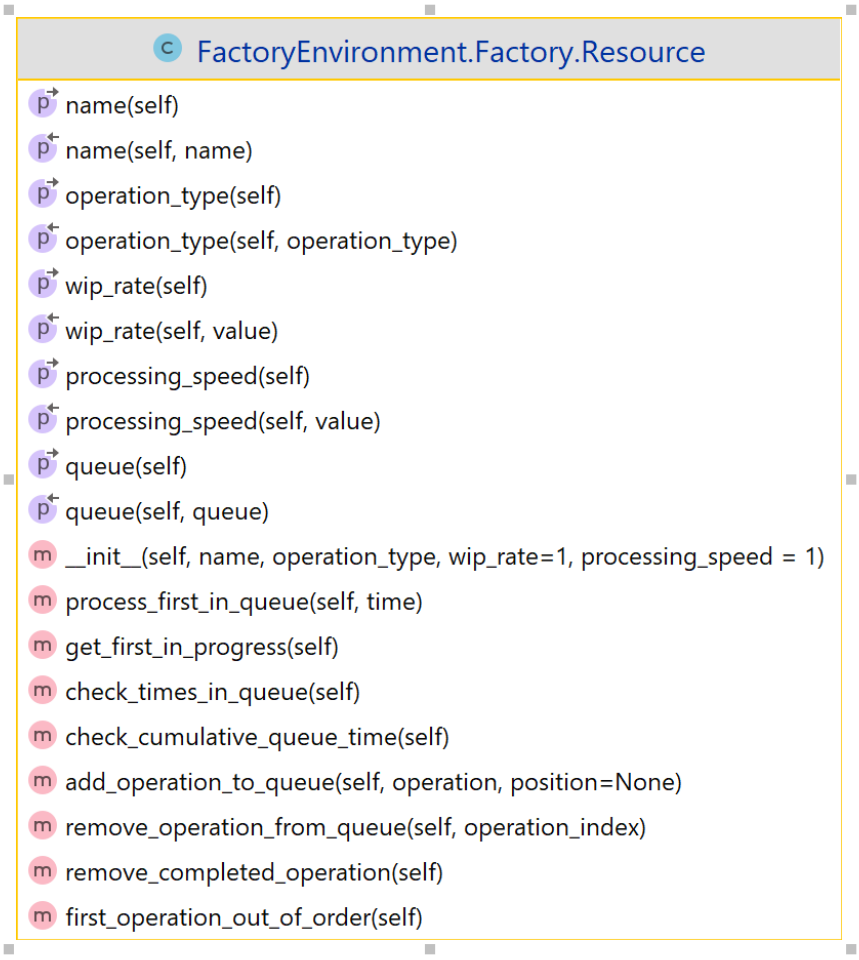

Figure 8: UML diagram for the resource class including class methods and properties.

Figure 9: UML diagram for the job class including class methods and properties.

to the respective operation objects within it. This is used for validation checks when processing those operations.

3.4. Operation Class

Operations are the smallest building blocks of jobs as they represent one process within a job which can be completed by one resource within the factory. Within this environment these resource behaviours are represented by the resource class. Operations contain a lot of information about their state that is then used by the other classes within the environment for their various methods and properties. The UML diagram for the operation class, including its class methods and properties, is shown in figure 10.

Operations are initiated with the `job_id` of the job that they are part of, the `operational_order`, which is the index of the operation within that job's `operations` list, the `processing_time` that it will take to complete the job, the `setup_time` required for a resource to start processing the job, a `completion_time`, which is the sum of the processing and setup times, an `operation_type`, for example, bending, and an initial `wip_value` of zero.

There are also properties that represent the operation's progress to completion. An operation object has additional properties for its `is_queued` state, `in_progress` state, and `completed` state.

The operation class has only one method, which is `process_operation`. This sets `in_progress`, reduces the `completion_time`, and increases the `wip_value`. If the completion time is zero, the object is marked as completed and `in_progress` is set to false.

