

Technical University of Crete
School of Electrical and Computer Engineering



Diploma Thesis

Production Line Control using Deep Reinforcement Learning Techniques

Georgios Pantazis

Submitted in partial fulfillment of the requirements for the Engineering Diploma in the School of Electrical and Computer Engineering at the Technical University of Crete

Chania, July 2025

Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών



Διπλωματική Εργασία

**Έλεγχος Γραμμών Παραγωγής με
χρήση Τεχνικών Βαθιάς Ενισχυτικής
Μάθησης**

Γεώργιος Πανταζής

Η εργασία εκπονήθηκε στο πλαίσιο των απαιτήσεων για την απόκτηση Διπλώματος Μηχανικού από τη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης

Χανιά, Ιούλιος 2025

Abstract

Modern production lines are facing challenges, when it comes to maintaining efficiency. These challenges are due to various reasons, including unpredictable machine failures, fluctuating demand, and buffer congestion. These issues result in increased costs and delays, that traditional scheduling methods struggle to handle. As manufacturing systems become more complex, there is a growing need for adaptive control strategies that can optimize production in real-time. In this diploma thesis, we present a deep reinforcement learning approach to optimizing production line control, leveraging SimEvents in Simulink and the Matlab Reinforcement Learning Toolbox. The production system consists of sequential machine stages, buffers, and an assembly mechanism, where inefficiencies, such as buffer congestion and machine downtime, lead to increased operational costs. A Proximal Policy Optimization (PPO)-based reinforcement learning agent was designed to handle machine operations by monitoring key system variables, including buffer levels and machine states. The agent is designed to minimize delays and prevent buffer accumulation, enabling it to learn adaptive scheduling policies that improve overall efficiency and minimize cost. Through a series of simulations, our approach proves to be more effective than traditional optimization methods and yields solutions very close to optimal solutions, which can be obtained for small problem instances by dynamic programming methods. The results of extensive experimental testing show that reinforcement learning successfully improves manufacturing efficiency through better decision-making, pointing to new ways of running production systems.

Περίληψη

Οι σύγχρονες γραμμές παραγωγής αντιμετωπίζουν προκλήσεις, όσον αφορά στη διατήρηση της αποδοτικότητας. Αυτές οι προκλήσεις οφείλονται σε διάφορους λόγους, όπως οι απρόβλεπτες βλάβες μηχανών, οι διακυμάνσεις της ζήτησης και η συμφόρηση των ενδιάμεσων αποθεμάτων. Αυτά τα ζητήματα οδηγούν σε αυξημένο κόστος και καθυστερήσεις, τα οποία οι παραδοσιακές μέθοδοι χρονοπρογραμματισμού δυσκολεύονται να διαχειριστούν. Καθώς τα συστήματα παραγωγής γίνονται ολοένα και πιο περίπλοκα, υπάρχει αυξανόμενη ανάγκη για προσαρμοστικές στρατηγικές ελέγχου που μπορούν να βελτιστοποιούν την παραγωγή σε πραγματικό χρόνο. Στην παρούσα διπλωματική εργασία, παρουσιάζουμε μια προσέγγιση βαθιάς ενισχυτικής μάθησης για τη βελτιστοποίηση του ελέγχου γραμμών παραγωγής, αξιοποιώντας το SimEvents στο Simulink και το Matlab Reinforcement Learning Toolbox. Το σύστημα παραγωγής αποτελείται από διαδοχικές μηχανές, ενδιάμεσους χώρους αποθεμάτων και έναν μηχανισμό συναρμολόγησης, όπου οι αναποτελεσματικότητες, όπως η συμφόρηση των αποθεμάτων και ο χρόνος αδράνειας των μηχανών, οδηγούν σε αυξημένο λειτουργικό κόστος. Σχεδιάστηκε ένας πράκτορας ενισχυτικής μάθησης, βασισμένος στον αλγόριθμο Proximal Policy Optimization (PPO), ο οποίος διαχειρίζεται τη λειτουργία των μηχανών, παρακολουθώντας βασικές μεταβλητές του συστήματος, όπως τα επίπεδα των ενδιάμεσων αποθεμάτων και τις καταστάσεις των μηχανών. Ο πράκτορας έχει σχεδιαστεί για να ελαχιστοποιεί τις καθυστερήσεις και να αποτρέπει τη συσσώρευση αποθεμάτων, επιτρέποντάς του να μαθαίνει προσαρμοστικές πολιτικές χρονοπρογραμματισμού που βελτιώνουν τη συνολική απόδοση και μειώνουν το κόστος. Μέσα από μια σειρά προσομοιώσεων, η προτεινόμενη προσέγγιση αποδεικνύεται πιο αποτελεσματική από τις παραδοσιακές μεθόδους βελτιστοποίησης και αποδίδει λύσεις πολύ κοντά σε βέλτιστες λύσεις, οι οποίες μπορούν να επιτευχθούν για μικρά στιγμιότυπα του προβλήματος μέσω μεθόδων δυναμικού προγραμματισμού. Τα αποτελέσματα εκτεταμένων πειραματικών δοκιμών δείχνουν ότι η ενισχυτική μάθηση βελτιώνει επιτυχώς την αποδοτικότητα της παραγωγής μέσω καλύτερης λήψης αποφάσεων, υποδεικνύοντας νέους τρόπους διαχείρισης των συστημάτων παραγωγής.

Thesis Committee

Professor Michail G. Lagoudakis (Supervisor)

School of Electrical and Computer Engineering

Professor Thrasyvoulos Spyropoulos

School of Electrical and Computer Engineering

Associate Professor Efstratios Ioannidis

School of Production Engineering and Management

Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this thesis.

Firstly, I would like to express my sincere gratitude to my supervising professors, Dr. Michail G. Lagoudakis and also Dr. Efstratios Ioannidis, for giving me the opportunity to work on a topic I truly enjoyed. I am grateful for their invaluable guidance, trust, and motivation throughout this thesis. Their support not only helped me navigate challenges, but also provided a solid foundation for understanding the core principles of production line optimization and reinforcement learning techniques.

I am also really grateful to Dr. Eleftherios Doitsidis, for his help and his willingness to assist and provide guidance, whenever I needed it.

Finally, I would also like to thank my family and my friends, who stood by my side and supported me in every possible way during this long journey.

Contents

CHAPTER 1: INTRODUCTION	10
1.1 Objectives of the Thesis.....	10
1.2 Contributions of the Thesis	10
1.3 Structure of the Thesis.....	11
CHAPTER 2: LITERATURE REVIEW	13
2.1 Optimization of Production Lines	13
2.1.1 Static Scheduling Approaches.....	14
2.1.2 Dynamic Optimization Techniques	15
2.2 What is Reinforcement Learning and its Applications in Industry	16
2.2.1 Use of RL in Manufacturing.....	18
2.3 Categorization of RL Algorithms	19
2.3.1 Model-Based Learning	20
2.3.2 Model-Free Learning	21
2.3.3 On-Policy vs. Off-Policy Learning	22
2.4 The Role of Hyperparameters.....	23
2.5 Policy-Based Methods vs Value-Based Methods.....	24
2.5.1 Overview of Policy-Based Method.....	24
2.5.2 Overview of Value-Based Method	25
2.6 Overview of PPO Algorithm	26
2.6.1 Actor-Critic Framework	27
2.6.2 Objective Function and Generalized Advantage Estimation (GAE)	29
CHAPTER 3: PRODUCTION LINE IMPLEMENTATION.....	31
3.1 Production Line Description and Analysis.....	31
3.1.1 Desing Philosophy and System Overview	32
3.2 SimEvents-Based Production Line Model	33
3.2.1 Product and Client Generators	34
3.2.2 Attributes and Event Actions	34
3.3 Overview of Machine States.....	36
3.3.1 Stateflow Chart Setup and MachineStatus Signal.....	36
3.3.2 State Definitions	38
3.3.3 Entity Gate Interaction with Machine States	38
3.3.4 Failure and Repair Modeling using Exponential Distributions	39
3.3.5 Transitions Between States	40
3.4 Queue Block Configuration (Buffers)	40
3.4.1 Buffer Capacities Selection.....	40
3.4.2 Queue Types in Buffer Management	41
3.4.3 Selection of FIFO	42
3.4.4 Potential Drawbacks of the other options.....	42
3.4.5 Importance of the Queue Block in SimEvents.....	42

3.5 Server Block Configuration (Machines).....	44
3.5.1 Main Settings of the Machine Server	44
3.5.2 Statistics for Monitoring and Optimization	45
3.6 Assembly Mechanism in Manufacturing.....	46
3.6.1 Assembly Logic in our Model	48
3.6.2 Composite Entity Creator Block.....	49
3.6.3 Synchronization and Deadlock Avoidance	50
3.7 Summary	50
CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION	53
4.1 Choice of Reinforcement Learning Algorithm.....	53
4.2 Transition from Simulink Model to RL Environment	54
4.2.1 Overview of the Environment Setup	54
4.2.2 Adding the RL Agent Block and Initial Configurations	55
4.3 Observation and Action Spaces	58
4.3.1 Observation Space Definition and Signals.....	58
4.3.2 Action Space Definition.....	61
4.3.3 Decoding Actions to Individual Machines.....	63
4.4 Reward Function Setup	64
4.4.1 Reward Function Logic in Simulink	64
4.4.2 Implementation of the Reward Function	65
4.5 Actor-Critic Network Architectures	69
4.5.1 Actor Network Design and Layers	69
4.5.2 Critic Network Design and Layers	70
4.5.3 Actor and Critic Representations	70
4.6 PPO Agent Configuration	72
4.6.1 PPO Hyperparameter Setup	72
4.7 Training Setup and Execution	73
4.7.1 Training Options and Parameters	73
4.7.2 Saving and Loading the Trained Agent	74
CHAPTER 5: RESULTS AND DISCUSSION	75
5.1 Introduction to the Evaluation Metrics.....	75
5.1.1 Total Reward.....	75
5.1.2 Buffer Occupancies	76
5.2 Optimal cost Baseline via Dynamic Programming.....	76
5.2.1 Simplified Model Description	76
5.2.2 Optimal Cost for Simplified Model Using Dynamic Programming Method	78
5.2.3 Training Results of the Simplified Model.....	81
5.2.4 Comparison of Agent Results with Dynamic Programming Solution	81
5.2.5 Comparison of Extracted Policies from Both Methods	82
5.3 PPO Agent Results	86
5.3.1 Total Reward Evolution Over Episodes	86
5.3.2 Buffer Management Improvement	88
5.4 Simulation of Trained Agent Across Different Scenarios	93

5.4.1 Baseline Scenario : Reproduction of Training Conditions	93
5.4.2 Scenario 1: Higher Demand.....	95
5.4.3 Scenario 2: Faster Production	95
5.4.4 Summary of Scenario Testing	96
5.5 Comparison between PPO vs A2C.....	96
5.5.1 Overview of A2C Implementation and Changes.....	97
5.5.2 Performance Comparison of PPO and A2C	97
5.6 Limitations and discussion	99
CHAPTER 6: CONCLUSIONS AND FUTURE WORK	101
6.1 Summary of Contributions	101
6.2 Future Extensions	101
REFERENCES.....	103

CHAPTER 1: INTRODUCTION

In the last few years, there has been a significant effort to make production lines work more efficiently and reduce costs, especially in the case of industrial environments where things are always evolving. As production systems get more complicated, the classic approach of control and production planning, fails to handle the case of systems with increased complexity due to significant difficulties in modeling and using the analytical tools which are essential for providing a feasible solution. To tackle these issues, intelligence-based approaches, including, but not limited to, neural networks [1], fuzzy logic [2], reinforcement learning (RL) [3], have been used to propose viable solutions. In the case of RL, production systems can adjust, while they are operational, resulting in reduced cost and enhanced efficiency.

1.1 Objectives of the Thesis

This thesis focuses on building a flexible system to improve production line performance by cutting down delays, easing buffer congestion, and reducing costs. Production lines are complex setups, where issues like changing demand, machine breakdowns, or uneven product flow can cause inefficiencies and raise expenses. Traditional fixed scheduling methods often cannot handle these changes, so there is a need to explore more adaptable and data-based approaches.

To face those challenges, this thesis aims to design and implement a Reinforcement Learning (RL) agent capable of learning and adapting in real time, without relying on predefined rules or strict mathematical models. The system is developed in a simulated manufacturing environment using SimEvents and Simulink, where the RL agent interacts with the production line, observes system states such as buffer levels and machine statuses, and learns near-optimal control policies. The main objective is to showcase that this learning-based approach can outperform classical methods in both adaptability and efficiency, showing the way for more intelligent and autonomous production systems.

1.2 Contributions of the Thesis

In this diploma thesis, we present a solution to the problem of controlling production systems by developing an RL-based solution. This solution is based on a Proximal Policy Optimization (PPO) reinforcement learning agent and it is implemented using SimEvents in Simulink. A scenario using a production line is studied, made up of a series of machine stations with different operation modes (working, idle, or out of order). There are buffers between the machines, acting as product storage entities and temporal clients. The overall goal is to produce adequate number of products to satisfy the demand, while simultaneously keep the overall cost to the minimum. The PPO-based approach considers among other, buffers levels, the status of the machines and based on the available information, it adjusts the machine's setting accordingly.

The results of the proposed approach are encouraging, and they are compared to the results of conventional optimization approaches (dynamic programming), to highlight the efficiency of our approach.

By integrating PPO into the production model, this thesis aims to demonstrate that reinforcement learning can provide a viable alternative to conventional optimization methods, especially in the case where modelling and solving the problem at hand might be difficult or in some cases impossible using the traditional methodologies. The outcomes include reduced waiting times for products and clients, optimized machine utilization, and enhanced production flow. Furthermore, this work explores the scalability of the approach and discusses how it could be applied in real-world manufacturing settings, showing the way for more efficient and adaptable production systems.

1.3 Structure of the Thesis

This thesis is organized into six main chapters, each covering a distinct aspect of the research, from background and methodology to results and conclusions, as depicted in Figure 1.1.



Figure 1.1 Structure of thesis

In Chapter 1, the problem of production line optimization is introduced and the limitations of traditional methods in handling dynamic production environments are highlighted. This chapter also outlines the objectives and motivation behind using reinforcement learning as a solution.

A review of existing literature on production line optimization and reinforcement learning is presented in **Chapter 2**. Various static and dynamic optimization approaches are discussed, followed by an overview of RL applications in industrial settings.

Chapter 3, describes the design and implementation of the production line model using SimEvents in Simulink. The key system components, including machines, buffers, and the assembly mechanism, are detailed. Configuration of every system component is being discussed.

The structure and functionality of the PPO-based RL agent, as well as the reward function and observation signals, are presented in **Chapter 4**. Actions and observations definitions are discussed. The selection of the Proximal Policy Optimization (PPO) algorithm is justified by comparing it with other RL methods. The integration of the agent with the production line model in Simulink is described in detail.

In Chapter 5 we look at the test results in detail, showing how well the system works. The analysis compares how the PPO solution performs against older optimization methods and dynamic programming approaches. Through studying how the PPO agent makes decisions, we see how it manages machine states to lower costs. The chapter also points out the limits and problems found during the research.

Chapter 6 summarizes the key contributions of the thesis and provides suggestions for future research. Potential extensions include real-world implementation, and scalability considerations.

CHAPTER 2: LITERATURE REVIEW

2.1 Optimization of Production Lines

Optimizing production lines, is crucial in manufacturing, as it directly influences a company's profitability and competitive edge. A production line usually has a series of machines or workstations, with buffers between them to hold products temporarily. A sample production line is presented in Figure 2.1.

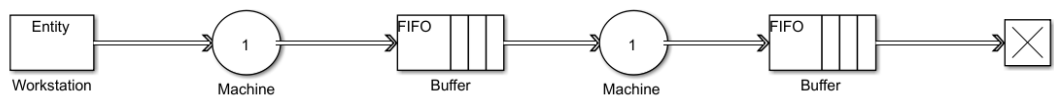


Figure 2.1 Basic Production Line Setup with Machines and Buffers.

To keep everything running smoothly and avoid problems like delays and idle machines, it's important to manage how the machines work and how products move through the line. In the past, a common way to optimize production lines was through static scheduling. This means making a set plan of time for what each machine should do, based on certain fixed rules. Static scheduling can work well, when everything goes as planned, but it has trouble adjusting, if something unexpected happens, like a machine breaking down or a sudden change in how many products are needed. As a result, static scheduling can lead to increased backorders, bigger delays in the holding spaces, and bad machine usage, which increases costs.

To deal with the problems of static scheduling, methods using dynamic optimization are been used [4]. These methods, decide in real-time based on the latest information of the system, allowing the production line to adjust as things change. Some of these methods include heuristic scheduling, model predictive control, and discrete event simulation, which help decide what tasks to assign and how to move products through the line more efficiently [5].

While these methods are often better than static scheduling, they also come with some disadvantages, as they need a lot of planning and can sometimes be hard to use in large systems. Recent progress and evolution of artificial intelligence in our days, especially in reinforcement learning (RL), could offer a reasonable alternative to optimize production lines.

RL methods are different, because they learn the best ways to make decisions by interacting directly with the system, rather than following a set plan. This ability to learn and adjust makes reinforcement learning a strong candidate to handle the challenges of complex and changing production environments. The main differences between the two methods can also be seen in Table 2.1 that follows [6].

Feature	Static Scheduling	Dynamic Scheduling
Decision Timing	Pre-planned	Real-time, based on system feedback
Adaptability	Limited	High—adjusts as conditions change
Complexity	Relatively simple	Can be complex, especially for large systems
Suitability	Works in static environments	Ideal for dynamic environments

Table 2.1 Differences between static and dynamic scheduling

2.1.1 Static Scheduling Approaches

Static scheduling has been the standard and most straightforward way to manage production lines for most of the years. These methods rely on a fixed schedule that decides when each machine should start and stop. It works by considering factors like production speed, machine capacity, waiting times, etc. For instance, in industries where the volume of products and production times remain consistent, static scheduling helps maintain a smooth workflow.

The proposed approach is problematic in handling unexpected changes that might happen. Machines can break down, people might suddenly want more or fewer products, and there can be problems getting supplies. Static schedules are pretty stiff and do not have the wiggle room to adjust when these surprises come up. As a result, things start to get delayed, products pile up in waiting areas, and machines either sit around doing nothing or work inefficiently. A well-known example of static scheduling is the job shop scheduling problem, where you allocate specific tasks to machines with predetermined processing times [7].

To provide a better explanation, a simple example of a job shop problem following a static scheduling approach is presented. Consider a small workshop with just 2 machines and 2 jobs, where everything is known in advance, as depicted in Table 2.2.

Time	Hour 1	Hour 2	Hour 3
Machine A (Cutting)	Job 1 Cut	Job 2 Cut	Idle
Machine B (Drilling)	Idle	Job 1 Drill	Job 2 Drill

Table 2.2 Simple Static Job Shop Schedule

Machine A: Cutting Machine

Machine B: Drilling Machine

Job 1: Make a Square Block

1. Cut (Machine A) - 1 hour
2. Drill (Machine B) - 1 hour

Job 2: Make a Rectangle Block

1. Cut (Machine A) - 1 hour

2. Drill (Machine B) - 1 hour.

In the problem described, three basic assumptions are made: (i) The schedule is created before starting work, (ii) No changes are made during execution, (iii) A fixed number of jobs, fixed processing times, predetermined order are considered.

This issue has been extensively studied, and techniques, such as the Earliest Due Date (EDD), Longest Processing time, FCFS or Shortest Processing Time (SPT) have been employed to develop the most efficient schedules possible [8].

But even with these techniques, the major drawback of static schedules persists. Once something disrupts the plan, the entire system begins to struggle.

Even considering the aforementioned problems, static scheduling is still used in a lot of manufacturing setups today, mostly because it is straightforward to implement and requires neither constant oversight nor significant computational resources. However, as production environments become increasingly complex and unpredictable, their limitations are becoming more apparent, emphasizing the need for more flexible methods, such as dynamic scheduling and approaches based on reinforcement learning.

2.1.2 Dynamic Optimization Techniques

The problematic behavior of static scheduling approaches in the case of unexpected changes, led to the use of dynamic optimization techniques. Unlike static scheduling, which follow a set plan, dynamic optimization adapts to unexpected changes. The system keeps track on what is happening and tweaks the schedule instantly. This means it can easily face unexpected events that could occur in such systems, like machines breaking down or changes in demand, in a more efficient way than static scheduling [9].

Dynamic optimization techniques use real-time data and information, like how machines are performing, storages levels, waiting times and others, to take the best actions. For instance, if a machine suddenly fails, a dynamic scheduling system can quickly reschedule tasks to the other machines or change the order of production to avoid delays. Or, if there is a sudden surge in demand, the system can make it a top priority, without needing someone to manually do the whole schedule.

A method for dynamic optimization is heuristic scheduling, where you apply straightforward rules or draw on past experiences to quickly come up with a solid solution. Simple rules, like "do the job that'll take the least time first," are easy to use and do not need a lot of computer power, making them perfect for quick decisions. But, these rules often do not find the best solution and can struggle in complicated or changeable environments where the rules might not fit as well.

Another technique is model predictive control (MPC), which uses a model to predict what will happen next and makes decisions based on that. MPC fine-tunes the current schedule, while considering how it might affect future production. For instance, it might hold off on a task for a bit, if it thinks that will lead to a better overall outcome. While MPC works well, it can be a heavy lift for computers and might not work as well for big production systems, where the model can get too complicated [10].

Simulation-based approaches are increasingly adopted in industrial settings, particularly when it is challenging to predict how machines and buffers will interact. In these complex environments, purely analytical models can become unwieldy or impractical, making simulation a more viable option. Simulations allow adjustments by running through different scenarios and picking the best one in real time. But, to be useful, simulations need accurate models of the production system and enough computational power to run these scenarios quickly enough to keep up with the production pace [11]. All those optimization techniques and their advantages and disadvantages are highlighted in Table 2.3 that follows.

Technique	Description	Strengths	Limitations
Heuristic Scheduling	Uses simple rules or past experiences to quickly make decisions	Fast, low computational cost	May miss optimal solutions in complex environments
Model Predictive Control	Predicts future system behavior and adjusts current tasks	Optimizes both current and future performance	High computational cost, needs accurate models
Simulation-Based approaches	Runs simulations to test different scenarios and select the best action	Effective in highly complex systems	Requires accurate models and significant computing power

Table 2.3 Comparison of Dynamic Optimization Techniques.[11]

Overall, dynamic optimization techniques are a big step up from static scheduling, because they are more flexible and can react to changes better. But they have got their own challenges, especially when it comes to using them in bigger, more complicated production setups. The need for detailed models, the heavy computer demands of making quick decisions, and the possibility of not always finding the best solutions are all things that need to be worked out.

2.2 What is Reinforcement Learning and its Applications in Industry

This is where reinforcement learning could really shine. Unlike traditional dynamic optimization methods, reinforcement learning can learn the best decisions directly through experience, without needing detailed models or simple rules. The key concept is presented in Figure 2.2. Guided by a reward function that tells the agent what is good and what is bad and by interacting with the environment (production line) over time, a reinforcement learning agent can figure out strategies that adapt to changing conditions and make things run more smoothly. This makes it an exciting new way to optimize production lines and could provide a solution that works well, even in big, complex systems, where static and traditional dynamic scheduling methods fall short.

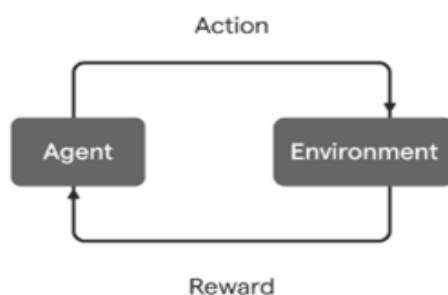


Figure 2.2 Reinforcement learning process

Reinforcement learning takes a totally different track from the usual ways of optimizing things. Instead of sticking to set schedules, using quick-and-dirty rules, or relying on models to predict what will happen next, RL learns the best ways to do things through trial and error. It is a simple, yet powerful, idea: an RL agent interacts with its environment, tries out different moves, sees what happens, and slowly gets better at making decisions based on the actual result. This process for learning from experience is what makes RL so exciting for tricky, ever-changing setups, like production lines [12].

Reinforcement learning (RL) is gaining momentum across all sorts of industries by providing a way to tackle optimization problems that the old methods just cannot handle. Some of the sectors that reinforcement learning is being widely used are shown in Figure 2.3 below.

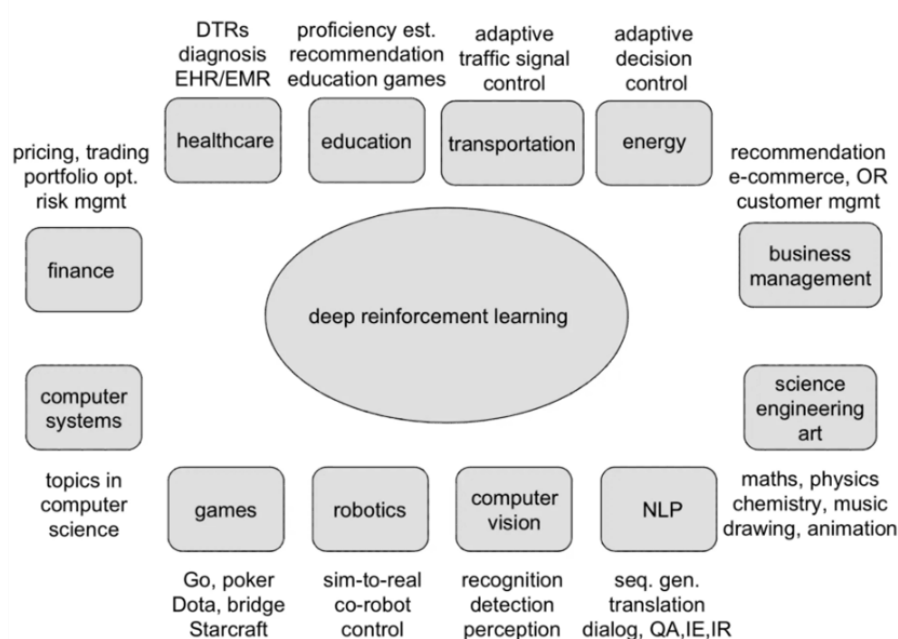


Figure 2.3 Reinforcement learning various applications [12].

By learning directly from interacting with its environment, RL can adjust to complex and ever-changing systems in ways that fixed schedules or simple rules cannot. Its ability to improve overtime makes it perfect for situations where you need to make quick decisions and conditions are constantly changing. RL has shown significant success in

robotics, where robots use sensor data to adjust their movements and perform tasks such as assembly, sorting, and material handling. [13].

In logistics and supply chain management, RL is used to figure out the best delivery routes, keep track of what is in the warehouse, and react to what customers want at any given moment [14].

Energy management systems also lean on RL to balance how efficiently things are made with saving energy, and when it comes to keeping machines in good shape, RL helps decide the best time to service them. These uses show how RL can make the most out of environments that are always changing, to reduce costs, and make things run better [15].

Applications of RL to manufacturing include fine-tuning of production lines by controlling when machines are on or off, deciding what tasks to do when, and keeping delays to a minimum, all while getting better based on what is happening right now. In the following section we will present in more detail how RL can be applied in manufacturing [12].

2.2.1 Use of RL in Manufacturing

Manufacturing systems are complex, constantly changing, and often hard to predict, which makes them a great fit for reinforcement learning (RL). Unlike the traditional approach of optimizing with set plans and simple rules, RL can keep learning and adjusting, as things change without needing external intervention. This is particularly useful in production systems, where unexpected machine breakdowns, diverse product types and constantly shifting demand can complicate operations. Over time, RL agents get better at figuring out how to make things run more smoothly, reduce on delays, and keep costs low [16].

One of the most significant ways RL is used in manufacturing, is for controlling machines and figuring out schedules. Machines on a production line can be in different states - working, idle, or being fixed - and deciding when to switch between these states can make a big difference in production systems performance. An RL agent can learn, when it is best to keep machines going, when to let them rest to save energy, or when to schedule maintenance to avoid surprises. By paying attention to what is happening in real-time, like how full the buffers are and how products are moving, the agent can tweak machine states to keep things flowing smoothly and avoid jams.

Another application for RL is in job-shop scheduling, where you have got to process different jobs on various machines in a particular order. The classic scheduling methods have a hard time adjusting to unexpected events, whereas RL systems can continuously fine-tune job assignments by learning tasks to prioritize and which machines should handle them. This helps reduce waiting times, dodge bottlenecks, and make the process more efficient [17].

Quality control is another area where RL shows much promise in manufacturing. As products go through the production line, their quality can be affected by all sorts of things, i.e. how the machines are set up, the temperature, or how long they are processed, etc. RL can help regulate these settings to ensure products meet the standards, without

needing constant manual adjustments. For example, an RL agent could learn how to adjust machine settings on the fly to get the best product quality, while keeping waste to a minimum [18].

In automated assembly lines, RL agents can manage robotic arms, conveyors, and assembly stations to make everything work together more efficiently. For instance, a robotic arm powered by RL can learn the best way to pick up, move, and put together parts while steering clear of crashes or slowdowns. As time goes on, the system gets better at dealing with different sizes and positions of parts, making it more adaptable to changes in how things are made [19].

