FPT UNIVERSITY QUY NHON AI CAMPUS

FACULTY OF INFORMATION TECHNOLOGY

REINFORCEMENT LEARNING

FPT UNIVERSITY

MINI CAPSTONE

CUTTING STOCK 2D PROBLEM

ARTIFICIAL INTELLIGENCE

STUDENT: BUI DINH THANH DANH (QE170016)

NGUYEN MINH PHAT (QE170068)

NGUYEN TRONG NGHIA (QE170151)

PHAM GIA THINH (QE170177)

CHE MINH QUANG (QE170206)

QUY NHON CITY, 3/2025

INSTRUCTOR: PhD. NGUYEN AN KHUONG

FPT UNIVERSITY QUY NHON AI CAMPUS

FACULTY OF INFORMATION TECHNOLOGY

REINFORCEMENT LEARNING

FPT UNIVERSITY

MINI CAPSTONE

CUTTING STOCK 2D PROBLEM

ARTIFICIAL INTELLIGENCE

STUDENT: BUI DINH THANH DANH (QE170016)

NGUYEN MINH PHAT (QE170068)

NGUYEN TRONG NGHIA (QE170151)

PHAM GIA THINH (QE170177)

CHE MINH QUANG (QE170206)

QUY NHON CITY, 3/2025

INSTRUCTOR: PhD. NGUYEN AN KHUONG

Acknowledgment

The successful completion of this course project would not have been possible without the support and guidance of many individuals. We would like to extend our heartfelt appreciation to our teachers and friends who have provided valuable assistance throughout the process.

A special thank you goes to Mr. Nguyen An Khuong for his continuous encouragement and insightful advice. His constructive feedback, thorough evaluations, and strategic direction from the early stages have played a crucial role in shaping our approach and refining our ideas. His dedication and expertise have had a lasting impact on the development of this project.

While we have strived to present a well-rounded project, we acknowledge that there may still be areas for improvement. We greatly appreciate any constructive criticism and recommendations that can help us enhance our understanding and refine our work for future endeavors.

Once again, we sincerely thank everyone who has contributed to this journey. Your support means a great deal to us!

Project Implementation Team

(Sign and print full name)

Bui Dinh Thanh Danh

Nguyen Minh Phat

Nguyen Trong Nghia

Pham Gia Thinh

Che Minh Quang

Commitment

Our graduation project team affirms that this project is developed based on certain existing documents while ensuring that no content or results have been copied from other works. All referenced materials have been properly cited in accordance with academic standards.

Project Implementation Team

(Sign and print full name)

Bui Dinh Thanh Danh

Nguyen Minh Phat

Nguyen Trong Nghia

Pham Gia Thinh

Che Minh Quang

Abstract

The Cutting Stock Problem (CSP) is a well-known optimization challenge in industrial manufacturing, particularly in the woodworking industry, where raw material utilization is crucial. This paper explores various approaches to solving CSP, including heuristic algorithms like First Fit and Best Fit, as well as advanced reinforcement learning techniques such as Q-learning and Actor-Critic. By integrating both traditional and AI-driven methods, our study aims to maximize material usage efficiency while minimizing waste. The proposed approach is evaluated on different stock sizes and product demands to assess its effectiveness in real-world scenarios.

Content

CHAPTER 1: OVERVIEW OF CUTTING STOCK PROBLEM

1.1 Problem Overview

The Cutting Stock Problem (CSP) arises in manufacturing industries that deal with cutting large raw materials into smaller pieces while minimizing waste. In the woodworking industry, this translates to cutting large wooden boards into predefined product sizes efficiently. The challenge lies in selecting cutting strategies that optimize material usage while ensuring that all required products are cut.

Traditional heuristic algorithms like First Fit and Best Fit provide quick, rule-based solutions but often fall short in handling dynamic and large-scale cases effectively. To enhance performance, machine learning techniques such as reinforcement learning (Q-learning, Actor-Critic) can be employed to learn optimal cutting strategies from experience. This research aims to compare the effectiveness of these methods and develop a more adaptive approach for CSP.

1.2 Application of Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment. The agent receives feedback in the form of rewards and gradually improves its decision-making policy through trial and error. RL is particularly suitable for optimization problems like CSP, where decisions at each step (cutting actions) influence future outcomes (remaining stock and waste).

1.3 Research Objectives

The primary objective of this research is to develop an **efficient and adaptive cutting strategy** for the **Cutting Stock Problem (CSP)**, specifically in the context of **wood cutting**. The study aims to minimize material waste while ensuring all required products are obtained from the available stock.

To achieve this, we investigate both **traditional heuristic methods** and **reinforcement learning (RL) approaches** to compare their performance in solving the problem effectively.

The specific targets of this research include:

- **Minimizing Waste:** Reduce the amount of unused material by optimizing cutting patterns.
- **Comparative Analysis:** Evaluate and compare heuristic approaches (**First Fit, Best Fit, Combination**) with RL-based techniques (**Q-learning, Actor-Critic**) to determine the most effective method.
- **Adaptive Decision-Making:** Enable the cutting strategy to dynamically adjust to different stock configurations and product demands.
- **Scalability:** Ensure that the proposed solutions can handle large problem instances with multiple stock pieces and product types.
- **Real-World Applicability:** Develop an approach that can be applied in industrial woodworking processes, improving resource utilization and cost efficiency.

By achieving these goals, this research contributes to both **optimization theory** and **practical manufacturing applications**, demonstrating how machine learning can enhance traditional problem-solving methods in industrial settings.

1.4 Study Constraints

While our approach aims to optimize the wood-cutting process using various algorithms, certain limitations exist:

- **Computational Complexity:** Advanced RL models like Actor-Critic require significant computational resources and training time, making real-time applications challenging.
- **State Space Limitations:** The environment's state representation is simplified for feasibility, which may not capture all real-world constraints, such as material defects or cutting precision.
- **Algorithm Performance Variability:** Traditional heuristics (First Fit, Best Fit) perform well in specific scenarios but may be suboptimal in dynamic environments compared to RL-based approaches.
- **Generalization Challenges:** The trained RL models may not generalize well to unseen stock and product configurations, requiring fine-tuning for different datasets.

- **Physical Constraints Ignored:** The study does not consider machine limitations, blade thickness, or material loss, which are crucial in industrial applications.

Future work should address these limitations by incorporating real-world constraints, improving computational efficiency, and refining state representations for better adaptability.

CHAPTER 2: DEFINE DATASET

2.1 Dataset

In this study, we utilize a **synthetic dataset** to simulate the wood-cutting optimization problem. The dataset consists of information about **input wood sheets (stocks)** and **products to be cut**, designed to evaluate the performance of optimization algorithms.

