Robotic Pallet Packing

Alfred Cueva

October 3, 2024

This report describes how the bin packing environment algorithm functions in terms of its **state**, **reward**, and **action** components. The environment simulates a two-dimensional bin packing problem using reinforcement learning, where an agent aims to efficiently place boxes into a fixed-size pallet.

1 Reinforcement Learning Model

1.1 State

The state of the environment is represented by a 2D grid, corresponding to the pallet dimensions ($Pallet\_x$ by $Pallet\_y$). Initially, the grid is empty, with each cell set to zero. As boxes are placed, the grid updates to reflect which cells are occupied. The state (i.e., observation space) is an array that represents the current state of the pallet, showing both occupied and available areas. This allows the agent to "observe" the layout of the boxes in the pallet and decide on the next action.

1.2. Action

The action space is discrete with two possible actions:

- **0:** Place the box without rotation.
- **1:** Rotate the box by 90 degrees before placing.

At each time step, the agent selects one of these actions to try to place an arriving box in the pallet, with the option to rotate it for a better fit.

1.3. Rewards

The reward structure is designed to encourage efficient packing, prioritize specific regions, and penalize inefficient placements:

- **Size Reward (r_{size}):** The size reward is calculated as the ratio of the filled area (arrangement_size) to the total pallet area (Pallet_size). This encourages the agent to maximize the utilization of the pallet space:
$$r_{\text{size}} = \frac{\text{arrangement\_size}}{\text{Pallet\_size}} \tag{1}$$
- **Edge Reward (r_{edge}):** A fixed reward of 0.5 is given when boxes are placed along the perimeter of the pallet. This reward incentivizes the agent to focus on filling the perimeter first, thereby creating an organized packing pattern:
$$r_{\text{edge}} = 0.5 \tag{2}$$
- **Squeezing Penalty (p_{squeeze}):** This penalty is based on the number of squeezing attempts, where a box is surrounded by other boxes, making placement more challenging. The penalty is calculated as follows:
$$p_{\text{squeeze}} = \text{penalty\_factor} \times \text{squeezing\_attempts} \tag{3}$$
The penalty factor is set to 0.3 from tuning. This penalty discourages inefficient packing by penalizing situations where boxes are squeezed into limited spaces.

The total reward is calculated as the sum of r_{size}, r_{edge} (if applicable), and a negative p_{squeeze}. When the perimeter is fully packed, r_{edge} is no longer applicable.

1.4. Environment Step

In each step, the agent must decide whether to rotate the newly arriving box and find a suitable placement. The steps include:

- **Perimeter Placement:** The agent first attempts to place the box along the perimeter of the pallet. If successful, the state is updated, and rewards are given.
- **Inner Space Placement:** If the perimeter is full or no valid perimeter placement is found, the agent tries placing the box in the inner space.
- **Reward Calculation:** If a valid placement is found, the reward is computed as described above, and the environment state is updated with the new placement.
- **Termination:** If no valid placement is found, the episode ends, indicated by `terminated = True`.