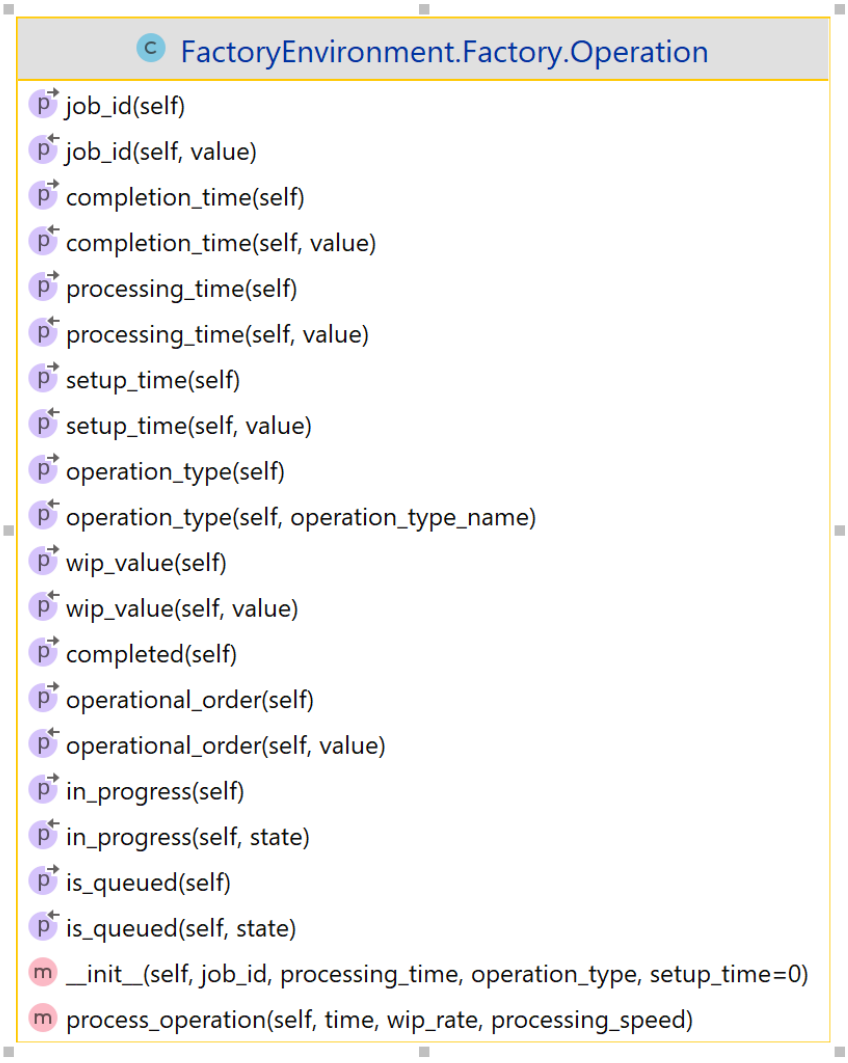

Figure 10: UML diagram for the operation class including class methods and properties.

4. Creating Factory Environments

This section outlines how the environments that can be created are fit for the purpose of research. The ability to create environments with various different machine setups and constraints is outlined in sections 4.1 and 4.2, as well as the ability to track the different optimisation targets in section 4.3.

The general process for creating a factory environment is to first initialise a factory object, some resource objects with their names and associated WIP rates, and some jobs with their IDs and deadlines. Operations are created and added to jobs using the job class's `add_operation` method, which takes as arguments the operation's type and time until completion. The operation type should match a resource operation type. Created resource and job objects are added to the factory using the factory class's `add_resource` and `add_job` methods, respectively. A flow diagram showing this process can be seen in figure 11.

4.1. Machine Setups

Simulating complex machine setups is straightforward. Single machine setups can trivially be created by initialising a single resource and adding it to the factory with as many jobs as required, each with a single operation. Extending to parallel machine setups then just requires creating and adding to the factory multiple resources which process the same type of operation but with different IDs. Heterogeneous machine setups can be achieved by setting the `processing_speed` variable to different values.

