机器学习大作业报告

李凤阳 522031910590
李佳豪 522031910614
宋泽瑞 522031910509

摘要

三维装箱问题是一个在运输和包装等行业中广泛存在的物资装载问题。它指的是在满足容积限制、外形几何限制等条件的情况下，将一定数量体积较小的物品放入具有较大容量的一个或多个箱子中，以达到空间利用率最高等目的的组合优化问题，是一个经典的 NP-hard 组合优化问题，本文使用基于深度强化学习的三维装箱问题 (3D-BPP) 解决方案。通过将传统的组合优化问题转化为序列决策问题，使用二位热力高度图的形式将原本的三维问题二维化以降低复杂度，使用策略网络加 PPO 进行决策，使用 CNN 提取高度图特征以解决问题。

1 问题描述

在 Bin Packing Problem (BPP) 装箱问题中，我们需要将定数量体积较小的物品放入具有较大容量的一个或多个箱子中并最大化箱子利用率。我们允许下物品生成时可以在三个正交维度方向进行正交旋转（旋转 90°）且物品与物品之间不能有碰撞，不能超过容器边缘，并且不允许新的物品放置在已放置物品的下方，以达到模拟现实场景的目的。Task1 中我们需要自行生成数据集，使用深度学习 + 强化学习的方式训练策略网络，自主进行决策，以完成将物品放入单个容器的目标。Task2 中我们需要在 Task1 网络中加树搜索相关的算法进行优化。Task3 允许我们使用多个容器进行装箱，并输出最终每个物品的位置和旋转情况。

2 具体实现

2.1 Task1

2.1.1 任务思路

我们采用强化学习来产生策略，该策略决定了物品在容器中的放置方式。Agent 与一个我们自定义的环境进行交互，该环境模拟了装箱过程。Agent 接收容器和可用物品的状态表示，基于学习到的策略选择动作，并根据放置的有效性获得奖励。学习过程的目标是最大化累积奖励，该奖励受到容器中利用体积 (utilization rate) 的影响。

2.1.2 数据生成

我们使用了如大作业任务书中的数据生成算法，即容器体积为 100×100×100，待放置的物品由单个的 100×100×100 立方体进行随机分割而来，每次分割需要：随机选择分割的维度，随机选择分割的比例，分割后新物品随机进行旋转，若不指定分割数量，则在一定范围内随机生成 n 个物品。因此，由该算法得到的物品理论上可以达到 100% 的利用率，即最优解是可以达到的。因此学习的目标是向更大的空间利用率靠拢。

2.1.3 环境定义

我们定义了一个强化学习环境 BoxEnv，旨在模拟 3D 装箱问题的解决过程。该环境负责管理容器的状态、物品的放置、碰撞检测以及奖励计算。通过与该环境的交互，Agent 可以学习如何有效地将物品放入容器中，从而最大化空间利用率。我们将环境分为状态空间和动作空间，以及奖励计算，放置策略。**状态空间**主要由以下几个部分描述：

- 高度图 (Height Map): 我们根据大作业任务书中的提示，用一个二维数组表示容器在 XY 平面上的高度信息。每个元素表示在该 (x, y) 坐标处的高度，即目前该位置最高的放置物品所占的高度，反映了已经放置物品的堆叠情况。但由于 BPP 是一个实际的物理问题，因此使用高度图，可以确保放置新物品时不会向已放置的物品下方放入物品，同时为了确保物理上的真实，不能将放入的箱子完全悬空，或者说支撑率必须大于一定范围（0.5）才允许放置新物品。由于多个箱子摞起来时重心计算过于复杂，因此本文只考虑箱子自身和下方的支撑面积。一个放置物品的示例如下 [1] 所示：