Algorithm 1 Perform a Step in the Environment

```
0: Input: action
0: Output: Updated grid state, reward, termination status, and additional information
0: Determine rotation choice: rotate_choice ← action % 2
0: Get new box type: box_type ← arrival()
0: Initialize placement and placed_box to None
0: If not perimeter_filled then
0:    Call find_best_placement(grid, box_type, rotate_choice, perimeter_only=True)
0:    Set placement, placed_box ← returned values
0:    If placement = None and check_perimeter() then
0:       Set perimeter_filled ← True
0:    End If
0: End If
0: If perimeter_filled = True or placement = None then
0:    Call find_best_placement(grid, box_type, rotate_choice, perimeter_only=False)
0:    Set placement, placed_box ← returned values
0: End If
0: If placement is not None then
0:    Extract placement coordinates: x, y ← placement
0:    Call place(grid, box_type, placed_box, x, y, placement_order)
0:    Update arrangement size: arrangement_size ← arrangement_size + (placed_box[0] * placed_box[1])
0:    Increment number of boxes placed: num_boxes_placed++
0:    Compute size reward: size_reward ← arrangement_size / Pallet_size
0:    Call count_squeezing_attempts(placement_order, Pallet_x, Pallet_y, distance_threshold=squeezing_distance_criteria)
0:    Set squeezing_attempts ← returned value
0:    Set penalty_factor ← 0.3
0:    Compute squeezing penalty: squeezing_penalty ← penalty_factor * squeezing_attempts
0:    If not perimeter_filled then
0:       Set edge_reward ← 0.5
0:       Compute reward: reward ← size_reward + edge_reward - squeezing_penalty
0:    Else
0:       Compute reward: reward ← size_reward - squeezing_penalty
0:    End If
0:    Set terminated ← False
0: Else
0:    Set reward components to zero: reward, size_reward, edge_reward, squeezing_penalty ← 0
0:    Set terminated ← True
0: End If
0: Create info dictionary:
      info ← { "size_reward": size_reward, "edge_reward": edge_reward, "squeezing_penalty": squeezing_penalty }
0: return Updated grid state, reward, terminated, False, info
```

- **Input:**
  - `action`: The action taken by the agent, determining whether to rotate the box before placement.
- **Output:** The function returns the updated grid state, the reward obtained, whether the episode is terminated, and additional information in a dictionary.
- **Explanation:** The function takes a step in the environment, attempting to place a new box:
  - **Determine Rotation Choice**:

    * The action is used to determine whether the box should be rotated before placement.
  - **Get New Box Type**:
    * The type of box to be placed is determined using the `arrival()` function.
  - **Placement Attempt Along Perimeter**:
    * If the perimeter is not yet filled, the function attempts to find a placement along the perimeter by calling `find_best_placement()` with `perimeter_only=True`.
    * If no valid placement is found and the perimeter is fully filled, the `perimeter_filled` flag is set to `True`.
  - **Placement Attempt in Inner Space**:
    * If no perimeter placement is found, or if the perimeter is filled, the function attempts to find a placement anywhere within the pallet by calling `find_best_placement()` with `perimeter_only=False`.
  - **Place Box if Valid Placement is Found**:
    * If a valid placement is found, the function calls `place()` to update the grid and adds the box to the placement order.
    * The reward is computed based on the arrangement size, perimeter placement, and a squeezing penalty to encourage efficient packing.
    * If the perimeter is not yet filled, an additional edge reward is added.
  - **Terminate if No Valid Placement is Found**:
    * If no valid placement is found, the episode is terminated and the reward is set to zero.
  - **Return Values**:
    * The function returns the updated grid state, computed reward, termination status, and additional info.

Auxiliary Function: `check_perimeter`

Algorithm 2 Check if All Perimeter Locations Are Filled

```
0: **Input:** grid, Pallet_x, Pallet_y
0: **Output:** True if all perimeter cells are filled, otherwise False
0:
1: for $i = 0$ to $Pallet\_x - 1$ do
2:   if $grid[i][0] = 0$ or $grid[i][Pallet\_y - 1] = 0$ then
3:     return False
4:   end if
5: end for
6: for $j = 0$ to $Pallet\_y - 1$ do
7:   if $grid[0][j] = 0$ or $grid[Pallet\_x - 1][j] = 0$ then
8:     return False
9:   end if
10: end for
11: return True
```

- **Input:** `grid`, `Pallet_x`, `Pallet_y`
- **Output:** `True` if all perimeter cells are filled, otherwise `False`
- **Explanation:** The function iterates over the rows and columns of the perimeter of the pallet:
  - First, it checks all cells along the left and right edges of each row of the pallet.

  - Then, it checks all cells along the top and bottom edges of each column of the pallet.
  - If any of these cells are unoccupied (value is 0), the function returns `False`.
  - If all perimeter cells are filled, the function returns `True`.