Setting up resources with different operation types that they can process allows shop scheduling problems to be simulated, with the type of shop scheduling problem then determined by the operation setup for the jobs. Flow shops require adding the same operations in the same order to the jobs. Job shops can be achieved by having jobs with different operation orders in various arrangements, with each operation being able to be processed by at least one resource within the factory. The same extension as moving from a single machine setup to a parallel machine setup of adding multiple resources with the same operation type can be applied here to allow the simulation of flexible shop problems of various types.

For research purposes, sets of jobs can be generated using loops with random times within a range, and a random amount of operations in order. These can maintain unique identities simply through applying the index as the job id, but more complicated ids can be derived if required using other methods. Using the index as an ID can help with applying first in first out and last in first out methods.

4.2. Constraints

Some constraints are handled by logic within the environment code, whereas others will need to be added manually when creating the factory. The code has been structured to be extensible for different additional constraints.

Setup times can be added to operations as part of the creation of operation objects. Setup times with a resource dependence aren't currently supported but could in theory be added by

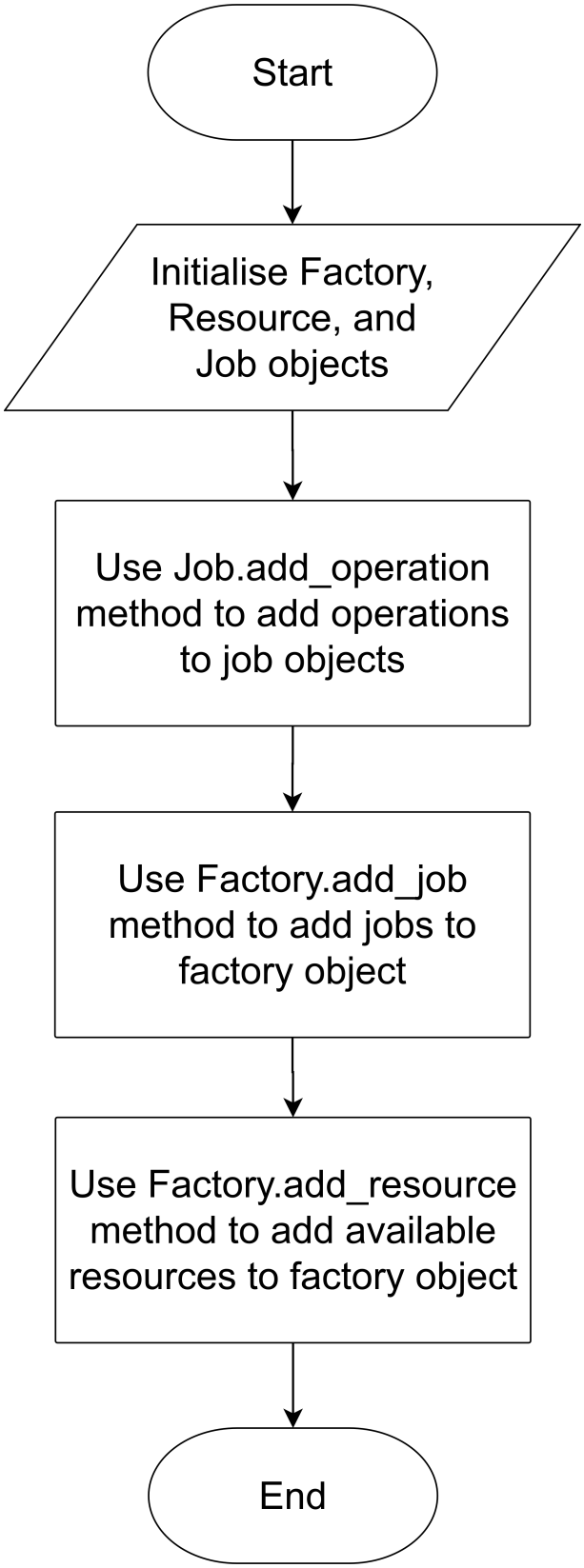

Figure 11: Flow diagram for the general process of creating a factory environment with different jobs, operations and resources.

creating a resource property that adds to the setup time of the operation.