Data Structure

The dataset consists of four main columns:

- **batch_id**: Batch identification number
- **type**: Object type (Stock/Product)
- **width**: Width of the stock or product
- **height**: Height of the stock or product

Stock Characteristics

- **Average size**: 76.7 × 81.5 (width × height)
- **Smallest size**: 15 × 15
- **Largest size**: 150 × 150

Product Characteristics

- **Average size**: 24.8 × 25.0 (width × height)
- **Minimum size**: 15 × 15
- **Largest size**: 35 × 35

Dataset Overview

- **Total Stocks**: 200 (10 stocks per batch × 20 batches)
- **Total Products**: 494
- **Average Product/Stock Ratio**: 2.47

This dataset was generated to provide a controlled and scalable environment for testing different optimization algorithms, including heuristic and reinforcement learning-based approaches.

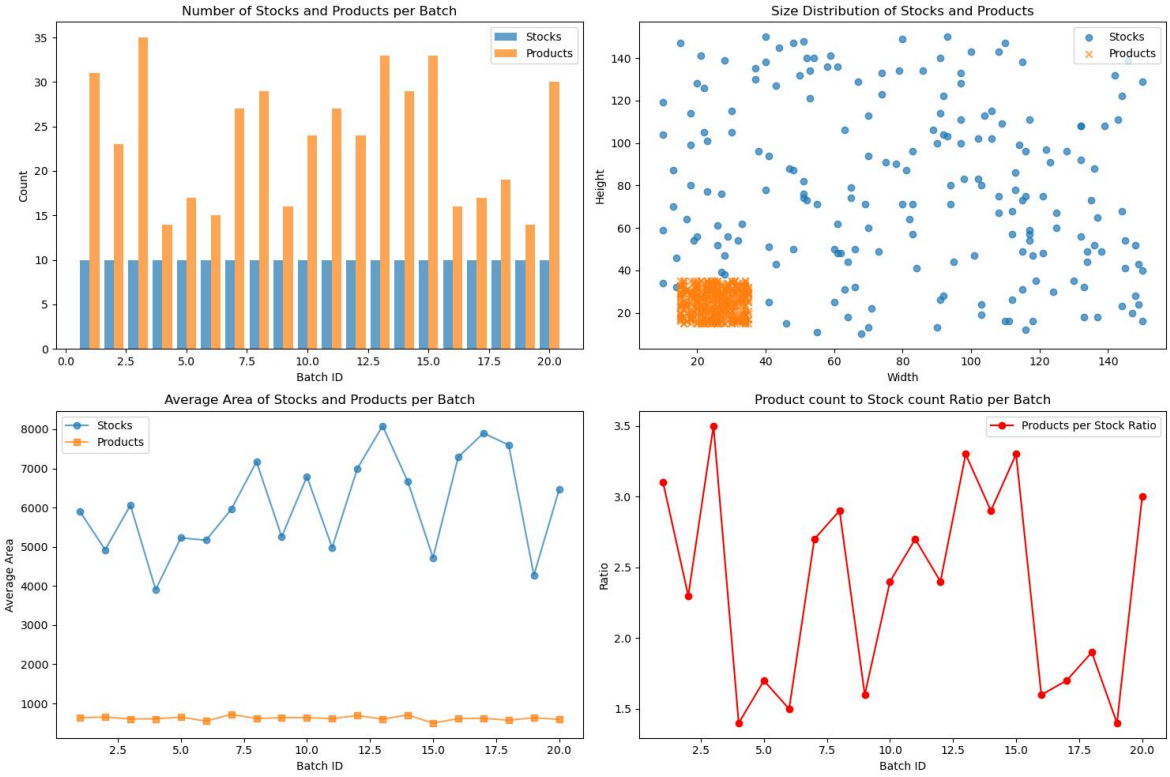


Chart visualization information of dataset

2.2 Time-line and Task List

The research process is structured into multiple phases to ensure a systematic and efficient approach to solving the Cutting Stock Problem using various algorithms. The timeline outlines key tasks, responsible individuals, supporting members, start and end dates, and completion percentages.

The project is divided into the following major sections:

- **General Work:** Defining requirements, problem statements, and planning.
- **Data Preparation:** Generating and analyzing datasets for training and evaluation.
- **Environments Setup:** Installing necessary libraries and building a simulation environment.
- **Algorithms Development:** Implementing heuristic-based and reinforcement learning (RL) methods, including First Fit, Best Fit, Combination, Q-learning, and Actor-Critic.
- **Final Review & Report:** Summarizing results, evaluating algorithm performance, and documenting findings.

Link details task list:

Reinforcement Learning_ Task list

TASK ID	TASK TITLE	PIC	Support	START DATE	END DATE	DURATION IN DAYS	PCT OF TASK COMPLETE
0	General work			21/01/2025	31/01/2025		
0.1	Clear requirements	Nghĩa,Danh, Phát, Thịnh, Quang	Mr. Khương	21/01/2025	22/01/2025		100%
0.2	Define solutions	Nghĩa,Danh, Phát, Thịnh, Quang		23/01/2025	25/01/2025		100%
0.3	Task list and timeline for project	Nghĩa,Danh, Phát, Thịnh, Quang		26/01/2025	27/01/2025		100%
0.4	Plan deatails	Nghĩa,Danh, Phát, Thịnh, Quang		28/01/2025	30/01/2025		100%
0.5	Research	Nghĩa,Danh, Phát, Thịnh, Quang		31/01/2025			
1	Data			1/2/2025	8/2/2025		
1.1	Build plan to collect data	Nghĩa,Danh, Phát, Thịnh, Quang		1/2/2025	1/2/2025		100%
1.2	Research dataset	Nghĩa,Danh, Phát, Thịnh, Quang		2/2/2025	6/2/2025		100%
1.3	Generate data	Nghĩa,Danh, Phát, Thịnh, Quang		2/2/2025	6/2/2025		100%
1.4	Review & Report	Nghĩa,Danh,Phát,Quang	Thịnh	7/2/2025	8/2/2025		100%
2	Environemnts			9/2/2025	16/2/2025		
2.1	Define the components of Reinforcement Learning	Nghĩa,Danh, Phát, Thịnh,Quang		9/2/2025	11/2/2025		100%
2.2	Build plan & setup environments (install library, algorithms,..)	Nghĩa,Danh, Phát, Thịnh, Quang		12/2/2025	13/2/2025		100%
2.3	Building a simulation environment - CuttingStockEnv	Nghĩa,Danh, Phát, Thịnh, Quang		14/02/2025	16/02/2025		100%
3	Algorithms			17/2/2025	15/3/2025		
3.1	Research and Select Algorithms (Q-Learning, Actor-Critic, Heuristic-based approaches)	Nghĩa,Danh, Phát, Thịnh, Quang		17/2/25	18/2/2025		100%
3.2	Implement Heuristic Policies (First Fit, Best Fit, Combined Approach)	Nghĩa,Danh,Thịnh	Quang,Phát	19/2/25	28/2/25		100%
3.3	Implement Q-Learning Algorithm	Nghĩa,Phát,Quang	Thịnh,Danh	19/2/25	28/2/25		100%
3.4	Implement Actor-Critic Algorithm	Thịnh,Danh	Nghĩa,Phát	19/2/25	28/2/25		100%
3.5	Train and Fine-tune RL Agents	Nghĩa,Danh, Phát, Thịnh, Quang		1/3/25	12/3/25		100%
3.6	Compare Performance between Heuristic and RL-based Approaches	Nghĩa,Danh, Phát, Thịnh, Quang		13/3/25	15/3/25		100%
4	Final Review & Report			16/03/2025	23/03/2025		
4.1	Check and review the entire project, create a doc file to describe in detail the operating principles and the team's problem solving process.	Nghĩa,Danh, Phát, Thịnh, Quang		16/03/2025	19/03/2025		100%
4.2	Prepare ppt file to report results	Nghĩa,Danh, Phát, Thịnh, Quang		20/03/2025	23/03/2025		100%