Auxiliary Function: `check_space`

Algorithm 3 Check if a Box Can Be Placed in the Grid

```
Input: grid, box, current_x, current_y
Output: True if the box can be placed at the specified coordinates, otherwise False

Extract dimensions of the box: $x\_box, y\_box \leftarrow$ box
If $(current\_x + x\_box > Pallet\_x)$ or $(current\_y + y\_box > Pallet\_y)$ then
return False
End If
Set $start\_x \leftarrow \max($int(current_x)$, 0)$
Set $start\_y \leftarrow \max($int(current_y)$, 0)$
Set $end\_x \leftarrow \min($int(current_x + x_box)$,$ Pallet_x$)$
Set $end\_y \leftarrow \min($int(current_y + y_box)$,$ Pallet_y$)$
for $i = start\_x$ to $end\_x - 1$ do
  for $j = start\_y$ to $end\_y - 1$ do
    if $grid[i][j] \neq 0$ then
      return False
    end if
  end for
end for
return True
```

- **Input:**
  - `grid`: A 2D array representing the current state of the pallet.
  - `box`: A tuple representing the dimensions of the box (`x_box`, `y_box`).
  - `current_x, current_y`: The coordinates at which to check if the box can be placed.
- **Output:** The function returns `True` if the box can be placed at the specified coordinates without overlapping any existing items, otherwise `False`.
- **Explanation:** The function checks if there is enough space to place a box at the specified coordinates:
  - First, the dimensions of the box are extracted from the input parameter `box`.
  - **Boundary Check**:
    * If the box, starting at `(current_x, current_y)`, extends beyond the boundaries of the grid (either horizontally or vertically), the function returns `False`.
  - **Set Start and End Coordinates**:
    * The starting coordinates (`start_x, start_y`) are determined, ensuring they are non-negative and within grid bounds.
    * The ending coordinates (`end_x, end_y`) are calculated, ensuring they do not exceed the grid boundaries.
  - **Check for Overlapping Items**:
    * The function iterates over the cells defined by `(start_x, start_y)` to `(end_x, end_y)`.
    * If any cell within the specified area is already occupied (`grid[i][j]` $\neq 0$), the function returns `False`, indicating that the box cannot be placed without overlapping.

– **Return True**:

  * If no overlapping cells are found and the box fits within the boundaries, the function returns True, indicating that the box can be placed at the specified coordinates.

Auxiliary Function: find_best_placement

Algorithm 4 Find the Best Placement for a Box

```
Input: grid, box_type, rotate_choice, perimeter_only
Output: A tuple representing the best placement coordinates and box dimensions, or None if no valid placement is found

Determine box dimensions: box ← rotation(boxes[box_type]) if rotate_choice else boxes[box_type]
Initialize an empty list for valid placements: valid_placements ← [ ]
If perimeter_only = True then
for $i = 0$ to $Pallet\_x - box[0]$ do
  if check_space(grid, box, i, 0) then
    Append ((i, 0), box) to valid_placements
  end if
  if check_space(grid, box, i, Pallet_y - box[1]) then
    Append ((i, Pallet_y - box[1]), box) to valid_placements
  end if
end for
for $j = 0$ to $Pallet\_y - box[1]$ do
  if check_space(grid, box, 0, j) then
    Append ((0, j), box) to valid_placements
  end if
  if check_space(grid, box, Pallet_x - box[0], j) then
    Append ((Pallet_x - box[0], j), box) to valid_placements
  end if
end for
if valid_placements is not empty then
  return valid_placements[0]
end if
End If
For Inner Placement:
for $i = 0$ to $Pallet\_x - box[0]$ do
  for $j = 0$ to $Pallet\_y - box[1]$ do
    if check_space(grid, box, i, j) then
      return (i, j), box
    end if
  end for
end for
return None, None
```

- **Input:**
  - grid: A 2D array representing the current state of the pallet.
  - box_type: The type identifier for the box to be placed.
  - rotate_choice: A boolean indicating whether the box should be rotated.
  - perimeter_only: A boolean indicating whether to attempt placement only along the perimeter.