Energy optimization is also becoming a key issue. Manufacturing uses extensive energy amounts, so figuring out how to use less, without having an effect in productivity, is a top priority. RL agents can help by learning when to turn off machines during slow times or how to balance production schedules with the available amount of energy by minimizing the cost. This not only saves money on energy, but also helps make the whole manufacturing process more sustainable [20].

RL's ability to adapt to continuous changes and learn from what is happening right now is a total game-changer for manufacturing. As production systems get more complicated, RL's role in making things run more efficiently, keeping costs down, and being more flexible, will only grow [16].

2.3 Categorization of RL Algorithms

To make decisions and improve over time, RL relies on specific learning algorithms. These algorithms help the RL agent balance exploration (trying new actions to discover better outcomes) and exploitation (using known strategies that have worked well before).

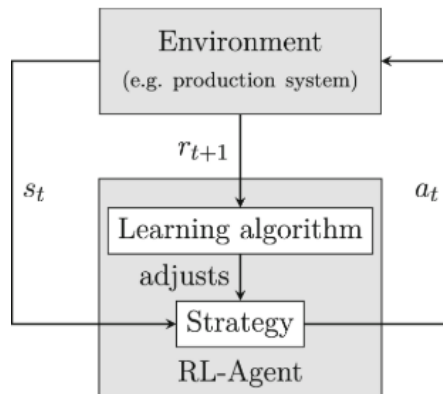


Figure 2.4 Interaction of an RL agent with an environment [21].

The interaction shown in Figure 2.4 is based on the following components.

Observation (s_t): At any given time t , the agent observes the current state s_t of the environment. In a manufacturing context, this state could include key variables like:

- The status of machines (working, idle, or failed).
- The number of products in buffers.
- Current production demands or bottlenecks.

- Energy consumption metrics.

This observation provides the information the agent needs to understand how the production line is operating at that moment.

Action Selection (a_t): Based on the observed state s_t , the agent selects an action a_t using its strategy (or policy), which is continuously refined by the learning algorithm. Examples of possible actions in a production line include:

- Turning a machine on or off.
- Prioritizing certain tasks or products.
- Rescheduling jobs to different machines.
- Adjusting machine speed or production flow.

The selected action is intended to optimize the system's performance by reducing delays, avoiding bottlenecks, or minimizing costs.

Interaction with the Environment: The selected action a_t is applied to the environment (the production system). For example, if the agent decides to switch a machine to idle mode, the environment will reflect this change by altering the machine's status and updating the production flow accordingly.

Reward Feedback (r_{t+1}): After the action is taken, the environment responds by providing feedback in the form of a numeric reward r_{t+1} . The reward tells us how good (high value) or bad (low value) the action was based on some criteria. In a production line, the reward could be based on:

- Reducing storage levels.
- Maximizing machine utilization.
- Saving energy.
- Meeting production deadlines.

Updating the Strategy: The chosen learning algorithm processes the feedback (reward r_{t+1}) and then updates the agent's strategy to improve future decision-making. During that process, the agent learns to take actions that maximize rewards in the long term by observing how the environment responds to different decisions. This process allows the agent to develop good and sometimes near-optimal scheduling and control strategies.

Reinforcement learning (RL) algorithms can be broadly categorized based on whether they rely on a model of the environment to make decisions or not. This distinction is crucial, as it affects how the agent learns, explores, and optimizes its actions. Understanding the key differences between model-based and model-free algorithms helps clarify why certain approaches work better in specific environments, such as production lines or dynamic manufacturing systems.

2.3.1 Model-Based Learning

Model-based RL algorithms, work by creating and using a model of the environment, which helps predict how the system will respond to different actions. This model acts like a virtual version of the real world, letting the agent run through and assess different

actions before trying them out. The idea is to cut down on expensive trial-and-error by "testing" decisions in the model and using what it learns to tweak its strategy [22].

As presented in Figure 2.5, in this setup, the model usually uses mathematical equations to describe the system changes, showing what happens when you move from one state to another and what rewards you might get.

Rather than just being used for simulation, the model helps to solve the Markov Decision Process (MDP) by using planning methods like value iteration or policy iteration. These methods find the best strategy (optimal policy) for making decisions. With this model, the algorithm can predict what might happen next and pick actions that lead to the highest long-term rewards.

This approach, called model-based reinforcement learning, is especially helpful in situations where making real-world decisions is expensive, dangerous, or difficult. For example, it is very useful in industries with heavy machinery or where stopping production would cause big problems.

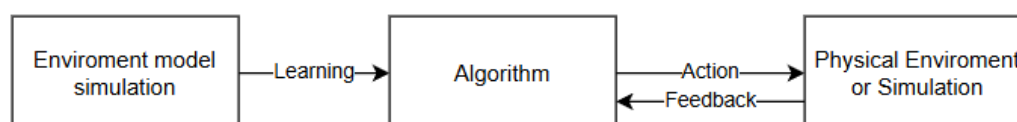


Figure 2.5 Model-Based algorithm

Note: By the term "simulation" in model-based algorithms, we refer to a mathematical process carried out by the algorithm before taking actions that are evaluated by the reward function. This process attempts to simulate the environment's response to the algorithm's upcoming actions.

The performance of the model-based RL depends on the accuracy of the actual model. If the model cannot handle the real world's complexities or unpredictability, the agent's choices might not be good or could even cause problems. Additionally, creating a good model can take a lot of computing power, which makes this method trickier to use in systems that change a lot or are big.

Model-based RL can be split into two types:

(i) *Given the model*: The way the system works is already known, and the agent uses this information to make its decisions. This works in the case where the rules are clear-cut, like in games like chess (for example, AlphaZero). (ii) *Learn the model*: The agent figures out how the system works, by playing around in the environment and collecting data. This is handy in situations where you do not have a clear model to start with or where things are too complicated to spell out ahead of time.

2.3.2 Model-Free Learning

Model-free RL algorithms do not depend on a detailed model of the environment. They learn from the interaction with the system, monitoring what actually happens after they take action, and using that feedback to improve their choices. This approach is convenient in constantly changing and hard to predict environments, where putting

together accurate models is tough or where things keep shifting. It is also useful in cases where the environment's model is so complex or large that it cannot be efficiently stored or used in practice.

Figure 2.6, shows how model-free learning focuses on figuring out the best values or policies. The agent tries out different moves, keeps track of which ones lead to better outcomes based on what it's experienced, and slowly zeroes in on the best way to do things.

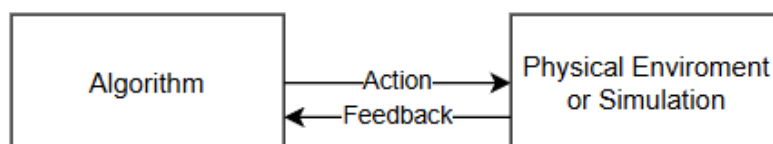


Figure 2.6 Model-Free Algorithm

The main advantage of model-free algorithms, is that you do not need to know how the system works beforehand, which makes them perfect for real-world applications, like production lines. In these setups, things like machines breaking down, products piling up in buffers, and demand going up and down can be all over the place. Getting to the point where the agent learns well can take a while and use a lot of computing power, because it needs to interact with the environment a ton to gather enough info to learn effectively [22].

2.3.3 On-Policy vs. Off-Policy Learning

Another important categorization of algorithms can be **on-policy** and **off-policy** learning algorithms. This categorization, defines how an algorithm explores its environment and learns from the gathered data, which directly impacts the stability, efficiency, and applicability of the learning process in real-world scenarios [23].

- **On-policy algorithms**, explore the environment by following a specific policy and improve that policy based only on the data collected from it. The agent interacts with the environment using the current policy, evaluates the outcomes, and adjusts its decisions to improve that same policy. This focused approach, often results in more stable training, as the agent is always refining the policy it is actively using. However, it can be less efficient, because the algorithm can only use recent interactions to make updates.
- **Off-policy algorithms** are more flexible. They explore actions not only by following the current policy, but also by collecting data from other strategies, like random actions. These algorithms evaluate and improve their policy based on a bigger range of data, which often includes historical data stored in replay buffers. This gives off-policy algorithms the ability to learn from a wider range of experiences and achieve higher efficiency.

2.4 The Role of Hyperparameters

Hyperparameters are key settings in reinforcement learning (RL) algorithms that directly influence the agent's learning process and performance. Unlike parameters learned during training, hyperparameters are *fixed* before training and determine how the agent interacts with the environment, updates its knowledge, and explores new strategies. Selecting appropriate hyperparameters is crucial to achieving efficient and stable learning. Below, we explore some of the most important hyperparameters and their impact on training performance [24].

Learning Rate (α):

The learning rate controls the size of updates to the policy or value function after each training step. It governs how quickly the agent adapts its understanding of the environment based on new observations.

- A high learning rate, allows faster adaptation, but risks overshooting a stable point, leading to unstable training.
- A low learning rate ensures more stable and gradual updates, but may result in slower learning and convergence.

In environments with highly dynamic or noisy conditions, setting a lower learning rate often helps maintain stability. However, many RL implementations use adaptive learning rate schedules, where the learning rate decreases over time to balance exploration in the early stages with fine-tuning in later stages.

Batch size:

The mini-batch size refers to the number of samples (time steps or experiences) used in each training iteration to update the policy. Mini-batches are created by splitting the experience data collected during the experience horizon.

- Small mini-batch sizes allow for faster updates, but may result in noisy gradient estimates, leading to instability in training.
- Large mini-batch sizes reduce the variance in updates by averaging over more data, leading to smoother and more stable learning.

An appropriate mini-batch size balances speed and stability. In highly variable and changing environments, bigger mini-batch sizes ensure updates capture a wide range of experiences, avoiding bias from short-term changes.

Clip Factor:

The clipping factor is another important hyperparameter, as it controls the stability of policy updates throughout the learning process. It limits how much the policy can change during each training iteration by limiting the probability ratio between the old and new policy.

- A small clipping factor keeps policy updates small, making learning slow and steady to avoid instability.
- A large clipping factor allows bigger updates, which could speed up learning but might cause instability.

In complex environments, like production lines, where sudden policy changes could occur, a small clipping factor is typically preferred to ensure smooth and steady improvements.

In conclusion, the list of hyperparameters can be big and differ based on the algorithm we are using. The important thing to keep in mind, is that we need proper tuning of these parameters to ensure that the agent can learn efficiently without becoming stuck in suboptimal strategies. In complex environments like production line optimization, these settings help ensure smooth learning while adapting to dynamic system conditions.

2.5 Policy-Based Methods vs Value-Based Methods

Reinforcement learning (RL) algorithms split into value-based and policy-based methods, each with its own pros and cons (Figure 2.7). Knowing how these methods differ helps pick the best one for optimizing production lines. This section compares those two methods in depth.

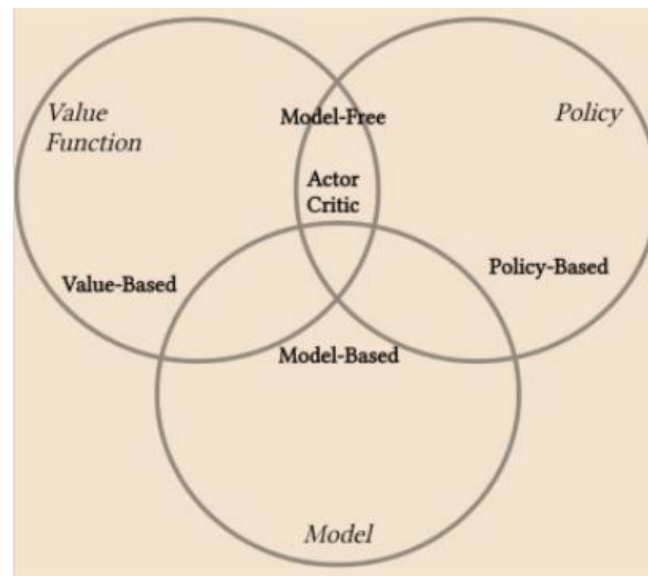


Figure 2.7 Reinforcement Learning Approaches [25].

2.5.1 Overview of Policy-Based Method

Policy-based methods learn how to pick the best action in each situation without trying to figure out how good each action is. They usually think of actions as a set of choices, with each choice having a certain chance of being picked. The aim is to tweak these chances to get the best overall results over time. These methods work well in places where there are a lot of different actions to choose from, or where the actions are not just simple yes-or-no decisions [26]. Some well-established policy-based methods are presented below:

Advantage Actor-Critic (A2C): It mixes learning, how to choose actions (actor) with guessing how good those actions are (critic). This makes the adjustments more stable and helps get to the best solution faster.

Trust Region Policy Optimization (TRPO): It keeps big changes to how actions are chosen in check, so things do not go in an unexpected manner, but it may require significant computational power.

Proximal Policy Optimization (PPO): It is a simpler way to keep changes to how actions are chosen under control, although it is not computationally intensive. It builds on what TRPO does, but makes it easier to use.

Several more algorithms can be mentioned here such as **DDPG, SAC, TD3**. These algorithms rely on off-policy updates and are often used in robotics and dynamic control tasks, due to their ability to handle complex continuous actions [27].

Policy-based methods are great for dealing with actions that need to be adjusted smoothly, like how fast machines should run or how full buffers should be. They also handle situations that keep changing well, because they focus on making the best series of choices over time.

2.5.2 Overview of Value-Based Method

Value-based methods are instrumental in evaluating the suitability of various actions or situations to facilitate decision-making processes. Instead of directly determining a good course of action, these methods employ an indirect approach by estimating a value function that surmises the potential rewards associated with specific actions in a given context [28]. The agent then picks actions that promise the best results based on what has been learned. These methodologies may include:

Q-Learning: This basic value-based RL method learns a Q-function, which shows the long-term reward for doing something in a specific situation. It updates this Q-function using the Bellman equation, but only works well, when you have a set number of choices.

Deep Q-Learning (DQN): It builds on Q-Learning by using a deep neural network to guess the Q-function, which lets it handle situations with many different or complicated details. However, continuous actions still need to be broken down into smaller pieces, which can be inefficient.

Double DQN: It makes DQN work better and more steadily by using two networks—one to pick actions and one to judge them. This helps avoid overestimating how good specific actions are, making it more reliable for demanding tasks.

Value-based methods are highly effective, when presented with a well-defined set of options. However, they encounter challenges in situations, where actions can be modified seamlessly. Breaking down continuous actions into smaller components can result in excessive complexity and hinder effective management.

In optimizing production lines that require precise and smooth adjustments, value-based methods tend to be less effective. The effort required to break down actions into manageable pieces, along with their difficulty in handling smooth adjustments, makes them less ideal than methods that focus on determining the most appropriate course of action directly.

2.6 Overview of PPO Algorithm

Proximal Policy Optimization (PPO) is a policy-based reinforcement learning algorithm that combines stability, scalability, and computational efficiency. It can handle both discrete and continuous observation and action spaces and it is a simpler, but effective, version of Trust Region Policy Optimization (TRPO), and it keeps policy updates steady, without needing a lot of computing power. This makes it perfect for real-world uses, like making production lines work better. [30]

Action Space and Observation Space Handling

In our production line optimization, discussed later, the action space includes 32 distinct actions. PPO excels in managing such action spaces by using softmax distributions, allowing the agent to choose from a large set of possible setups efficiently.

On-Policy Learning for Real-Time Adaptation

PPO works as an on-policy algorithm, meaning it updates the policy with data from the agent's latest interactions with the environment. This helps the agent adapt fast to changing conditions, like machine breakdowns or sudden shifts in product demand. Unlike off-policy algorithms such as DDPG or SAC, PPO's on-policy approach makes sure policy updates match the current system state, making it perfect for real-time optimization in dynamic settings.

Stochastic Policy for Effective Exploration

PPO uses a stochastic policy, meaning it picks actions based on probabilities instead of fixed choices. This lets the agent try out different strategies during training, avoiding getting stuck in less-than-ideal solutions. Stochastic policies work well in production lines, where testing various machine setups and strategies helps find the most efficient options.

Stable Learning with clipping mechanism

A big challenge in reinforcement learning involves preventing policy updates from becoming too large, which could cause unstable behavior or slow learning. PPO tackles this with a clipping mechanism that limits how much the policy can change at once. The clipping parameter keeps the new policy close enough to the old one. By controlling the size of updates, PPO ensures steady learning, even in unpredictable environments with sudden changes or varying rewards. This stability is important in production settings, where sudden changes in machine operations could mess up the whole process.

Keeping Exploration Going with Entropy Regularization

To maintain a balance between experimentation and maintaining established practices, PPO incorporates an entropy term into its objective function. This encourages the agent to explore fewer common actions, thereby preventing it from settling on an inferior solution prematurely. In the context of production line optimization, this enables the agent to continue seeking novel methods of controlling machinery and utilizing buffers, potentially leading to more cost-effective or efficient production processes.

Summarizing, PPO's inherited advantages for training, are depicted in Figure 2.8.



Figure 2.8 PPO agent advantages [31]

2.6.1 Actor-Critic Framework

PPO operates using an actor-critic network structure, where:

- Actor Network is responsible for generating actions based on the current policy.
- Critic Network, evaluates the expected cumulative reward (value function) from a given state, to help improve the actor's decision-making.

The actor proposes actions for each time step, while the critic provides feedback by estimating the quality of the current state or the anticipated reward based on the selected action [32]. The actor's policy is updated based on the critic's feedback, ensuring that only actions that are likely to lead to higher rewards are reinforced. This interaction between actor and critic is highlighted in Figure 2.9.

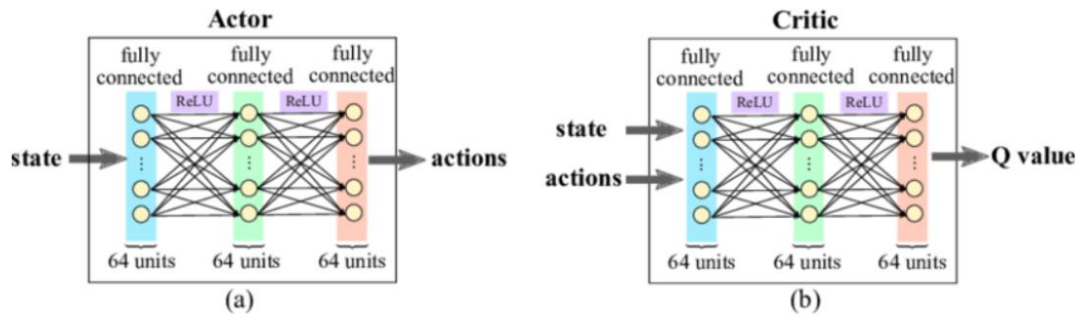


Figure 2.9 Overview of the actor-critic structure, illustrating how the actor selects actions, while the critic evaluates them to guide updates. [25].

As far as it concerns the network layer structure and functions, we have the input layer, the hidden layers and the output layer for both actor and critic networks.

Input Layer:

- Both the actor and critic networks share the same input layer, which takes in the observation vector s . This vector contains key state variables relevant to the environment.
- The input layer feeds this information into hidden layers, which learn to capture patterns and relationships between the features.

Hidden Layers:

- The actor and critic networks typically have multiple fully connected (dense) hidden layers.
- Each hidden layer is followed by an activation function (commonly ReLU) to introduce non-linearity, allowing the network to learn more complex relationships between inputs.
- These layers are responsible for learning how various features (i.e., machine status, buffer congestion etc.) and influence good action selection and expected future rewards.

To sum up the structure of a neural network is depicted in Figure 2.10:

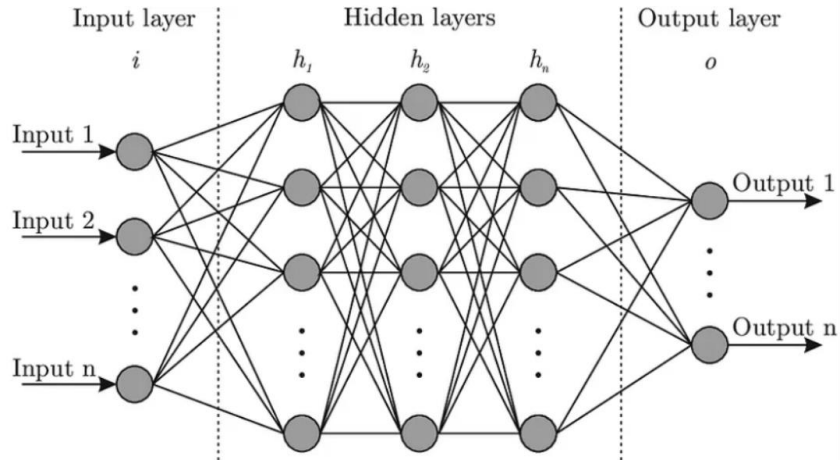


Figure 2.10 General architecture of neural networks, with input, hidden, and output layers [33]

Actor Network Output Layer:

- The actor's output layer uses a softmax activation function to produce a probability distribution over the action space [30].
- This probability distribution ensures that the network outputs a valid probability for each action, allowing the agent to select an action based on the likelihood of achieving higher rewards.

$$\pi(a | s) = \text{softmax}(W_{out} \cdot h_{hidden})$$

where:

- W_{out} is the output weight matrix of the actor network.
- h_{hidden} is the activation vector from the final hidden layer.
- softmax function transforms the output scores into a probability distribution over actions.

Critic Network Output Layer:

- The critic outputs a **single scalar value** $V(s)$, representing the estimated cumulative reward from the given state s .
- This value helps guide the actor's policy updates by comparing the actual rewards received with the expected value.

$$V(s) = W_{out} \cdot h_{hidden}$$

To sum up, adding hidden layers helps the model understand complex connections between different production factors. The actor's output layer makes sure the agent picks the best action while encouraging exploration through probability-based choices. The critic's output layer gives the actor reliable feedback to improve the policy, keeping the agent from getting stuck in poor choices. This balanced mix of layered learning allows PPO to handle the changing and complex environment of a production line, leading to smart decisions and smooth learning updates.

2.6.2 Objective Function and Generalized Advantage Estimation (GAE)

The key aspect of PPO lies in its objective function, which aims to maximize the expected reward, while preventing large, destabilizing policy updates [34].

The standard PPO objective can be written as :

$$L(\theta) = E[\min(r(\theta) \cdot A, \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \cdot A)]$$

Where:

- θ represents the policy parameters.
- $\pi_{\theta}(a | s)$ is the probability of taking action a in state s under the current policy.
- $\pi_{\theta_{old}}(a | s)$ is the same, but under the previous version of the policy.
- $r(\theta) = \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)}$ is the probability ratio of the new and old policies.
- A is the advantage estimate
- ϵ is the clip factor

The clipping mechanism is the fundamental component of PPO. By clipping the probability ratio $r(\theta)$, the algorithm restricts the magnitude of policy updates. This mechanism guarantees stable and controlled learning, thereby preventing overshoots during the training process.

PPO often relies on Generalized Advantage Estimation (GAE) to compute the advantage term A . GAE finds a balance between bias and variance, making the policy updates more reliable. The advantage is defined as the difference between the actual return and the estimated value:

$$A_t = \delta_t + (\gamma\lambda)\delta_t + 1 + \dots$$

Where:

- δ_t is the temporal difference (TD) error, capturing the discrepancy between the estimated value and the actual reward received.
- γ is the discount factor for future rewards.
- λ controls the trade-off between bias and variance.

This interaction is schematically presented in Figure 2.11 :

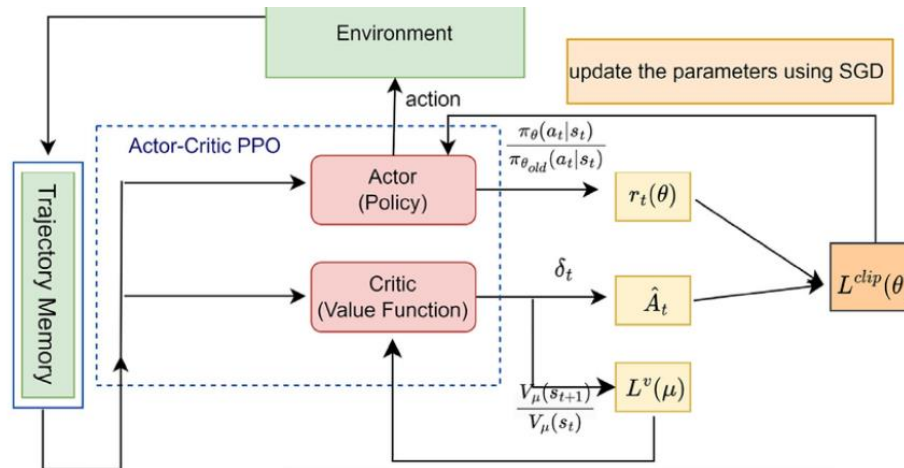


Figure 2.11 Flow of the Proximal Policy Optimization (PPO) algorithm, illustrating the interaction between the actor-critic networks, environment, and policy updates using the clipped objective [35].

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

3.1 Production Line Description and Analysis

The central focus of this thesis is the design and optimization of a production line model. The production line serves as the foundation of the system, where raw materials undergo sequential processing by machines, products temporarily reside in buffers, and client orders are fulfilled through an assembly process. The primary objective of this production line model is to simulate a realistic manufacturing process and investigate the application of reinforcement learning (RL) in enhancing its operational efficiency.

To achieve this, SimEvents in Simulink was chosen as the modeling environment, due to its discrete-event simulation capabilities and integration with MATLAB, which allows seamless implementation of control algorithms [36].

Traditional simulation tools can be cumbersome when modeling complex systems like production lines. They often require extensive customization to manage various events, such as machines starting or stopping, products moving through buffers, or real-time decision-making. SimEvents, however, offers a distinct approach. Designed specifically for event-driven modeling, it is the perfect choice for capturing the details of production lines. SimEvents simplifies modeling machine operations, buffer usage, product movement, and real-time decisions, eliminating the challenges of traditional simulation tools.

The choice of SimEvents has the significant advantages that are depicted in Figure 3.1.

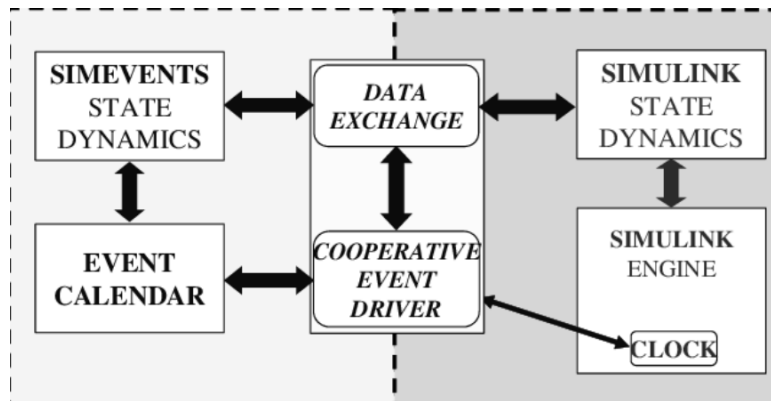


Figure 3.1 Synchronization and data exchange between SimEvents' discrete-event dynamics and Simulink's continuous-time simulation for hybrid system modeling [37].

Discrete-Event Modeling: SimEvents models systems, where changes occur at discrete points in time, such as product arrivals or machine breakdowns, which aligns with the nature of production lines.

Scalability: The modular design of SimEvents models facilitates the addition or modification of machines, buffers, and assembly mechanisms, without disrupting the overall system.

Pre-Built Components: SimEvents offers pre-built components for queues, servers, and entity routing, thereby expediting the development process and ensuring optimal simulation performance.

Visualization and Debugging: Simulink’s user-friendly graphical interface facilitates straightforward debugging and analysis of product movement through the production process.

Integration with MATLAB: SimEvents facilitates seamless interaction with MATLAB-based control logic, facilitating the direct implementation of the Proximal Policy Optimization (PPO) agent for dynamic scheduling and optimization. This functionality is particularly advantageous, as MATLAB provides pre-built reinforcement learning agents for numerous algorithms. Consequently, Simevents offers a time-saving solution.

3.1.1 Desing Philosophy and System Overview

The production line model was designed with ease of use, scalability, and adaptability in mind. It was structured into distinct components, including machines and buffers, and the sequential movement of products through the line. Additionally, it incorporates realistic production challenges, such as bottlenecks, machine failures, and varying job durations. This approach serves as the foundation for our production line simulation, as depicted in Figure 3.2 [2].

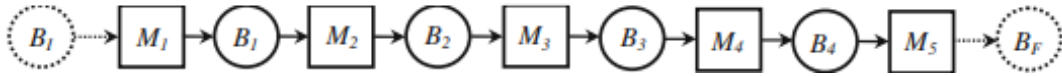


Figure 3.2 Production Line without assembly [20].

Key Design Principles:

Modularity:

We have designed each machine and buffer as individual components, allowing for easy setup and expansion. This modular approach enables us to add more machines or larger buffers without the need to restart the entire process.

Scalability:

The system can handle production setups, without considering their size as a problem.

Event-Driven Simulation:

We used SimEvents to run the simulation based on independent events. When a product comes in, a job finishes, or a machine breaks, the system updates itself. This is a realistic approach considering the limitations and functionalities of a real production floor.

Handling Uncertainty:

Real production lines encounter unexpected challenges, such as varying product arrivals and machine breakdowns. To simulate these scenarios, we introduced randomness into the model. This includes varying product arrival times and the probability of machine failures. By incorporating these elements, we can assess how the system would respond to the fluctuations inherent in actual production.

A set of specific system components was used from the blocks of the SimEvents library, as they are depicted in Figure 3.3, for setting up a sample production line.