Each task is assigned to team members with specified deadlines, ensuring progress tracking and project completion within the planned schedule.

CHAPTER 3: ENVIROMENT

3.1. INTRODUCTION

The cutting stock problem is a classic NP-hard combinatorial problem in the field of optimization. The problem requires cutting large sheets of material (stocks) into smaller products (products) in an optimal way, minimizing waste. The problem has many applications in industries from cutting metals, wood, paper, fabric to applications in microchip design and architecture.

In this paper, we present a standardized reinforcement learning environment for the material cutting problem, following the Gymnasium specification (the successor to OpenAI Gym). This environment allows the implementation and evaluation of reinforcement learning algorithms (Q-learning, DQN, PPO, etc.) as well as traditional heuristic methods, providing a basis for performance comparisons between methods.

3.2. MATERIAL CUTTING PROBLEM

3.2.1. Mathematical definition

The two-dimensional material cutting problem can be defined as follows:

- A set $S = \{s_1, s_2, ..., s_m\}$ of material plates (stocks), each plate s_i has size $w_i \times h_i$
- A set $P = \{p_1, p_2, ..., p_n\}$ of products, each product p_j has size $w'_j \times h'_j$ and quantity to be cut q_j

Requirement: Find a way to place the products on the material panels so that:

1. No products overlap
2. Each product is completely contained within the material sheet.
3. Minimize the number of panels used and/or maximize the usable area

3.2.2. Characteristics and constraints

Our environment considers the following characteristics and constraints:

- The cutting method is orthogonal (horizontal and vertical)
- Products can be rotated 90° (horizontal or vertical)
- No constraints on cutting order
- Sheets of material and products of original size

3.3. ENVIRONMENTAL DESIGN

3.3.1. Overall architecture

The CuttingStockEnv environment is designed to follow the Gymnasium specification, with the following main components:

CuttingStockEnv

│

├── **__init__()** - Initialize the environment

├── **reset()** - Reset state

├── **step()** - Performs action and transitions state

├── **render()** - Displays the current state

└── **close()** - Release resources

The environment is designed according to the principle of modularity, with components separated for easy maintenance and expansion:

- constants.py: Contains constants and metadata
- utils.py: Utility functions
- actions.py: Defines the action space and related functions
- renderer.py: Handles rendering and visualization

3.3.2 Data representation

3.3.2.1 Representation of material plates (stocks)

Each material sheet is represented as a two-dimensional array with the values:

- -2: Area outside the actual size of the material sheet
- -1: Empty area, can place products
- ≥0: The product index has been placed at that position

This representation allows:

- Quickly identify empty areas
- Track the location of each product
- Efficiently calculate metrics such as utilization and waste

3.3.2.2 Product presentation (products)

Each product is represented as a dictionary with the following information:

- size: Size (width, height) of the product
- quantity: Number of products to be cut

This design allows easy extension to add other attributes such as priority weights, deadlines, etc.

3.3.3 State and action space

3.3.3.1 Observation Space

The state of the environment is represented by a dictionary with two main components:

```
    {
        "stocks": spaces.Tuple([spaces.MultiDiscrete(upper, start=lower)] * self.num_stocks),
        "products": spaces.Sequence(
            spaces.Dict({
                "size": spaces.MultiDiscrete(np.array([max_w, max_h]), start=np.array([1, 1])),
                "quantity": spaces.Discrete(max_product_per_type + 1, start=0),
            })
        ),
    }
)
```

The status provides complete information about:

- Current status of all material panels
- List of remaining products to be cut with corresponding quantities

3.3.3.2 Action Space

Each action is a dictionary with three components:

```
action_space = spaces.Dict(
        {
            "stock_idx": spaces.Discrete(env.num_stocks),
            "size": spaces.Box(
                low=np.array([1, 1]),
                high=np.array([env.max_w, env.max_h]),
                shape=(2,),
                dtype=int,
            ),
            "position": spaces.Box(
                low=np.array([0, 0]),
                high=np.array([env.max_w - 1, env.max_h - 1]),
                shape=(2,),
                dtype=int,
            ),
        }
    )
```

Such action design allows for full specification of a cut decision:

- stock_idx: Select which material plate
- size: Product size to be cut (can be rotated)
- **position: Product placement on the material sheet**

3.3.4 State transition mechanism

The step() method handles the state transition logic when performing an action:

1. Check the validity of the action:
    - Does the selected material exist?
    - The product has the size that meets the requirements and the quantity is > 0
    - Is the placement located within the material sheet?
    - Is the booking area still available?

2. If the action is valid:
    - Marking of used material
    - Update position on material sheet
    - Reduce the number of products remaining

3. Check the termination condition:
    - If all products have been cut, status terminated = True

4. Calculate rewards and return new information

3.3.5 Reward mechanism

In the basic version, the environment uses sparse rewards:

- +1 when all products have been successfully cut
- 0 in all other cases

This design encourages the agent to search for the complete solution, but may cause learning difficulties due to insufficient reward signals. Therefore, in practical algorithms, we often add a reward shaping mechanism.

3.4 EVALUATION METRICS

The _get_info() method provides important metrics to evaluate the effectiveness of the solution:

```python
def _get_info(self):
    filled_ratio = np.mean(self.cutted_stocks).item()
    trim_loss = []
    for sid, stock in enumerate(self._stocks):
        if self.cutted_stocks[sid] == 0:
            continue
        tl = (stock == -1).sum() / (stock != -2).sum()
        trim_loss.append(tl)
    trim_loss = np.mean(trim_loss).item() if trim_loss else 1
    return {"filled_ratio": filled_ratio, "trim_loss": trim_loss}
```

3.4.1. Filled Ratio

Filled ratio measures the proportion of sheet material used:

Filled_ratio = Used stock \ Total stock

Note:

- **Used stock** : is the number of units (panels, cells, elements...) that have been used, occupied or filled.
- **Total stock** : is the total quantity that can be used.
- **filled_ratio** : is a ratio that represents the level of utilization compared to the overall capacity. Usually ranges from 0 to 1 (or expressed as a percentage from 0% to 100%).