- **Output:** The function returns a tuple containing the best placement coordinates and the box dimensions, or `None` if no valid placement is found.
- **Explanation:** The function attempts to find the best position to place a box in the grid:
  - First, it determines the dimensions of the box based on whether a rotation is required.
  - It initializes an empty list (`valid_placements`) to store potential valid placements.
  - **Perimeter Placement**:
    - If `perimeter_only` is `True`, it iterates over possible positions along the perimeter.
    - It tries placing the box along the left and right edges of each row (‘i‘ from 0 to $Pallet\_x - box[0]$).
    - It then tries placing the box along the top and bottom edges of each column (‘j‘ from 0 to $Pallet\_y - box[1]$).
    - If a valid position is found (using `check_space()`), it appends the placement details to `valid_placements`.
    - If there are any valid placements, the function returns the first placement from the list.
  - **Inner Placement**:
    - If no perimeter-only placements are allowed or none were found, the function tries placing the box in any available inner position.
    - It iterates over all possible positions in the grid (‘i‘ from 0 to $Pallet\_x - box[0]$ and ‘j‘ from 0 to $Pallet\_y - box[1]$).
    - If a valid position is found, it returns the coordinates and box dimensions.
  - If no valid placements are found after attempting both perimeter and inner placement, the function returns `None, None`.

Auxiliary Function: `place`

Algorithm 5 Place a Box in the Grid

```
Input: grid, box_type, box, current_x, current_y, placement_order
Output: Updates the grid with the placed box and records the placement details

Extract dimensions of the box: $x\_box, y\_box \leftarrow$ box
Retrieve box mapping information: box_mapped ← boxes_mapping[box_type]
Append placement details: placement_order.append((box_type, current_x, current_y, x_box, y_box))
for $i = 0$ to $x\_box - 1$ do
  for $j = 0$ to $y\_box - 1$ do
    grid[current_x + i][current_y + j] = box_mapped
  end for
end for
```

- **Input:**
  - `grid`: A 2D array representing the current state of the pallet.
  - `box_type`: The type identifier for the box to be placed.
  - `box`: A tuple representing the dimensions of the box (`x_box`, `y_box`).
  - `current_x, current_y`: The coordinates on the grid where the box will be placed (starting position).
  - `placement_order`: A list that keeps track of the placement details of each box.

- **Output:** The function modifies the `grid` by marking the cells occupied by the box and updates the `placement_order` list with the details of the placed box.
- **Explanation:** The function performs the following steps to place the box in the grid:
  - Extracts the dimensions of the box from the input parameter `box`, assigning them to `x_box` and `y_box`.
  - Retrieves the mapping information for the box type from the dictionary `boxes_mapping`, which is stored in `box_mapped`.
  - Appends the details of the box placement (including the box type, start position, and dimensions) to the `placement_order` list. This information will be useful for tracking the placement history.
  - Iterates over the range of the box dimensions to update the corresponding cells in the grid:
    * The outer loop iterates from 0 to $x\_box - 1$, representing the rows of the box being placed.
    * The inner loop iterates from 0 to $y\_box - 1$, representing the columns of the box.
    * For each cell in the box dimensions, the grid is updated at the position `(current_x + i, current_y + j)` to mark it as occupied by the box type (`box_mapped`).