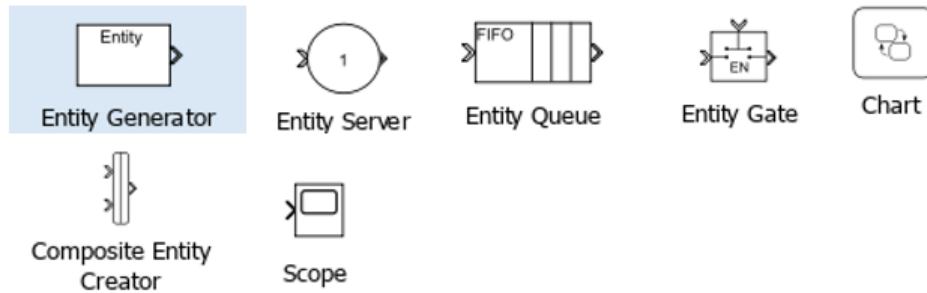


Figure 3.3 SimEvents Blocks used for production line model simulation

The **Entity Generator** block is used to simulate products and clients (entities) arrival into the system at adjustable rates.

The **Entity Server** is utilized to represent the machines of the production line. Each machine corresponds to a processing stage, where products undergo specific tasks, before proceeding to the subsequent stage. Machines function based on predefined processing times and can transition between states, such as operational, idle, and malfunctioning.

The **Entity Queue** is used to represent buffers. These are placed between machines to store products temporarily, when downstream machines are unavailable. They are also used to store waiting clients.

The **Entity Gate** is used to prevent entities flowing, when the upstream machine is not in working state. We use the machine status signal of the machine to either block or allow products (entities) to flow.

The **Stateflow chart** is used to implement the control logic and transitions between the states of each machine.

The **Composite Entity Creator** is used to represent the assembly logic. This block takes two different entities (products and clients) and creates a new one with the attributes of both. Here, we match clients and products to create sales.

The **Scope** is used for the visualization of the outputs of the simulation. It is used to observe different type of signals, such as buffer capacities and machine states.

3.2 SimEvents-Based Production Line Model

The production line model was implemented using SimEvents in Simulink, which provides a robust framework for modelling discrete-event systems. By treating production processes as sequences of events - such as product arrivals, machine processing, and failures - the system dynamically simulates real-world manufacturing behavior.

This section details the key SimEvents components, their configuration, and their roles in simulating product flow and production performance.

3.2.1 Product and Client Generators

The simulation commences with the generation of products and client orders. Product generators introduce raw materials into the system, simulating the supply of production inputs. Similarly, client generators introduce demand by generating orders from clients at predefined rates. Both generators are configurable, enabling the simulation of diverse production conditions, such as consistent demand or fluctuating orders.

One key parameter is the generation rate, the time between consecutive product generations and also the rate of clients being generated [38]. These two play a pivotal role, as they are directly responsible for the quantity of products and clients that will traverse the production line.

For our case, we have chosen the following rates:

The client generation rate is determined using an exponential distribution. This is commonly used in modeling interarrival times, where the time between the arrival of two consecutive clients follows a probabilistic distribution that reflects realistic variability.

As shown in Figure 3.4, $\mu I = 4.5$ represents the mean interarrival time between two consecutive clients. This is the average time, in simulation units (e.g., seconds), between client arrivals, whereas $dt = -\mu I * \log(1 - \text{rand}())$ generates the actual interarrival time (dt) for each client using an exponential distribution. That means that clients do not arrive at fixed intervals, but instead at varying intervals that average to 4.5 time units. This configuration is depicted in Figure 3.4.

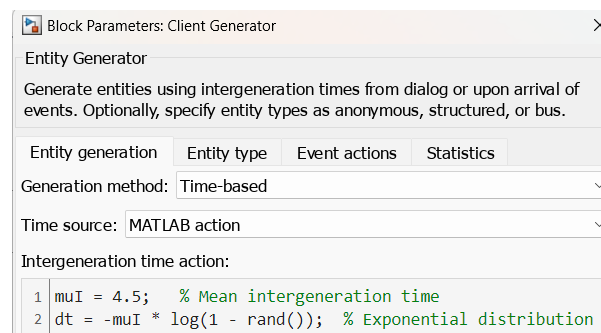


Figure 3.4 Client generator block configuration

For the case of products, one product is generated every 2.7 time units, therefore creating a steady input flow. Unlike the client generator (which uses a probabilistic distribution), this generator provides a constant and predictable product supply, useful for modeling stable production conditions.

3.2.2 Attributes and Event Actions

In SimEvents, upon the creation of an entity, the user has the ability to give certain attributes, using the entity generator block via the “Entity Type” tab in the block [39].

The products of the production line at hand have two attributes (Figure 3.5), namely processing time and product number.

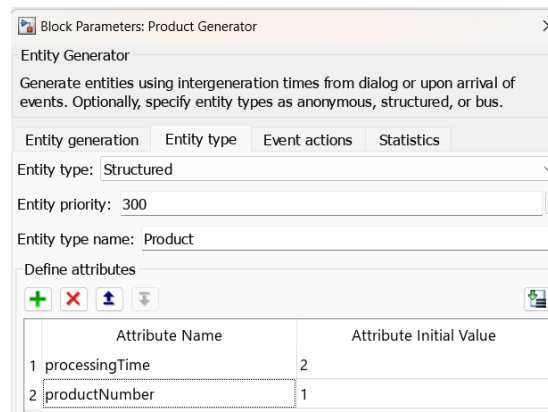


Figure 3.5 Product Attributes

The processing time encompasses the duration a product spends traversing each machine in its production cycle, from initial input to final output and subsequent movement to the next processing station.

To assign realistic variability to the processing times, each product is given a random duration drawn from an exponential distribution, which is commonly used to model time between events in production systems.

Specifically, for a given average processing time μ , the processing time T for a product is computed as: $T = -\mu \ln(1-u)$

where u is a uniformly distributed random number in the interval $(0,1)$ and \ln denotes the natural logarithm.

Using the “Event actions” (Figure 3.6), we can initialize or give random values to those attributes.

This is analogous to the client generation rate. Each product generated will be assigned a random processing time with an average value of 2. For instance, if a product has a processing time of 2.5, it implies that for our production line comprising five machines, it will require $5 \times 2.5 = 12.5$ time units to traverse through all the machines.

This task is straightforward to implement, using a server block which has a service time source parameter. We simply enter the attribute of each product, allowing each product to spend a different amount of time being processed compared to other products, making the simulation even more realistic.

The second attribute, “productNumber,” is utilized for monitoring specific entities. This functionality proved instrumental in verifying the correctness of the assembly logic introduced later (Figure 3.6).

```
%assign productNumber attribute to every
persistent productNumber
if isempty(productNumber)
    productNumber = 0;
end

% Assign productNumber attribute
entity.productNumber = productNumber + 1;
productNumber = productNumber + 1;
```

Figure 3.6 Assignment of product attribute using event actions

We initiate the process by assigning “productValue=1” to the initial product and subsequently incrementing the value for each subsequent product. Consequently, the first product will have “productValue=1,” the second product will have “productValue=2,” and so on.

3.3 Overview of Machine States

Each one of the five machines in our production line has three possible states. The working state, the idle state and the failed state.

The working state is when the machine is actively processing products, the idle state is when the machine is available, but currently not processing products, and the failed state is when a machine has encountered a failure and needs to be repaired.

3.3.1 Stateflow Chart Setup and MachineStatus Signal

The most efficient and straightforward method to represent the states was by employing stateflow chart blocks. The logic employed for all machines was identical. We defined the three states and their transitions. The default state that every machine should commence in is the working state (indicated by the blue arrow). The setup is depicted in Figure 3.7.

Based on the state of the machine, we update the machineStatus signal. This signal serves two crucial purposes. Firstly, it will be subsequently provided to the agent as an observation signal. Secondly, it is utilized in conjunction with the entity gate block to enable or disable the flow of products through the production line, based on whether the upstream machine is operational or not.

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

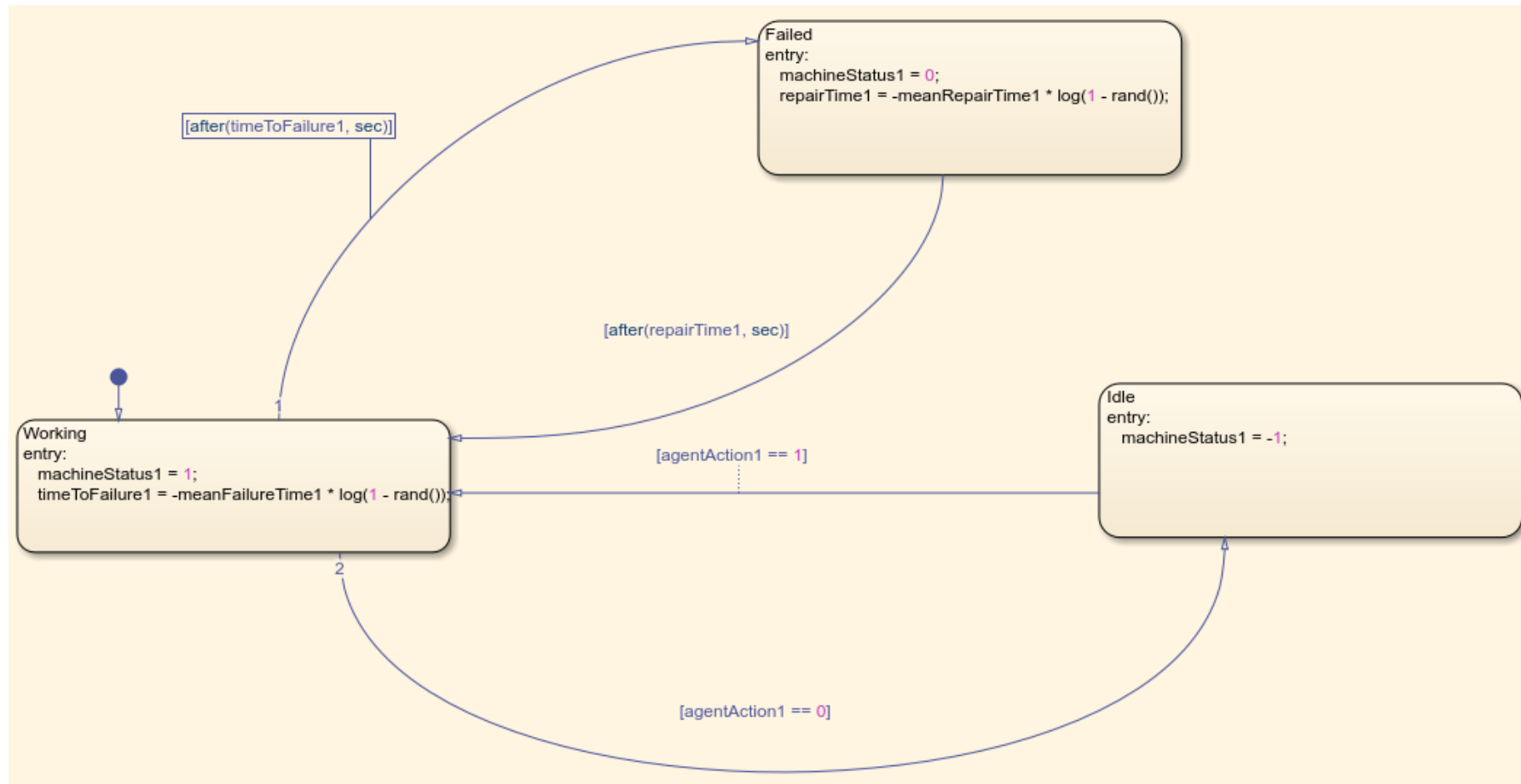


Figure 3.7 Machine states definitions using stateflow chart

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

3.3.2 State Definitions

According to Figure 3.7 we have defined three different states:

1. Working State:

- The machine is actively processing products.
- Entry action: $\text{machineStatus} = 1$ (indicates the machine is working).
- The transition to failure is governed by an exponential distribution using $\text{timeToFailure} = -\text{meanFailureTime} * \log(1 - \text{rand}())$, simulating real-world stochastic failures.

2. Failed State:

- The machine has encountered a failure and is undergoing repairs.
- Entry action: $\text{machineStatus} = 0$ (indicates the machine is not operational).
- Repair time is also calculated using an exponential distribution:
 $\text{repairTime} = -\text{meanRepairTime} * \log(1 - \text{rand}())$. Once the repair is complete, the machine transitions back to Working.

3. Idle State:

- The machine is powered off or not actively processing any products.
- Entry action: $\text{machineStatus} = -1$ (indicates the machine is idle).
- Transitions between the Idle and Working states depend on the control action from the RL agent.

3.3.3 Entity Gate Interaction with Machine States

Overall the interaction between different machine states, is handled through an Entity Gate interaction mechanism, using switches, as the one depicted in Figure 3.8.



Figure 3.8 Gate block acting as a switch

The procedure is executed and managed according to the rationale outlined in Figure 3.9. When the signal arriving into a switch attains the desired value, it closes and permits the flow of products. Conversely, when the signal does not possess the desired value, the gate remains open, preventing any product flow [40].

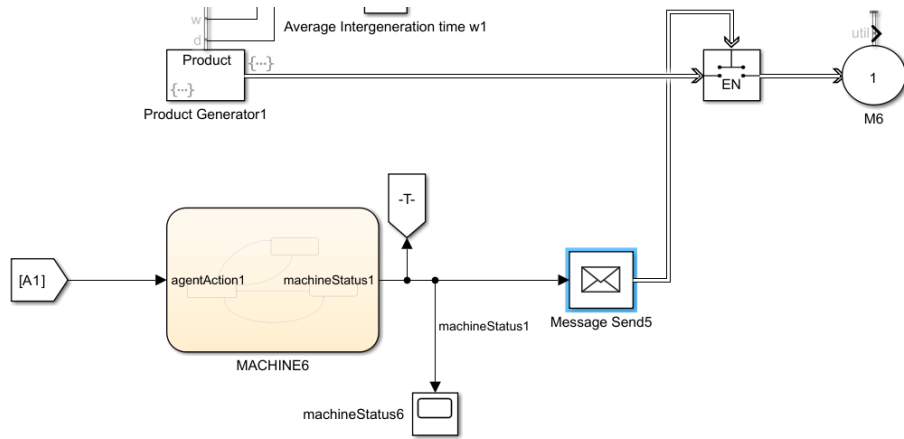


Figure 3.9 Entity Gate managing products flow based on machine Status signal

Specifically, if a machine is in a working state ($\text{machineStatus}=1$), the gate permits the movement of products. Conversely, if the machine is not in a working state ($\text{machineStatus}=0$ or $\text{machineStatus}=-1$), the gate is closed, and no products can flow until the upstream machine achieves a working state. This behavior is determined by the parameterization of the enable gate block, which allows entities to pass only when it receives a positive value ($\text{machineStatus}=1$).

3.3.4 Failure and Repair Modeling using Exponential Distributions

In the production line model, failures and repairs are modeled using exponential distributions. The idea behind this is that the time between failures and repair durations are often random, but follow a memoryless property, making exponential distributions ideal.[40].

Failure Time(timeToFailure):

- Formula: $\text{timeToFailure} = -\text{meanFailureTime} * \log(1 - \text{rand}())$
- The mean time to failure time (MTTF) defines the average time between machine failures.
- An exponentially distributed variable is used to model the random nature of failures, ensuring that some machines fail sooner or later than others.

Similarly, repair time(repairTime):

- Formula: $\text{repairTime} = -\text{meanRepairTime} * \log(1 - \text{rand}())$
- The mean time to repair (MTTR) defines the average time needed to fix a machine.
- Exponential distribution ensures that repair durations vary, reflecting real-world conditions, where some repairs take longer than expected.

3.3.5 Transitions Between States

We will later cover this more analytically, once we have introduced the reinforcement learning agent, but here is a brief description of how the transitions between states occur:

As mentioned, the agent we will later present takes the following possible actions:

Based on observed signals, it will either decide if a machine should be put to working state (1), or if a machine should be put to idle state(1). Also

From Working → Failed:

- Triggered by the condition **[after(timeToFailure, sec)]**, indicating that the predefined failure time has elapsed.

From Failed → Working:

- Triggered by the condition **[after(repairTime, sec)]**, representing the completion of the repair.

From Working → Idle:

- Triggered by the RL agent's action **[agentAction == 0]**, meaning the agent has decided to turn the machine off to optimize performance or reduce energy consumption.

From Idle → Working:

- Triggered by the RL agent's action **[agentAction == 1]**, meaning the agent has decided to power the machine on to resume production.

Self-Loop Transitions:

We can see that the machine remains in the current state, until the respective condition (failure, repair, or control action) triggers a transition.

This setup models real-world production scenarios, where machine performance can be unpredictable, and adaptive decision-making is crucial for efficient system operation.

3.4 Queue Block Configuration (Buffers)

In any production line, buffers play a critical role in ensuring smooth product flow by temporarily storing products, when downstream machines are busy or unavailable. The capacity of each buffer must be set to a value chosen, after considering the balance between preventing production bottlenecks and minimizing excess storage costs. In our model, we have set the buffers capacities to the value of 50. The choice for those capacities is explained below.

3.4.1 Buffer Capacities Selection

One of the primary reasons for this choice is that a buffer capacity of 50 units provides sufficient space to temporarily store products during peak production periods or in cases

where a downstream machine encounters delays or failures. This mitigates the risk of upstream machines becoming blocked and entering idle states.

Another crucial factor is cost minimization. While larger buffers offer greater flexibility, they also lead to increased storage and handling expenses. A buffer size of 50 represents a strategic trade-off, ensuring adequate flexibility, while avoiding excessive costs.

Finally, this capacity was determined through simulation experiments and trials, demonstrating its effectiveness in balancing demand fluctuations and processing variations, while minimizing the risk of frequent overflows.

3.4.2 Queue Types in Buffer Management

In SimEvents, the Queue block provides different queue management options that determine how products are stored and processed within the buffer. These options are listed under the Queue Type selection, as shown in *Figure 3.10* and each method has distinct characteristics that impact production line performance.

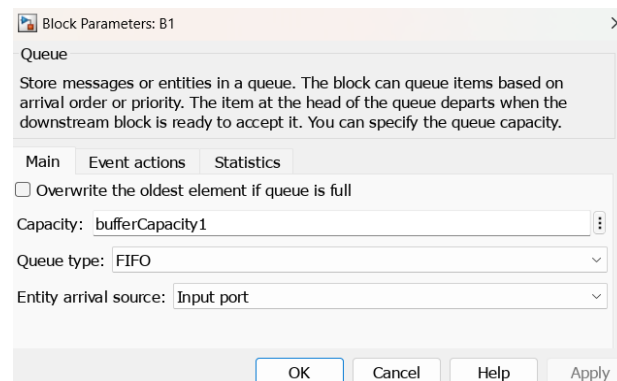


Figure 3.10 Configuration of Queue Block

Available queue types are shown in Figure 3.11:

FIFO (First-In, First-Out):

- The first product to enter the queue is the first to leave it.
- Products are processed in the order they arrive, maintaining the sequence of production[41].

LIFO (Last-In, First-Out):

- The most recently added product is the first to be processed.
- This approach prioritizes the newest arrivals while older products wait longer.

Priority-Based Queuing:

- Products are prioritized based on custom-defined criteria (e.g., priority levels or urgency).
- Higher-priority products are processed first, regardless of arrival order.

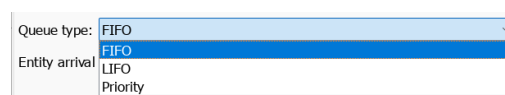


Figure 3.11 Queue Types in Simevents

3.4.3 Selection of FIFO

As Figure 3.12 suggests, when using FIFO, products are enqueued at the back and served from the front in arrival order.

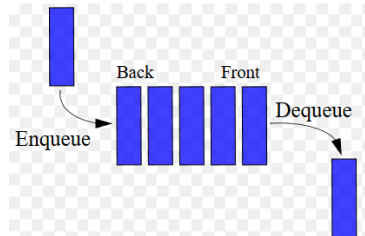


Figure 3.12 Illustration of the FIFO (First-In, First-Out) queue mechanism. Products are enqueued at the back and served from the front in arrival order. [41]

Maintains production order: The FIFO (First-In, First-Out) method ensures products get processed in the order they arrive, supporting a smooth and steady production flow. This proves especially important when the order of processing affects product quality and consistency.

Prevents products deterioration: FIFO minimizes the time products spend in the buffer, reducing the risk of degradation or quality loss. This is especially important in industries, where product freshness is an important factor.

Simple and efficient implementation: FIFO is easy to set up and uses minimal computing power compared to other complex scheduling methods. It delivers a predictable product flow without requiring complicated rules.

Ensures equitable resource allocation: By handling items based on the order they arrive, FIFO avoids long delays for certain products and promotes fair resource access. This helps prevent starvation, a frequent problem in LIFO-based systems.

3.4.4 Potential Drawbacks of the other options

- **LIFO:** While LIFO can be useful in scenarios where the most recent arrivals need immediate processing, it could cause significant delays for older products, leading to inefficiencies and quality issues [42].
- **Priority Queues:** Priority-Based systems are beneficial when certain products require urgent processing, but they introduce complexity and may lead to starvation of lower-priority items, if not carefully managed.

By choosing FIFO, the system ensures that production runs smoothly, without unnecessary complexity, making it easier for the reinforcement learning agent to monitor and manage buffer levels efficiently.

3.4.5 Importance of the Queue Block in SimEvents

The Queue block in SimEvents represents the buffer, where entities (products or clients) are stored temporarily. It is critical to the simulation, as it models real-time product accumulation, overflow, and flow regulation between machines. This block offers several key outputs through its statistics tab, which can be used for performance

monitoring and decision-making. Figure 3.13 shows the available outputs Simevents offer.

Main	Event actions	Statistics
<input type="checkbox"/>		Number of entities departed, d
<input checked="" type="checkbox"/>		Number of entities in block, n
<input type="checkbox"/>		Average wait, w
<input type="checkbox"/>		Average queue length, l
<input type="checkbox"/>		Number of entities extracted, ex

Figure 3.13 Statistics tab

- 1) **Number of entities departed (d):** Keeps track of how many entities (products or clients) have passed through the buffer, providing insights into production throughput.
- 2) **Number of entities in block (n):** Tracks the current number of entities waiting in the buffer. This signal is particularly important, because it is used as an observation signal for our reinforcement learning (RL) agent to assess buffer congestion and adjust machine states accordingly. For example, in Figure 3.14 below we can see the products flow in the first buffer and how that queue statistic signal help us monitor products and clients in our line.

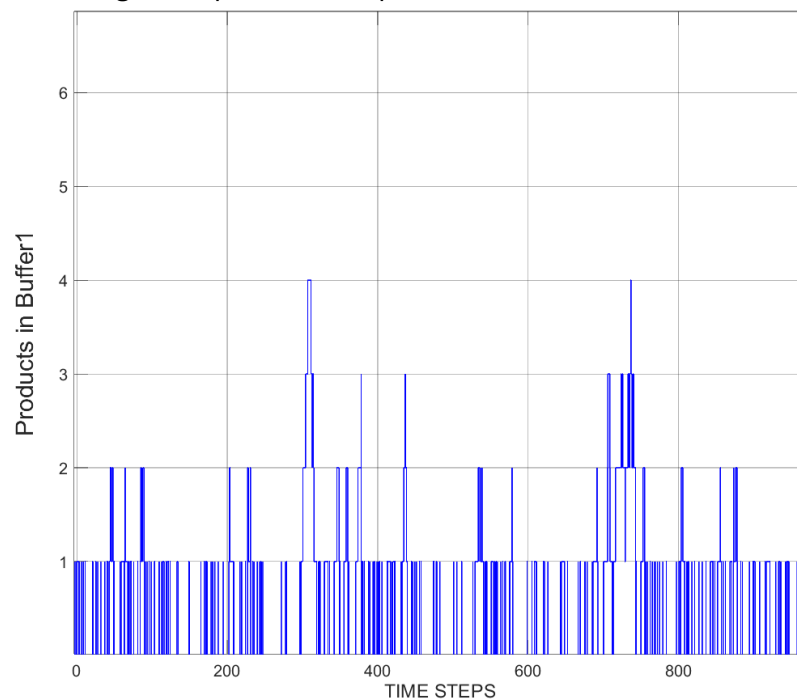


Figure 3.14 Products flow in buffer 1

- 3) **Average wait time (w):** Represents the average time an entity spends in the buffer, which is useful for analyzing delays and system efficiency.
- 4) **Average queue length (l):** Shows how often the buffer operates near capacity, helping in evaluating whether buffer sizes are adequate.

3.5 Server Block Configuration (Machines)

The server block is a crucial component of the production line model, representing the processing stations, where entities (products) undergo procession. The configuration of this block directly affects the production flow, machine utilization, and system efficiency. This section details the key parameters, functionality, and their role in optimizing machine behavior.

3.5.1 Main Settings of the Machine Server

The server block, as all of the blocks in SimEvents, has parameters someone can change and experiment with. Here, we will discuss those parameters and the parametrization we chose and the reasons behind those choices. The parameters of server block are shown in Figure 3.15 below:

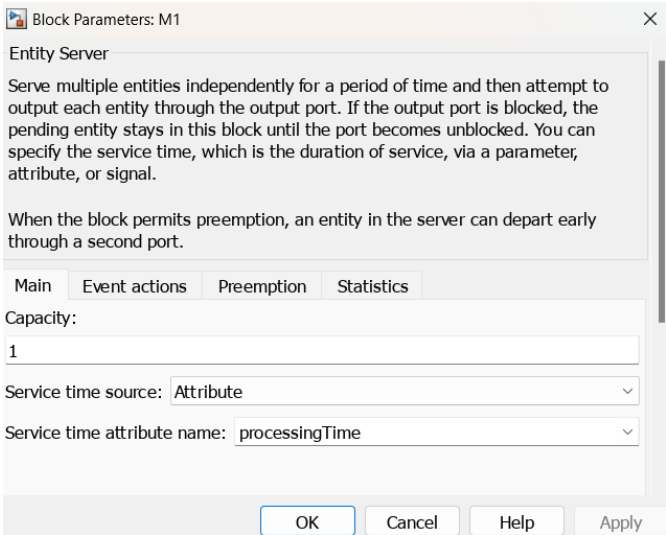


Figure 3.15 Server block parameters

Capacity:

The server capacity is set to **1**, meaning each machine processes one product at a time. This is ideal for ensuring sequential processing, which is common in many real-world production lines.

For scenarios where parallel processing is possible, the capacity could be increased to reflect multi-product handling capabilities, but that is not our case.

Service Time Source:

The service time source setting in the server block shown in Figure 3.16 defines how the processing time for each entity (product) is done. This configuration is important for accurately simulating varying processing durations. SimEvents provides the following options for specifying service time:

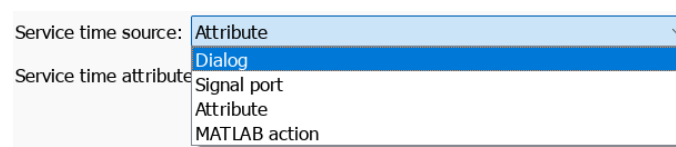


Figure 3.16 Service Time Source Options

1) **Dialog:**

With that option, service time is fixed and specified directly within the block's parameter settings. This option is best for systems, where processing times are constant or predetermined (e.g. each product always takes exactly 2 seconds to process).

2) **Signal Port:**

In that case, the service time is provided dynamically through an external signal connected to the block. This setting could be useful, when service times are influenced by external conditions or real-time inputs, such as machine temperature or product-specific attributes.

3) **Attribute:**

Here, the service time is defined as an attribute of the entity entering the server. Each entity carries its own service time, which can vary based on a distribution (e.g., exponential or normal).

The reason we chose this option is because by using the attribute `processingTime`, we ensure that each entity carries its unique processing duration, allowing for a realistic representation of variability in the system.

4) **Matlab action:**

The service time is calculated using a MATLAB script or function that is executed during simulation. Could be useful for scenarios, where the service time depends on complex calculations or multiple factors beyond simple distributions.

3.5.2 Statistics for Monitoring and Optimization

Similar to the queue block, the statistics tab provides key performance indicators related to machine operation, allowing real-time monitoring and optimization through feedback loops. Those statistics are presented in Figure 3.17, as follows:

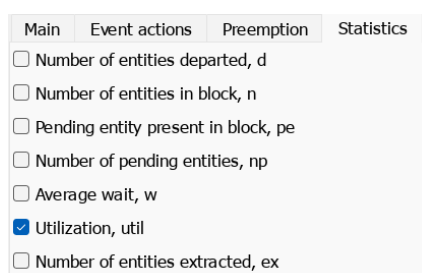


Figure 3.17 Statistics tab for server block

- **Utilization, util:** Tracks how often the machine is actively processing products versus being idle. This metric is crucial for identifying underused or overworked machines. Also used to get a grasp of how effective our system is.
- **Pending entity present in block, pe:** Indicates whether any product is currently waiting for processing within the machine server.

- Average wait, w : Used to track the average duration that entities spend waiting in the server block before being processed. This value is important for identifying potential bottlenecks in the production line.

3.6 Assembly Mechanism in Manufacturing

The assembly process is a fundamental part of manufacturing systems, where individual components, subassemblies, and parts are combined to create a finished product. It is a crucial stage in many production lines, directly impacting efficiency, product quality, and overall manufacturing success.