Stochasticity in different parameters can be handled in a variety of different ways. Stochastic release dates can be achieved by creating a job generator that generates random times until the job is released to be a visible part of the environment to the agent. The total environment time can be tracked as part of the environment if required to help with this.

Stochastic processing times can be achieved by generating a random time for the representation of the processing time to the agent. The environment then knows the processing time and will process that amount of time for the operation, but the model will have a different amount of time expected.

Breakdowns can be handled as forced operations at the front of a resource queue that can't be moved with a randomly decided length. The agents used can queue the held up operations elsewhere, and if they are completed in another resource they will mark themselves as such. Then, when they reach the front of the resource queue which had the breakdown, they will be automatically removed as they have no completion time left.

Alternatively, the resource's processing speed can be set to 0. Stochastic speeding up and slowing down of resources can be handled in a similar way to add further uncertainty to the system.

4.3. Optimisation Targets

Completed jobs are stored with their final state to allow tracking of many different optimisation targets. Job level throughput can be tracked by the amount of completed jobs in a certain amount of time, and operation and resource level throughput requires a simple tracker to be added to the operation to measure which resource processed it.

The time left before the deadline for a job is stored within the final state. Late jobs are easily identifiable by those that are completed with a negative deadline. Early jobs similarly have a positive final deadline, and a comparison can be made between the deadline value and a variable to determine if a job was too early and should be penalised. Therefore earliness and lateness metrics are available within the environment.

Work in process is tracked at an operational level and is then summed at a job level to give the total WIP for a job. It is then also available at a factory level, with each of the values for the different levels accessible through each classes' `wip_value` property. The factory WIP value by default only tracks jobs that have currently not yet been completed, but a similar sum can be done for all completed jobs to see how much work units and therefore how much cost has been involved in the processing of a certain job.

5. Model Interaction

In general, models can be applied to a constructed scenario by either building the algorithm programatically, in the case of simple heuristics, or by creating a state-action space and appropriate agent that interacts with the environment, in the case of reinforcement learning methods.

The rest of this section aims to demonstrate how the different methods outlined in section 2.2 can be applied to a factory environment that has been set up using the code provided in this work. First, heuristic based methods, including expert systems and approximation algorithms, are examined in section 5.1, then reinforcement learning and deep reinforcement learning methods are examined in 5.2.

5.1. Heuristic Based Methods

Simple heuristics in general can be applied to the environment by programming the relevant algorithm to determine the optimal operation to queue at any given decision point for a given resource.

For example, the first in first out and last in first out methods can be achieved by, at each decision step, determining which operation that is applicable to the given resource is the attached to the job which is earliest or latest arriving. Possible ways of finding which job was the earliest or latest to arrive is to use the IDs, if the jobs arrive in ID order, or a timestamp if not.

The earliest due date heuristic can be applied by sorting the operations available for a given resource by their respective jobs' due dates and then adding the lowest result to the queue. This information is easily found within the environment by creating a list of valid jobs that can be queued by searching each job for a valid operation that can be queued into the resource, and then sorting the list of valid jobs by the deadline, and then ordering the operation from the job that is first in the queue.

More advanced heuristics can be constructed following this same basic process based on the ability to combine the information provided by the environment. For example, the apparent tardiness cost can be computed for each job and operation within the factory to provide the basis for which operation to queue next.

As operations can be removed and inserted via inbuilt factory class methods the NEH and iterated greedy methods are both able to be employed via coding the algorithm using these inbuilt methods to find the correct sorting of jobs and operations within the environment.

Approximation algorithms can be applied in a similar method to create plausible solutions that are then compared based on the optimisation targets. They operate in a similar manner to heuristics in regards to the algorithm needing to be inputted and the flow of information that they need.

Rules based expert systems rely on information to use their inference machines. The information provided within the factory environment should be able to provide enough information at a given decision point. The main difficulty would then be interfacing the expert system with the environment to create an adequate representation for it.

It can therefore be seen that the application of a variety of the differing complexities of heuristics can be achieved with the information available within the factory environment. Small

tweaks may be needed in order to achieve the full functionality of the exact heuristic being examined, but this can be done within the code.