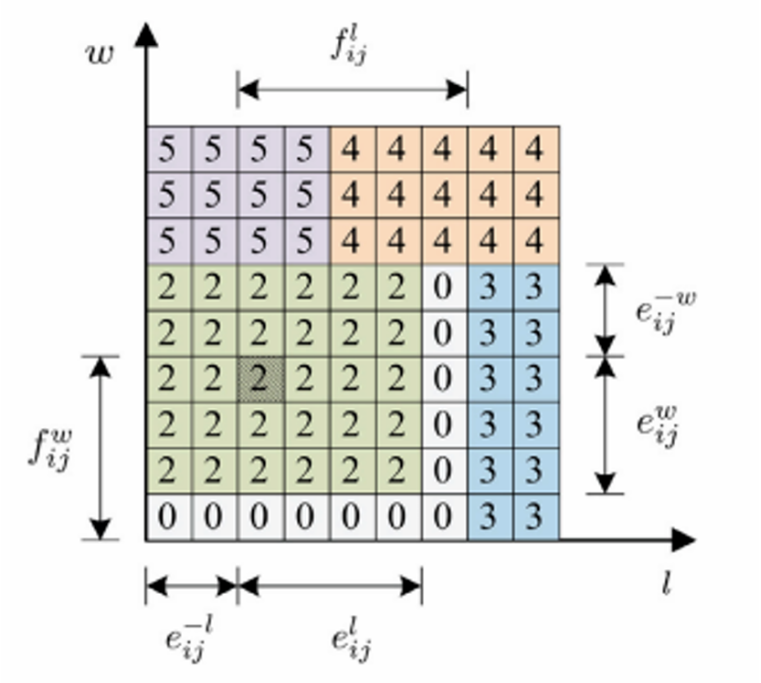


图 1: 高度图示例

- 可用物品掩码 (Available Mask): 一个布尔数组，指示哪些物品可以被放置。值为 1 表示可用，0 表示不可用。这是因为在强化学习环境中动作空间始终在变化，因此使用掩码以确定哪些物品是当前可用而哪些不是，也即哪些物品已经放入容器当中。

- 统计特征: 包括当前高度图的均值、最大值和方差等参数，这些特征可以帮助 Agent 理解容器的状态。

动作空间的设计较为直接，每一个动作对应于选择一个物品进行放置。每个物品都有一个唯一的索引，Agent 可以通过选择该索引来指定要放置的物品。

奖励计算的设计旨在鼓励代理进行有效的物品放置。奖励可以基于以下几个方面进行计算：

- 体积利用率: 代理成功放置物品后，计算当前已放置物品的总体积与容器总体积的比率。

- 高度奖励: 放置物品的高度越低，奖励越高，以鼓励在容器底部放置更多物品。

- 失败惩罚: 如果代理尝试放置物品失败，则增加惩罚，促使代理学习更有效的放置策略。

奖励计算公式如下:

$$Reward = \alpha * U + \beta * H + \gamma * F \tag{1}$$

其中

$$U = \frac{\sum W_{placed} * L_{placed} * H_{placed}}{W_{All} * L_{All} * H_{All}}, H = 1 - (\frac{H_{placed}}{H_{MAX}}), F = Failuretime \tag{2}$$

2.1.4 强化学习部分定义

在强化学习部分，我们使用 PPO 算法进行运作，PPO（Proximal Policy Optimization）是一种基于策略的强化学习算法，它通过优化代理的策略来最大化预期的累积奖励。我们根据上述算法设计与实现了 PPOAgent，以下是算法的细节。

PPO 的核心思想是通过限制策略的更新幅度来提高训练的稳定性。具体而言，PPO 使用以下目标函数:

$$L^{CLIP}(\theta) = E_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

其中：$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ 是当前策略与旧策略的概率比。$\hat{A}_t$ 是优势估计，表示在状态 s_t 下采取动作 a_t 的相对好坏。ϵ 是一个小的超参数，控制更新的幅度。

在每个训练回合中，PPOAgent 将与环境交互以收集经验。这些经验包括状态、动作、奖励和下一个状态。收集经验的过程可以描述为：

1. 初始化环境并获取初始状态 s_0。
2. 重复以下步骤直到达到终止条件：
    - 根据当前策略 π 选择动作 a_t。
    - 执行动作 a_t，观察奖励 r_t 和下一个状态 s_{t+1}。
    - 存储经验 (s_t, a_t, r_t, s_{t+1})。