Assembly processes can vary in complexity, ranging from manual assembly, performed by workers, to fully automated systems that utilize robotics, conveyor belts, and advanced machinery [43].

A critical aspect of the assembly process is ensuring that all required components arrive at the assembly station on time. Any delays or mismatches can disrupt the workflow, leading to production bottlenecks, increased idle time, and reduced overall output. For example, in an automotive manufacturing facility, such as the one depicted in Figure 3.18, components, like the engine, transmission, and electronic systems, must be delivered precisely when needed. This synchronization is essential for maintaining a seamless production flow and minimizing downtime.



Figure 3.18 Assembly production line in a car factory [44]

The assembly process is not just about physical parts, though. In industries focused on services or logistics, "assembly" can also mean matching different entities, like pairing products with client orders or combining production tasks. These kinds of assembly processes need good scheduling and resource management to balance what is available with what is needed and get the best performance overall.

When designing an effective assembly system, there are several things to keep in mind:

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

- **Resource Coordination:** Making sure all the necessary parts or entities are at the assembly station at the same time.
- **Buffer and Inventory Management:** Keeping buffer sizes and inventory to the right level to avoid running out or having too much.
- **Assembly Speed and Efficiency:** Minimizing assembly time while preserving high product quality.
- **Error Detection and Rework:** Having ways to spot and fix defects during or after assembly to keep up the production standards.

In modern manufacturing, automated assembly transforms production by boosting efficiency and cutting costs. Advanced technologies, such as robotic arms (Figure 3.19), automated guided vehicles (AGVs), and programmable logic controllers (PLCs), increases operational speed and reduces the need for human involvement.



Figure 3.19 Robotic Assembly Line [45].

Plus, those smart scheduling systems are effective in cutting down on downtime and ramping up production.

In conclusion, the assembly process serves as a critical link in the production chain, ensuring the seamless integration of all necessary components, as illustrated in Figure 3.20, to manufacture the final product. Its efficiency directly impacts key factors, such as production speed, cost-effectiveness, and the overall quality of the finished product.

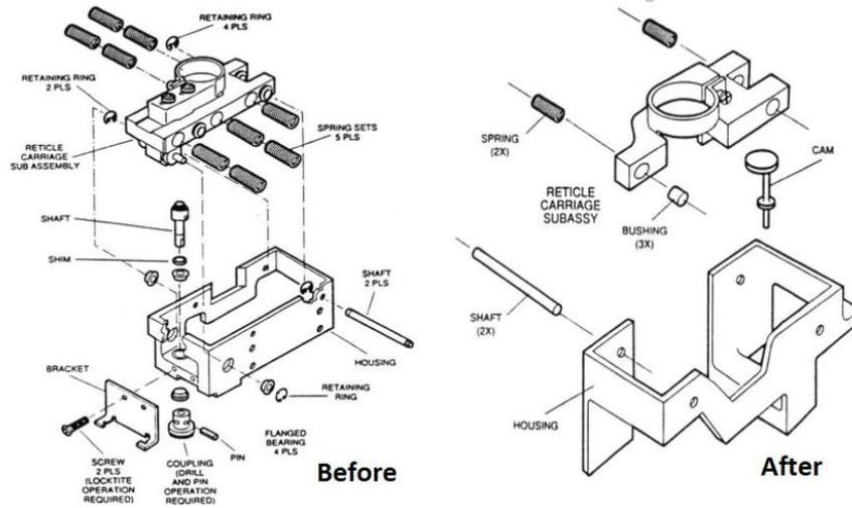


Figure 3.20 Before and After Assembly [46].

3.6.1 Assembly Logic in our Model

In our production line model, the assembly mechanism is utilized to match waiting clients with processed products to generate orders. Specifically, two distinct entities—clients and products—are combined to form a new entity: an order. The creation of an order requires the simultaneous availability of both a processed product and a ready (waiting) client. This ensures that each product leaving the production line is immediately assigned to a client, signifying a completed order. The logic behind this product-client matching is critical for the efficient operation of the system, as any imbalance between product supply and client availability may result in delays, idle machinery, or buffer congestion.

To implement this logic, we relied on the Composite Entity Creator block from SimEvents. Figure 3.21 that follows shows the assembly mechanism implemented in our production line model.

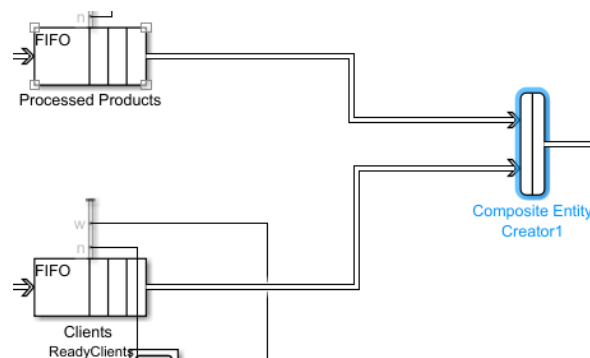


Figure 3.21 Assembly process in our production line

This block combines entities arriving simultaneously from its input ports into a single composite entity. Specifically, processed products from the upstream buffer and waiting clients from their respective buffer arrive at the Composite Entity Creator. When both

entities are available, the block merges them into an "Order" entity, which represents a successful product-client match.

The matching logic ensures that products are not left waiting indefinitely in the production line and that clients are not underserved. By coupling the flow of products and clients, the system maintains a steady production pace, while optimizing resource utilization and minimizing bottlenecks.

So, at the end, we have the new entity "Orders", that includes the attributes of both combined entities coming out of the system.

This matching mechanism mirrors real-world manufacturing systems, where production output must be aligned with customer demands to minimize waste and inefficiencies.

3.6.2 Composite Entity Creator Block

As previously discussed and illustrated in Figure 3.22, the **Composite Entity Creator Block** in **SimEvents** was utilized to simulate the assembly process.

The configuration of this block was straightforward. It functions by receiving entities from multiple input ports and merging them into a single composite entity. In our setup, these input ports correspond to:

- **Port 1:** Processed products from the production line
- **Port 2:** Ready clients awaiting product allocation

Each time an entity arrives at both input ports simultaneously, the block combines them into a composite entity labeled "**Order**". This newly formed entity retains attributes from both the product and the client, facilitating downstream operations, such as order tracking and performance analysis.

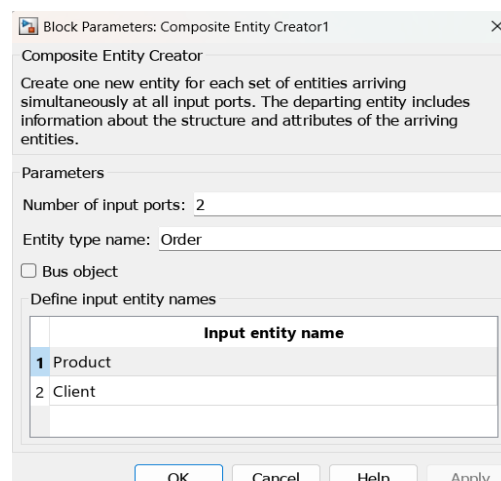


Figure 3.22 Composite Entity Creator Block Configuration

3.6.3 Synchronization and Deadlock Avoidance

In most production systems, where multiple products must be matched, ensuring synchronization without introducing bottlenecks is crucial [47]. In our model, synchronization is achieved through the **Composite Entity Creator Block**, which guarantees that an order (composite entity) is generated only when both a processed product and a ready client arrive simultaneously in their respective buffers. However, careful consideration is required to prevent scenarios, where one type of entity—either products or clients—remains idle indefinitely, leading to system inefficiencies.

To maintain synchronization, while minimizing the risk of deadlock, the following strategies are implemented:

1. **Balanced Generation Rates:** The production rates of both clients and products are carefully calibrated to sustain a steady flow and prevent significant mismatches.
2. **FIFO Buffers:** As depicted in Figure 3.26, both clients and processed products are temporarily stored in queue blocks, before being sent to the Composite Entity Creator Block. These buffers operate on a **First-In, First-Out (FIFO)** basis, ensuring that entities are processed in their arrival sequence, thereby maintaining order and preventing disruptions.

3.7 Summary

In this chapter, we analyzed the design and implementation of the production line model using SimEvents in Simulink, covering its major components and configuration decisions. This includes the generation of products and clients, the use of state machines for machine behavior, buffer and queue configurations, and the integration of an assembly mechanism to create product-client orders. All this can be shown in Figure 3.23, where our whole production line SimEvents model is presented.

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

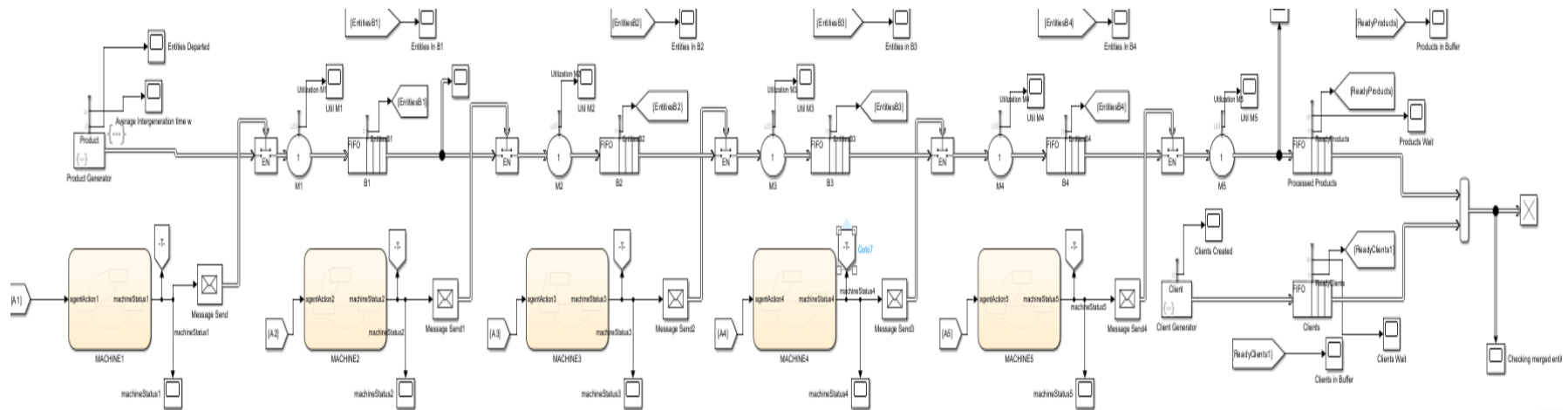


Figure 3.23 Production line implementation in SimEvents

CHAPTER 3: PRODUCTION LINE IMPLEMENTATION

The main takeaway message is that every part of the production line creates a dynamic and ever-changing environment for the reinforcement learning (RL) agent. It faces real challenges, like shifting buffer levels, unexpected machine breakdowns, and the unpredictable timing of product and client arrivals. The integration of RL provides the opportunity to make real-time decisions, such as when to switch machines on or off, to optimize performance.

The following key observations and signals were identified as critical for the RL agent's decision-making:

- **Machine Status Signals:** These are generated through the stateflow charts and reflect whether each machine is working, idle, or failed. They determine when machines can process products and when upstream product flow should be halted.
- **Buffer Levels:** The levels of entities in buffers serve as indicators of congestion or underutilization. Managing inventory levels is crucial to avoiding bottlenecks in production.

Additionally, aspects, such as mean time to failure (MTTF) and mean time to repair (MTTR), introduce stochastic behaviors that are difficult to optimize using traditional approaches. This dynamic and unpredictable environment showcases the importance of adaptive learning through reinforcement learning.

CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION

4.1 Choice of Reinforcement Learning Algorithm

Before proceeding with anything else, the first step is to selecting the appropriate reinforcement learning approach . Given the nature of production lines and our objectives for near-optimal operation, we opted for policy-based methods rather than value-based ones [29]. The main reasons are summarized below:

A Variety of Actions: In a production line, there are numerous decisions to be made regarding the best functioning of machines. Value-based methods, like Q-Learning or DQN, need to categorize these options systematically and monitor their performance in every situation. This might be challenging, since there is a variety of choices. Policy-based methods primarily concentrate on determining a good course of action, thereby facilitating and expediting the process, even in the presence of numerous options.

Making Choices Directly: Policy-based methods are capable of immediately selecting the most suitable action for each situation, eliminating the need for prior estimation of the potential benefits of each action. This is of paramount importance in the case of a production line, in situations where rapid and precise decisions are required regarding the operation of machines and the status of their buffers. As a result, the overall performance is smoother and more efficient.

Handling Changes: Production systems often face sudden issues, like machine breakdowns or shifts in demand. Policy-based methods adjust quickly by using real-time data. Value-based methods, however, may need a lot of recalculating when changes happen, making them less flexible.

Trying New Things: Policy-based methods, especially those using entropy regularization, keep exploring new actions to find better solutions. This has an impact in production settings where testing different processes is key. Value-based methods might stick to weaker solutions too soon, especially when there are many options, because they do not explore as much.

Keeping Things Stable: Value-based methods are more appropriate and work fine for smaller problems but can become trickier in bigger problems due to overestimating certain actions. Policy-based methods, like PPO with its clipping factor, manage updates carefully, making them more reliable for large production setups.

Based on these points, policy-based methods seem better for improving production line operations. They handle a wider range of options, easily adapt to changes. They boost efficiency, scale well, and remain steady, outperforming value-based methods.

So after selecting policy-based methods, we have to choose the specific reinforcement learning algorithm we are going to use.

We selected the Proximal Policy Optimization (PPO) algorithm for our problem for the following reasons:

Stability in Policy Updates: The clipping mechanism ensures that PPO does not take large steps that could destabilize the policy, which is crucial in dynamic environments.

Effective in Large Action Spaces: PPO is capable of handling large, discrete action spaces, making it ideal for production systems with multiple machines requiring simultaneous decisions.

Scalability: PPO can scale well to complex systems, which is beneficial for future extensions beyond simulation

4.2 Transition from Simulink Model to RL Environment

In the earlier chapters, we looked at how the production line model was built and run using SimEvents in Simulink. This model acts as a changing environment where reinforcement learning can take place. Now, the next step is to connect this model with a reinforcement learning (RL) agent that can help make better decisions in real time [48].

This transition includes capturing observations from the production line model and providing them as inputs to an RL agent, which learns to make fairly good control decisions. In this chapter, we will focus on developing the Proximal Policy Optimization (PPO) agent, the design of its observation space using the key signals identified, and the reward function that is used for the learning process. By using PPO, we aim to show how the RL agent can learn smart strategies that help manage buffer congestion, cut down costs, and make the production line more efficient.

The reinforcement learning agent is designed to learn a good (and possibly even optimal) policy for managing machine states and minimizing production costs. These costs primarily accumulate due to buffer overflows and increased waiting times. Unlike static or heuristic-based scheduling approaches, the RL agent continuously interacts with the system, dynamically observing its state and making data-driven decisions to maximize long-term rewards.

4.2.1 Overview of the Environment Setup

To implement reinforcement learning within the production line system, we developed an environment using the `rlSimulinkEnv` function in MATLAB, which establishes a connection between the dynamic Simulink model and the PPO agent. The production line model, introduced in Chapter 3, serves as the interactive environment, where the reinforcement learning (RL) agent engages with machine states and buffer conditions.

In reinforcement learning, the environment provides state observations to the agent, receives action inputs, and returns rewards based on system performance. The key components of this setup are as follows:

- **Observation Space:** The environment continuously monitors real-time system data, including machine statuses and buffer levels, which collectively define the

system's current state. These observations are critical for enabling the agent to make informed decisions.

- **Action Space:** The agent is granted control over the operational status of each machine (active or idle) and can regulate product flow accordingly. The available actions are represented as binary vectors, as elaborated in Section 4.3.2.
- **Reward Signal:** The reward function assesses system performance, following each action executed by the agent. Key factors influencing the reward include waiting clients, waiting processed products, and unit buffer costs (further detailed in **Section 4.4**). These signals guide the agent's learning process, encouraging decisions that optimize production efficiency.

The Simulink-based environment dynamically changes, as the agent interacts with it. For example, when an action is taken to switch a machine to the idle state, downstream product movement is affected, which in turn influences the reward calculation and the observation signal at the next time step.

The Simulink environment is connected to the RL framework by specifying the model, observation space, action space, and the block where the agent receives and processes data, using the following code appeared in Figure 4.1:

```
%% Step 1: Define the Environment
mdl = 'productionLine';
agentBlk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl, agentBlk, obsInfo, actInfo);
```

Figure 4.1 Environment and reinforcement learning framework connection

This is exactly where our connection between our model and our agent happens.

ObsInfo, actInfo and agentBlk are all later described in this chapter.

Note: 'productionLine' is the exact name of our Simulink model. This must match exactly in order for the agent to be connected to our environment

4.2.2 Adding the RL Agent Block and Initial Configurations

To insert the RL agent into the production line model, we used the RL Agent block in Simulink, which acts as an interface between the Simulink environment and the PPO agent. The RL Agent block shown in Figure 4.2 is where the agent receives observation signals, sends actions, and collects rewards.

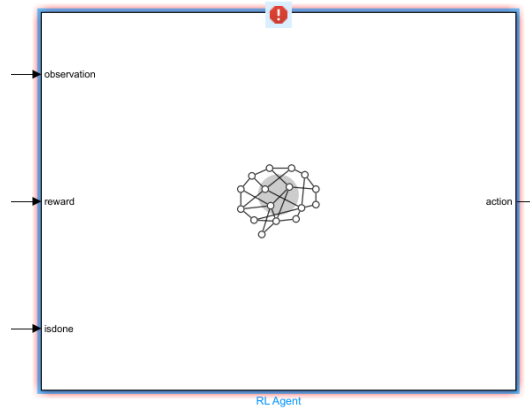


Figure 4.2 RL agent block in Simulink

The RL Agent block is connected to the Simulink model at critical points, where observations and actions are exchanged. Specifically, it is linked to the following signals:

- Observation Signals: Machine statuses, buffer levels.
- Action Signals: The block outputs control decisions (whether machines should be working or idle) to the production line system.
- Reward Signal: The reward from the environment is fed back into the block to guide learning.

The “IsDone” signal serves as a termination condition, indicating when an episode should end, such as when a maximum time step is reached or when a specific reward threshold is met. This mechanism ensures that the agent focuses its learning within meaningful episode boundaries.

In the following sections, we will cover how these observation and action signals are defined, encoded, and decoded within the Simulink environment, enabling seamless interaction between the agent and the production line.

As every Simulink block, RL agent block has its own parameters that can be configured. (Figure 4.3).

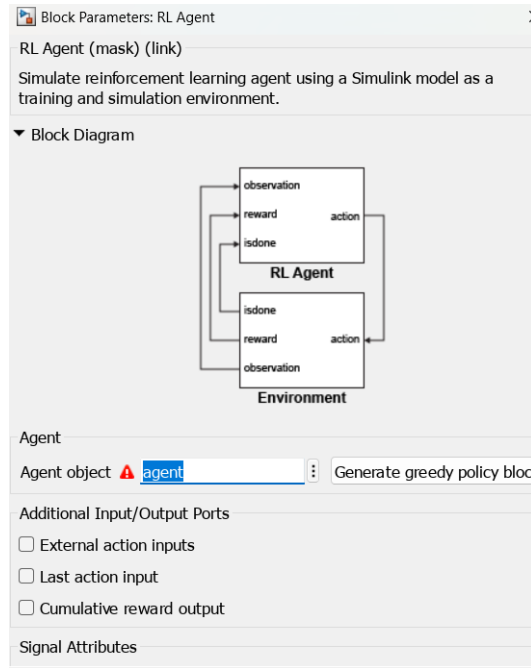


Figure 4.3 RL Agent block parameters

The RL Agent Block requires precise configuration to ensure seamless communication between the PPO agent and the production line model. The key configuration parameters include:

- **Agent Name (RL Agent Block Name):** The name assigned to the RL Agent Block in Simulink (e.g., *RL Agent*, as depicted at the bottom of Figure 4.4) must exactly match the name specified in the MATLAB script when defining the Simulink environment.

For instance, in the MATLAB script, when creating the Simulink environment using `rlSimulinkEnv`, the variable `agentBlk` should explicitly reference this block name to establish a correct linkage between the RL agent and the model. This integration is illustrated in Figure 4.4.

```
agentBlk = [mdl '/RL Agent'];
```

Figure 4.4 Agent block connection to environment

- **Agent Object:** The *Agent Object* field inside the block specifies the variable name of the agent created in MATLAB (e.g., *agent*).

In our case, this refers to the PPO agent defined using the `rlPPOAgent` function, as shown in Figure 4.5. This variable contains the agent's network, configurations, and learning parameters [49].

```
% Create PPO Agent
agent = rlPPOAgent(actor, critic, agentOpts);
```

Figure 4.5 rlPPOAgent Matlab function

The block also allows for additional settings like external action inputs or cumulative reward outputs, which can be toggled based on the specific needs of the system.

4.3 Observation and Action Spaces

To ensure proper communication between the PPO agent and the production line model, we need to define two essential components:

1. **Observation Space:** The signals sent to the RL agent that reflect the current state of the production line.
2. **Action Space:** The set of actions the agent can take to control the production system.

4.3.1 Observation Space Definition and Signals

In our case, there are 11 observation signals. These include machine status signals for each of the five machines and buffer level signals for the six buffers in the model.

Specifically, we utilize the statistic "number of entities in block, n" from the queue blocks. This signal represents the number of entities (products or clients) currently present in each buffer, allowing us to monitor waiting clients, waiting processed products, and the exact number of entities in each buffer at any given time.

The observation signals are as follows:

- **6 buffer levels:** These include the four intermediate buffers and the two final buffers, which store processed products and waiting clients. The value range from 0 to 50 (maximum buffer capacity).
- **5 machine status signals:** Each machine status signal can take one of the following values:
 - -1 (Failed)
 - 0 (Idle)
 - 1 (Working)

These signals form the observations vector for the agent.

As a result the total number of possible observations in the environment is: $51^6 * 3^5 = 17$ billion distinct states. Because the number of possible observations is so huge it is impossible for the algorithm to check and learn from every single one. Instead of trying to explore them all, we use tools like neural networks to help the agent recognize patterns and learn how to act, even from just a small amount of experience.

Before being fed to the agent, the observations are processed using the `gatherObservations` function that we implemented. The `gatherObservations` function, presented in Figure 4.6, is responsible for collecting and normalizing key signals from the production line, before passing them to the RL agent.

This normalization is particularly important, when using actor-critic architectures, as it ensures that all observation values fall within a common range. This, in turn, enhances training stability and improves the convergence of the reinforcement learning algorithm.

CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION AND INTEGRATION

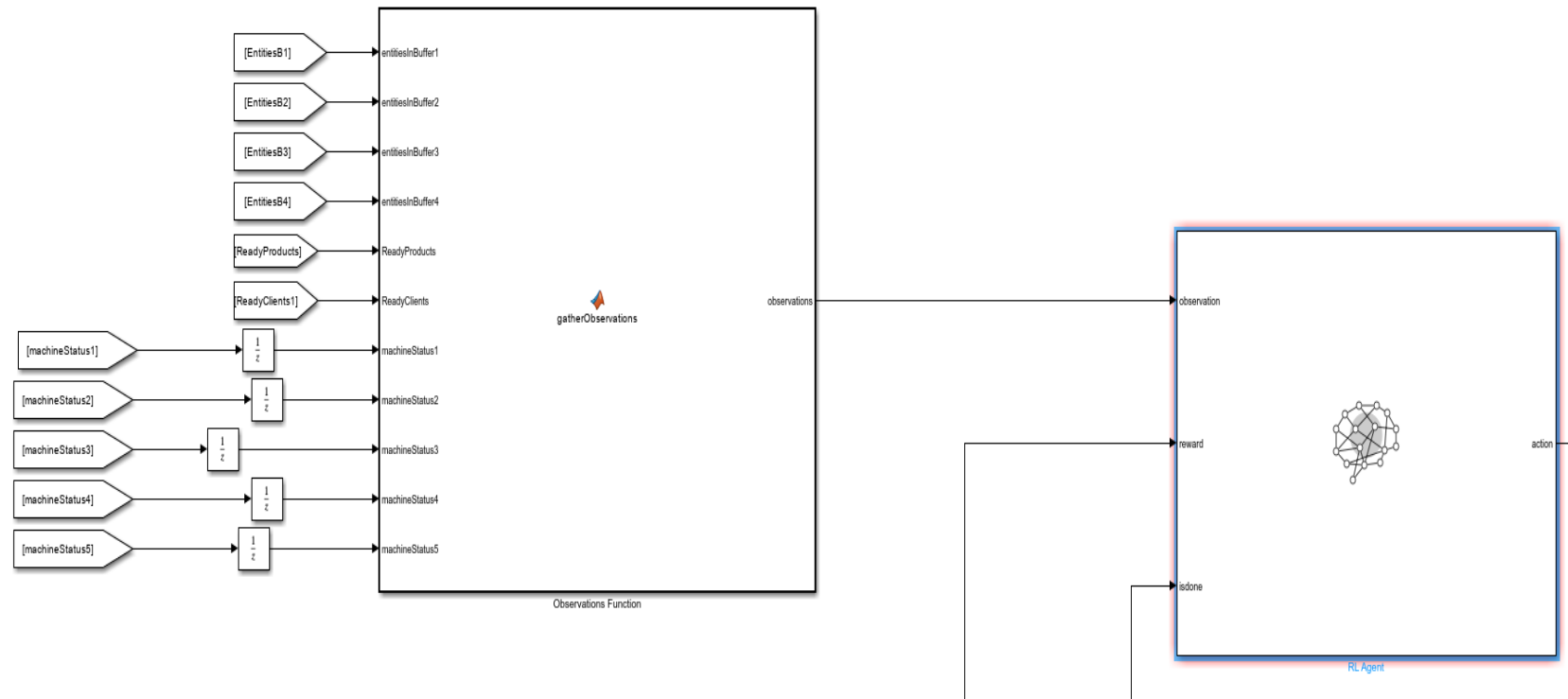


Figure 4.6 Observation signals fed into the agent via `gatherObservations` function

CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION AND INTEGRATION

The code for that function is presented in Figure 4.7 below and its pretty straightforward:

```
function observations = gatherObservations(entitiesInBuffer1, entitiesInBuffer2, entitiesInBuffer3, ...
                                         entitiesInBuffer4, ReadyProducts, ReadyClients, ...
                                         machineStatus1, machineStatus2, machineStatus3, ...
                                         machineStatus4, machineStatus5)

    % Define maximum buffer capacity
    maxBufferCapacity = 50;

    % Normalize buffer levels
    bufferLevels = [entitiesInBuffer1; entitiesInBuffer2; entitiesInBuffer3; entitiesInBuffer4] / maxBufferCapacity;

    % Normalize ready products and clients
    readyStates = [ReadyProducts; ReadyClients] / maxBufferCapacity;

    % Normalize machine states to [0, 1]
    machineStates = [machineStatus1; machineStatus2; machineStatus3; machineStatus4; machineStatus5];

    % Combine normalized observations into a single vector
    observations = [bufferLevels; readyStates; machineStates];
end
```

Figure 4.7 *gatherObservations* custom Matlab function

Each buffer level is divided by the maximum capacity to obtain a normalized value between 0 (empty) and 1 (full).

Similar to buffer levels, the ready products and clients are normalized between 0 and 1.

The machine states are not normalized, since their values are limited compared to buffer levels.

Finally, all normalized signals are concatenated into a single observation vector of dimension **11**, matching the specifications defined by the observation space (obsInfo).

Now that these elements have been configured in our model, it is essential to ensure that the signals are correctly interpreted and fed into the agent. This is where the **rlNumericSpec** is used.

rlNumericSpec is a MATLAB function used to define the observation space in reinforcement learning environments. It specifies the structure, range, and characteristics of the numerical observations that the agent receives from the environment.

The **rlNumericSpec** function ensures that:

- The dimensionality of the observation vector is correctly defined.
- The values of the observations remain within specified limits.
- The agent correctly interprets the meaning and constraints of each observation dimension.

So, we used this function in the following way as shown in Figure 4.8:

```
% Define observation specifications
obsInfo = rlNumericSpec([11, 1], ...
    'LowerLimit', [0; 0; 0; 0; 0; 0; -1; -1; -1; -1; -1], ...
    'UpperLimit', [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1]);
obsInfo.Name = 'Production Line States';
```

Figure 4.8 *rlNumericSpec* usage and initialization

The first argument defines the dimension of the observation vector, [11, 1] indicates that the observation space is a column vector with 11 elements.

LowerLimit: Specifies the minimum allowable value for each observation dimension. After normalization, it is obvious that the first 6 signals (buffer levels) are going zero as the lowest possible value and the machine statuses are going to have -1.

Similarly, upper limit specifies the maximum allowable value for each observation dimension. First 6 signals are going to have maximum possible value of 1(if buffer is full) and machine statuses are going to have maximum value of 1(working state).

After all, we have ensured observations are correctly defined and properly connected with our agent and we are ready to move to the actions of the agent.

4.3.2 Action Space Definition

The action space represents how the agent controls the production line. For our problem, the agent needs to decide whether each machine should be put to working or idle at any given time step, based on the observations signals.

The action space is structured as a **binary decision space** for five machines, meaning that each machine has two possible actions:

0: Put the machine in the idle state.

1: Keep the machine in the working state.