This index needs to be maximized.

3.4.2. Trim Loss

Trim loss measures the percentage of wasted area on used panels:

Trim_loss = Number of Empty Cells \ Total Valid Cells

Note:

- **Number of Empty Cells** : Total unused area on used panels (including gaps that cannot be utilized due to geometric limitations).
- **Total Valid Cells** : Total area of all panels used to cut the part.
- **Trim_loss** : Value from 0 to 1 (or 0% to 100%), the lower the higher the material utilization efficiency.

This index should be minimized.

3.4.3. Balance between metrics

In practice, there is a balance between filled ratio and trim loss. An optimal solution will use as few panels as possible and maximize the area utilization per panel.

CHAPTER 4: ALGORITHM

4.1 Heuristic algorithms

Heuristic algorithms serve as practical approaches to problem-solving that emphasize obtaining near-optimal solutions with high efficiency, particularly in situations where computing exact solutions would be too time-consuming or impractical. These techniques make use of simplified rules, experiential insights, or strategy based shortcuts to reduce the size and complexity of the solution space. By deliberately overlooking less favorable options, heuristics aim to produce sufficiently good results within acceptable computational limits, even if those results are not guaranteed to be optimal.

For the Cutting Stock Problem, heuristic approaches play a vital role in constructing viable cutting layouts while maintaining a balance between accuracy and computational performance. This balance is often achieved by applying constraints such as restricting the number of cutting phases or adhering to specific cutting rules like guillotine-only cuts. These problem-specific limitations streamline the decision space, making it feasible to derive solutions that fulfill demand requirements while minimizing offcuts and material loss.

Though heuristic solutions are inherently approximate, they prove to be exceptionally useful in industrial environments where rapid processing and efficient resource allocation are critical. In particular, two-dimensional cutting and packing problems benefit greatly from heuristics, as they can deliver high-quality outcomes in a fraction of the time required by exact algorithms. Their ability to effectively address the combinatorial complexity of the NP-hard Cutting Stock Problem without exhaustive search makes them an essential tool in both research and practical applications.

4.1.1 First - Fit Algorithm for 2D Cutting - Stock

The First-Fit Algorithm is a practical heuristic for solving 2D cutting-stock problems, aiming to place rectangular flooring wood pieces onto stock sheets while minimizing waste. It efficiently places each piece on the first suitable stock sheet, balancing speed and feasibility. Although not always optimal, its computational efficiency makes it suitable for practical scenarios like flooring wood cutting.

In our implementation, the algorithm prioritizes larger items and optimizes stock usage, operating as follows:

1. **Sort Products by Area:** Products are sorted by area $W_i \times H_i$ in descending order to place larger pieces first.
2. **Sort Stocks by Usable Space:** Stock sheets are sorted by the number of usable cells (not -2) in descending order.
3. **Process Each Product:** Iterate through products, skipping those with fulfilled demand (quantity ≤ 0).
4. **Find the First Suitable Position:** For each product, check stock sheets in order. If a stock sheet's usable dimensions (W_j^{Stock}, H_j^{Stock}) fit the product, scan from top-left to bottom-right for the first empty region (all cells = -1) of size $W_i \times H_i$
5. **Place or Add New Stock:** Place the product if a position is found; otherwise, introduce a new stock sheet (index *m*, increment *m*) and place the product at (0,0).
6. **Handle Completion:** Return a default action (stock index 0, size (0,0), position (0,0)) if no products remain.

Let:

- n: number of products to cut.
- m: number of stock sheets used.
- W_i, H_i : width and height of the *i*-th product.
- W_j^{Stock}, H_j^{Stock} : usable width and height of the *j*-th stock sheet.

The objective is to minimize the number of stock sheets while respecting their dimensions. The placement condition is:

- Find the first stock sheet *j* such that $W_i \leq W_j^{Stock}$, with an empty region for placement.
- If no stock sheet fits, create a new one (m=m+1).

The time complexity is O($n \cdot m \cdot W \cdot H$), where *W* and *H* are the maximum stock sheet dimensions. This implementation respects the non-rotatable constraint of flooring wood cutting, but

its greedy nature may lead to suboptimal material usage, prompting exploration of advanced methods in later sections.

```
Function First_fit_2DCSP:
    push one of each stock_type into the stock_list.
    for stock_i from the largest to smallest area in stock_list:
        for prod_j from the largest to smallest area in product_list:
            if prod_j can fit into stock_i:
                place prod_j into stock_i
            else:
                let "stockType" be the first stock_type that fits prod_j
                create a new stock of type "stockType"
                place prod_j into the new stock
            if the textbfdemand for prod_j is textbfmet:
                remove prod_j from the item list
```

For each product, the algorithm may need to evaluate all existing stock sheets to identify a suitable position, and if none is found, a new stock sheet is added. The First-Fit Algorithm serves as a straightforward and efficient heuristic for tackling 2D cutting-stock problems, particularly in flooring wood cutting, due to its compliance with non-rotatable constraints. Although it may not yield an optimal solution, its simplicity and speed make it effective for practical applications. However, its limitations in optimizing material usage and managing cutting complexity highlight areas for enhancement. In this report, we seek to investigate an improved heuristic that better balances material efficiency and computational simplicity, with further details explored in the upcoming sections.

4.1.2 Best- Fit Algorithm for 2D Cutting - Stock

The Best-Fit Algorithm is a strategic heuristic for 2D cutting-stock problems, designed to place rectangular flooring wood pieces onto stock sheets while minimizing material waste. Unlike the First-Fit approach, it selects the stock sheet that results in the least remaining unused space after placement, aiming for optimal material utilization. While more computationally intensive, this method often achieves better efficiency in resource-constrained scenarios like flooring wood cutting.

In our implementation, the algorithm prioritizes larger items and optimizes stock usage, operating as follows:

1. **Sort Products by Area:** Products are sorted by area $W_i \times H_i$ in descending order to place larger pieces first.
2. **Process Each Product:** Iterate through products, skipping those with fulfilled demand (quantity ≤ 0).
3. **Find the Best-Fit Stock:** For each product, evaluate all stock sheets. If a stock sheet's usable dimensions (W_j^{Stock}, H_j^{Stock}) can accommodate the product, scan for all possible positions and calculate the remaining unused area after placement. Select the stock sheet and position that minimize this remaining area.
4. **Place or Add New Stock:** Place the product in the selected position if a suitable stock sheet is found, updating the stock and reducing the product's quantity. If no stock sheet fits, introduce a new stock sheet (index m, increment m m m) and place the product at (0,0).