优势函数 $\hat{A}_t$ 用于评估在特定状态下采取某个动作的好坏。优势的计算通常采用 Generalized Advantage Estimation (GAE)，其公式为：

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$

其中：$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ 是时序差分误差。γ 是折扣因子，控制未来奖励的影响。λ 是平滑参数，控制优势的估计偏差。

在收集到一定数量的经验后，PPOAgent 进行策略优化，使用梯度上升法来更新参数：

$$\theta \leftarrow \theta + \alpha \nabla_\theta L^{CLIP}(\theta)$$

在本任务中，在基础的 PPO 算法基础上，我们增加了经验回放的设计，并基于放置难度和稀有性进行采样，保持困难样本和成功样本的平衡，考虑放置序列的依赖关系并通过 episode 片段进行训练，训练步骤按照难度递进，分阶段学习不同的放置策略，从较小总数量物品 (num_min) 开始训练，直到达到最大物品数量（num_max）最后逐步整合多个子目标。同时我们采用了动态裁剪阈值的方法，根据训练进展调整 PPO 的裁剪参数，在早期允许较大更新以加快探索，后期收紧约束确保稳定性。

使用 PPO 的直接目标是最大化空间利用率，保证放置稳定性并提高装箱效率。同时需要维持放置策略的连续性，平衡探索与利用并适应不同尺寸物品的分布。同时由于三维装箱问题中每个 episode 的计算成本较高，PPO 可以多次利用同一批数据进行多轮策略更新，这样可以提高样本利用效率，加快收敛速度。

2.1.5 编码器-解码器模式的深度强化学习

我们根据大作业任务书的提示，采用 Encoder-Deocoder 的模型进行深度学习部分的实现。

Encoder 的主要目的是将输入的状态信息（高度图 height map）转换为一个低维的特征表示。输入为二维的张量（1，L，W），也即高度图的特征表示，这种特征表示能够捕捉到输入数据的关键特征，使得后续的网络（Actor 和 Critic）能够更有效地进行决策和价值评估。

Decoder 的主要作用是将 Encoder 提取的低维特征映射回高维空间，通常用于生成与输入相似的输出。在本例中，Decoder 没用到（我后面再改这一部分）。

ActorNetwork 使用 Encoder 提取输入状态的特征图，通过两个全连接层和一个 Policyhead, 通过 softmax 函数将 logits 转换为概率分布，表示在给定状态下选择每个动作的概率。接着根据输出的概率分布，使用 Categorical 分布随机选择动作。在生成动作概率时，应用掩码将不可用的动作的概率设置为负无穷，确保这些动作不会被选择。在这里我们使用方式是在输出概率后才用掩码进行掩盖，但我们认为可以对输入进行处理，也即对动作空间进行掩盖，效果可能更优。

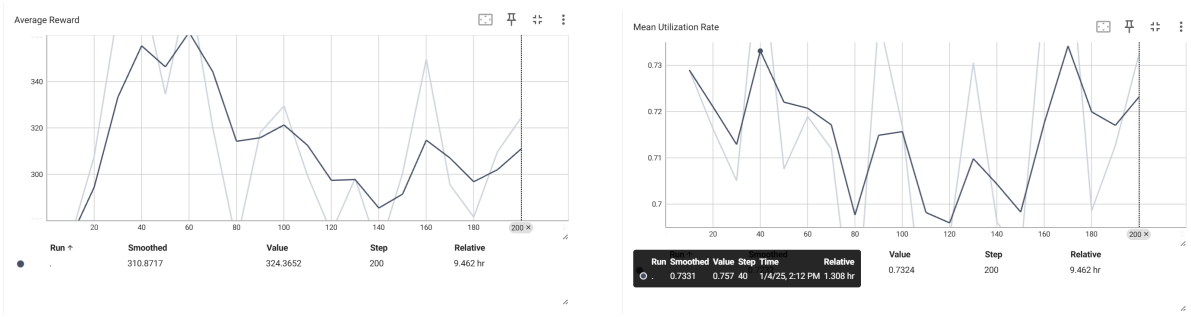
CriticNetwork 同样使用 Encoder 提取输入状态的特征图，通过两个全连接层和一个 Valuehead，输出一个单一的值，表示当前状态的预期价值。这是对当前状态下未来奖励的估计。CriticNetwork 的输出与实际获得的奖励进行比较，以计算价值损失（均方误差），这用于更新 CriticNetwork 的参数，接着提供反馈以更新 ActorNetwork 的