5.2. Reinforcement Learning

The process for applying reinforcement learning algorithms begins similarly to the process for applying simple heuristics. Reinforcement learning algorithms, including deep reinforcement learning algorithms, require an environment to act as a representation of the problem that the agent can interface with so that it can make decisions. From this environment, a set of states, actions, and rewards need to be constructed so that the reinforcement learning agent can learn about the environment and the best actions to take. Therefore, in addition to the process of creating the environment used for simple heuristics, a state, action and reward set must be created.

States are a representation to the agent of what the environment looks like. Within production scheduling, this can be an overview of the whole factory with all of its jobs and operations, with all of their statuses, as well as resources with their full queue information, or a smaller slice of this, for example information about operations that can be queued into a given machine. The action space can then be constructed to provide the possible state transitions for this process. Finally, the rewards are dictated by the optimisation objective, and so by measuring the optimisation target as part of the process the reward for the agent after each action can be determined. It is also common to add a negative reward for each action so that the agent attempts to maximise its reward in the fewest amount of steps possible.

A simple example of a possible configuration for the state action space is to create a state representation containing the current resource cumulative queues, as well as operation information including whether it is queued, its time until completion, and the job deadline. The action space is then queueing each operation into the relevant resource, and a separate action to advance time to process the shortest operation in any of the resources' queues.

There are many other possible ways of creating a state action space for researching production scheduling supported by this environment. For example, a decision can be taken only when a resource queue is empty, and the state representation could then include all possible operations that could be queued to that resource, with information about each of the relevant completion times and job deadlines. Cumulative queue times for other resources could then be included with their processing speeds and WIP rates, so that the agent can judge which operation to prioritise. If the state representation gets too large this method could be combined with heuristics to create a simplified decision space. The available actions for the agent are then to choose which operation to queue only, and the passing of time is handled within the environment within the step function, until a new decision point is reached.

The action space can be further expanded if the ability to remove items and reposition them in queues is included. Similarly, when extra constraints are added such as breakdowns, the representation can include information about processing speeds.

Rewards are then just structured by the controller based on the desired optimisation target and the chosen action. For example, the agent can receive a negative reward for allowing a job to be late before completion, or for allowing WIP value to be raised, and a positive reward for

the completion of a job. If a benchmark is known for a particular test, the agent can be penalised for its distance away from optimality. The information flow allowed by the environment allows these rewards to be applied to the agent.

Once an environment with a state representation and action space with rewards for the transitions has been created, an agent can then be added to interact with the system. The general process for creating a factory environment, and applying an agent can be seen in figure 12.