Let:

- n: number of products to cut.
- m : number of stock sheets used.
- $W_i \times H_i$: width and height of the i-th product.
- W_j^{Stock}, H_j^{Stock}: usable width and height of the j -th stock sheet.

The objective is to minimize material waste by selecting the stock sheet j that minimizes the remaining area after placing the product, where the remaining area is calculated as the number of empty cells (marked as -1) minus the product's area $W_i \times H_i$. If no stock sheet fits, a new one is created (m=m+1).

The time complexity is O ($n \cdot m \cdot W \cdot H$), where *W* and *H* are the maximum stock sheet dimensions, as the algorithm evaluates all stock sheets and positions for each product. This approach is well-suited for flooring wood cutting, respecting the non-rotatable constraint, and typically achieves better material efficiency than First-Fit. However, its higher computational cost prompts exploration of hybrid methods, as discussed in later sections.

4.1.3 Combination Policy for 2D Cutting-Stock

The Combination Policy integrates the strengths of First-Fit, Best-Fit, and stock merging strategies to address 2D cutting-stock problems, aiming to place rectangular flooring wood pieces onto stock sheets with both speed and material efficiency. This hybrid approach balances computational

efficiency with optimal resource utilization, making it highly effective for practical applications like flooring wood cutting.

The algorithm operates in three sequential phases:

1. **First-Fit Placement:** Initially, the First-Fit strategy is applied to quickly place products into the first available stock sheet. Products are sorted by area $W_i \times H_i$ in descending order, and stock sheets are prioritized based on the number of usable cells (not -2). This phase focuses on speed, placing each product in the first empty region (marked as -1) that fits, ensuring rapid initial placement.
2. **Best-Fit Refinement:** If First-Fit fails to place a product, the Best-Fit strategy is employed to optimize placement. It evaluates all possible positions across stock sheets, selecting the one that minimizes the S_{ij} metric, defined as $S_{ij} = (x + W_i) \times (y + H_i$, where (x,y) is the top-left position of the product. This ensures the product is placed as close to the top-left corner as possible, reducing fragmentation and improving space utilization.
3. **Stock Merging:** Finally, the algorithm attempts to reduce the number of stock sheets by merging smaller stocks into larger ones. Stock sheets are sorted by usable area, and if a smaller stock (with area A_i) can fit into a larger stock (with area A_i, where $A_i < A_j$), the contents of the smaller stock are transferred to the larger one at position (0,0), minimizing the total number of stock sheets used.

Let:

- n: number of products to cut.
- m: number of stock sheets used.
- $W_i \times H_i$: width and height of the i-th product.
- W_j^{Stock} , H_j^{Stock} : usable width and height of the j-th stock sheet.

```
Function Combination_2DCSP:
    for stock_i from the largest to smallest area in stock list:
        for prod_j from the largest to smallest area in product list:
            for (x, y) in all position can cut prod_j from stock_i:
                if cut prod_j at position (x, y):
                    let S_ij be the area of the smallest rectangle contain cut area
            let (x_0, y_0) be the position where the value of S_ij is smallest
            cut the prod_j from stock_i at position (x_0, y_0)
    for stock_i from the smallest to largest area in cut stock list:
        for stock_j from the smallest to largest area in uncut stock list:
            if cut area of stock_i is smaller than stock_j then:
                move cut set of stock_i to stock_j
```

The objective is to minimize both the number of stock sheets and material waste while maintaining computational efficiency. The following pseudo-code outlines the Combination Policy:

The time complexity is $O(n \cdot m \cdot W \cdot H)$ where W and H are the maximum stock sheet dimensions, as each phase may need to evaluate all stock sheets and positions. If no action is found after all phases, a default action (stock index 0, size (0,0), position (0,0)) is returned.

This combination policy is particularly effective for flooring wood cutting, as it respects the non-rotatable constraint and combines the speed of First-Fit with the optimization of Best-Fit and stock merging. By addressing both speed and material efficiency, it offers a robust solution, paving the way for further exploration of advanced techniques like Reinforcement Learning in the next sections.

4.1.4 Conclusion

In this section, we explored three heuristic algorithms First-Fit, Best-Fit, and Combination Policy to address the 2D cutting-stock problem for flooring wood cutting, focusing on minimizing material waste while respecting the non-rotatable constraint due to fixed grain patterns.

The **First-Fit Algorithm** prioritizes speed and simplicity, placing products into the first available stock sheet that can accommodate them. Its time complexity of $O(n \cdot m \cdot W \cdot H)$ ensures rapid execution, making it ideal for scenarios where computational efficiency is critical. However, its greedy approach often leads to suboptimal material utilization, as it does not consider the overall efficiency of stock usage.

The **Best-Fit Algorithm** improves upon First-Fit by selecting the stock sheet that minimizes remaining unused space after placement,achieving better material efficiency.While it shares the same time complexity of $O(n \cdot m \cdot W \cdot H)$, its exhaustive evaluation of all possible positions increases computational cost, making it more suitable for applications where material optimization is a priority over speed.

The **Combination Policy** integrates the strengths of both approaches, using First-Fit for rapid initial placement, Best-Fit for space optimization, and stock merging to reduce the number of stock sheets. This hybrid strategy, with a time complexity of $O(n \cdot m \cdot W \cdot H)$, effectively balances speed and material efficiency, making it a robust solution for flooring wood cutting. However, its multi-phase nature can still be computationally intensive for large-scale problems.

Overall, while these heuristic algorithms provide practical solutions for the 2D cutting-stock problem, their limitations in achieving optimal material utilization and handling complex scenarios highlight the need for more advanced methods. In the following sections, we will explore Reinforcement Learning approaches to further enhance performance, aiming to overcome the trade-offs between computational efficiency and material optimization observed in these heuristics.

4.2 Reinforcement Learning Algorithm

4.2.1 Q-learning Method

4.2.1.1 Overview about Q-learning and Q-tables

Before addressing the Cutting Stock environment, we provide a brief introduction to Q-learning and the concept of a Q-table.

4.2.1.1.1 What is a Q-Table?

- A Q-table is a 2D matrix where each row corresponds to a state s and each column corresponds to an action a.
- The entry Q(s,a) represents the expected future reward when taking action a in state s, followed by the current or a greedy policy.

- In tabular Q-learning, Q(s,a) values are stored and updated through environment interactions, making it suitable for discrete problems like Cutting Stock.

4.2.1.1.2 Q-learning Algorithm

- Q-learning is an off-policy Reinforcement Learning method introduced by Watkins & Dayan (1992). Its core update rule is:

  $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max Q(s',a') - Q(s,a)]$
  - s: current state.
  - a: action taken.
  - r: immediate reward.
  - s': next state.
  - α ($0 \leq \alpha \leq 1$): learning rate.
  - γ ($0 \leq \gamma \leq 1$): discount factor.
- Over multiple episodes, the Q-table converges to optimal values if all state-action pairs are sufficiently explored.