Auxiliary Function: Counting Squeezing Attempts

Algorithm 6 Count Squeezing Attempts

```
0: Input: placement_order, grid_width, grid_height, distance_threshold
0: Output: Number of squeezing attempts
1: Initialize squeezing_attempts ← 0
2: Set latest_box ← placement_order[num_boxes - 1]
3: if is_perimeter(latest_box) then
4:   return squeezing_attempts
5: end if
6: Initialize sides_coverage ← { 'top': [], 'bottom': [], 'left': [], 'right': [] }
7: for box2 in placement_order except latest_box do
8:   if are_boxes_adjacent(latest_box, box2) then
9:     $y, x, height, width \leftarrow box2[1], box2[2], box2[3], box2[4]$
10:    if $y + height =$ latest_box.y and overlaps horizontally then
11:      Add box2 to sides_coverage['top']
12:    end if
13:    if $y =$ latest_box.y + latest_box.height and overlaps horizontally then
14:      Add box2 to sides_coverage['bottom']
15:    end if
16:    if $x + width =$ latest_box.x and overlaps vertically then
17:      Add box2 to sides_coverage['left']
18:    end if
19:    if $x =$ latest_box.x + latest_box.width and overlaps vertically then
20:      Add box2 to sides_coverage['right']
21:    end if
22:  end if
23: end for
24: Set covered_sides ← [side for side, boxes in sides_coverage.items() if boxes]
25: Set squeeze_counted ← False
26: if len(covered_sides) = 2 then
27:   if ('top' in covered_sides and 'bottom' in covered_sides) or ('left' in covered_sides and 'right'
      in covered_sides) then
28:     squeezing_attempts += 1
29:     Set squeeze_counted ← True
30:   end if
31: end if
32: if not squeeze_counted and len(covered_sides) = 3 then
33:   if (('top', 'left', 'right') in covered_sides) or (('bottom', 'left', 'right') in covered_sides) or (('top',
      'bottom', 'left') in covered_sides) or (('top', 'bottom', 'right') in covered_sides)) then
34:     squeezing_attempts += 1
35:     Set squeeze_counted ← True
36:   end if
37: end if
38: if not squeeze_counted and len(covered_sides) = 4 then
39:   squeezing_attempts += 1
40:   Set squeeze_counted ← True
41: end if
42: if not squeeze_counted and len(covered_sides) ≥ 3 and sum(len(boxes) for boxes in
    sides_coverage.values()) ≥ 3 then
43:   squeezing_attempts += 1
44: end if
45: return squeezing_attempts
```

- **Input:**
  - `placement_order`: A list of boxes representing the order in which they were placed.
  - `grid_width`, `grid_height`: The dimensions of the pallet grid.
  - `distance_threshold`: The threshold to determine if two boxes are adjacent.
- **Output:**
  - `squeezing_attempts`: The number of successful squeezing attempts.
- **Explanation:**
  - The function identifies squeezing situations for the latest box placed in the grid based on the boxes already in place.
  - It starts by checking if the latest box is on the perimeter. If it is, no squeezing is counted.
  - **Determining Adjacent Boxes:**
    * The function iterates over all previously placed boxes to check whether they are adjacent to the latest box.
    * For each adjacent box, it identifies which side of the latest box is covered and stores this information.
  - **Squeezing Attempts Counting:**
    * **Two Sides Covered:** If the latest box is covered on **two sides that are aligned** (either top-bottom or left-right), it is counted as a squeezing attempt. This condition ensures that the box is tightly enclosed in a "corridor" formed by the two adjacent boxes.
    * **Three Sides Covered (U-Shape):** If three sides of the box are covered and they form a U-shape, this is considered a squeezing attempt. A U-shape indicates a situation where the box is effectively trapped from three directions, reducing further flexibility in packing.
    * **Four Sides Covered:** If all four sides of the latest box are covered by other boxes, it is counted as a squeezing attempt since the box is completely surrounded with no space left for movement.
    * **Three or More Boxes Contributing to a Squeeze:** If there are **three or more adjacent boxes collectively** covering three or more sides, and at least three boxes are involved, it counts as a squeezing attempt. This ensures that partial contributions by multiple boxes leading to a restricted enclosure are also taken into account.

1.5. Termination Condition

The episode terminates if the agent cannot place a box (`placement is None`), implying that there is no more available space for the current box configuration.