策略。

2.1.6 训练及测试

我们在训练初期采用简单难度的数据进行训练，即生成的物品数量在 20 个以下，同时对于每一轮生成的数据使用多个 eposide 进行放置，并综合进行评估和更新，在训练中期，我们将生成的物品数量增加至 35 个以下，同时逐渐降低每轮数据使用的 eposide 数，在训练后期，我们将生成的物品数量控制在 40 个以上，并每 5 个 eposide 进行数据的更换，促进模型可以使用更丰富的数据进行训练。

我们对模型总共训练了约 800 轮，其中我记录了后两百轮，也即训练后期的数据变化如下：



(a) reward 变化　　(b) Mean Utilization Rate 变化

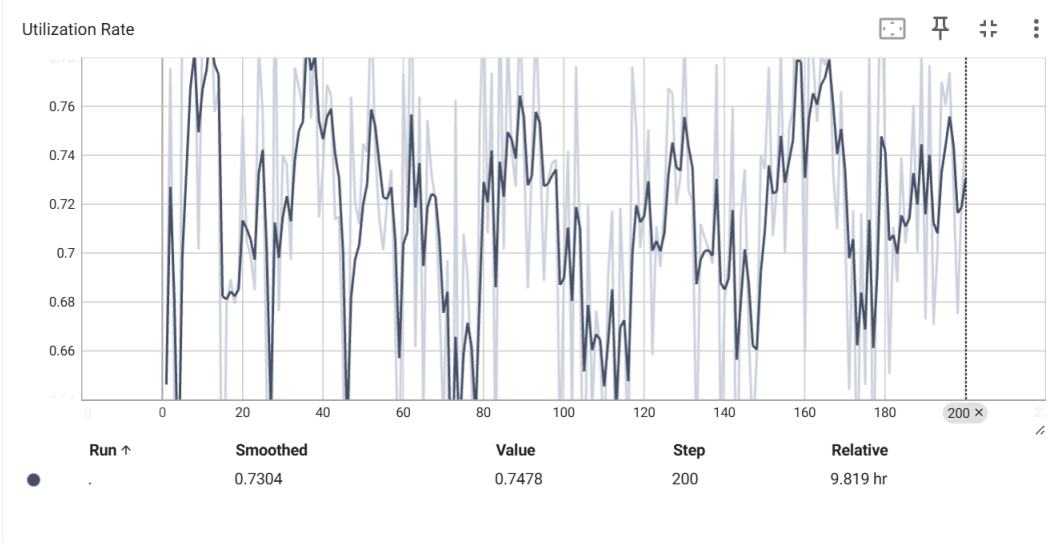


图 3: Utilizate rate 变化

可以看到，训练的利用率基本在 0.65 至 0.76 变化，最大利用率达到了 0.83，我们

也生成了 50 组不同的数据集用作测试集（物品数量在 10 到 50 之间），平均的利用率约为 0.72，平均 reward 为 313.94。当然，由于时间和硬件原因，我们训练的次数其实过于少了，仅仅有 800 轮左右，每个阶段锁训练的次数都没有达到要求，导致每个阶段模型并未收敛，因此最后的利用率为百分之七十多。训练初期模型性能的提升较为明显，由不到 0.4 左右的利用率快速增长到 0.65，但随着难度逐渐提高，同时因为模型训练轮数过少没有收敛，导致在训练后期每次放置物品的利用率波动较大，基本看不出模型的优化。我们估计，如果让模型训练至收敛，模型的性能，即利用率，能达到接近 0.8。