4.2.1.1.3 Action Selection: Epsilon-Greedy Strategy

- Q-learning balances exploration and exploitation:
  - With probability ε, a random action is chosen (exploration).
  - Otherwise, the action argmax Q(s,a) is selected (exploitation) from the Q-table.
- During training, ε decays from a high value (1.0) to a low value (0.01) to shift toward exploitation.

4.2.1.2 Q-learning Agent

4.2.1.2.1 State Encoding

- Directly encoding the full 2D layout into a Q-table is impractical due to size. Instead, two features are extracted:
  - empty_space: Total count of free cells (-1) in stocks.
  - remaining_products: Number of uncut products.

- Combined into a state index:
  state = (empty_space × 1000 + remaining_products) % state_size
  (where state_size = 100000).
- This simplification reduces spatial detail but keeps the Q-table manageable.

4.2.1.2.2 Action Selection

- Actions are represented as integers from 0 to action_size - 1 (action_size = 1000).
- The get_env_action(action, observation) function converts this into a specific action:
  - prod_idx = action % len(products): Selects the product to cut.
  - stock_idx = (action // len(products)) % len(stocks): Selects the stock to use.
  - Placement: Scans for a free region (if stock is an array) or picks randomly (if stock is a tuple).
- If no valid placement is found, a default action (stock_idx=0, size=(0,0), position=(0,0)) is returned.

4.2.1.2.3 Q-update

- After receiving reward r and transitioning to state s', the Q-table is updated:
  $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max Q(s',a') - Q(s,a)]$
  - $\alpha = 0.3$, $\gamma = 0.9$ in the code.
- ε decays via epsilon = max(min_epsilon, epsilon * epsilon_decay) (min_epsilon = 0.01, epsilon_decay = 0.995).

4.2.1.3 Overview of Reward and Penalty in the Project

The reward in this implementation, computed by get_reward(), comprises three components:

- **Filled Ratio**: Proportion of used stock area occupied by products.
- **Trim Loss**: Proportion of unused space in partially cut stocks.
- **Bonus for Unused Stocks**: Incentive for leaving stocks untouched, reducing material use.

The default environment reward is 1 when cutting is complete (done) and 0 otherwise, but the custom reward integrates these factors for finer feedback.

4.2.1.4 How Rewards Are Computed

4.2.1.4.1 Filled Ratio and Trim Loss

1. Filled Ratio

- Retrieved from info["filled_ratio"] (default 0.5), it measures the efficiency of stock usage.
- Higher values positively contribute to the reward, promoting effective cutting.

2. Trim Loss

- Retrieved from info["trim_loss"] (default 0.2), it quantifies wasted space in used stocks.
- This is subtracted from the reward, penalizing inefficient strategies.

4.2.1.4.2 Bonus for Unused Stocks

- Counts num_stocks_unused by checking stocks with no cuts (values not differing from -2).
- Computed as: num_stocks_unused / total_stocks.
- Multiplied by lambda_bonus = 0.2 to add a bonus, encouraging minimal stock usage.

4.2.1.4.3 Combining the Components

- Total reward:

  Reward = (Filled Ratio - Trim Loss) + Bonus for Unused Stocks
  - **Positive**: Filled Ratio (efficient use), Bonus (fewer stocks used).
  - **Negative**: Trim Loss (waste penalty).

4.2.1.4.4 Additional Penalties or Edge Cases

- Zero Reward During Ongoing Steps

  If done = False, the default reward is 0, indirectly penalizing unproductive steps.

- Invalid Cutting Actions

  Invalid actions (e.g., out-of-bounds cuts) result in a default action, with no direct penalty but no state improvement.

4.2.1.4.5 Impact on Agent Behavior

- The reward structure guides the agent to:

- Maximize product density in used stocks (high Filled Ratio).
- Minimize waste (low Trim Loss).
- Concentrate cuts on fewer stocks (high Bonus).

- This aligns with practical goals: reducing material consumption and waste.

4.2.2 Actor Critic Methods:

4.2.2.1 Overview of Actor-Critic Methods:

Actor-Critic (AC) methods are a popular class of **reinforcement learning (RL)** algorithms that combine both policy-based and value-based learning to improve stability and efficiency in training. Unlike traditional policy gradient methods that suffer from high variance or value-based methods that struggle in high-dimensional continuous action spaces, Actor-Critic methods leverage the strengths of both approaches.

Key Components

- **Actor**
  - The actor is responsible for selecting actions based on the current policy $\pi(s;\theta)$.
  - It updates the policy parameters in the direction of better-performing actions using feedback from the critic.
  - The actor ensures that the policy gradually improves over time, leading to better decision-making.

- **Critic**
  - The critic evaluates how good an action taken by the actor is, using either a value function $V(s)$ or an advantage function $A(s,a)$.
  - It provides feedback to the actor by estimating the expected future rewards.
  - By reducing variance in policy updates, the critic helps make learning more stable and sample-efficient.

4.2.2.2 Architecture of Actor-Critic:

- **The Actor** observes the current state from the environment and selects an action.
- The selected action is executed in the **environment**, resulting in a **reward** and a new state.
- **The Critic** receives the new state and reward, then estimates the state value using the **value function** $\boldsymbol{V(s)}$.
- **TD Error** is computed to measure the discrepancy between the estimated value and the actual received reward.
- **The Actor** updates the policy based on feedback from the Critic, aiming for more optimal actions in the future.
- **The Critic** updates the state value to improve the accuracy of future estimations.

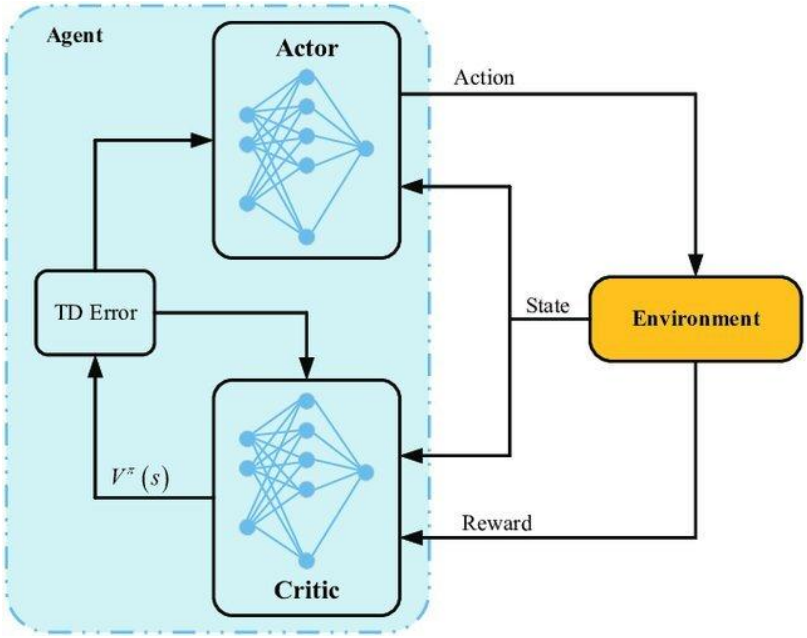


Figure 5 - available via license: Hybrid NOMA/OMA-Based Dynamic Power Allocation Scheme Using Deep Reinforcement Learning in 5G Networks