In reinforcement learning, the action space defines the possible actions the agent can take at each step. For our production line model, the action space is discrete, because the agent needs to make binary decisions (on/off) for each of the five machines. To handle this discrete nature, we use **rlFiniteSetSpec**, which is specifically designed for environments, where the action space consists of a finite set of discrete actions.

rlFiniteSetSpec specifies a finite set of allowable actions, therefore is ideal when the agent's actions belong to a fixed set like ours. Actions definitions using **rlFiniteSetSpec** are shown in Figure 4.9:

```
%% Define Action Specifications Using dec2bin
actions = dec2bin(0:31, 5) - '0';
actInfo = rlFiniteSetSpec(mat2cell(actions, ones(1, 32), 5));
actInfo.Name = 'MachineActions';
```

Figure 4.9 Actions Definition

To gain a deeper understanding of how actions are defined and distributed across the machines in our production line, we consider the following approach:

The MATLAB command `dec2bin(0:31, 5)` generates the binary representation of integers ranging from 0 to 31. Since our system consists of five machines, each action is encoded as a 5-bit binary string, where each bit represents the operational state of a corresponding machine.

For example:

CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION AND INTEGRATION

- 00000 represents turning all machines off.
- 11111 represents keeping all machines on.
- 10101 represents machines 1,3,5 on and machines 2,4 off.

The expression - '0' converts the binary strings into numeric arrays:

So are actions would look like this at that point:

[0 0 0 0 0; 0 0 0 0 1; 0 0 0 1 0; ... 1 1 1 1 1]

Then we use : “mat2cell(actions, ones(1, 32), 5)”

This splits the 32×5 action matrix into 32 individual rows, each representing a separate action .

The function **mat2cell** takes two arguments:

- ones(1, 32) specifies that each row of the matrix should be split into its own cell.
- 5 indicates that each action configuration has 5 binary decisions (one per machine).

The result is a cell array where each cell contains a 1×5 action vector.

actInfo = rlFiniteSetSpec(mat2cell(actions, ones(1, 32), 5));

The rlFiniteSetSpec function takes the cell array of discrete actions as input and defines the agent’s action space.

Agent Output Example

So to summarize, there are 32 possible combinations of actions. The agent selects one of these 32 combinations at each time step.

When the agent selects an action during a time step, it outputs an index corresponding to one of the 32 possible actions. For example, suppose the agent selects **Action Index 21**. Using our action space mapping, we get:

dec2bin(21, 5) → [1 0 1 0 1]

This means:

Machine 1: ON (1)

Machine 2: OFF (0)

Machine 3: ON (1)

Machine 4: OFF (0)

Machine 5: ON (1)

4.3.3 Decoding Actions to Individual Machines

This raises an important question: How do we distribute these actions to the individual machines, given that they are encoded as a 5-bit binary vector?

The solution is straightforward: we must decode the binary vector and assign each bit to the corresponding machine's control signal.

To achieve this, we developed a custom MATLAB function designed to efficiently distribute the actions. The setup of this function is presented in Figure 4.10.

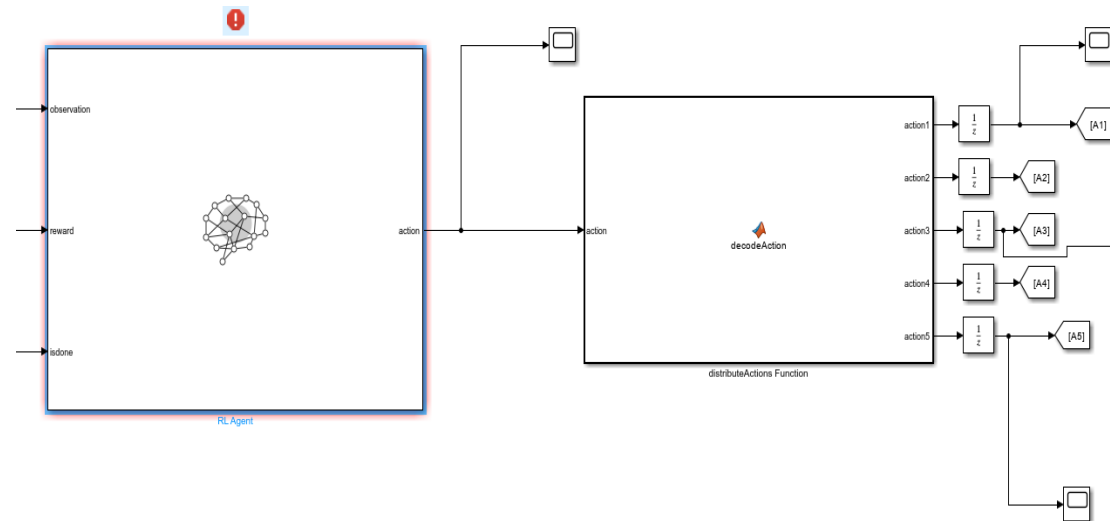


Figure 4.10 Actions get decoded and distributed to each machine

The code of the function is presented in figure 4.11 that follows.

```
function [action1, action2, action3, action4, action5] = decodeAction(action)
% Assign each binary digit to the respective machine's action
action1 = action(1);
action2 = action(2);
action3 = action(3);
action4 = action(4);
action5 = action(5);
end
```

Figure 4.11 decodeAction Function

Suppose the agent output is the action vector [1 0 1 0 1]. The function decodeAction will distribute this vector as follows:

```
[action1, action2, action3, action4, action5] = decodeAction([1 0 1 0 1]);
```

action1 = 1 → Machine 1 ON

action2 = 0 → Machine 2 OFF

action3 = 1 → Machine 3 ON

action4 = 0 → Machine 4 OFF

action5 = 1 → Machine 5 ON

So, now we have ensured that each machine receives its correct ON/OFF (0/1) signal based on the agent's output.

And here comes the stateflow chart in play (Figure 4.12) :

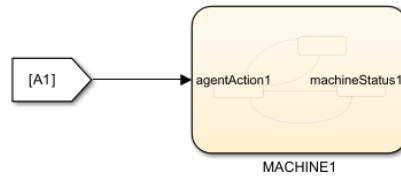


Figure 4.12 Action 1 being fed to stateflow of machine1

Each action is being fed to the stateflow chart of each machine.

Action 1 goes to stateflow of machine 1, action 2 goes to stateflow of machine 2, etc.

Once the action is fed into the stateflow, it decides on what state should the machine move to.

If agent has output 1, the machine moves to working state, whereas if agent has output 0, the machine is being set to idle until a new action comes.

4.4 Reward Function Setup

In reinforcement learning, the reward function is one of the most important parts, because it tells the agent what it should aim for. It acts like feedback, giving the agent a number (reward) that shows how good or bad its actions are in reaching the goal.

The reward function is designed to guide the agent toward making the system run more efficiently, while avoiding actions that slow things down. In a production setup, this means reducing wait times, using machines efficiently, preventing buffers from overflowing, and keeping costs low. The agent continuously learns from these rewards, improving its decision-making over time.

It is crucial to design the reward function carefully, so it truly reflects real-world goals. If it is not well thought out, the agent might focus too much on one thing, while ignoring others, which could lead to problems—like getting stuck in a strategy that seems good in the short term but is not the best overall (also known as getting trapped in local optima or minima) [50].

In our model, the reward function discourages excessive buffer usage and long wait times for products or clients. This helps keep the production line running smoothly, while minimizing overall costs. The next section explains exactly how this reward system is set up.

4.4.1 Reward Function Logic in Simulink

As mentioned earlier, the reward function plays a crucial role in our reinforcement learning setup. It helps guide the agent's learning by evaluating system performance at each time step and providing feedback. This allows the agent to understand whether its actions are making the production process more efficient.

In Simulink, the reward function logic is built into the model, using a custom MATLAB function block, called `rewardFunction`. This function processes key signals from the production line, such as buffer levels and the number of waiting clients.

The `rewardFunction` continuously tracks the system's state and calculates the reward based on congestion in the intermediate buffers, the number of ready products, and the number of waiting clients. The goal is to minimize buffer costs, while keeping production as efficient as possible. To encourage better decision-making, the agent receives a negative reward (penalty), when buffers become full or clients wait too long. This pushes the agent to develop optimal machine control strategies.

The setup is illustrated in Figure 4.13.

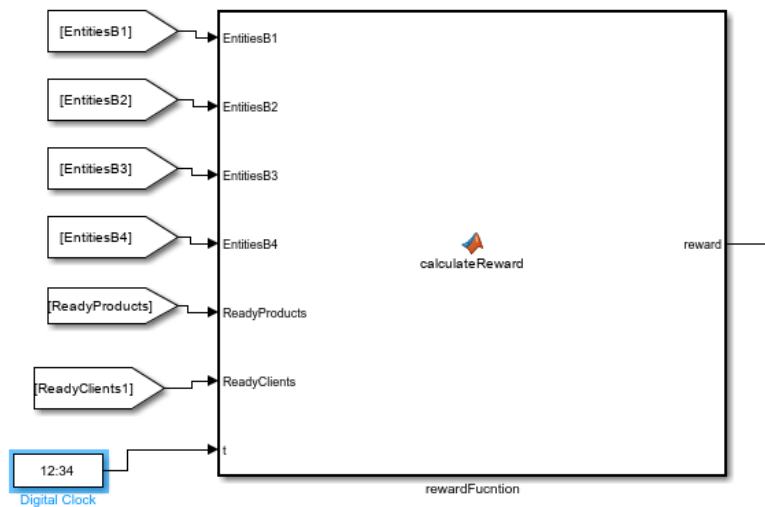


Figure 4.13 Reward function integration into Simulink Model

4.4.2 Implementation of the Reward Function

As shown in Figure 4.13, the reward function is implemented using a custom MATLAB function inside the `rewardFunction` block. The function is connected to the following input signals from the production line model:

- **EntitiesB1, EntitiesB2, EntitiesB3, EntitiesB4:** Represent the number of entities in the 4 intermediate buffers.
- **ReadyProducts:** Tracks how many finished products are waiting in the final buffer.
- **Ready Clients:** Indicates how many clients are waiting.
- **Digital Clock:** Provides the simulation time (t) to calculate the time step (dt), ensuring that the reward calculation accounts for changes over time.

Key Aspects of the Reward Function

1.Inputs:

The reward function receives buffer levels (`EntitiesB1`, `EntitiesB2`, etc.), the number of ready products and clients, and the simulation time (t) as inputs.

2.Time Interval (dt):

The time interval (dt) between time steps is calculated as:

$$dt = t - lastTime$$

The variable lastTime stores the last simulation time, allowing us to compute how much time has passed between reward calculations.

This dt scaling is crucial, as it ensures that the penalties for buffer congestion and waiting clients are related to the duration of the delay. The longer a product or client waits, the higher the penalty.

3. Cost Weights:

Different cost weights are applied to buffers, ready products, and ready clients:

- **Buffer cost:** Buffers 1 to 4 are penalized with a cost of 0.5 per product.
- **Ready products cost:** A cost of 1 is applied to products waiting to be processed.
- **Ready clients cost:** Clients waiting for products are penalized more heavily, with a cost of 1.3 per client.

4. Reward Calculation Formula: The total cost is computed as:

$$Total\ Cost = \sum (Buffer\ Level \times Buffer\ Cost + Ready\ Products \times Products\ Cost + Ready\ Clients \times Clients\ Cost) \times dt$$

The reward is the negative of this total cost: Reward = -Total Cost

5. Digital Clock Block Usage:

The Digital Clock block outputs the current simulation time (t), which is used to compute the time interval (dt). The sample time is set to 0.1 to ensure that the reward is updated frequently, but without excessive computational overhead. This setup ensures that longer delays result in higher penalties, encouraging the agent to minimize waiting times.

All the above were included in the reward function that can be seen below in Figure 4.14 :

```

function reward = calculateReward(EntitiesB1, EntitiesB2, EntitiesB3, EntitiesB4, ReadyProducts, ReadyClients, t)
    persistent lastTime
    if isempty(lastTime)
        lastTime = t;
        reward = 0;
        return;
    end

    dt = t - lastTime;
    lastTime = t;

    bufferCost = 0.5;
    productsCost = 1;
    clientsCost = 1.3;

    totalCost = (EntitiesB1 + EntitiesB2 + EntitiesB3 + EntitiesB4) * bufferCost ...
        + ReadyProducts * productsCost ...
        + ReadyClients * clientsCost;

    reward = -totalCost * dt;
end

```

Figure 4.14 Reward Function

By implementing the reward function this way, we ensure that the agent :

- Keep buffers from getting congested.
- Minimize the time products and clients wait.
- Optimize machine states and overall system performance.

To sum up, our production line model, after integrating the reinforcement learning agent and the custom function blocks, follows in Figure 4.15:

CHAPTER 4: REINFORCEMENT LEARNING AGENT IMPLEMENTATION AND INTEGRATION

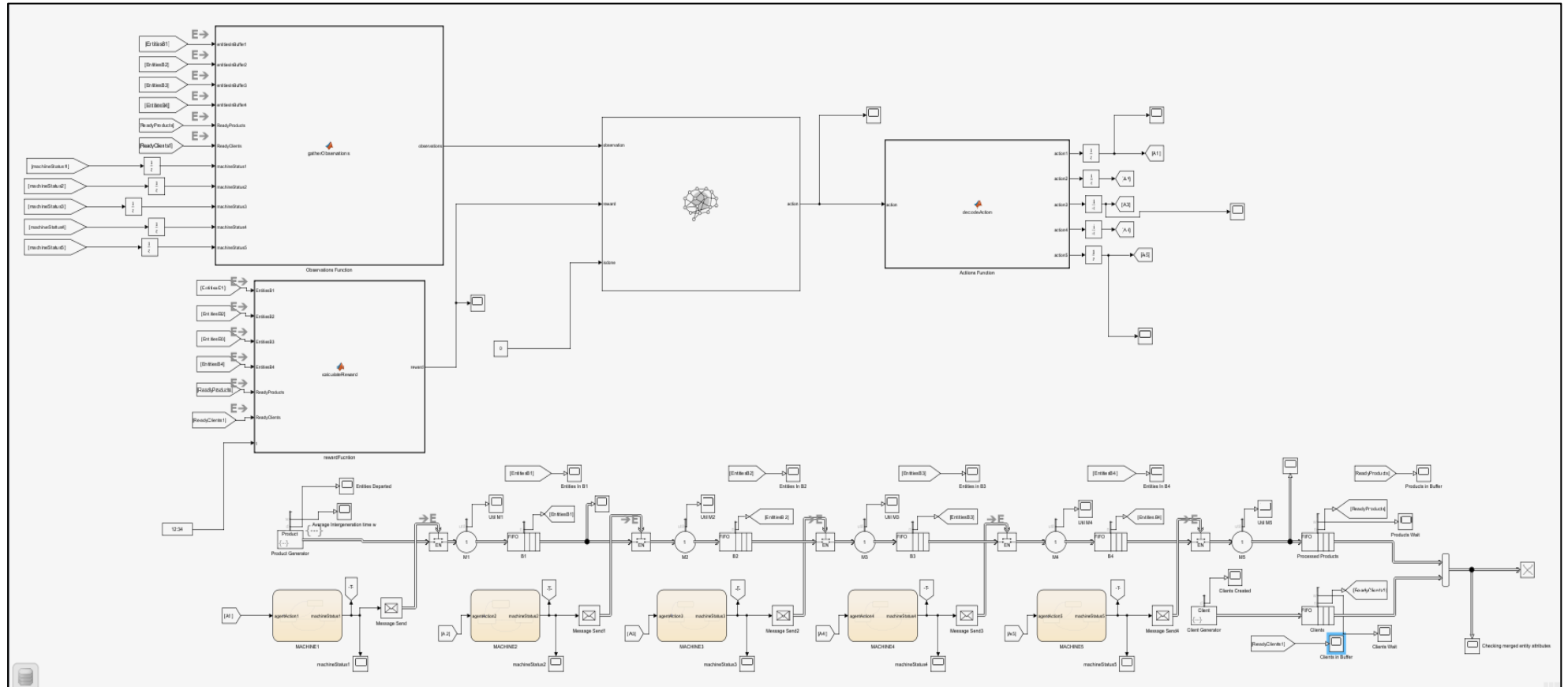


Figure 4.15 Production line with reinforcement learning agent integrated

4.5 Actor-Critic Network Architectures

As we earlier discussed in ‘2.6 Overview of PPO Algorithm’, the Proximal Policy Optimization (PPO) algorithm relies on an actor-critic architecture, where the actor is responsible for choosing actions and the critic evaluates the quality of those actions. In our implementation, both the actor and critic networks are neural networks, each designed to process observation inputs and provide necessary outputs for learning. The interaction between those networks was discussed earlier and also presented in figure 4.16:

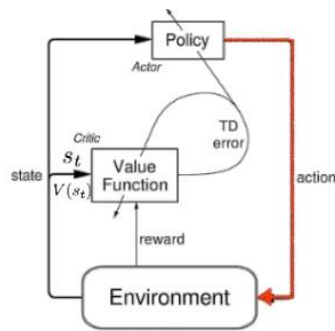


Figure 4.16 Actor-Critic Interaction [51].

4.5.1 Actor Network Design and Layers

The actor network takes the observation vector from the environment and generates a probability distribution over the action space. The structure of the actor network in our MATLAB code is as follows in Figure 4.17:

```

% Actor Network with initialized weights
actorNet = [
    featureInputLayer(obsDim, 'Normalization', 'none', 'Name', 'obsInput')
    fullyConnectedLayer(128, 'Name', 'fc1', ...
        'WeightLearnRateFactor', 1, ...
        'WeightsInitializer', 'glorot')
    reluLayer('Name', 'relu1')
    fullyConnectedLayer(64, 'Name', 'fc2', ...
        'WeightLearnRateFactor', 1, ...
        'WeightsInitializer', 'glorot')
    reluLayer('Name', 'relu2')
    fullyConnectedLayer(actDim, 'Name', 'fcOutput', ...
        'WeightsInitializer', 'zeros')
    softmaxLayer('Name', 'actionProb')];
  
```

Figure 4.17 Actor network

Let's break the actor network and explain its structure.

featureInputLayer: Takes in the 11-dimensional observation vector (defined by obsDim) without normalization, as the gatherObservations function already handles normalization.

fullyConnectedLayer1 (fc1): Contains 128 neurons and uses the Glorot initializer to set initial weights.

ReLU activation 1 (relu1): Introduces non-linearity to enable the network to learn complex patterns.

fullyConnectedLayer 2 (fc2): Contains 64 neurons, reducing the dimensionality to a more abstract representation.

Output Layer (fcOutput): Maps the internal representation to the number of possible actions (actDim = 32), with initial weights set to zero.

Softmax layer (actionProb): Converts the output to a probability distribution over the 32 possible actions, ensuring the sum of probabilities equals 1.

Overall, the actor network learns a policy that maps the current state (observations) to an action probability distribution, allowing the agent to select actions based on likelihood.

4.5.2 Critic Network Design and Layers

The critic network estimates the expected cumulative reward (value) from the current state. Its architecture mirrors the actor network, with minor differences in the output layer (Figure 4.18) :

```
% Critic Network with initialized weights
criticNet = [
    featureInputLayer(obsDim, 'Normalization', 'none', 'Name', 'obsInput')
    fullyConnectedLayer(128, 'Name', 'fc1', ...
        'WeightLearnRateFactor', 1, ...
        'WeightsInitializer', 'glorot')
    reluLayer('Name', 'relu1')
    fullyConnectedLayer(64, 'Name', 'fc2', ...
        'WeightLearnRateFactor', 1, ...
        'WeightsInitializer', 'glorot')
    reluLayer('Name', 'relu2')
    fullyConnectedLayer(1, 'Name', 'value', ...
        'WeightsInitializer', 'zeros')];
```

Figure 4.18 Critic Network

The only difference between the actor network and the critic network is that the critic network outputs a single scalar value ($V(s)$) representing the expected return from the given state of the environment.

Overall, the critic guides the learning process by estimating how good the current state is, providing feedback to the actor on which actions lead to higher future rewards.

4.5.3 Actor and Critic Representations

Now that we have decided on the actor and critic network architectures and layers, we need to connect them to the reinforcement learning framework. This is handled by the following code shown in Figure 4.19:

```
% Create actor and critic representations
actor = rlDiscreteCategoricalActor(actorNet, obsInfo, actInfo);
critic = rlValueFunction(criticNet, obsInfo);
```

Figure 4.19 Actor and critic representations.

The `rlDiscreteCategoricalActor` defines an actor network that selects actions from a discrete action space. In our case, the action space consists of 32 possible combinations of on/off signals for the five machines.

Why categorical?

The actor chooses from a finite set of discrete actions, so a categorical distribution is suitable, allowing it to sample actions based on their probability.

How it Works:

For each time step, the actor receives the observation vector from the environment. The observation vector passes through the actor network, and the output layer (softmax) produces probabilities for each action.

The action with the highest probability is typically selected during training.

So for `actor = rlDiscreteCategoricalActor(actorNet, obsInfo, actInfo);` :

- **actorNet:** The neural network architecture defined in Section 4.4.1.
- **obsInfo:** Specifies the structure and limits of the observation space.
- **actInfo:** Defines the action space with the 32 discrete actions.

Similarly for the critic, the `rlValueFunction` defines the critic network that estimates the expected cumulative reward from the current state, known as the state value $V(s)$.

The critic guides the actor by evaluating how good the current state is in terms of expected rewards. This helps the actor update its policy to favor actions that lead to better states.

How it works?

For each time step, the critic receives the same observation vector as the actor. The observation vector passes through the critic network, and the output layer produces a scalar value $V(s)$. This value is used to calculate the advantage estimate $A(s,a)$, which tells the actor whether taking a specific action was better or worse than expected.

So for `critic=rlValueFunction(criticNet, obsInfo);`

- **criticNet:** The neural network architecture defined in Section 4.4.2.
- **obsInfo:** Specifies the structure and limits of the observation space.

With these actor and critic representations configured, the PPO algorithm can effectively train the agent by updating the policy based on both short-term feedback (immediate rewards) and long-term performance (value function estimates).

4.6 PPO Agent Configuration

In this section, we set up the PPO agent by adjusting key settings that control how it learns. The effectiveness of PPO depends on fine-tuning parameters, such as the learning rate, horizon (how far it looks ahead), clip factor, and other important factors. Here, we will go over the specific choices we made and explain why they are crucial for optimizing our production line's performance.

4.6.1 PPO Hyperparameter Setup

The PPO agent is created using the `rlPPOAgent` function, which requires the actor and critic networks, along with a set of hyperparameters to guide the training process. Below in Figure 4.20 is the configuration and a brief explanation of each parameter:

```
%% Step 3: PPO Agent Options
agentOpts = rlPPOAgentOptions(...
    'ExperienceHorizon', 2048, ...
    'MiniBatchSize', 512, ...
    'ClipFactor', 0.1, ...
    'EntropyLossWeight', 0.02, ...
    'DiscountFactor', 0.995, ...
    'AdvantageEstimateMethod', 'gae', ...
    'GAEFactor', 0.95, ...
    'ActorOptimizerOptions', rlOptimizerOptions('LearnRate', 1e-4, 'GradientThreshold', 1), ...
    'CriticOptimizerOptions', rlOptimizerOptions('LearnRate', 5e-5, 'GradientThreshold', 1));
```

Figure 4.20 PPO Agent Hyperparameters

Before diving into the details of each parameter, it is important to note that not all hyperparameters available for tuning are shown in this Figure. Some additional hyperparameters for our agent are kept at their default values.

If we wanted to train our agent using only the default settings, we could simply use:
`agentOpts = rlPPOAgentOptions();`

For each hyperparameter we want to tune, we need to specify its name followed by the desired value. The hyperparameters we experimented with are as follows:

Experience horizon: The experience horizon is set to 2048, which defines how many time steps of experience are collected, before performing an update. A larger horizon helps the agent better estimate long-term rewards, which is important in our production line, where costs and waiting times accumulate overtime.

Mini Batch Size: The mini-batch size is configured to 512, which controls the size of data batches used during each gradient update. A larger batch size improves stability, but can slow down learning, so our choice balances stability and training speed.

ClipFactor: The clip factor is set to 0.1, representing the core mechanism of PPO. A small clip factor prevents the policy from being updated too aggressively, ensuring stability.

EntropyLossWeight: To encourage exploration, the entropy loss weight is set to 0.02, adding a penalty, if the agent's action probabilities become too deterministic. This exploration is critical during the initial learning stages, helping the agent discover optimal strategies.

DiscountFactor: The discount factor, set to 0.995, defines the importance of future rewards. A value close to 1 emphasizes long-term rewards, making the agent focus on system-wide optimization rather than short-term gains.

AdvantageEstimateMethod: We use Generalized Advantage Estimation (GAE) to compute the advantage function $A(s,a)$ balancing bias and variance in the estimation.

GAEfactor: The GAE factor is set to 0.95, giving more weight to long-term future rewards, while maintaining stability.

Actor and critic learning rates: The learning rates for the actor and critic networks are set differently to allow for stable updates. The actor's learning rate is slightly higher ($1e-4$) to adapt faster to changing environments, while the critic's learning rate is lower ($5e-5$) to ensure stable and accurate value estimates. Both optimizers include a gradient threshold to prevent excessively large updates during training, which could destabilize the learning process.

These hyperparameters were selected through extensive testing to balance exploration, stability, and learning efficiency. Our production line has complex, long-term dependencies, so parameters like a large experience horizon and a near 1- discount factor were necessary to achieve optimal performance. Once the hyperparameters are set, the PPO agent is created using the following code (Figure 4.21).

```
% Create PPO Agent
agent = rlPPOAgent(actor, critic, agentOpts);
```

Figure 4.21 Creation of PPO agent using rlPPOAgent function

When we call rlPPOAgent with these arguments, MATLAB initializes a PPO agent with the specified actor, critic, and configuration options. Now, the agent is ready to be trained. Training options and setup is fully covered in the upcoming section.

4.7 Training Setup and Execution

To successfully train the PPO agent within the production line environment, specific training options and parameters were configured to ensure convergence, stability, and efficiency. This section covers the setup of training options, the process of monitoring the training progress, and the mechanism for saving the trained agent for future use.

4.7.1 Training Options and Parameters

The training process was configured using the rlTrainingOptions function in MATLAB. This function allows for the customization of key training parameters that control how long the agent trains and how performance is evaluated. The main parameters configured are as shown in Figure 4.22.

```
%% Step 4: Training Options
trainOpts = rlTrainingOptions(...
    'MaxEpisodes', 20000, ...
    'MaxStepsPerEpisode', 2000, ...
    'ScoreAveragingWindowLength', 100, ...
    'Verbose', true, ...
    'Plots', 'training-progress', ...
    'StopTrainingCriteria', 'AverageReward', ...
    'StopTrainingValue', -100);
```

Figure 4.22 Training options

Max Episodes: The maximum number of training episodes was set to 20,000 to give the agent sufficient time to explore and learn near-optimal policies.

Max Steps Per Episode: Each episode has a maximum of 2,000 steps to ensure the agent can observe meaningful production line behavior, without overly long training sessions.

Score Averaging Window Length: The average reward is calculated over the most recent 100 episodes. This helps monitor the agent's performance and detect whether it is improving consistently.

Verbose Output: This option was enabled to print detailed updates during training, making it easier to monitor the agent's progress in real time.

Stop Training Criteria: Training is automatically stopped, when the average reward over 100 episodes reaches a specified threshold (-1000 in this case). This value was chosen based on our expected reward structure and system goals.

These options were carefully chosen to balance efficient training and ensure the agent can learn optimal behavior within the given options (Figure 4.23).

```
% Train the agent
trainingStats = train(agent, env, trainOpts);
```

Figure 4.23 Training the agent with the specified training options

4.7.2 Saving and Loading the Trained Agent

After training is complete, the trained agent is saved to a file for future use and evaluation using the following MATLAB commands shown in Figure 4.24:

```
% Save the trained agent
save('trainedAgent_dec2bin.mat', 'agent');
save('trainingStats_dec2bin.mat', 'trainingStats');
```

Figure 4.24 Saving the trained agent

These files contain the agent's learned policies, actor and critic networks, and training statistics. The saved agent can be reloaded and used in future simulations or real-world implementations without requiring retraining using the command:

```
load('trainedAgent_dec2bin.mat', 'agent');
```

With the completion of training, we have a PPO agent that has learned to optimize machine states, minimize production delays, and balance buffer loads. In the next chapter, we evaluate the effectiveness of this agent through simulation results and comparisons with traditional optimization methods.

CHAPTER 5: RESULTS AND DISCUSSION

5.1 Introduction to the Evaluation Metrics

In this section, we introduce the key metrics used to evaluate the performance of the PPO agent. These metrics provide valuable feedback on how well the agent optimizes the production line and minimizes costs. They will be analyzed and discussed in the following sections.

5.1.1 Total Reward

When evaluating a reinforcement learning agent's performance, the first thing that typically comes to mind is analyzing the reward behavior.

The total reward is one of the most important metrics in reinforcement learning. It represents the sum of all rewards the agent accumulates across each episode. Tracking this total reward helps determine whether the agent is improving. If the reward steadily increases, it usually indicates that the agent is learning effectively and making meaningful progress toward its goal. However, if the total reward remains low or fluctuates without improvement, it suggests that the agent is not learning efficiently. In such cases, adjustments may be needed, such as fine-tuning hyperparameters, modifying agent options, or making other changes to improve reward behavior.