1.6. Tensorboard Reward Callback Class (`TensorboardRewardCallback`)

The `TensorboardRewardCallback` class logs the rewards and penalties at each step to provide visualization of the agent's performance over time. It tracks the following metrics:

- **Total Episode Reward:** Sum of all rewards in the current episode.
- **Component Rewards:** Individual components, including r_{size}, r_{edge}, and p_{squeeze}, are logged separately to understand their contributions to the total reward.

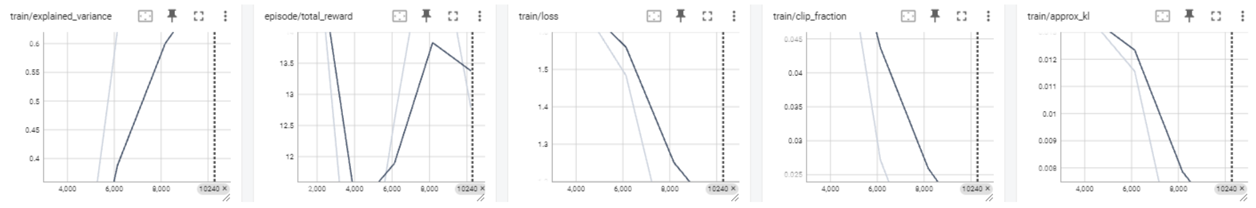


Figure 1: Training metrics for the RL model

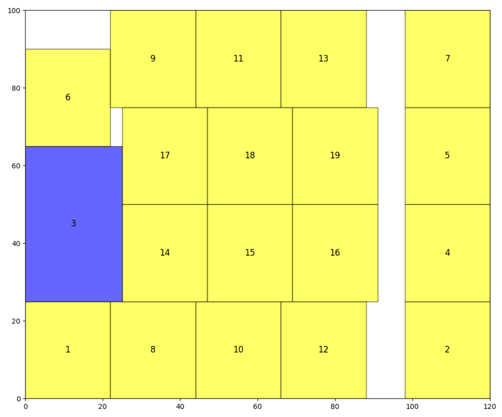
2 Evaluation

The metrics shown in Figure 1 provide insights into the training behavior of the RL model:

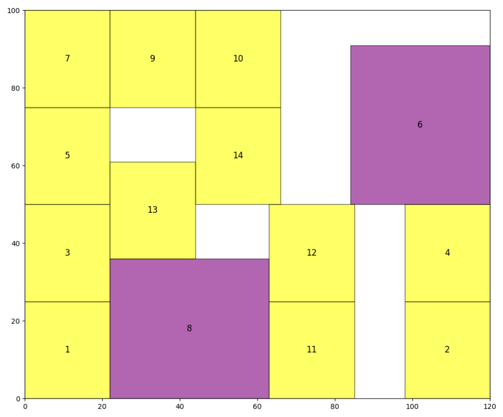
- **Explained Variance (`train/explained_variance`):** The explained variance is a measure of how well the value function predicts the returns. Initially, the variance is low, suggesting that the model struggled to make accurate predictions. As training progresses, explained variance improves significantly, approaching a value close to 1, which indicates that the model's value predictions are becoming more accurate and reliable over time.
- **Total Reward (`episode/total_reward`):** The total reward graph exhibits a gradual upward trend with some fluctuations. These fluctuations early in training indicate exploration, while the general rise suggests that the agent is learning to improve its performance, gradually achieving higher cumulative rewards per episode.
- **Training Loss (`train/loss`):** The training loss decreases significantly as training progresses, which is a positive indicator of the model's optimization. Lower loss values suggest that the model is successfully minimizing the error in predicting the value function. The decrease in the loss around the middle of the training process signifies that the model has effectively adjusted its parameters to fit the training data well.
- **Clip Fraction (`train/clip_fraction`):** The clip fraction shows how much clipping was applied to the policy during the training process. Initially, the clip fraction is high, indicating more clipping occurred to maintain a stable learning process. Over time, it decreases, suggesting the model is converging and policy updates are becoming smoother, thus needing less clipping.
- **Approximate KL Divergence (`train/approx_kl`):** The approximate KL divergence decreases throughout the training. This suggests that the updated policy remains relatively close to the previous policy, indicating stable learning without significant deviations. A lower KL divergence means that the policy updates are small and controlled, which is beneficial for PPO (Proximal Policy Optimization) to maintain stable improvements.

