SOLVING THE PROBLEM OF “CUTTING STOCK” USING THE Q-LEARNING METHOD

# Introduction

## Context and Movation

The Cutting Stock Problem (CSP) is a classic challenge in Operations Research and Production Management. In its basic two-dimensional (2D) form, we have a set of stocks—rectangular raw materials (width × height)—and a set of products to be cut out of these stocks. Each product has specified dimensions and a required quantity. The fundamental objective is to minimize material waste (often called “trim loss”) while ensuring that all product requirements are satisfied.

In recent years, Reinforcement Learning (RL) has gained attention for its ability to learn from trial-and-error interactions with an environment. This project explores how Q-Learning, one of the simplest yet foundational RL algorithms, can be applied to the 2D Cutting Stock Problem. Q-Learning provides a mechanism to iteratively update what is known as the Q-Table, capturing the desirability (or “quality”) of particular actions in particular states, aiming to maximize long-term rewards.

## Research Objectives

* Construct a simulation environment (Environment) for the Cutting Stock Problem using the Gymnasium interface.
* Implement a Q-Learning agent that interacts with the environment, incrementally learning which cutting actions yield better material utilization.
* Analyze performance in terms of reducing trim loss, improving fill ratio, and effectively using fewer stock sheets where possible.
* Discuss limitations and future work, including potential improvements with more advanced RL techniques or domain-specific heuristics.

# Overview of Q-Learning and Q-Tables

Before addressing the Cutting Stock environment, we provide a concise introduction to Q-Learning and the concept of a Q-Table.

## What is a Q-Tabel?

* A Q-Table is a matrix (or 2D array) in which each row corresponds to a possible state and each column corresponds to a possible action
* The entry represents the expected future reward when the agent takes action in state then continues to follow the current policy or a greedy policy thereafter.
* In tabular Q-Learning, we store and update each as we gain experience from interacting with the environment. This is especially useful in discrete or semi-discrete problems, although it can become large for high-dimensional tasks.

## Q-Learning Algorithm

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

]

* is the **current state**.
* is the **action** taken.
* is the **immediate reward** received after taking action in state .
* is the **next state**.
* is a possible **action** in the next state.
* is the **learning rate**, controlling how much new information overrides old information.
* is the **discount factor**, determining how much future rewards are valued compared to immediate rewards.

Over time, through repeated interactions (episodes) in the environment, the Q-Table converges to values that approximate the optimal Q-function—if each state-action pair is visited infinitely often and if certain learning conditions are met.

## Action Selection: Epsilon-Greedy Strategy

An essential component of Q-Learning is balancing **exploration** and **exploitation**:

* With probability ε, the agent picks a random action (exploration).
* Otherwise, it picks the action arg (exploitation), based on the current Q-Table.

During training, ε typically decays from a high value (encouraging exploration) to a lower value (favoring exploitation) over many episodes.

# Cutting Stock Problem Overview

## Problem Statement

* We are given a set of rectangular stock sheets, each with a specific width and height.
* We have a set of products (also rectangles) of various widths and heights, each requiring a certain quantity.
* We want to cut these products out of the given stock sheets in a way that minimizes trim loss and potentially the number of sheets used.

## Challenges

* The search space is extremely large, as each cut can be placed in many possible positions on a stock.
* Traditional optimization methods can become computationally expensive, especially for 2D cutting with large stocks and numerous product types.
* A Reinforcement Learning approach, particularly Q-Learning, allows an agent to learn incrementally by trying different cutting actions and receiving feedback in the form of rewards.

# Enironment

We implemented a Gymnasium-style environment, with the following key points:

# State (Observation Space)

* Each stock is represented as a 2D array.
* -2 = out-of-bound region (beyond the real stock dimensions),
* -1 = free/unused space,
* >= 0 = indicates an occupied cell, storing an index of a product that has been placed.
* A **product list** providing the (width, height) of each product type, along with the remaining quantity to cut.

# Action Space

* An action is a dictionary of the form:

{

"stock\_idx": <index of chosen stock>,

"size": (width, height),

"position": (x, y)

}

* The environment checks if to is free (-1) before allowing the cut.

# Step Function

* Validates the action. If valid, it places the product in the specified position and decrements the required quantity.
* **Termination**: If all products (of all types) have quantity zero, the environment returns terminated = True.

# Rendering

* Implemented via Pygame. Each stock is displayed in a window cell, where each product index is color-coded.

# Reward

* By default, the environment returns if cutting is fully done, otherwise.
* However, we override or complement this in the training script to incorporate measures like trim loss, fill ratio, and a bonus for unused stock.