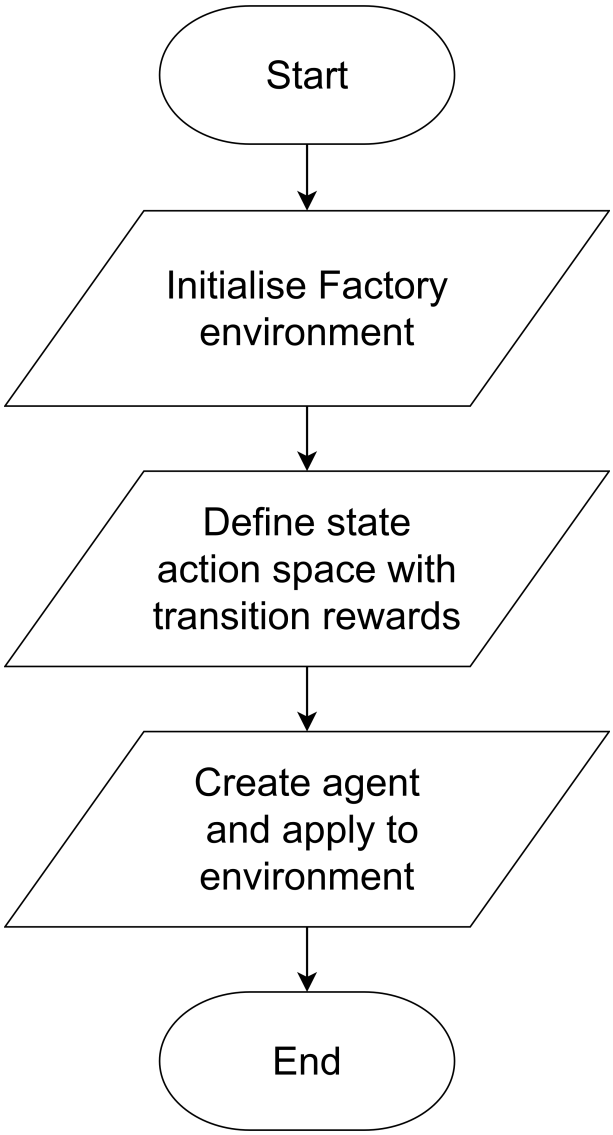

Figure 12: Flow diagram showing the general process of creating an environment and applying an agent.

Following this general process, the different reinforcement learning techniques can be applied to the created environments with various different setups, optimisation targets and constraints. Training these algorithms is achieved in the normal way by allowing the agent to interact with the environment to learn about the interactions between the state-action spaces and rewards.

6. Conclusions

This work aimed to produce a factory environment which could be used for production scheduling research purposes with modern techniques and methods.

The factory environment produced is generalisable to many different possible machine setups, constraints and optimisation objectives and can be used alongside reinforcement learning methods due to its structure and methods.

Currently, the environment cannot properly handle all machine setups constraints, such as job and operations dependent setup times and flexible shops which allow machines to process more than one type of resource. The latter can be approached by allowing resources to hold arrays of types of operations they can process and then managing to display a representation to the model that adequately shows this. The main next stage would be to move this environment into the OpenAI framework as this is more familiar with many researchers.

Overall, this advancement will allow production scheduling researchers and controllers to reduce the amount of time that they spend producing and curating environments and instead focus on applying novel techniques to make advancements in the field of production scheduling. These advancements can impact productivity and profitability within manufacturing by providing dynamic schedules for different environments.

7. References

[1] B. Waschneck, T. Altenmüller, T. Bauernhansl, and A. Kyek, "Production scheduling in complex job shops from an industrie 4.0 perspective: A review and challenges in the semiconductor industry," CEUR Workshop Proceedings, vol. 1793, 2016.

[2] M. L. Pinedo, Scheduling: Theory, algorithms, and systems. 2008.

[3] R. Ruiz, "Scheduling Heuristics," in Handbook of Heuristics, pp. 1–24, Springer International Publishing, 2016.

[4] A. Rinciog and A. Meyer, "Towards Standardizing Reinforcement Learning Approaches for Stochastic Production Scheduling," Procedia CIRP, vol. 107, pp. 1112–1119, 2022.

[5] K. R. Baker, "Minimizing earliness and tardiness costs in stochastic scheduling," European Journal of Operational Research, vol. 236, pp. 445–452, 7 2014.

[6] M. S. Fox and S. Smith, "ISIS: a knowledge-based system for factory scheduling," Expert Systems, vol. 1, no. 1, pp. 25–49, 1984.

[7] J. R. Jackson, "Scheduling a production line to minimize maximum tardiness," Tech. Rep. 43, University of California, Los Angeles, 1955.

[8] I. Sabuncuoglu and O. B. Kizilisik, "Reactive scheduling in a dynamic and stochastic FMS environment," International Journal of Production Research, vol. 41, no. 17, pp. 4211–4231, 2003.

[9] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," SIAM Journal on Applied Mathematics, vol. 17, no. 2, pp. 416–429, 1969.

[10] A. P. J. Vepsalainen and T. E. Morton, "Priority Rules for Job Shops with Weighted Tardiness Costs," Management Science, vol. 33, no. 8, pp. 1035–1047, 1987.

[11] M. Nawaz and E. E. Enscore, "A Heuristic Algorithm for the m-Machine, n-Job Flow-shop Sequencing Problem," Omega, vol. 11, no. 1, pp. 91–95, 1983.