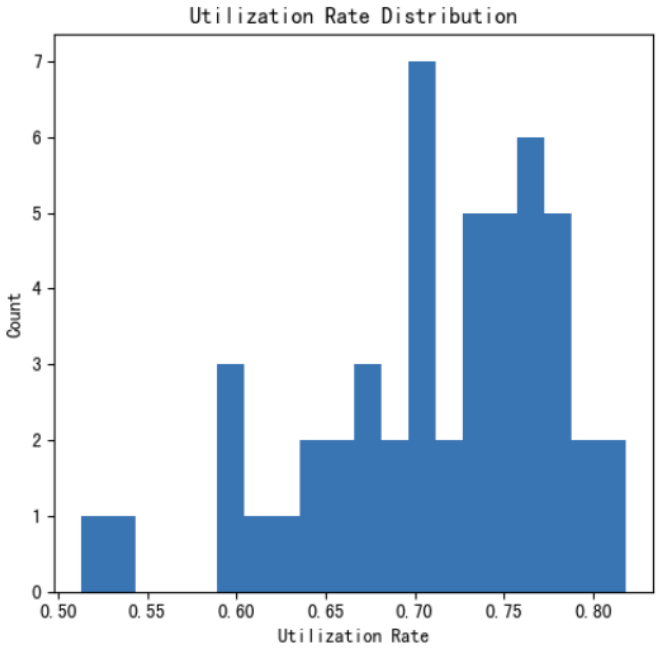


图 4: 测试集利用率分布

2.2 Task2

2.2.1 任务思路

在任务 2 中，我们结合策略网络与搜索算法（例如深度优先搜索，DFS）来优化装箱策略。通过将策略网络输出的概率分布作为启发，结合 DFS 对装箱顺序进行优化，旨在提高空间利用率。

2.2.2 搜索算法的实现

我们选择使用深度优先搜索（DFS）算法，结合策略网络提供的动作概率分布，来探索最优的装箱顺序。DFS 通过递归探索所有可能的放置顺序，并使用策略网络来指导每一步的决策。

- **DFS 结合策略网络**：在搜索过程中，策略网络的输出用于排序可能的动作，以优先探索高概率的动作。
- **状态保存与回溯**：在每一步操作后，保存当前环境状态，以便在需要回溯时能够恢复到之前的状态。
- **异常处理**：在搜索过程中处理可能的异常情况，确保算法的鲁棒性。

2.2.3 具体实现

我们实现了一个简单的 DFS 算法，结合策略网络以优化装箱策略。以下是实现的关键函数：

```
import copy
import torch
import numpy as np
from BPP import *

def dfs_search(env, state, agent, depth=0, max_depth=10, path=None):
    if path is None:
        path = []

    if depth >= max_depth:
        return None

    available_mask = state["mask"].to(agent.device)
    action_probs = agent.actor(state, available_mask)
    _, sorted_actions = torch.sort(action_probs, descending=True)

    for action in sorted_actions[0].tolist():
        if not available_mask[action]:
            continue

        original_env = copy.deepcopy(env)

```

```
        try:
            next_state, reward, done, _ = env.step(action, False)
            next_state = {
                "height_map": next_state["height_map"].to(agent.device),
                "mask": next_state["mask"].to(agent.device)
            }

            path.append((action, reward))

            if done:
                return path
            else:
                result = dfs_search(env, next_state, agent, depth + 1,
                    max_depth, path)
                if result is not None:
                    return result
        except Exception as e:
            pass

        env = original_env
        path.pop()

    return None
```

2.2.4 测试与结果

我们对结合策略网络和 DFS 的算法进行了测试。结果表明，通过这种方法，可以进一步优化装箱策略，提高空间利用率。特别是在复杂的装箱场景中，DFS 结合策略网络能够探索到更优的放置顺序。