The total reward presented in this section refers to the discounted cumulative reward, with a discount factor $\gamma = 0.995$, in accordance with the agent's training objective. This ensures consistency between the learning goal and the evaluation metric.

In our setup, the total reward is simply the negative of the total cost (reward = -cost). This means that a higher total reward (less negative) indicates a reduction in wasted time and inefficiencies in the system. Our reward function penalizes buffer congestion and long wait times for both products and clients. As the total reward increases over multiple episodes, the total cost decreases.

This metric provides valuable insights into the following improvements:

- **Reduced buffer congestion:** The agent optimizes product flow, preventing bottlenecks and overflows.
- **Improved machine utilization:** Efficient machine operation reduces idle time and ensures machines are used optimally.
- **Lower waiting times for products and clients:** The agent minimizes delays in buffers, leading to lower costs and improved system efficiency.

In this chapter, we'll show how the total reward changes over time as the agent trains. We'll also compare the reward from the PPO agent to a perfect-case scenario (the baseline) to see how close it gets to the best possible performance.

5.1.2 Buffer Occupancies

Keeping these levels under control is important, as it helps avoid slowdowns in production and makes sure client needs are met. If buffers get too full, it means there's congestion, which can cause delays. But if buffers are mostly empty, it might mean the system is not being used well.

To see how well the agent is learning, we'll look at how buffer levels change during training:

- **Early episodes:** Buffer occupancies may fluctuate significantly due to poor flow management, with processed products or waiting clients accumulating in buffers.
- **Mid-training:** Gradual improvements are expected, as the agent learns to manage machine operations more effectively.
- **Late episodes:** The agent should have optimized the flow, resulting in stable and balanced buffer occupancies with minimal congestion.

By comparing buffer occupancies in initial episodes with later episodes, we can observe how the agent learns to balance product flow.

5.2 Optimal cost Baseline via Dynamic Programming

To see how well the PPO agent performs, we first set a baseline using Dynamic Programming (DP). DP is a strong optimization method that looks at every possible situation and action, then works out the best solution by breaking the problem into smaller steps. Unlike reinforcement learning, which learns through trial and error, DP finds the exact policy that gives the lowest possible cost for the system.

So in our problem, we use it to compare its results with the training results of a simplified production line model we will later introduce.

5.2.1 Simplified Model Description

To effectively compare the performance of the PPO-based reinforcement learning agent with an optimal solution from dynamic programming method, we must simplify the production line model. The simplified model consists of:

- Two machines
- Three buffers (two final buffers and one intermediate buffer)

And is presented in Figure below 5.1:

CHAPTER 5: RESULTS AND DISCUSSION

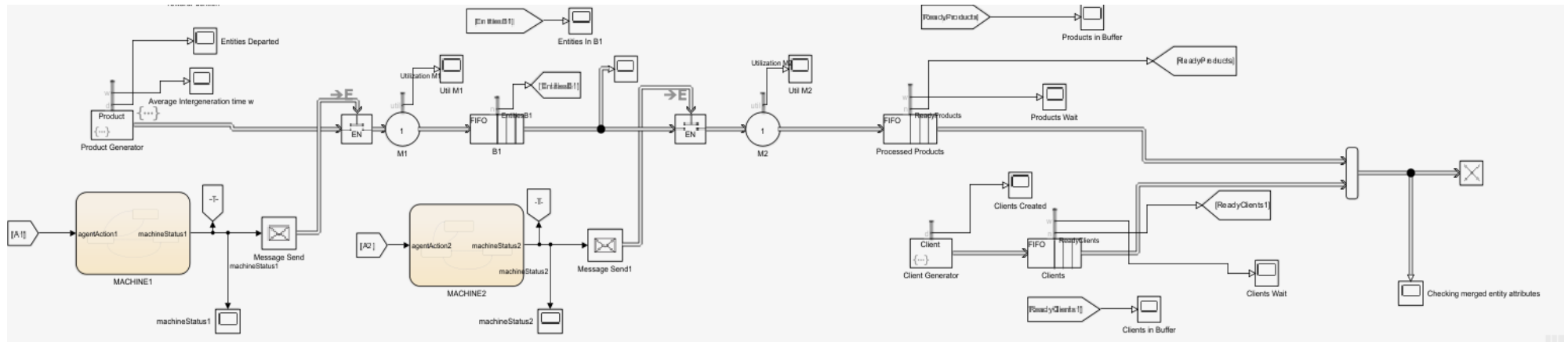


Figure 5.1 Simplified production line model

CHAPTER 5: RESULTS AND DISCUSSION

This simplification allows us to apply dynamic programming (DP) to find an optimal scheduling policy and resulting reward as, Dynamic programming is used for smaller-scale models due to its ability to systematically evaluate all possible states and actions, providing a good benchmark.

The simplified system configuration is as follows:

- Machine 1 and Machine 2: Sequential machines
- Intermediate Buffer: Holds products between Machine 1 and Machine 2.
- Two Final Buffers: Holds waiting clients and processed products

Key Assumptions:

- Each machine processes one product at a time.
- Processing times follow an exponential distribution, as in the original model.
- Buffers have same capacities with the original model.
- Machine failures and repairs are also modeled using exponential distributions.

5.2.2 Optimal Cost for Simplified Model Using Dynamic Programming Method

In this section, we look at the optimal cost found using the dynamic programming (DP) method for the simpler version of the production model. The goal of the DP algorithm is to lower the average cost at each time step, while also considering machine breakdowns, repairs, how fast products are made, and the cost of storing inventory. To do this, we set up the problem as a Markov Decision Process (MDP) and solve it using value iteration.

State Space Representation

The system is characterized by the following state variables:

- x : Inventory level of intermediate products.
- y : Inventory level of finished products (positive) or backorders (negative).
- $i_1 \in \{0,1\}$: Operational state of Machine 1 (1 if operational, 0 if under repair).
- $i_2 \in \{0,1\}$: Operational state of Machine 2 (1 if operational, 0 if under repair).

Thus, the state space is given by:

$$S = (x, i_1, y, i_2), \quad 0 \leq x \leq M_x, \quad -M_y \leq y \leq M_y, \quad i_1, i_2 \in \{0,1\}$$

where $M_x = M_y = 50$.

Note: In the dynamic programming model, the variable y captures both finished product inventory (when $y > 0$) and customer backorders ($y < 0$) using a single variable. This representation is fully consistent with the RL environment, where these two quantities appear as separate observation signals (ReadyProducts and ReadyClients). In practice, the agent combines these two inputs to recover the same information carried by the signed y , ensuring full compatibility between the state spaces of both methods.

Decision Variables

At each time step, for each machine that is operational ($i_1 = i_2 = 1$) the system must decide on production actions for the two machines:

- $a_1 \in \{0,1\}, a_2 \in \{0,1\}$

Therefore we have a total of 4 actions (00, 01, 10, 11), just exactly as our reinforcement learning setup, where:

- $a_1 = 1 \rightarrow$ Machine 1 processes an intermediate product.
- $a_1 = 0 \rightarrow$ Machine 1 remains idle (not processing products).
- $a_2 = 1 \rightarrow$ Machine 2 processes an intermediate product into a finished product.
- $a_2 = 0 \rightarrow$ Machine 2 remains idle.

Cost Function

The total cost function consists of:

1. Inventory Holding Costs:

$$C_h(x, y) = h_1 x + h_2 \max(y, 0) + b \max(-y, 0)$$

where:

- h_1 is the cost per intermediate product in inventory.
- h_2 is the cost per finished product in inventory.
- b is the backorder penalty per missing unit.

This cost function is also used in the reinforcement learning model, but in the form of a reward function defined as the negative of the total cost. This way, minimizing cost in the dynamic programming model is equivalent to maximizing reward in the PPO agent.

2. Failure and Repair Costs (implicitly modeled in transition probabilities).

The total expected cost at a given state is:

$$C(S, a) = C_h(x, y)$$

Transition Rates

The system dynamics are governed by the following transition probabilities:

- Production Transitions:
 - If Machine 1 is operational ($i_1 = 1$), procession occurs with probability μ_1 .
 - If Machine 2 is operational ($i_2 = 1$), it processes products with probability μ_2 .
- Demand Effects: Finished goods inventory decreases by 1 unit with probability λ .
- Machine Failures: Machine 1 fails with rate f_1 , and Machine 2 fails with rate f_2 .
- Repairs: Machine 1 is repaired with rate r_1 , and Machine 2 is repaired with rate r_2 .

The total event rate is given by:

$$v = \lambda + \mu_1 + \mu_2 + f_1 + f_2 + r_1 + r_2$$

In this model, v denotes the total transition rate out of a given state, summing the rates of all possible events: arrivals, productions, failures, and repairs. In the Bellman update,

CHAPTER 5: RESULTS AND DISCUSSION

dividing by v serves to normalize the cost per unit time, which is standard in average cost formulations for Markov decision processes.

Bellman Equation

The Bellman update equation is:

$$V(S) = \max_a [C(S, a) + \sum P(S' | S, a) V(S')]$$

Value Iteration Algorithm

To compute an optimal policy, we employ the Value Iteration Algorithm:

1. Initialize the value function:

$$V_0(S) = 0, \forall S$$

2. Iterate until convergence:

$$V_{k+1}(S) = \max_a [C(S, a) + \sum P(S' | S, a) V_k(S')]$$

3. Stopping Criterion: The process stops when:

$$\max_s |V_{k+1}(S) - V_k(S)| < \varepsilon$$

where ε is the predefined tolerance level.

Results of dynamic programming method

After executing the code for the dynamic programming method, we get the following results: The average optimal cost per time step is calculated to 0.26102, with 1541 iterations needed to converge, as shown in Figure 5.2.

Final Results:

Number of iterations: 1541
Final difference: 0.000001
Optimal average cost per time step: 0.26102

Figure 5.2 Dynamic Programming results

The convergence of DP cost per step is shown in figure 5.3 below:

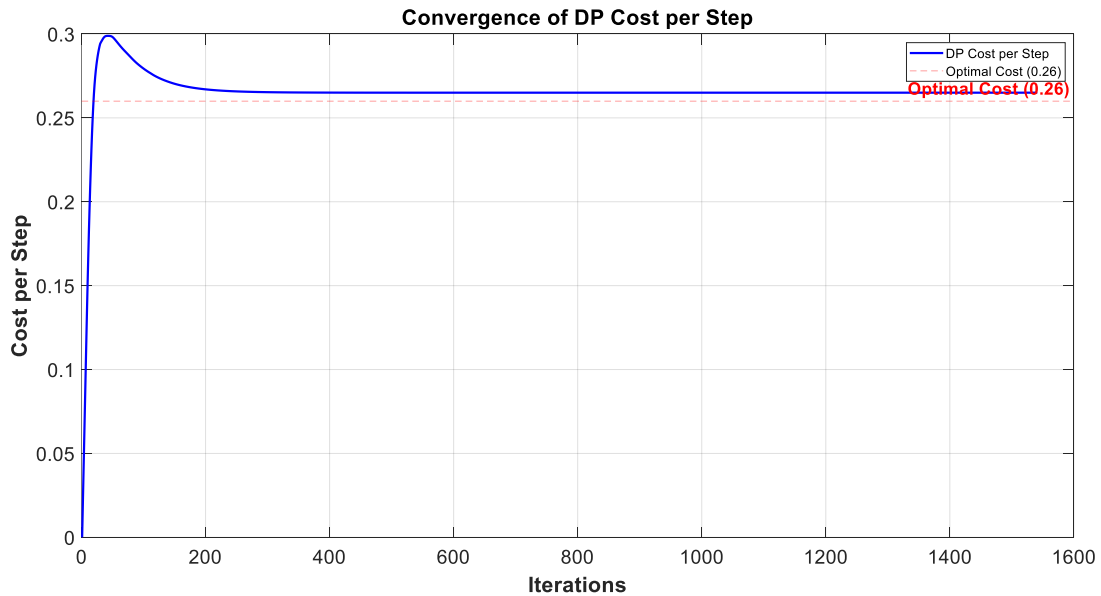


Figure 5.3 Convergence of DP Cost per step

Initially, the cost fluctuates, but stabilizes at 0.261, which represents the optimal average cost per step. This confirms that the algorithm successfully minimizes production costs under system constraints.

So, in conclusion, the dynamic programming method achieves an optimal cost of **0.261 cost units per time step**. These results demonstrate that dynamic programming effectively determines an optimal production policy, while accounting for machine failures, repairs, and demand uncertainty.

5.2.3 Training Results of the Simplified Model

The training results of the simplified model are presented in Figure 5.4 below:

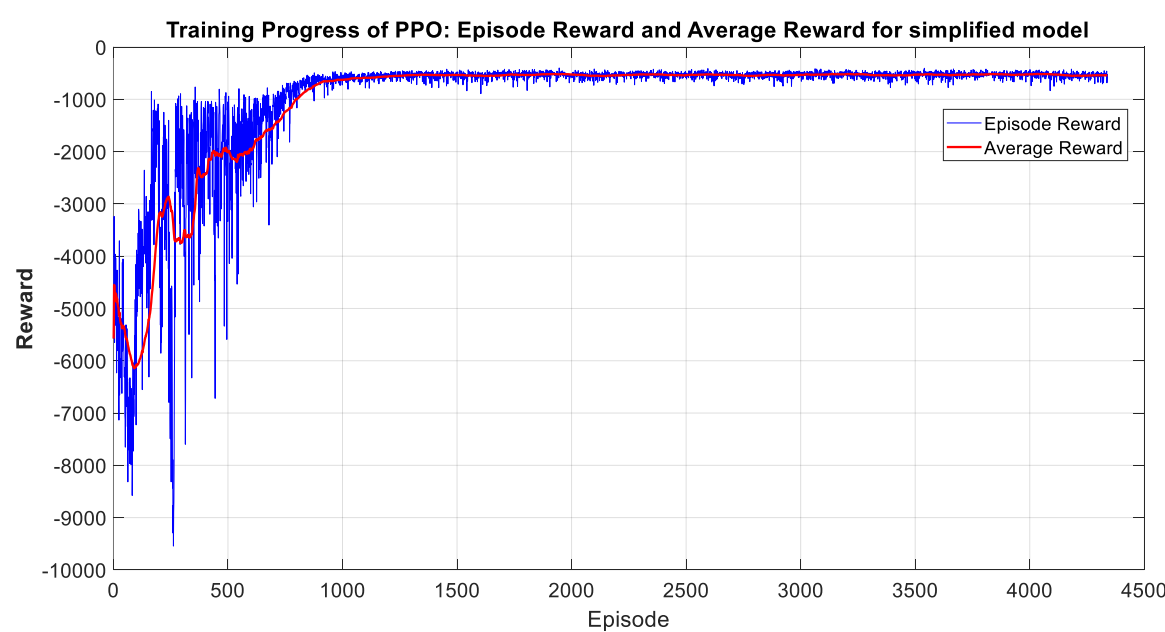


Figure 5.4 Training results of PPO agent for simplified model

We can see the agent achieving an average reward of -540.99 with about 13 hours and 4500 episodes needed, as seen in Figure 5.5.

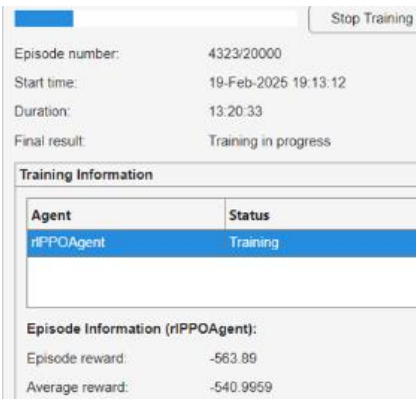


Figure 5.5 Training stats for simplified model

5.2.4 Comparison of Agent Results with Dynamic Programming Solution

Previously we calculated the optimal cost achieved both by dynamic programming method and reinforcement learning agent setup.

We know that each episode has 2000 time steps. The agent achieves average reward of -540.99, so average cost of 540.99 cost units per episode. That's translated to 540.99 cost units per 2000 time steps. So, the cost units per time step is $\frac{540.99}{2000} = 0.2704$ cost units per time step.

In Section 5.2.2, we found that the optimal cost achieved by the dynamic programming method was 0.261 cost units per time step, so the final results are presented in Table 5.1 below:

Method	Cost Step	per Total cost per episode(2000 steps)	Iterations/ Episodes to converge
Dynamic Programming	0.261	522.0	1541 Iterations
PPO Agent	0.2704	540.99	~ 1600 episodes

Table 5.1 Comparison of dynamic programming and reinforcement learning results

The results show that the DP method achieves a lower cost per step (0.261), which serves as the optimal baseline. The PPO agent learns a near-optimal policy, reaching a cost per step of 0.2704, which is roughly 3.8% higher than DP, showcasing that our agent is well trained and executes near optimal scenario.

In conclusion, the results for the simplified model validate both dynamic programming (DP) and reinforcement learning (RL) approaches for optimal production control. Since DP successfully finds an optimal policy, and PPO achieves a near-optimal solution with a small gap, we expect these findings to extend to larger production systems, like our more complex production line (5 machines, 6 buffers). Thus, while additional computational effort is required, we anticipate that the PPO agent will still find near-optimal policies in the larger production system, making reinforcement learning a viable solution for real-world industrial applications.

5.2.5 Comparison of Extracted Policies from Both Methods

In this section, we compare optimal policies extracted using Dynamic Programming (DP) and Reinforcement Learning (RL) for the simplified production line. DP computes the exact optimal policy through value iteration, providing a benchmark for the system's behavior, while RL learns an approximate policy through interaction with the environment, as implemented in the trained agent.

The comparison aims to evaluate the alignment between the two methods, focusing on their decision-making at critical points in the state space, specifically near the DP switching curves, where the policy transitions between decisions (e.g., turning machines on or off).

Before presenting the extracted DP policy, it is essential to understand the meaning of the state variables. As earlier mentioned in 5.2.2, the **intermediate buffer level (x)** represents the number of semi-finished products stored in the buffer between the two machines, whereas the **backorder level (y)** represents unmet customer demand, where negative values indicate backorders (unfulfilled demand), and positive values indicate excess stock. A high negative y value means that production does not meet demand, while a positive y value suggests overproduction.

DP Policy

Figure 5.6 depicts an optimal policy extracted using Dynamic Programming (DP) for the production system. Each subplot represents the machine's operational status as a function of the intermediate buffer level (x) and backorder level (y). The blue regions indicate states, where the machine is actively working, while the white regions indicate states where the machine is not working. These policies are computed through value iteration, ensuring an optimal balance between production rates, buffer costs, and backorder penalties.

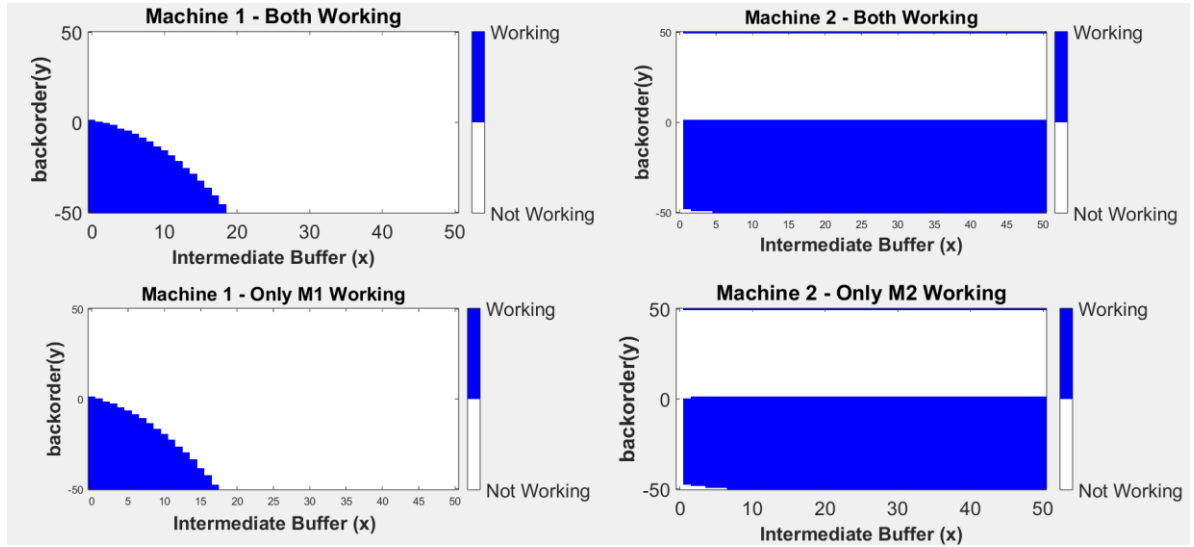


Figure 5.6 Dynamic Programming Policy

Examining the extracted DP policy, several patterns emerge in machine operation. When analyzing the case where only Machine 1 is available, a similar pattern is observed. Machine 1 mainly operates when backorders are significantly negative (approximately in the range of -50 to 0), indicating high demand pressure. However, without Machine 2 in the system, its activation region becomes slightly narrower, reaching up to an intermediate buffer level of around 16 (compared to 19 when both machines are available). This suggests that Machine 1 adjusts its behavior in the absence of additional production capacity by covering a slightly different region of the state space.

The case where only Machine 2 is working further confirms its strong dependence on backorder levels. The clear separation between the working and non-working states in this scenario shows that Machine 2's decisions are unaffected by the intermediate buffer level, as it consistently activates across all x-values from 0 to 50 when backorders are negative.

From these observations, the DP policy reveals a structured and optimal decision-making pattern. Machine 1's operation is influenced by both intermediate buffer levels and backorders, while Machine 2 follows a strict backorder-driven policy. The decision boundaries are well-defined, ensuring minimal cost, while maintaining system efficiency. These structured activation regions confirm that DP successfully finds an optimal balance between production, buffer, and backorder costs, leading to a highly efficient and interpretable policy.

The switching curves that follow on Figure 5.7 highlight the structured nature of the DP policy, showing clear boundaries where machines transition between active and inactive states.

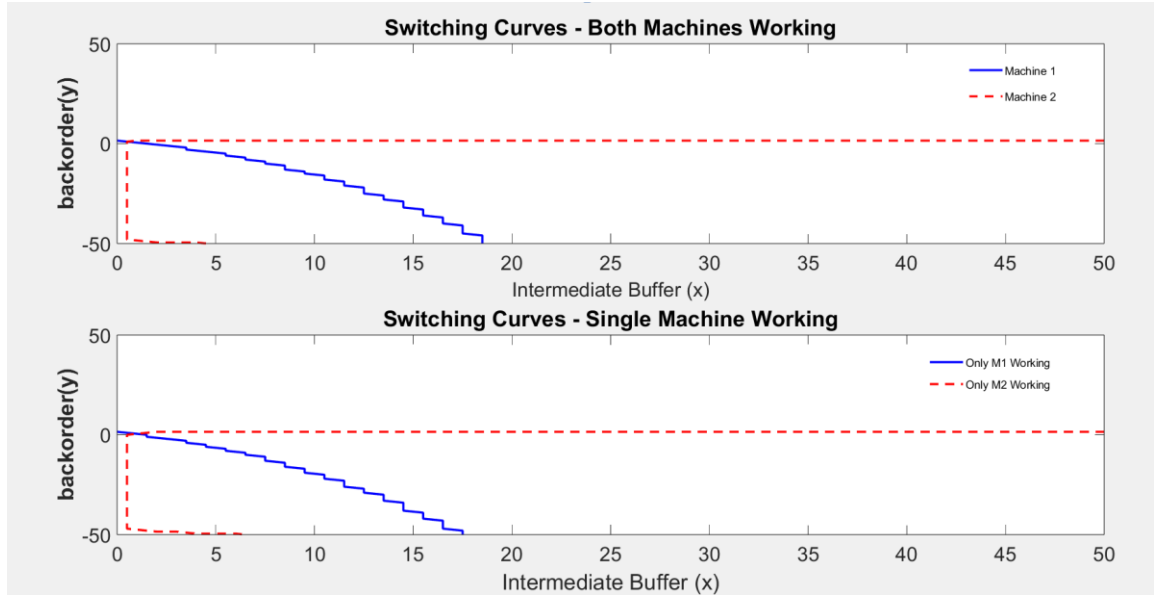


Figure 5.7 DP Switching Curves

Machine 1 operates based on both intermediate buffer levels and backorders, remaining active longer when the buffer is low, and demand is high. In contrast, Machine 2 follows a strict backorder-driven policy, activating only when backorders are negative, regardless of buffer levels.

When only one machine is available, Machine 1 slightly extends its working region to compensate for the absence of Machine 2, while Machine 2's decision boundary remains unchanged. This confirms that Machine 2 responds purely to demand, while Machine 1 balances production and storage constraints.

RL Policy

The RL policy is obtained by loading the trained Proximal Policy Optimization (PPO) agent, which has learned an approximate policy through interaction with the environment. To extract the policy, we follow a structured approach, ensuring that all possible states are systematically evaluated.

First, we iterate over all possible values of the intermediate buffer level (x) and the backorder level (y), covering the full state space. At each state, the trained RL agent is

queried with the state variables, and the agent returns an optimal action. The extracted decision is stored in a matrix corresponding to the respective machine availability.

The decisions are stored in matrices that map each state to an action, forming a complete representation of the RL policy. These matrices are then visualized using heatmaps, where blue regions indicate production activity, while white regions denote inactivity.

The results are presented in Figure 5.8 below :

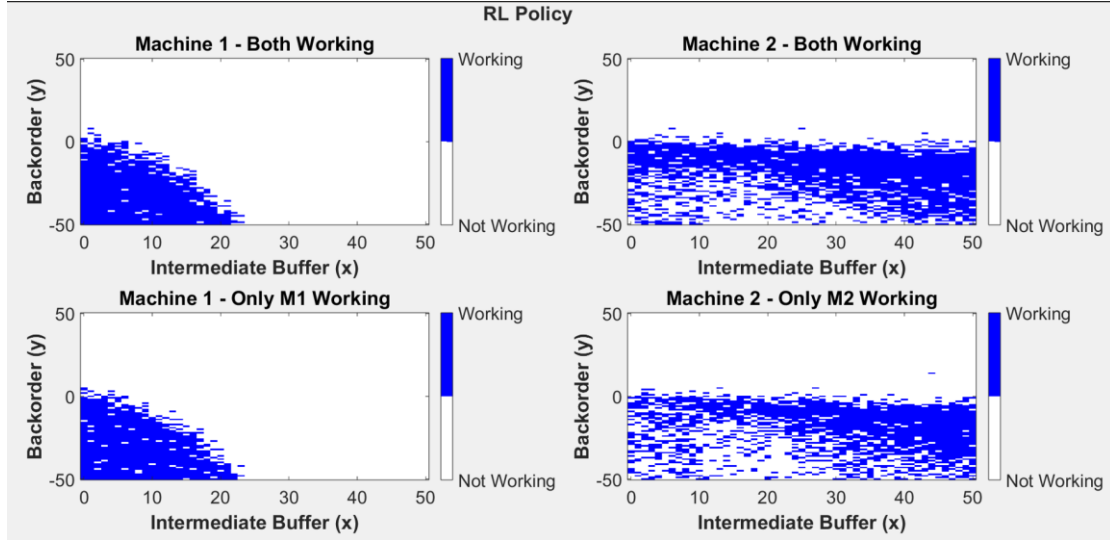


Figure 5.8 RL Policy

Similar Patterns with DP:

- Both policies show that Machine 1 operates when the intermediate buffer is low, and demand (backorders) is high.
- Machine 2 primarily follows a backorder-driven strategy, working mostly when there is a deficit in demand (negative y).
- When only one machine is available, Machine 1 compensates by extending its operating region, while Machine 2 remains strictly reactive to demand.

Differences:

- The RL policy is slightly less structured than the DP policy, showing some deviations in decision boundaries.
- The DP policy has clear switching curves, while the RL policy exhibits more flexible, but noisier transitions.

Even though there are some small differences, the RL policy mostly matches the DP policy. This shows that the trained agent has learned a good way to make decisions. The

differences we see happen because reinforcement learning is based on approximations, unlike dynamic programming, which finds the exact mathematical solution.

5.3 PPO Agent Results

In this section, we look at how well the PPO agent performs after training and how well it handles the production line compared to how the system worked at the beginning. We check the agent's progress using key performance indicators, like how the total reward changes over time and how well it manages the buffers.

By following these indicators, we can clearly see improvements in how smoothly products move through the system, how much buffer congestion is reduced, and how overall costs go down. This section also includes a comparison between the PPO agent and the A2C algorithm to better understand how effective the PPO agent is.

5.3.1 Total Reward Evolution Over Episodes

As mentioned earlier, the evolution of total reward over training episodes provides key insights into the learning process of the PPO agent and its ability to optimize machine scheduling and buffer management in the production line. The results we are presenting in this section illustrate the progression of the total reward in our production line.