[12] E. Taillard, "Theory and Methodology Some efficient heuristic methods for the flow shop sequencing problem," European Journal of Operational Research, vol. 47, no. 1, pp. 65–74, 1990.

[13] P. J. Kalczynski and J. Kamburowski, "An improved NEH heuristic to minimize makespan in permutation flow shops," Computers and Operations Research, vol. 35, pp. 3001–3008, 9 2008.

[14] W. Liu, Y. Jin, and M. Price, "A new improved NEH heuristic for permutation flowshop scheduling problems," International Journal of Production Economics, vol. 193, pp. 21–30, 11 2017.

[15] R. Ruiz and T. Stützle, "A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem," European Journal of Operational Research, vol. 177, pp. 2033–2049, 3 2007.

[16] M. Welbank, "A Review of Knowledge Acquisition Techniques for Expert Systems," tech. rep., British Telecom Research Laboratories, 1983.

[17] K. Metaxiotis, D. Askounis, and J. Psarras, "Expert systems in production planning and scheduling: A state-of-the-art survey," Journal of Intelligent Manufacturing, vol. 13, pp. 253–260, 2002.

[18] S. F. Smith, M. S. Fox, and P. S. Ow, "Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems," AI Magazine, vol. 7, pp. 45–61, 9 1986.

[19] C. Badie, G. Bel, E. Bensana, and G. Verfaillie, "Operations research and artificial intelligence cooperation to solve scheduling problems: the OPAL and OSCAR systems," in Expert Planning Systems, pp. 1–5, 1990.

[20] S. F. Smith, "Knowledge-based production management: Approaches, results and prospects," Production Planning and Control, vol. 3, no. 4, pp. 350–380, 1992.

[21] E. Ignall and L. Schrage, "Application of the Branch and Bound Technique to Some Flow-Shop Scheduling Problems," Operations Research, vol. 13, no. 3, pp. 400–412, 1965.

[22] R. F. Tavares Neto and M. Godinho Filho, "An ant colony optimization approach to a permutational flowshop scheduling problem with outsourcing allowed," Computers and Operations Research, vol. 38, no. 9, pp. 1286–1293, 2011.

[23] S. Wang, X. Xiao, F. Li, and W. Ce, "Applied research of improved hybrid discrete PSO for dynamic job-shop scheduling problem," in 2010 8th World Congress on Intelligent Control and Automation, pp. 4065–4068, 2010.

[24] C. Bierwirth and D. C. Mattfeld, "Production Scheduling and Rescheduling with Genetic Algorithms," Evolutionary Computation, vol. 7, pp. 1–17, 3 1999.

[25] D. P. Williamson and D. B. Shmoys, The design of approximation algorithms. Cambridge university press, 2011.

[26] A. Dey, "Machine Learning Algorithms: A Review," International Journal of Computer Science and Information Technologies, vol. 7, no. 3, pp. 1174–1179, 2016.

[27] J. Schmidhuber, "Deep learning in neural networks : An overview," Neural Networks, vol. 61, pp. 85–117, 2015.

[28] F. Zhao, Q. Zhang, D. Yu, and Y. Yang, "A Novel Method Based on Artificial Neural Network to Production Scheduling," International Journal of Information Technology, vol. 11, no. 6, pp. 20–29, 2005.

[29] M.-T. Kechadi, K. S. Low, and G. Goncalves, "Recurrent neural network approach for cyclic job shop scheduling problem," Journal of Manufacturing Systems, vol. 32, pp. 689–699, 10 2013.

[30] A. Essien and C. Giannetti, "A Deep Learning Model for Smart Manufacturing Using Convolutional LSTM Neural Network Autoencoders," IEEE Transactions on Industrial Informatics, vol. 16, pp. 6069–6078, 9 2020.

[31] L. Pack Kaelbling, M. L. Littman, A. W. Moore, and S. Hall, "Reinforcement Learning: A Survey," Journal of Artiicial Intelligence Research, vol. 4, pp. 237–285, 1996.

[32] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. The MIT Press, 1 ed., 2008.

[33] S. Edelkamp and S. Schrödl, "Introduction and Guidelines," in Heuristic Search, pp. 3–46, Elsevier, 2012.