2.3 Task3

2.3.1 任务思路

在电子商务包装问题中，目标是使用给定尺寸的多个快递包装盒，将订单中的各类物品进行高效包装，以提高容器的空间利用率，降低包装成本。本算法采用贪心策略来解决这一问题。首先，对每个订单内的物品按照体积进行降序排序。这是基于贪心思想，优先放置体积较大的物品，因为大物品对空间的占用和布局影响更大，先放置

大物品可以为后续小物品的放置留出相对规整的空间，更有利于整体空间的有效利用。在放置物品时，优先尝试将物品放入已有的容器中。通过 find_fit_location 方法在容器中搜索合适的放置位置。该方法允许底面存在高度差异，它遍历容器的底面，针对每个可能的放置区域，计算区域内的高度差异以及基准高度。优先选择高度差异小且基准高度低的位置，这样可以使放置后的容器空间更加规整，减少因物品堆叠不整齐导致的空间浪费。同时，在放置物品前，会调用 check_collision 方法检查物品与已放置物品是否发生碰撞，调用 check_suspension 方法检查物品是否处于悬浮状态，只有通过这两项检查，物品才会被放置在该位置。如果在所有已有容器中都找不到合适的放置位置，算法会创建一个能容纳该物品的最小尺寸新容器，并将物品放置其中。最后，通过计算容器的利用率来评估包装效果。利用率的计算方法是已放置物品的总体积与所用容器总体积的比值，这一指标能够直观地反映算法在包装过程中对容器空间的利用程度。

2.3.2 数据结构定义

Item 类用于表示物品，它包含物品编号 sku_code、尺寸 dimensions、放置位置 position 和放置状态 isplaced。将物品尺寸放大 10 倍并转换为整数，有助于在后续计算中避免浮点数精度问题。container_class 类表示容器，包含容器的长 L、宽 W、高 H、体积 volume、高度映射 height_map、利用率 utilization 和已放置物品列表 placed_items。其中，height_map 是一个二维数组，用于记录容器底面每个位置的高度信息，这对于判断物品的放置位置和更新容器状态非常关键

2.3.3 核心方法

对于放置物体的逻辑，可以用任务一的函数进行改编。

findfitlocation 方法通过遍历容器底面的所有可能位置，计算每个位置对应的区域高度差异和基准高度，从而找到适合放置物品的最佳位置。它首先判断物品尺寸是否超出容器的长和宽，如果超出则直接返回 None。然后，在遍历过程中，对于每个可能的放置区域，计算区域内的高度差异和基准高度，并选择高度差异最小且基准高度最低的位置作为最佳位置。update_height_map 方法在物品成功放置后被调用，它更新容器的高度映射 height_map，将物品放置区域的高度更新为物品放置后的高度，同时更新容器的利用率 utilization。check_collision 方法通过遍历已放置物品列表，检查新物品与已放置物品在三维空间中是否发生碰撞。check_suspension 方法用于检查物品放置后是否处于悬浮状态，它通过计算物品底部支撑区域的面积与物品底部总面积的比值来判断，如果该比值小于 0.5，则认为物品处于悬浮状态。

```python
    def find_fit_location(self, item):
        """允许底面有高度差异的放置位置查找"""
        width, depth, height = item.dimensions[0],item.dimensions[1],item.
            dimensions[2]
        rows, cols = self.height_map.shape

        if width > rows or depth > cols:
            return None

        best_location = None
        min_height_diff = float('inf')
        min_base_height = float('inf')

        # 遍历所有可能的放置位置
        for x in range(rows - width + 1):
            for y in range(cols - depth + 1):
                # 获取当前区域
                region = self.height_map[x:x+width, y:y+depth]
                base_height = np.max(region)  # 使用区域最大高度作为基准

                if base_height + height > self.H:  # 检查是否超出容器高度
                    continue

                # 计算区域内的高度差异
                height_diff = np.max(region) - np.min(region)

                # 选择高度差异最小且基准高度最低的位置
                if height_diff < min_height_diff or (height_diff ==
                    min_height_diff and base_height < min_base_height):
                    min_height_diff = height_diff
                    min_base_height = base_height
                    best_location = [x, y, base_height]

        return best_location
```