Reward Progression and Learning Behavior

Our PPO agent results are presented in Figure 5.9 below:

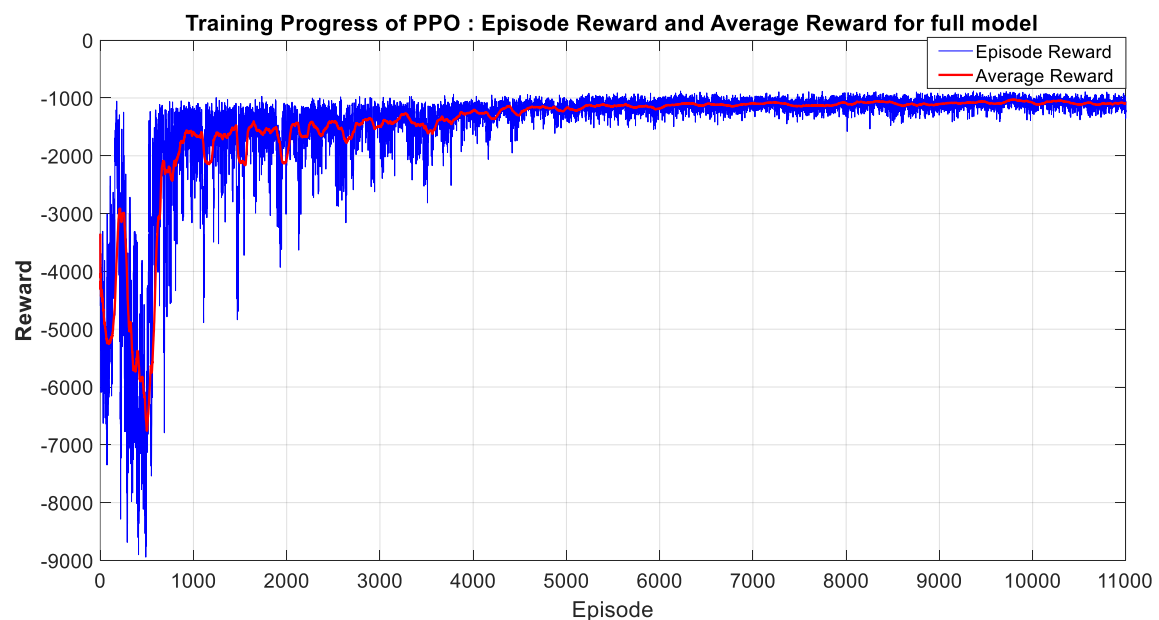


Figure 5.9 Total Reward progression over episodes

Let's see how those results are exactly interpreted and why they show that our agent has been successfully trained.

The graph clearly shows three distinct learning phases:

- **Exploration Phase (0 - ~500 episodes):**
Initially, the agent explores different actions randomly, leading to highly negative

rewards due to inefficient scheduling, frequent buffer congestion, and suboptimal machine operation. The episode reward fluctuates heavily, indicating that the agent is testing different strategies without yet having a clear understanding of which actions are beneficial.

- **Learning and Optimization Phase (500 - ~1200 episodes):**

As training progresses, the agent starts recognizing patterns in the environment, leading to a gradual increase in total reward (becoming less negative). The fluctuations begin to smooth out, suggesting that the agent is starting to stabilize its decision-making. This phase marks the shift from random exploration to more structured learning, as the PPO algorithm adjusts the policy towards optimizing machine utilization and buffer flow.

- **Convergence and Policy Stability (1500+ episodes):**

Beyond 1500 episodes, we observe a consistent behavior in the average reward, showing that the agent has successfully learned an effective scheduling policy. The episode-to-episode fluctuations decrease, and the average reward stabilizes, indicating that the agent has found an efficient way to balance machine states and product flow.

Training stats are presented in Figure 5.10 that follows:

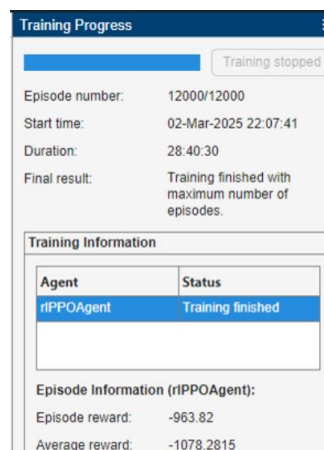


Figure 5.10 PPO training stats

Training lasted 28 hours for a total of 11.000 episodes (22.000.000 steps)

We also observe that the final average reward achieved by the PPO agent is about (-1070) per episode (per 2000 time units/ time steps).

In the next section, we will further analyze how this improved policy translates into better buffer management, showing the agent's impact on balancing products and clients flow over different training phases.

5.3.2 Buffer Management Improvement

As earlier mentioned, one of the key goals of the reinforcement learning agent is to improve buffer management by balancing the flow of products and clients in the system. Initially, the agent has no understanding of how to properly regulate production, leading to inefficient product accumulation and poor client servicing. Over time, through trial and error, it learns an optimal strategy to maintain balanced buffer levels, ensuring smooth production flow and reducing overall costs.

Episodes 0-100: Poor Buffer Regulation (Episodes 1-50)

At the beginning of training, the agent fails to regulate buffer levels properly, causing a severe imbalance. The processed products buffer accumulates excessively, as machines continue producing, without considering the system's actual needs. Meanwhile, very few clients are waiting, as the processed products buffer is always near to full, so when a client reaches the client's buffer, it immediately gets matched with one processed product.

As shown in Figure 5.11 and Figure 5.12 below, this imbalance results in high operational costs due to overproduction and inefficient flow management.

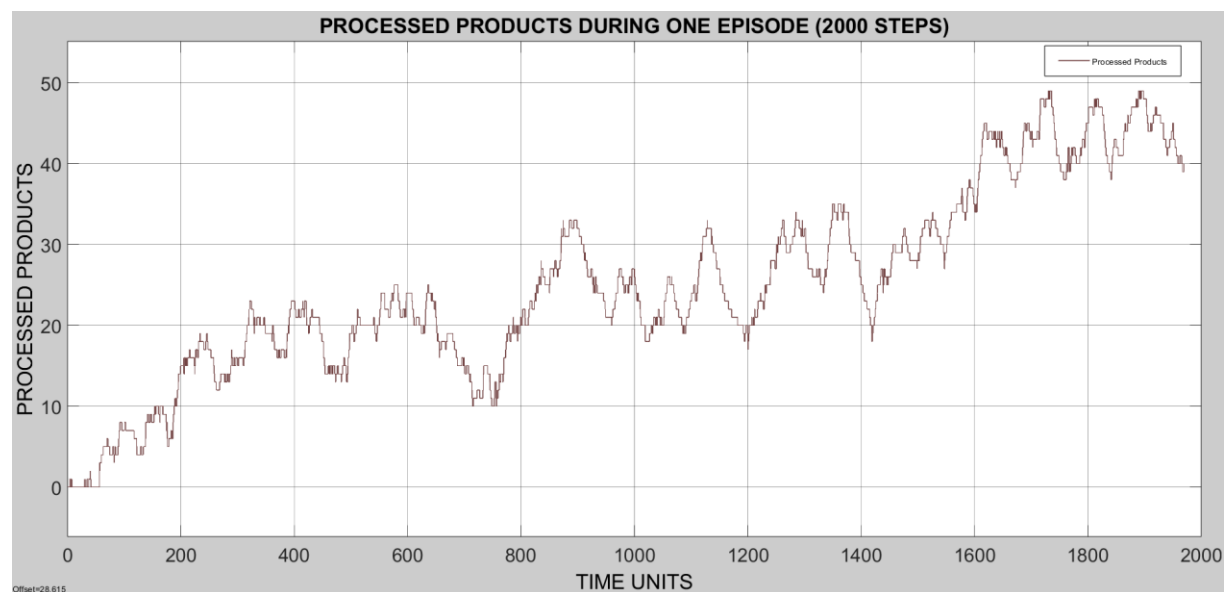


Figure 5.11 Processed products accumulated in early episodes

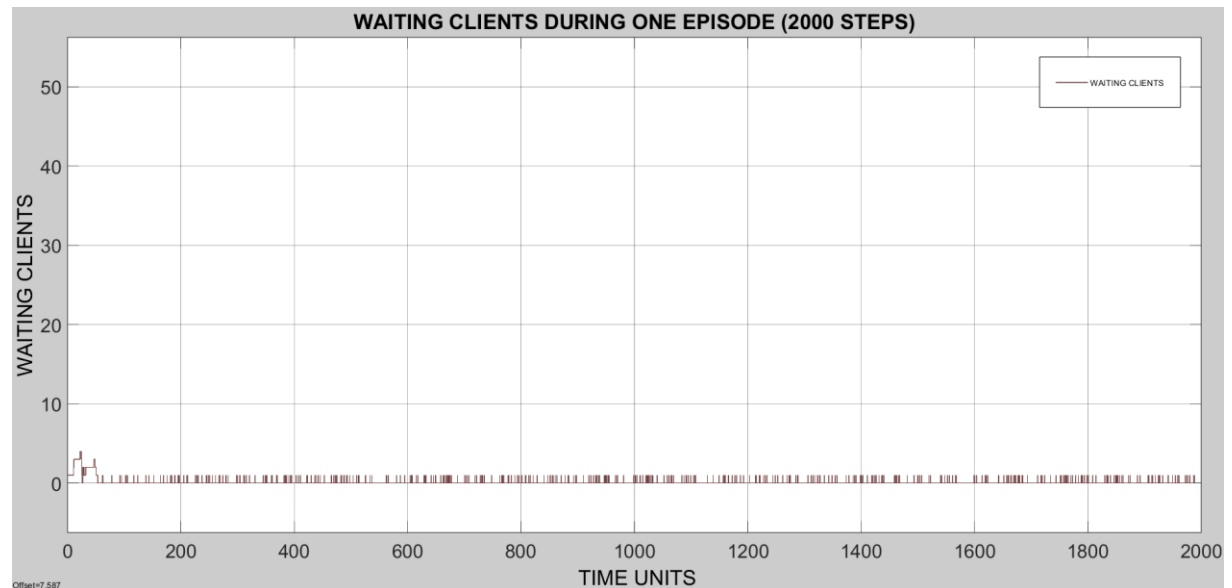


Figure 5.12 Waiting clients in early episodes

Episode 100-150: Early Exploration Phase

In Figures 5.13 – 5.14, we notice that the agent starts experimenting with different production strategies. The buffer for processed products still fluctuates, but in lower levels compared to initial episodes and there is also a noticeable increase in the number of waiting clients. This suggests that the agent is beginning to explore different machine operation patterns, shifting towards a better balance between supply and demand.

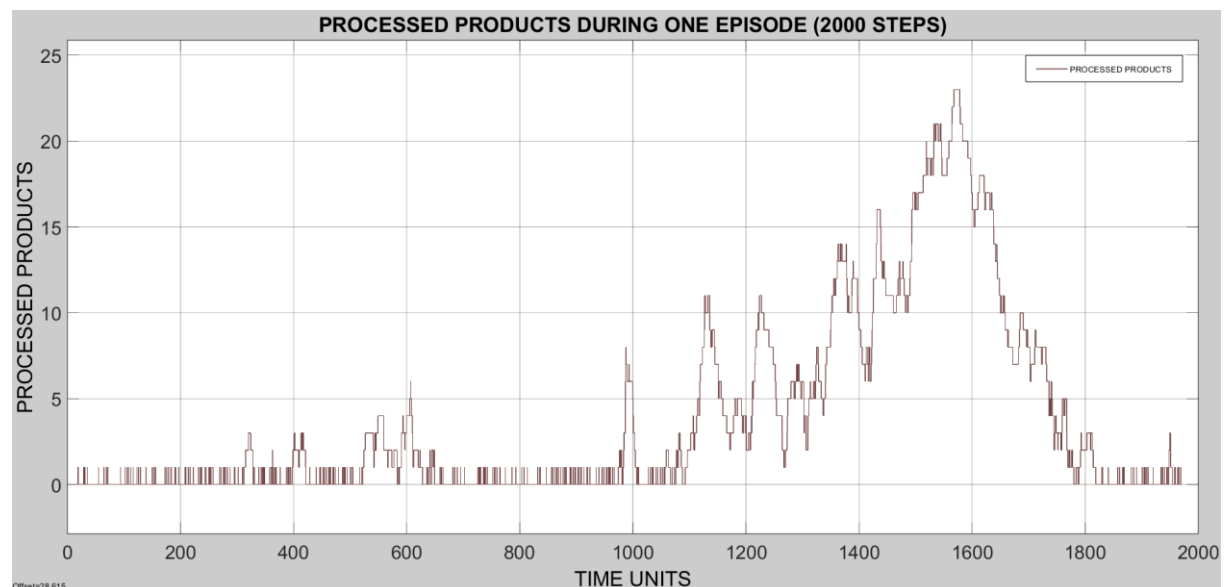


Figure 5.13 Processed products in early exploration phase

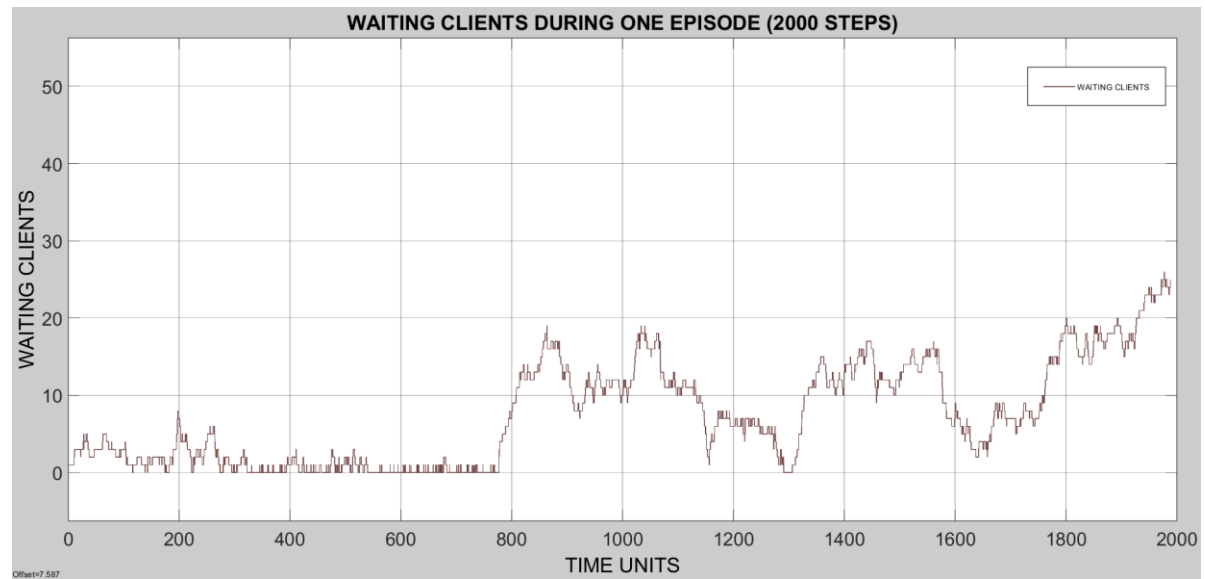


Figure 5.14 Waiting clients increasing in early exploration phase

Episode 500: Later exploration phase:

By episode 500, we are still in exploration phase, but now the agent is trying actions that accumulate the clients buffer and keep the processed products at minimal levels as shown in Figures 5.15 and 5.16. This is the point where our agent has the most negative reward, since the client's buffer have the highest cost compared to products buffers. We can see almost zero products waiting in the buffer and clients reaching the maximum capacity of their buffer.

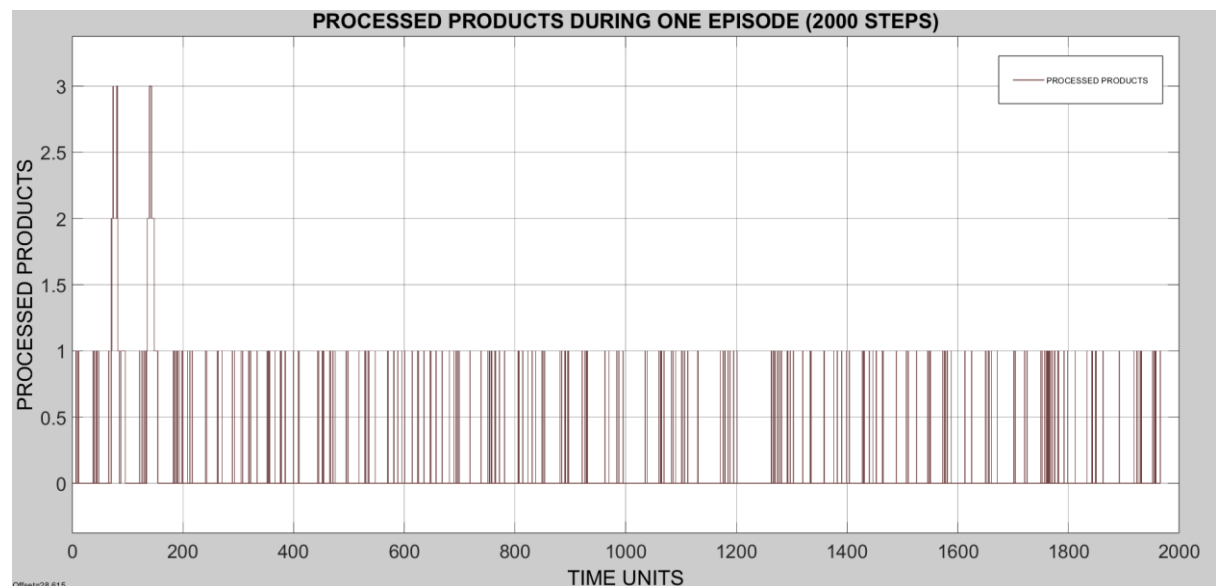


Figure 5.15 Processed Products being minimized as the agent is still exploring

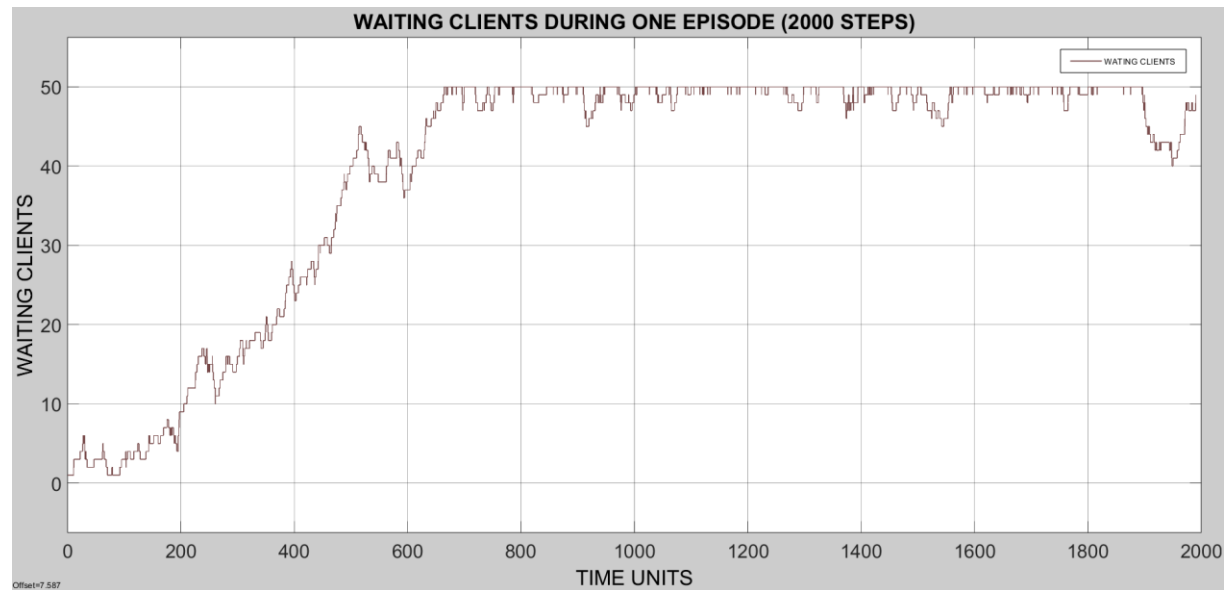


Figure 5.16 Waiting clients accumulating heavily reaching maximum buffer capacity

Summarizing, the agent in those first 500 episodes was exploring the environment and the impact of its actions. It explored by accumulating processed products and minimizing clients, then it tried accumulating clients and minimizing processed products. Now that the exploration phase is complete, the agent starts trying to balance waiting clients and processed products, aiming to minimize the cost of the system.

FINAL RESULTS

Episode 11.000: Optimal Buffer Balancing

By episode 11.000 (Figures 5.17 and 5.18), the system exhibits clear signs of optimal flow regulation. The buffer for processed products no longer experiences extreme peaks, as seen in earlier training phases. Similarly, the number of waiting clients is significantly more stable. This suggests that the agent has successfully learned to synchronize machine operations with real-time buffer conditions, preventing both overproduction and supply shortages.

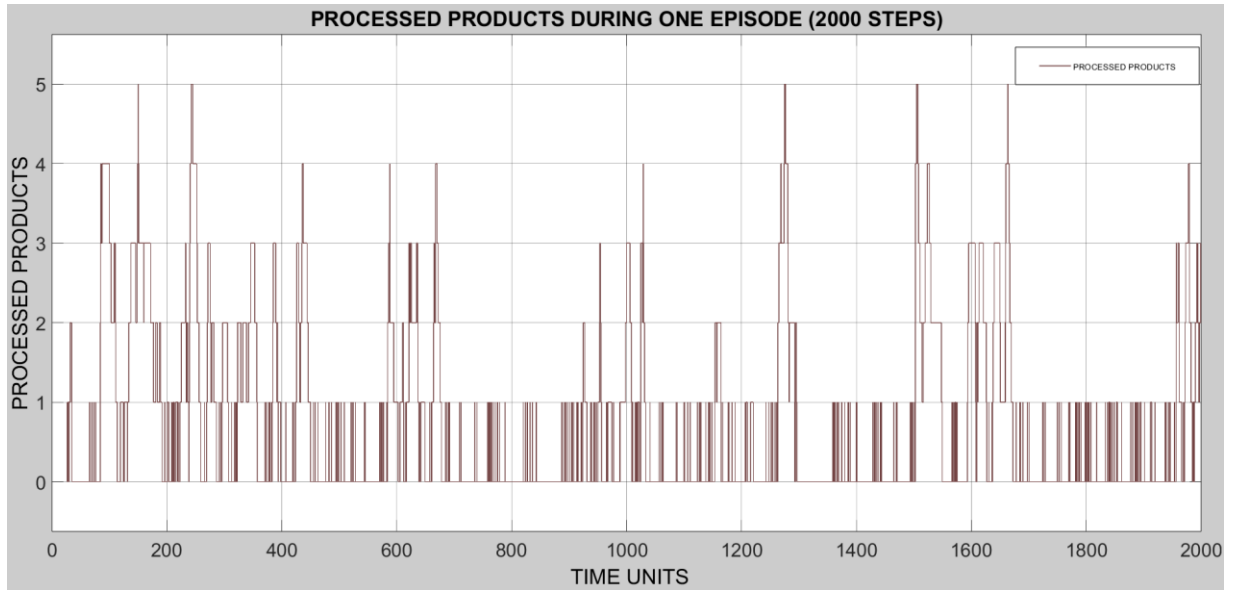


Figure 5.17 Processed products at the end of training

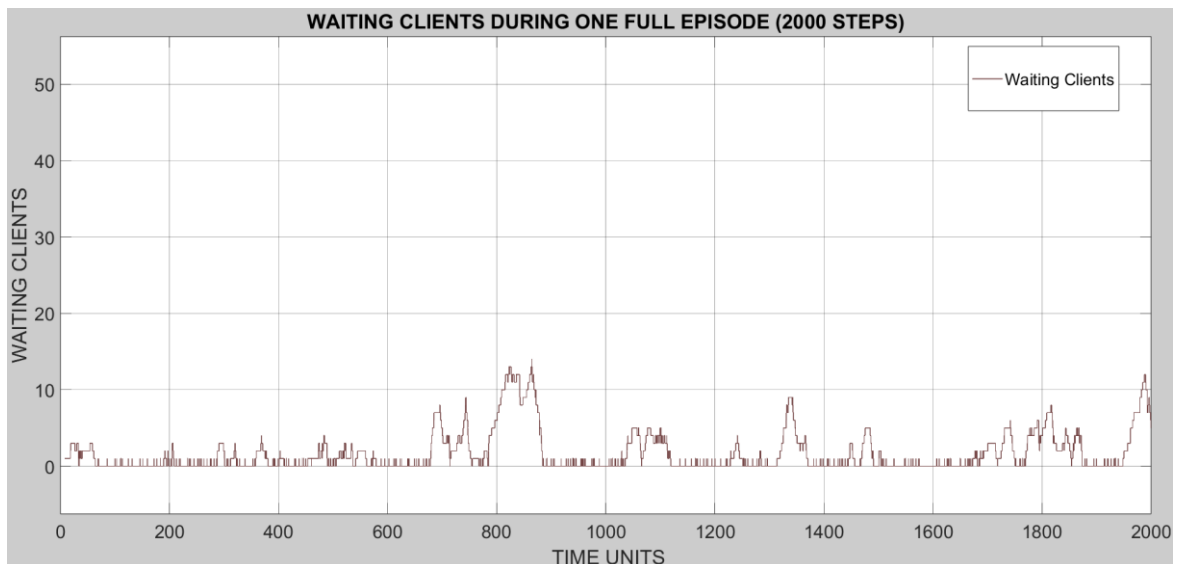


Figure 5.18 Waiting clients at the end of training

Numerical Breakdown for training results:

- Processed Products Buffer: Most of the time, no processed products are accumulated. Occasionally, we have some fluctuates to 5-6 units, and then the system immediately recovers bringing products back to near-zero values. This behavior means products are moving efficiently without unnecessary buildup.
- Waiting Clients Buffer: Typically remains between 0 and 5 clients, with very few peaks to 10 that are immediately brought down to near zero clients, with no prolonged congestion or overflow.

Overall, we can see the agent significantly improving the system, but someone could ask: **Why does not the agent minimize waiting clients and products to zero?**

Achieving zero waiting clients or a completely empty processed products buffer is not feasible, due to the fundamental nature of the system:

1. Exponential processing times and variability

- The machines' processing times follow exponential distributions, meaning that some products are processed quickly, while others take longer. This randomness in service time prevents a perfectly synchronized system, as some fluctuations are unavoidable.

2. Client Arrival Following an Exponential Distribution

- Client demand also follows an exponential distribution, meaning arrival times are not the same for every client. Some periods experience high demand spikes, while others see fewer arrivals, leading to natural buffer fluctuations.

3. System Imperfections and practical constraints

- In real-world production settings, achieving perfect synchronization is unrealistic, due to machine breakdowns, stochastic service times, and unexpected variations. Even in an ideal simulation, random variations in entity movement and processing times prevent a perfectly flat buffer profile.

5.4 Simulation of Trained Agent Across Different Scenarios

In this section, we focus on testing and validating the performance of the trained Proximal Policy Optimization agent across multiple scenarios to evaluate its robustness and generalization capabilities. After completing the training process and achieving results of total reward and buffer management, it is essential to ensure that the saved agent performs similarly with same efficiency, when exposed to both familiar and altered conditions. To do so, we loaded the trained agent into our environment and tested it through different scenarios, but also with the initial conditions it was trained, to verify its efficiency. During scenario testing, learning was turned off. The agent was evaluated in inference mode, using a fixed policy obtained after training. No further learning or policy updates occurred during the 100 test episodes. This allows for a consistent and fair assessment of generalization and robustness.

5.4.1 Baseline Scenario : Reproduction of Training Conditions

The first step in validating the trained PPO agent is testing it under the exact same conditions in which it was initially trained. This scenario is important, as it serves as a baseline to confirm that the agent has retained its learned policy and can achieve similar results when reloaded without requiring any further retraining.

To make this test, the trained PPO agent is loaded in the same simulation environment, with identical parameters and settings. The load and simulation of the training agent for all the scenarios that are going to be covered is done in the exact same way and is presented in Figure 5.19 below:

```

%% Step 4: Simulate the agent for 100 episodes
simOptions = rlSimulationOptions(...
    'MaxSteps', 2000, ...      % 2000 steps per episode
    'NumSimulations', 100);    % 10 episodes

disp('Starting simulation for 10 episodes...');
experience = sim(env, agent, simOptions);
disp('Simulation completed.');
```

Figure 5.19 Simulation settings for trained agent

The simulation was executed for 100 episodes, with each episode consisting of 2000 steps. The `rlSimulationOptions` function was used to define the simulation parameters, and the `sim` function was employed to run the agent-environment interaction. The simulation results were recorded for further performance analysis.

The calculation of total reward per episode was calculated using the code shown in Figure 5.20 that follows:

```

%% Step 5: Calculate and display total rewards per episode
totalRewards = arrayfun(@(x) sum(x.Reward), experience);
fprintf('\nAverage Total Reward over 100 Episodes: %.2f\n', mean(totalRewards));

for ep = 1:length(totalRewards)
    fprintf('Episode %d - Total Reward: %.2f\n', ep, totalRewards(ep));
end
```

Figure 5.20 Calculation and display of total reward per episode

The `arrayfun` function was used to compute the sum of rewards for each episode.

Returning to our simulation, we tested the trained agent over 100 episodes under the exact same conditions used during training. The results were as expected—during the simulation, the agent achieved an average reward similar to what it attained when fully trained. This confirms that the agent’s learned policy is functional and transferable when deployed in the same environment.

As discussed earlier in Chapter 5, the agent achieved an average reward of -1070 during training. As shown in Figure 5.21, the simulation of the trained agent over 100 episodes resulted in an average reward of -1079, further validating the agent’s performance.