# Q-Learning Agent

# State Encoding

* Directly encoding the entire 2D layout in a Q-Table can be prohibitively large.
* As a simplification, we compute two features from the environment’s observation:
* empty\_space: the total count of free cells (-1).
* remaining\_products: the sum of quantities of all uncut products.
* We combine them into a single integer state index:

mod

Although this approach may lose spatial information, it keeps the Q-Table manageable in size.

## Action Selection

* We maintain an integer action index from 0 up to action\_size - 1.
* In get\_env\_action(action, observation), we break this index down into which product to cut and where to place it on a chosen stock.
* If no valid placement is found (the stock is fully or incorrectly used), the agent returns a dummy action.

# Q-Update

* After receiving reward and transitioning to new state , the agent updates:

]

* We also include the epsilon-decay to gradually shift from exploration to exploitation.

# Overview of Reward and Penalty in the Project

In your implementation, the overall reward an agent receives at each step (or when querying get\_reward(...)) is composed of three main parts:

* Filled Ratio: Represents how much of the used stock area has been effectively occupied by products.
* Trim Loss: Measures how much space remains unused in each stock that has been at least partially cut.
* Bonus for Unused Stocks: Provides an extra positive incentive for any stock sheets that remain completely untouched, thus potentially reducing overall material consumption.

Additionally, there is a default environment reward that returns 1 if cutting is completely finished (all products are allocated) and 0 otherwise. However, you incorporate a custom reward function that combines filled ratio, trim loss, and unused stock bonus. This custom reward can override or supplement the environment’s default reward, ensuring the agent gets more fine-grained feedback on how effectively it is cutting the materials.

# How Rewards Are Computed

# Filled Ratio and Trim Loss

## Filled Ratio:

* Calculated as the proportion of space occupied by cut products in the stocks that the agent has actually used.
* A higher filled ratio implies more efficient packing or cutting, so it contributes positively to the reward.

## Trim Loss:

* Defined as the ratio of unused (but usable) area in any stock that has been partially cut.
* Trim loss is effectively subtracted (or penalized) from the reward because large unused spaces indicate poor cutting strategies.

In my code, the environment tracks these two quantities (filled\_ratio and trim\_loss) and stores them in the “info” dictionary after each step. When the training script calls your reward function, those values are retrieved to guide the reward calculation.

# Bonus for Unused Stocks

* After each step, the code checks how many stock sheets have not been touched at all—meaning they remain fully intact and free of any cuts
* The assumption is that using fewer stock sheets, while still meeting product demand, can be beneficial.
* The code then calculates the fraction of completely unused stocks:
* This fraction is multiplied by a small factor (often referred to as lambda\_bonus or similar in your code) to create a bonus term.
* This bonus is added to the reward, thus encouraging the agent to concentrate its cuts on as few sheets as possible.

# Combining the Components

These parts are brought together with an expression along the lines of:

## Positive Contribution

* FilledRatio (encourages efficient usage).
* Bonus for Unused Stocks (incentivizes minimizing the total stock used).

## Negative Contribution

* TrimLoss (penalizes wasted space).

The result is that if the agent manages to cut products in a way that:

* Occupies a high proportion of space in the sheets it actually uses,
* Keeps some sheets completely untouched,
* Minimizes leftover or wasted space on used sheets,

it will receive **higher overall rewards**.

# Additional Penalties or Edge Cases

Beyond these main reward terms, your project also has a couple of implicit or potential penalty mechanisms:

## Zero Reward During Ongoing Steps:

If the environment is not yet “done” and you have not overridden the default reward for each intermediate step, the agent might receive 0 from the environment. This effectively penalizes the agent indirectly for lengthy, unproductive moves if it does not improve the state meaningfully.

## Invalid Cutting Actions (If Implemented):

If the agent attempts to cut a product in an invalid position—outside boundaries or overlapping existing cuts—the environment typically ignores or fails the action. While your core code does not impose a direct numeric penalty for invalid moves, such a penalty can be added if desired. The agent would then learn more quickly to avoid pointless or impossible cuts.

# Impact on Agent Behavior

Because the Filled Ratio and Trim Loss are directly opposed in the reward formula, and because leaving entire stocks unused provides an additional bonus, the agent learns to:

* Fill each used sheet with products as densely as possible.
* Avoid spreading cuts across multiple sheets unnecessarily.
* Minimize leftover space on any partially used sheet.

When these objectives align well with real manufacturing goals (reducing total sheets consumed and waste), the reward function drives more efficient cutting strategies.

# Summary