2.3.4 主要流程

greedy_packing 函数: 该函数实现了贪心包装策略。首先创建一个空列表 containers，用于存放放置物品的容器。将物品列表 items 按照体积从大到小排序，这样优先

放置大物品，能为后续小物品的放置留出相对规整的空间，符合贪心算法的思想。对于每个物品，首先尝试将其放入已有的容器中。遍历已有的容器列表 containers，调用 find_fit_location 方法查找合适的放置位置。若找到合适位置，并且物品放置后不超出容器边界、不与已放置物品碰撞且不处于悬浮状态，就将物品的位置信息更新，调用 update_height_map 方法更新容器状态，并将物品的放置状态设置为 True，标记物品已放置，然后跳出循环。如果在所有已有容器中都找不到合适位置，就遍历容器尺寸列表 container_sizes，找到能容纳该物品的最小尺寸容器，创建新容器并将物品放置其中，同时更新物品和容器的相关状态。若最终物品仍未被放置，则输出提示信息。最后返回包含所有放置了物品的容器的列表 containers

main 函数则是整个程序的入口。

2.3.5 测试与小结

测试数据：使用 task3.csv 作为测试数据，该文件包含多个订单，每个订单包含多个物品的信息，包括物品编号 sku_code、长、宽、高和数量 qty。这些数据模拟了真实的电子商务订单场景，为算法的测试提供了实际的输入。

测试发现使用 task3.csv 作为数据集，平均空间利用率在 0.49

我们尝试了在任务一强化学习方法上增加选择容器的步骤，但是效果并不理想，空间利用率无法超过 0.4，这也是我们选择使用常规算法的原因。

本算法在处理一些复杂订单时，可能由于贪心策略的局限性，无法找到最优的包装方案，导致部分物品无法被放置。例如，当订单中存在一些形状不规则或尺寸特殊的物品时，算法可能无法有效地利用容器空间，从而出现物品无法放置的情况。未来，可以考虑对算法进行优化，例如引入启发式算法或改进贪心策略，以提高算法的性能和物品的放置成功率。

3 总结

在本次作业中，我们探讨了三维装箱问题（3D Bin Packing Problem, 3D-BPP），并通过深度强化学习算法提出了一种有效的解决方案。我们结合策略网络与深度优先搜索（DFS）算法，进一步优化了装箱策略，通过启发式方法提高了空间利用率。我们采用了贪心策略来处理多盒装箱问题，优先放置体积较大的物品，确保了算法的高效性。

4 小组分工

Name	Score	Work
宋泽瑞	33%	task1 相关代码、报告编写
李佳豪	33%	task2 相关代码、报告编写
李凤阳	33%	task3 相关代码、报告编写

附 github 仓库链接：https://github.com/yyyang2004/CS3308project

参考文献

[1] Que, Quanqing, Fang Yang, and Defu Zhang. "Solving 3D packing problem using Transformer network and reinforcement learning." *Expert Systems with Applications* 214 (2023): 119153.

[2] Zhang, Jingwei, Bin Zi, and Xiaoyu Ge. "Attend2Pack: Bin packing through deep reinforcement learning with attention." arXiv preprint arXiv:2107.04333 (2021).

[3] Zhu, Qianwen, et al. "Learning to pack: A data-driven tree search algorithm for large-scale 3D bin packing problem." In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021.

[4] Huang, Shengyi and Ontañón, Santiago. "A closer look at invalid action masking in policy gradient algorithms." *The International FLAIRS Conference Proceedings*, 35, May 2022.

[5] Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, and Klimov, Oleg. "Proximal policy optimization algorithms." 2017.