[34] R. Bellman, "A Markovian Decision Process," Indiana University Math Journal, vol. 6, no. 4, pp. 679–684, 1957.

[35] R. A. Howard, Dynamic Programming and Markov Processes. New York, London: The Technology Press of The Massachusetts Institute of Technology and John Wiley and Sons, Inc, 1 ed., 1960.

[36] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., 1st ed., 1994.

[37] C. J. C. H. Watkins and P. Dayan, "Q-learning," Machine Learning, vol. 8, pp. 279–292, 5 1992.

[38] T. M. Mitchell, "Machine learning," in Machine Learning, ch. 13, pp. 367–386, New York: McGraw-Hill Science/Engineering/Math, 1997.

[39] F. S. Melo, "Convergence of Q-learning: a simple proof," tech. rep., Institute for Systems and Robotics, Lisboa, 2001.

[40] Y. C. Wang and J. M. Usher, "Application of reinforcement learning for agent-based production scheduling," Engineering Applications of Artificial Intelligence, vol. 18, pp. 73–82, 2 2005.

[41] L. C. Rabelo, A. Jones, and Y. Yih, "Development of a real-time learning scheduler using reinforcement learning concepts," in IEEE International Symposium on Intelligent Control, pp. 291–296, IEEE, 1994.

[42] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," Science, vol. 362, pp. 1140–1144, 12 2018.

[43] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[44] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," 12 2013.

[45] L.-J. Lin, "Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching," Machine learning, vol. 8, no. 3, pp. 293–321, 1992.

[46] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting Fundamentals of Experience Replay," in International Conference on Machine Learning, pp. 3061–3071, 7 2020.

[47] Z. Wang, T. Schaul, M. Hessel, and M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning Hado van Hasselt," in International conference on machine learning, pp. 1995–2003, PMLR, 2016.

[48] T. E. Thomas, J. Koo, S. Chaterji, and S. Bagchi, "Minerva: A reinforcement learning-based technique for optimal scheduling and bottleneck detection in distributed factory operations," in 2018 10th International Conference on Communication Systems & Networks (COMSNETS), pp. 129–136, 2018.

[49] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek, "Optimization of global production scheduling with deep reinforcement learning," Procedia CIRP, vol. 72, pp. 1264–1269, 2018.

[50] M. S. A. Hameed and A. Schwung, "Reinforcement learning on job shop scheduling problems using graph networks," arXiv, pp. 1–8, 2020.

[51] J. Oren, C. Ross, M. Lefarov, F. Richter, A. Taitler, Z. Feldman, C. Daniel, and D. Di Castro, "SOLO: Search Online, Learn Offline for Combinatorial Optimization Problems," in International Symposium on Combinatorial Search, vol. 1, pp. 97–105, 2021.

[52] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in International Conference on Machine Learning, pp. 1928–1937, PMLR, 2 2016.

[53] F. T. Gerpott, S. Lang, T. Reggelin, H. Zadek, P. Chaopaisarn, and S. Ramingwong, "Integration of the A2C Algorithm for Production Scheduling in a Two-Stage Hybrid Flow Shop Environment," Procedia Computer Science, vol. 200, pp. 585–594, 2022.

[54] 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.

[55] A. K. Elsayed, E. K. Elsayed, and K. A. Eldahshan, "Deep Reinforcement Learning based Actor-Critic Framework for Decision-Making Actions in Production Scheduling," in 2021 Tenth International Conference on Intelligent Computing and Information Systems (ICICIS), pp. 32–40, Institute of Electrical and Electronics Engineers (IEEE), 2 2021.

[56] J. Liu, F. Qiao, and Y. Ma, “Real time production scheduling based on Asynchronous Advanced Actor Critic and composite dispatching rule,” in 2020 Chinese Automation Congress (CAC), pp. 7380–7383, Institute of Electrical and Electronics Engineers Inc., 11 2020.

A. Environment code

The Python code used for the environment can be found in a repo at https://github.com/C-McGowan/FactorySimulation.