Overall, the training metrics suggest that the model is learning effectively, with increasing explained variance and initially higher rewards. However, the fluctuation in KL divergence and clipping fraction indicates potential instability, which could be addressed by fine-tuning hyper-parameters like learning rate, clipping range, or batch size. The decrease in total reward after 15k steps suggests that the policy may need adjustment to maintain generalizability.

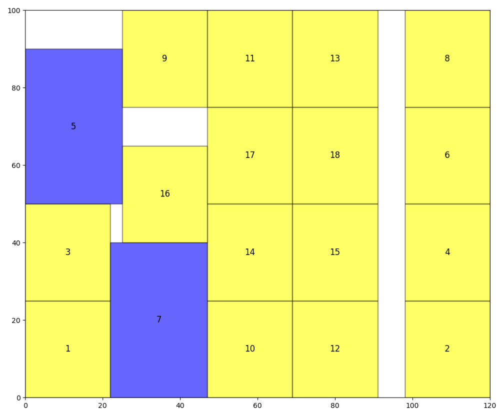
3 Examples



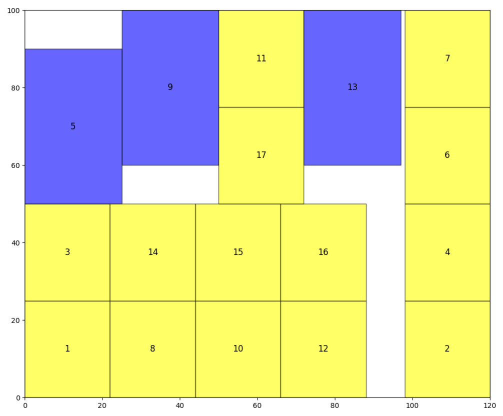
(a) Load Factor = 0.91, Squeezing Attempts = 3



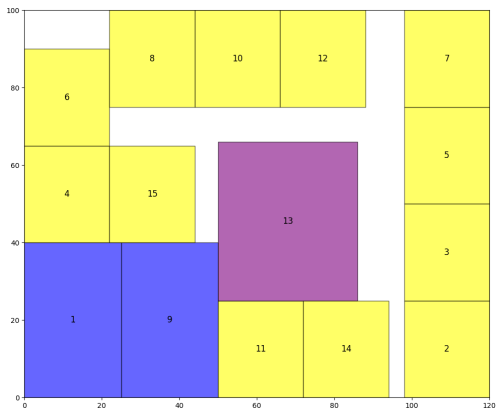
(b) Load Factor = 0.8, Squeezing Attempts = 2



(c) Load Factor = 0.90, Squeezing Attempts = 1



(d) Load Factor = 0.89, Squeezing Attempts = 1



(e) Load Factor = 0.84, Squeezing Attempts = 0

Figure 2: Trained policy example runs

From the figure we see that the policy consistently has preference for filling the pallet perimeter before going to the inner space. The shown examples consistently show load factors over 0.8. It also avoids squeezing thanks to the squeezing reward added. The squeezing attempts are **1.4** in average.

4 Conclusion

The presented approach effectively simulates a packing strategy considering load factor, squeezing attempts and perimeter filling. The approach uses a custom Reinforcement Learning environment with appropriate rewards, as well as heuristic functions for placing incoming boxes and counting squeezing attempts. The hardest part was the tuning of the model and considering all possible squeezing attempts. The final model runs over 30 000 time steps and used learning rate scheduler, the final rewards were found through tuning.