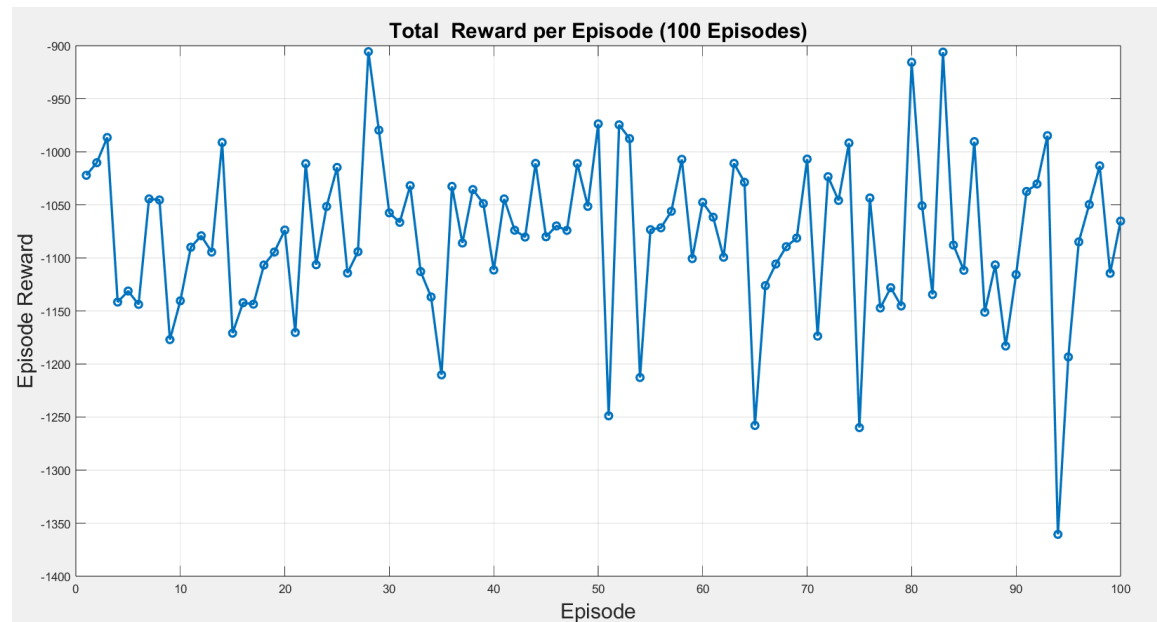


Figure 5.21 Simulation of trained agent in initial conditions

5.4.2 Scenario 1: Higher Demand

In the current scenario, we test the agent in different conditions than the conditions it was trained. Specifically, we have increased the demand in our system, by changing the client's generation rate from 1 client every 4.5 time units to 1 client every 4 time units.

The results are shown in Figure 5.22, where we can see that the agent achieved an average reward of -1203 reward, showcasing that the agent adapted reasonably well to that increase in client demand (approximately 12%).

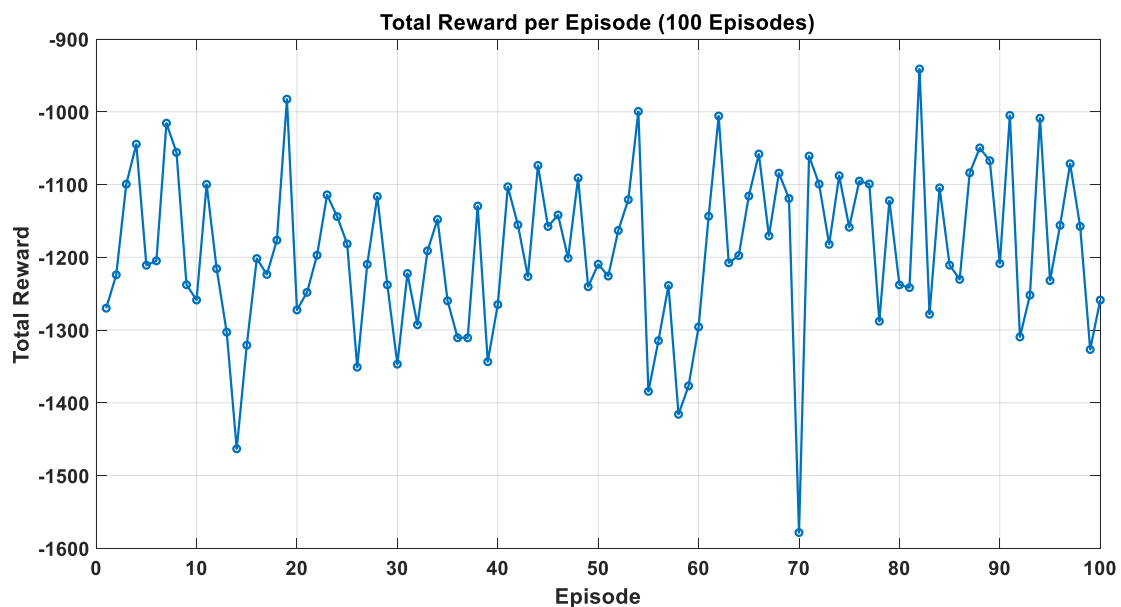


Figure 5.22 Agent's performance in higher demand conditions

5.4.3 Scenario 2: Faster Production

For this last scenario, we test the agent under more pressure, by increasing the product's generation rate, by increasing it from one product per 2.7 time units, to one

product per 2.3 units. The results are very satisfying, since the average reward is -1076, which suggests that the agent's policy is robust in handling more demanding product arrival rates, maintaining near-optimal efficiency. The results are presented in Figure 5.23 below.

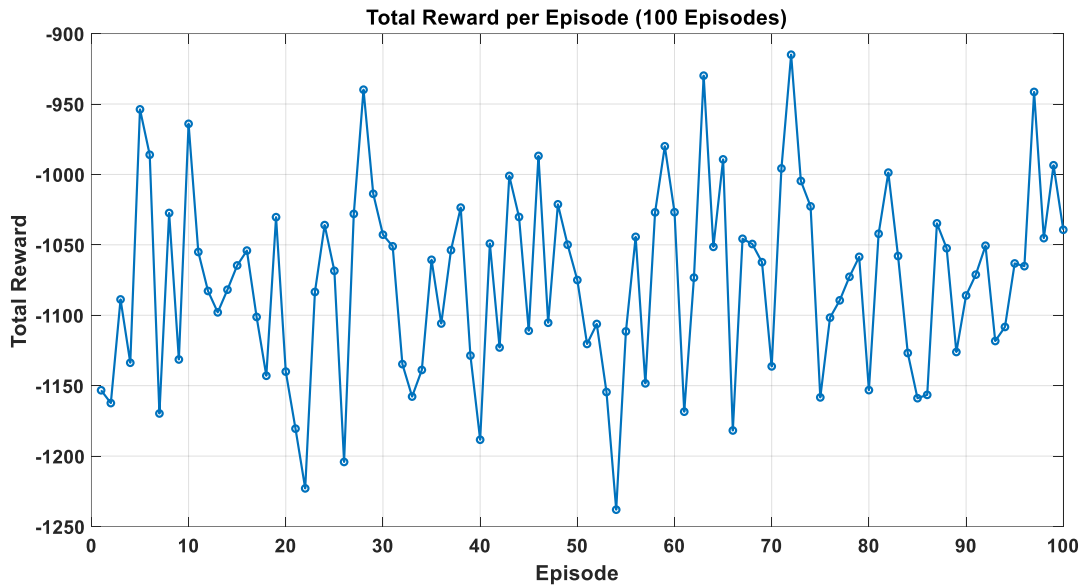


Figure 5.23 Agent's performance in higher production rate conditions

5.4.4 Summary of Scenario Testing

In summary, the scenario testing and simulation of the trained agent for a total of 100 episodes demonstrates that the trained PPO agent exhibits strong robustness and generalization across various conditions. In the baseline scenario, where the agent was tested under the exact training conditions, it achieved an average reward of -1079, closely matching the -1070 reward from the training phase, confirming that the agent retained its learned policy after being reloaded. In Scenario 1, where client demand was increased by 12% (reducing client generation time from 4.5 to 4 time units), the agent managed to achieve an average reward of -1203, indicating a reasonable ability to adapt despite the higher workload. Lastly, in Scenario 2, where product generation rate was increased from 2.7 to 2.3 time units (17%), the agent achieved a reward of -1076, demonstrating strong resilience and minimal performance degradation under faster production conditions. Overall, the results validate the agent's ability to handle both familiar and altered operating environments effectively.

5.5 Comparison between PPO vs A2C

In this section, we compare how the Proximal Policy Optimization (PPO) algorithm performs against Advantage Actor-Critic (A2C) to see how different reinforcement learning methods handle production line optimization. Both algorithms are policy-based, but they use different training methods and have varying levels of stability and efficiency.

We chose PPO as our main algorithm, because it works well with large discrete action spaces and tends to be more stable. To see how it truly measures up, we also ran A2C under the same settings. This comparison focuses on how quickly each algorithm learns,

how their rewards change over time, and how effectively they manage machine states and buffer levels.

The next parts of this section explain how we set up A2C for our system, then compare PPO and A2C in detail. We specifically look at total reward trends and buffer management to highlight any differences.

5.5.1 Overview of A2C Implementation and Changes

A2C is a policy-based reinforcement learning method that updates the policy using the gradient of an actor network, while a separate critic network estimates the value function to reduce variance and stabilize learning. Unlike PPO, A2C applies updates synchronously across multiple agents, making it simpler but potentially less stable than PPO.

The A2C implementation follows a similar structure to the PPO agent, but with key modifications in agent options and hyperparameters:

Entropy Regularization:

- In PPO, the entropy loss weight was set to **0.02**, ensuring controlled exploration.
- In A2C, we increased the entropy loss weight to **0.1** to encourage more exploration and reduce premature convergence. This change was needed, since with lower entropy loss weight values, the A2C agent was getting stuck early in local minima.

Learning Rates:

- PPO utilized different learning rates for the actor (**1e-4**) and critic (**5e-5**) networks.
- A2C used a slightly higher learning rate of **2e-4** for the actor and **1e-4** for the critic, allowing faster updates.

Training Update Frequency:

- PPO utilized a large experience horizon (**2048 steps**) and mini-batches (**512 samples per update**), stabilizing policy updates through a clipped loss function.
- A2C follows a more online learning approach, performing updates every **32 steps**, meaning the network adapts quickly, but may require more training episodes to converge.

Finally, we also had to change the agent definition in the training script. Instead of using `rlPPOAgent`, we define the A2C agent using `rlACAgent`. The same was done for the training options. Instead of `rlPPOAgentOptions`, we used `rlACAgentOptions`.

These modifications demonstrate the different ways PPO and A2C approach reinforcement learning. The next section presents a direct performance comparison between the two agents.

5.5.2 Performance Comparison of PPO and A2C

To compare the performance of the Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) algorithms, we analyze their reward progression and stability during training. Both agents were trained under the same conditions, using our production line

environment, allowing for a direct comparison of their effectiveness in optimizing the production line. The results of the A2C agent can be seen in Figure 5.24 that follows.



Figure 5.24 A2C agent results

Phase 1: Initial Exploration (0 - ~3000 episodes)

At the beginning of training, A2C starts exploring, leading to fluctuating and highly negative rewards. The average reward starts around **-4000** and shows some signs of improvement. The higher entropy loss weight (0.1) forces the agent to keep exploring, preventing it from settling into efficient behaviors early, as PPO did. Unlike PPO, which steadily increases its reward after exploration phase, A2C experiences a further drop in performance, with rewards falling below -7,000 as it keeps exploring. The higher entropy loss weight was needed, because we tried several lower values for that hyperparameter, and the agent was getting stuck early in local minima. At this stage, A2C is still struggling to balance machine operations and buffer management effectively, leading to an inefficient system, where waiting clients and congested buffers accumulate.

Phase 2: Learning and Optimization Phase (3000 - ~5000 episodes):

After the performance drop, A2C gradually recovers, and the average reward improves. However, even at this point, fluctuations in rewards remain, suggesting that the policy is still unstable. Unlike PPO, which maintains a consistent performance gain, A2C continues to show variability in decision-making.

Phase 3: Convergence and Policy Stability (5000+ episodes):

Beyond 5000 episodes, we observe a gradual upward trend in the average reward, indicating that the agent is beginning to recover from its earlier performance drop. The agent successfully learns a functional policy, with the average reward stabilizing around -1500. However, unlike PPO, the episode-to-episode fluctuations remain significant, reflecting instability in the agent's policy. This lack of consistency is largely due to the higher entropy loss weight, which encourages continued exploration at the cost of reduced stability. While the agent improves its ability to manage machine states and

CHAPTER 5: RESULTS AND DISCUSSION

product flow, its decision-making remains less reliable compared to PPO, resulting in slower convergence and less stable performance over time.

In Figure 5.25, we can see that A2C training took about 48 hours, significantly slower than PPO that needed 27 hours of training. Also, PPO managed to effectively optimize our production line, achieving an average reward of -1000 whereas A2C not only did take more time to train, but it also did not achieve the training results of PPO (-1500 reward).

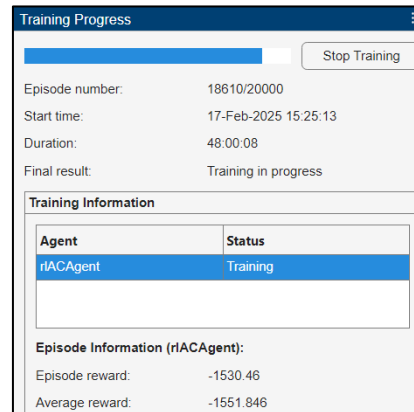


Figure 5.25 Training stats for A2C

In the Table 5.2 below, we have the comparison of the two agents after training

METRIC	A2C	PPO
Final Reward	-1500	-1070
Learning Stability	Higher fluctuations, instability	Steady learning curve, smaller fluctuations
Exploration	Excessive (entropy loss weight=0.1)	Controlled (entropy loss weight= 0.02)
Training Updates	Online updates (every 32 steps)	Batch updates (2048 steps)
Convergence Speed	Very slow (48 hours)	Faster, consistent improvements (28 hours)

Table 5.2 Comparison of A2C and PPO agent training results

In conclusion, while A2C does eventually learn a functional policy, its instability and slow convergence make it less effective for this production line optimization task. PPO proves to be the superior algorithm, achieving a much more reliable and optimized scheduling policy for our production line.

5.6 Limitations and discussion

While reinforcement learning works well for optimizing production lines, there are still a few important limits to consider.

A key challenge is the heavy computational cost of training. The PPO algorithm needs many training episodes to become stable, and this can be slow, especially when

CHAPTER 5: RESULTS AND DISCUSSION

simulating multiple machines that interact with each other. Using parallel computing or more efficient exploration methods, might help speed this up.

Another challenge is that the reward function depends on fixed cost values. But in real factories, these costs can change depending on things like production demand, material availability, or market changes. If the reward function were more flexible, the system could adjust better to these changing conditions.

Also, while the PPO agent managed to find a good balance between buffer usage and product flow, it still depends a lot on settings like the learning rate, exploration level, and batch size. Small changes to these hyperparameters can make a big difference, and finding the best combination often takes a lot of trial and error. A solution to that could be the usage of tools that automatically tune these hyperparameters and make the process easier and of course more efficient.

Even with these limitations, this study shows that reinforcement learning is a flexible and adaptable option compared to older scheduling methods. Because the agent learns directly from the system's feedback, it offers a strong way to improve how the production line works. With better reward design, faster training, and real-world testing, reinforcement learning could become an even more useful and scalable solution for industrial automation.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Summary of Contributions

In this thesis, we developed and implemented a reinforcement learning-based control strategy using the Proximal Policy Optimization (PPO) algorithm to optimize production line performance. Through the combination of dynamic SimEvents modeling in Simulink and reinforcement learning, we addressed key challenges related to buffer congestion, machine utilization, and client waiting times. The system demonstrated significant improvements over traditional methods by dynamically adapting machine states based on real-time conditions and feedback.

We also established a baseline using dynamic programming to evaluate the optimal cost for smaller configurations of the production line. The agent's performance was compared to this baseline, showing that the PPO algorithm was able to achieve near-optimal results, validating in that way the effectiveness of reinforcement learning for industrial optimization problems.

Additionally, we evaluated and compared PPO's performance with another RL algorithm, Advantage Actor-Critic (A2C), highlighting the effectiveness of PPO in handling large, multi-variable systems and optimizing long-term rewards.

6.2 Future Extensions

Several areas of improvement and extension are suggested for future research:

- **Expansion to Larger Systems:** While this work focused on a production line with five machines, future research could explore scalability to larger systems with more complex configurations. Techniques like hierarchical reinforcement learning could be considered to decompose the problem into manageable subproblems.
- **Exploration of Alternative RL Algorithms:** While PPO proved effective, future work could test other RL algorithms. Discrete-action algorithms, such as Deep Q-Networks (DQN) and Advantage Actor-Critic (A2C), can be tested and evaluated for stability and further performance improvements. Also, continuous-action algorithms, like Twin Delayed Deep Deterministic Policy Gradient (TD3) and Deep Deterministic Policy Gradient (DDPG), could be explored under the following condition. Applying these algorithms would require changing the way actions are defined and distributed in the production line model by the agent. Instead of having binary (0 for idle or 1 working) decisions for machine states, the action space could be designed to represent continuous signals. For example, those signals could be a percentage of machine power, processing speed, or other parameters. This modification could potentially allow for finer control over machine operations.
- **Hyperparameter Tuning:** We noticed that even small changes to hyperparameters made a big difference in how well the agent performed. Moving forward, we could use

automated techniques, like Bayesian optimization, to find the best settings for different production scenarios.

- **Real-World Implementation:** A critical next step is to validate the results in a real-world production line. Integrating the agent into real-time control systems and adapting it to real-world variations (e.g., sensor noise, unexpected failures) will be key.
- **Multi-Agent Systems:** As production lines become more complex, we could introduce multiple agents, each managing its own set of machines. This can lead to better overall performance and make the system easier to scale.

Overall, this thesis is an important step toward establishing production line optimization through adaptive and data-driven decision-making. The future extensions mentioned earlier offer solutions and ideas for further research and practical implementation, with the ultimate goal of bridging the gap between simulation-based models and real-world environments.

REFERENCES

- [1] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press. <https://doi.org/10.1093/oso/9780198538493.001.0001>
- [2] Tsourveloudis, N., Doitsidis, L., & Ioannidis, S. (2006). *Work in process control using fuzzy controllers*. Springer-Verlag London Limited. <https://doi.org/10.1007/s00170-006-0636-x>
- [3] Martins, M. S. E., Sousa, J. M. C., & Vieira, S. (2025). A Systematic Review on Reinforcement Learning for Industrial Combinatorial Optimization Problems. *Applied Sciences*, 15(3), 1211. <https://doi.org/10.3390/app15031211>
- [4] Priore, P., de la Fuente, D., Gómez, A., & Puente, J. (2001). Dynamic scheduling of manufacturing systems with machine learning. *International Journal of Foundations of Computer Science*. <https://doi.org/10.1142/s0129054101000849>
- [5] Vieira, G. E., Kück, M., Frazzon, E., & Freitag, M. (2017). Evaluating the robustness of production schedules using discrete-event simulation. *IFAC-PapersOnLine*. <https://doi.org/10.1016/j.ifacol.2017.08.896>
- [6] Ouelhadj, D., & Petrovic, S. (2008). A survey of dynamic scheduling in manufacturing systems. In *Journal of Scheduling* (Vol. 12, Issue 4, pp. 417–431). Springer Science and Business Media LLC. <https://doi.org/10.1007/s10951-008-0090-8>
- [7] Pinedo, M. L. (2022). *Scheduling* (pp. 187–224). Springer International Publishing. <https://doi.org/10.1007/978-3-031-05921-6>
- [8] Putra, M. R. S., Wibowo, A. S., & Utomo, A. R. P. (2021). Single machine production scheduling analysis using FCFS, SPT, LPT, and EDD methods. *Nusantara Science and Technology Proceedings*. <https://nstproceeding.com/index.php/nusciencetech/article/download/1281/1235/3969>
- [9] Zhang, Y., Hou, Z., & Liu, Z. (2022). Dynamic scheduling optimization of production workshops based on digital twin. <https://doi.org/10.3390/app122010451>

- [10] Qin, S. J., & Badgwell, T. A. (2003). A survey of industrial model predictive control technology. In *Control Engineering Practice* (Vol. 11, Issue 7, pp. 733–764). Elsevier BV. [https://doi.org/10.1016/s0967-0661\(02\)00186-7](https://doi.org/10.1016/s0967-0661(02)00186-7)
- [11] Streif, S., & Engell, S. (2021). Model predictive control for flexible job shop scheduling in manufacturing systems. *Applied Sciences*. <https://doi.org/10.3390/app11178145>
- [12] Li, Y. (2019). A deep reinforcement learning overview: Applications, challenges, and solutions. <https://doi.org/10.48550/arXiv.1908.06973>
- [13] Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. In *The International Journal of Robotics Research* (Vol. 32, Issue 11, pp. 1238–1274). SAGE Publications. <https://doi.org/10.1177/0278364913495721>
- [14] Yan, Y., Chow, A. H. F., Ho, C. P., Kuo, Y.-H., Wu, Q., & Ying, C. (2022). Reinforcement learning for logistics and supply chain management: Methodologies, state of the art, and future opportunities. In *Transportation Research Part E: Logistics and Transportation Review* (Vol. 162, p. 102712). Elsevier BV. <https://doi.org/10.1016/j.tre.2022.102712>
- [15] Ponse, K., Kleuker, F., Fejér, M., Serra-Gómez, Á., Plaat, A., & Moerland, T. (2024). Reinforcement Learning for Sustainable Energy: A Survey (Version 1). arXiv. <https://doi.org/10.48550/ARXIV.2407.18597>
- [16] Panzer, M., & Bender, B. (2021). Deep reinforcement learning in production systems: A systematic literature review. *International Journal of Production Research*. <https://doi.org/10.1080/00207543.2021.1973138>
- [17] Tassel, P., Gebser, M., & Schekotihin, K. (2021). A Reinforcement Learning Environment For Job-Shop Scheduling (Version 1). arXiv. <https://doi.org/10.48550/ARXIV.2104.03760>
- [18] Chung, J., Shen, B., Law, A. C. C., Zhenyu, & Kong. (2022). Reinforcement Learning-based Defect Mitigation for Quality Assurance of Additive Manufacturing (Version 1). arXiv. <https://doi.org/10.48550/ARXIV.2210.17272>
- [19] Luo, J., Solowjow, E., Wen, C., Ojea, J. A., & Agogino, A. M. (2018). Deep Reinforcement Learning for Robotic Assembly of Mixed Deformable and Rigid Objects. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp.

2062–2069). 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE. <https://doi.org/10.1109/iros.2018.8594353>

[20] Zhu, D., Yang, B., Liu, Y., Wang, Z., Ma, K., & Guan, X. (2022). Energy Management Based on Multi-Agent Deep Reinforcement Learning for A Multi-Energy Industrial Park. arXiv. <https://doi.org/10.48550/ARXIV.2202.03771>

[21] Kuhnle, A., Kaiser, J.-P., Theiß, F., Stricker, N., & Lanza, G. (2020). Designing an adaptive production control system using reinforcement learning. <https://doi.org/10.1007/s10845-020-01612-y>

[22] Olivares, D., Fournier, P., Vasishta, P., & Marzat, J. (2024). Model-free and model-based reinforcement learning for attitude control in fixed-wing UAVs. <https://doi.org/10.48550/arXiv.2409.17896>

[23] Degris, T., White, M., & Sutton, R. S. (2019). Off-policy and on-policy evaluation: A unified perspective. <https://arxiv.org/pdf/1905.01756>

[24] Shianifar, J., Schukat, M., & Mason, K. (2023). Optimizing deep reinforcement learning for adaptive robotic arm control: PPO and SAC hyperparameter optimization. (pp. 1-6) <https://doi.org/10.48550/ARXIV.2407.02503>

[25] Kapitan, V., Vasiliev, E., Kapitan, D. Y., & Rybin, A. (2023). Application of machine learning in solid state physics. Solid State Physics. <https://doi.org/10.1016/bs.ssp.2023.08.001>

[26] Weng, L. (2018). Policy gradient algorithms: An overview of policy-based reinforcement learning methods. <https://lilianweng.github.io/posts/2018-04-08-policy-gradient/>

[27] Lehmann, M. (2024). The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations. arXiv preprint arXiv:2401.13662. <https://doi.org/10.48550/arXiv.2401.13662>

[28] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. <https://doi.org/10.1109/MSP.2017.2743240>

[29] Nachum, O., Norouzi, M., Xu, K., & Schuurmans, D. (2017). Bridging the gap between value and policy-based reinforcement learning. <https://doi.org/10.48550/arXiv.1702.08892>

- [30] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. <https://doi.org/10.48550/arXiv.1707.06347>
- [31] <https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html>
- [32] Lim, H. K., Kim, J. B., Heo, J. S., & Han, Y. H. (2020). Federated reinforcement learning for training control policies on multiple IoT devices. *Sensors*, 20(5), 1359. <https://doi.org/10.3390/s20051359>
- [33] García-Hernández, R. A., Celaya-Padilla, J. M., Luna-García, H., García-Hernández, A., Galván-Tejada, C. E., Galván-Tejada, J. I., Gamboa-Rosales, H., Rondon, D., & Villalba-Condori, K. O. (2023). Emotional State Detection Using Electroencephalogram Signals: A Genetic Algorithm Approach. In *Applied Sciences* (Vol. 13, Issue 11, p. 6394). MDPI AG. <https://doi.org/10.3390/app13116394>
- [34] Rafailov, D., Dabney, W., & Van De Wiele, T. (2021). Generalized advantage estimation revisited: Bridging theory and practice in PPO. <https://arxiv.org/pdf/2109.06099>
- [35] Lim, H.-K., Kim, J.-B., Heo, J.-S., & Han, Y.-H. (2020). Federated reinforcement learning for training control policies on multiple IoT devices. *Sensors*, 20(5), 1359. <https://doi.org/10.3390/s20051359>
- [36] MathWorks. (n.d.). SimEvents documentation: Model and simulate discrete-event systems. <https://www.mathworks.com/help/simevents>
- [37] Clune, M., Mosterman, P., & Cassandras, C. (2006). Discrete Event and Hybrid System Simulation with SimEvents. In 2006 8th International Workshop on Discrete Event Systems (pp. 386–387). Proceedings. Eighth International Workshop on Discrete Event Systems. IEEE. <https://doi.org/10.1109/wodes.2006.382398>
- [38] Chastek, G., & McGregor, J. D. (2002). Guidelines for developing a product line production plan. Software Engineering Institute, Carnegie Mellon University. https://insights.sei.cmu.edu/documents/676/2002_005_001_14024.pdf (pp. 25-31)
- [39] MathWorks. Setting attributes of entities in SimEvents. <https://www.mathworks.com/help/simevents/ug/setting-attributes-of-entities.html>

- [40] Wikipedia contributors. Mean time between failures. https://en.wikipedia.org/wiki/Mean_time_between_failures
- [41] Ladan-Mozes, E., & Shavit, N. (2004). An optimistic approach to lock-free FIFO queues. Proceedings of the 18th International Symposium on Distributed Computing (DISC). https://people.csail.mit.edu/shanir/publications/FIFO_Queues.pdf
- [42] Natsheh, E. F. (2020). Queue Scheduling Algorithms: A Comparative Analysis. International Journal of Innovative Computing, 10(1), 21–25. <https://doi.org/10.11113/ijic.v10n1.257>
- [43] Mohd, F., Hassan, M. R., Rahim, A., & Saad, R. (2020). Design for manufacturing and assembly: A review. https://www.researchgate.net/publication/344661030_Design_for_Manufacturing_and_Assembly-Review
- [44] *Production line in a car factory – high-resolution image showing industrial robotics and automation.* Science Photo Library <https://www.science-photo.de/bilder/13405569-Production-line-in-a-car-factory>
- [45] *Robotic assembly line – conceptual image illustrating automated production processes.* Stockcake: https://stockcake.com/i/robotic-assembly-line_185854_32318
- [46] Mora, B., Retolaza, I., Campos, M. A., Ramirez, A., Cabello, M. J., & Martinez, F. (2020). DEVELOPMENT OF A NEW DESIGN METHODOLOGY FOR LARGE SIZE PRODUCTS BASED ON DSM AND DFMA. In Proceedings of the Design Society: DESIGN Conference (Vol. 1, pp. 2315–2324). Cambridge University Press (CUP). <https://doi.org/10.1017/dsd.2020.2>
- [47] Mishra, R., Zeeshan, M., & Singh, S. (2012). Two-way concurrent buffer system without deadlock in various time models using timed automata. <https://doi.org/10.48550/arXiv.1209.2376>
- [48] MathWorks. (n.d.). Reinforcement learning toolbox documentation. <https://www.mathworks.com/help/reinforcement-learning/>
- [49] MathWorks. rlPPOAgent (Reinforcement Learning Toolbox). <https://www.mathworks.com/help/reinforcementlearning/ref/rl.agent.rlppoagent.html>

[50] Icarte, R. T., Klassen, T. Q., Valenzano, R., & McIlraith, S. A. (2019). Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 64, 1-52. <https://doi.org/10.1613/jair.1.12440>

[51] Arwa, E. O., & Folly, K. A. (2020). Reinforcement Learning Techniques for Optimal Power Control in Grid-Connected Microgrids: A Comprehensive Review. In *IEEE Access* (Vol. 8, pp. 208992–209007). Institute of Electrical and Electronics Engineers (IEEE). <https://doi.org/10.1109/access.2020.3038735>