(https://www.mdpi.com/2076-3417/10/12/4236)

4.2.2.3 Key algorithm elements in Actor Critic:

1. Temporal Difference (TD) Error:

- TD Error measures the discrepancy between the estimated value of a state and the actual received reward from the environment. It is calculated using the formula:

$$TD\ Error = r_t + \gamma V(s_{t+1}) - V(s_t)$$

- In that:
  - $V(s_t)$: Estimated value of the current state.
  - r_t: Reward received after taking an action at state sts_tst .
  - $\gamma V(s_{t+1})$: Discounted value of the next state, reflecting future potential benefits.
- TD Error helps the Critic update the value function $V(s)$while also guiding the Actor to adjust its policy for better action selection.
- As TD Error approaches zero, the Critic's value estimation becomes more accurate.

2. Advantage Function $A(s, a)$:

- The advantage function A(s,a) measures how much better an action aaa is compared to the expected value of the state:

$$A(s, a) = Q(s, a) - V(s)$$

- In that:
  - $Q(s, a)$: Action-value function, representing the expected reward when taking action aaa in state s.
  - $V(s)$: State-value function, representing the expected reward of being in state s, considering optimal actions.
- If $A(s, a) > 0$, action a is better than average and should be selected more frequently.
- If $A(s, a) < 0$, action a is suboptimal and should be chosen less often.

3. Policy Update (Gradient Ascent for Actor):

- The policy function $\pi(a|s;\theta)$ is updated based on the Advantage Function using the policy gradient theorem:

$$\nabla_\theta J(\theta) = E\left[\nabla_\theta log\pi\left(a|s;\theta\right) A(s,a)\right]$$

- In that:
  - $J(\theta)$: The objective function to maximize (expected reward).
  - $\pi(a|s;\theta)$: The policy of the Actor, defining the probability of selecting action aaa given state sss.
  - $A(s,a)$: The advantage function, guiding the Actor in choosing better actions.

- Update Mechanism:
  - If $A(s,a)$is large (good action), the Actor increases the probability of selecting that action.
  - If $A(s,a)$is small (poor action), the Actor reduces the probability of selecting that action
  - This iterative process refines the policy towards optimal decision-making.

4. Value Function Update (Gradient Descent for Critic):

- The Critic updates the value function $V(s)$ using Gradient Descent, minimizing TD error with the following update rule:

$$w \leftarrow w + \alpha\delta_t \nabla_w V(s;w)$$

- In that:
  - w: Parameters of the Critic's value function.
  - α: Learning rate, controlling the step size of updates.
  - $\delta_t = TD\ Error$: Temporal Difference error, used to correct the Critic's value estimation.
  - $\nabla_w V(s;w)$: Gradient of the value function with respect to parameters w.
  -

- **Objective:**
  - Reduce the estimation error in $V(s)$ for more accurate future value predictions.
  - Improve the reliability of the Critic's feedback, leading to better policy updates.

4.2.2.4 Actor-critical algorithm in the Cutting Stock problem (cutting wooden boards):

4.2.2.4.1 Parameters:

- Learning Rates:

  $\alpha_\theta = 0.001$(Learning rate for the Actor, applied via the Adam optimizer).

  $\alpha_w = 0.001$(Learning rate for the Critic, applied via the Adam optimizer).

  - Discount Factor:

    $\gamma = 0.99$ (GAMMA)
  - Epsilon (Exploration):

    $\epsilon_{start} = 0.5$, $\epsilon_{end} = 0.01$, $\epsilon_{decay} = 0.995$
  - Maximum Episodes:

    MAX_EPISODES=1000
  - Maximum Steps per Episode:

    MAX_STEPS=200

4.2.2.4.2 Initialization:

- Environment:

  $env = CuttingStockEnv$ with stock and product data from a CSV file, maximum dimensions $MAX\_W = 150$, $MAX\_H = 150$, and $MAX\_PRODUCTS = 100$.
- Actor-Critic Model:
  - θ: Parameters of the Actor (neural network with layers of 512, 256, 128 units and output of action_size).
  - w: Weights of the Critic (neural network with output as the state value).
- Optimizer:

  $optimizer = optim.Adam(model.parameters(), lr = LR)$

4.2.2.4.3 Training Loop (for each episode):

1. **Initialization:**

- Reset environment: $(s) \leftarrow env.reset()$
- Process state: $s \leftarrow process\_state(s, num\_stocks)$ (vectorize stock and product information).

- Initialize lists: $log\_probs = [], values = [], rewards = []$
- Update epsilon: $\epsilon \leftarrow max\,(\epsilon_{end}, \epsilon \cdot \epsilon_{decay})$

2. Loop while the episode is not done and $steps < MAX\_STEPS$ (for each time step):

- Select Action:

$$(a, log\pi(a|s,\theta), v\hat{}(s,w)) \leftarrow select\_action(model, s, env, \epsilon)$$

  - If $random < \epsilon$: randomly select stock_idx.
  - Otherwise: sample from the distribution $\pi(a|s,\theta)$ (Categorical).
  - Determine a valid position on the stock using $find\_valid\_position$.
- Execute Action:

$$(s', r, done, info) \leftarrow env.step(a)$$

  - $s' \leftarrow process\_state(s', num\_stocks)$.
  - Adjust reward r:
    - $If\ r > 0: r = 1.0 - info['trim\_loss']$.
    - $If\ info['filled\_ratio'] > 0: r = 0.5 - info['trim\_loss']$
    - Otherwise: $r = -0.1$
- Store Information:
  - $log\_probs.append(log\pi(a|s,\theta))$.
  - $values.append(v\hat{}(s,w))$.
  - $rewards.append(r)$.
- Update State:

$$s \leftarrow s'$$

3. Compute Returns:

- Initialize $R = 0$
- For each r in rewards (from end to start):

  $R \leftarrow r + \gamma R$

  Add R to returns
- Normalize:

$$returns = (returns - mean(returns))/(std(returns) + 10 - 5)$$

4. Compute Loss and Update Model:

- Policy Loss:

For each $(log\ \pi(a|s,\theta), v\hat{}(s,w), R)$ in (log_probs,values,returns):

$advantage = R - v\hat{}(s,w)$

- Value Loss:

$$\text{value\_loss.append}\left(\text{smooth\_l1\_loss}\left(\hat{v}(s,w), R\right)\right)$$

Where:

$$\text{smooth\_l1\_loss}\left(\hat{v}(s,w), R\right) = \begin{cases} 0.5\left(\hat{v}(s,w) - R\right)^2 & \text{if } \hat{v}(s,w) - R < 1 \\ \hat{v}(s,w) - R - 0.5 & \text{if } \hat{v}(s,w) - R \geq 1 \end{cases}$$

- Total Loss:

$$\text{loss} = \sum \text{policy\_loss} + \sum \text{value\_loss}$$

- Update Parameters:

optimizer.zero_grad()

loss.backward()

optimizer.step()

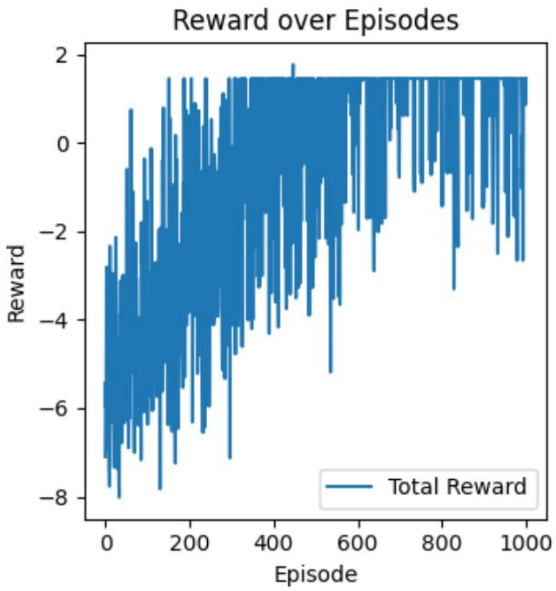
5. Save and display results:

- Store total reward per episode in episode_rewards.
- Store policy and value losses in policy_losses,value_losses.
- Every 10 episodes, print information: total reward, trim loss, filled ratio, epsilon.
- After training, save the model: actor_critic_batch_batch_id.pth
- Plot graphs: rewards, policy loss, value loss.

4.2.2.4.4 Evaluation of Actor-Critic Training Results:

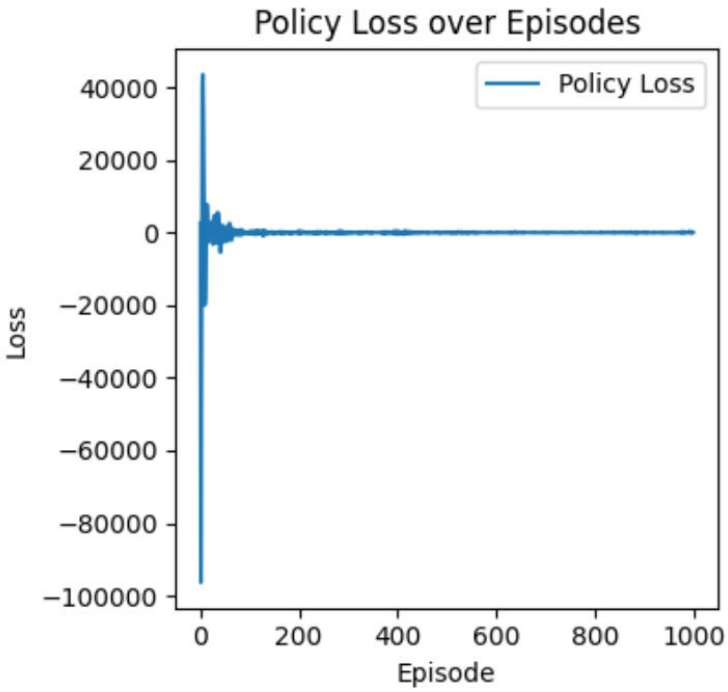
Based on the given training results of the Actor-Critic algorithm, we can analyze its performance using the three plots:

1. Reward over Episodes



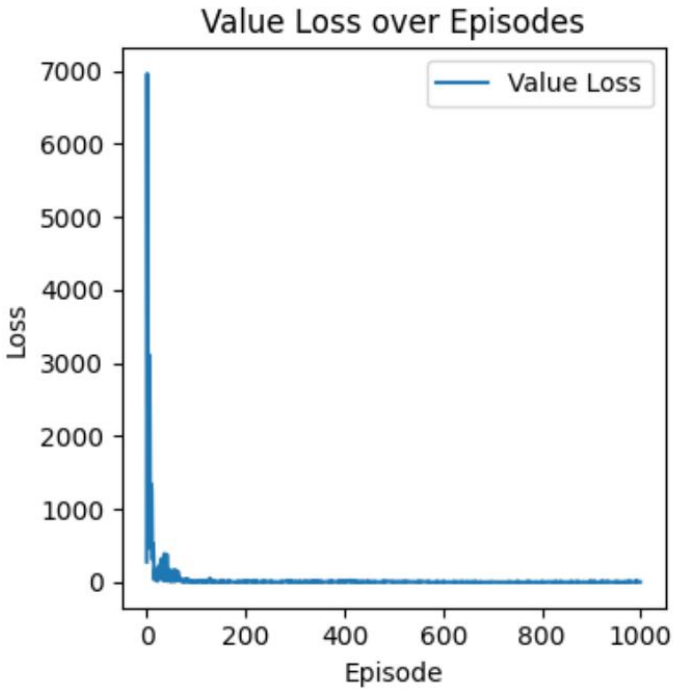
- Initially fluctuates around -8, then steadily increases.
- Stabilizes near 1.5 to 2 after 600 episodes, but with some fluctuations.
- Indicates effective learning but suggests minor instability.

2. Policy Loss over Episodes



- Starts at -100,000, decreases rapidly, and stabilizes near zero after 300 episodes.
- Shows that the Actor has learned a stable policy.

3. **Value Loss over Episodes**



- Begins at 7000, drops quickly, and remains near zero after 200 episodes.
- Suggests that the Critic has successfully learned value estimation.

4.2.2.5 Conclusion:

The Actor-Critic algorithm successfully learns an optimal policy, as shown by increasing rewards, stable policy updates, and low value loss. While minor fluctuations indicate some instability, overall, the model converges effectively and is suitable for real-world applications with potential fine-tuning for further improvement.

References

Gilmore, P. C., & Gomory, R. E. (1961). A Linear Programming Approach to the Cutting-Stock Problem. Operations Research, 9 (6), 849-859.

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. Mit Press.

Gymnasium Documentation. (2023). Retrieved from https://gymnasium.farama.org/

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai Gym. ARXIV Preprint Arxiv: 1606.01540.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level Control Through Deep Reinforcement Learning. Nature, 518 (7540), 529-533.