Solving the 3D Bin Packing Problem With Reinforcement Learning

Siddharth Nayak Harshad Khadilkar Ansuma Basumatary
Richa Verma

Overview

The Problem Statement

- Given a set of boxes, pack them in a container such that the volume packed is optimised.
- Also, for a real-world deployment on robots, certain physical constraints should be satisfied along with real-time decision making.

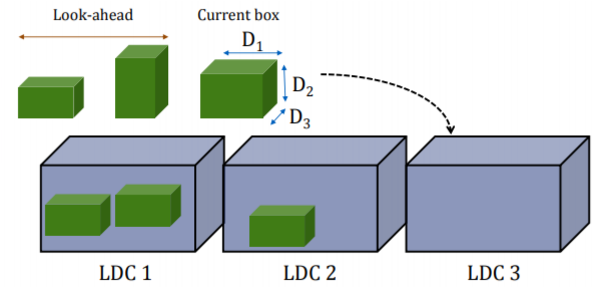


Figure: The setup to place boxes in a new container if no feasible positions are found in any of the previous containers. (LDC: Long Distance Container)

Why Reinforcement Learning?

- Recent success of RL for combinatorial problems like the travelling salesman problem, vehicle-routing problem, job-scheduling problem, etc.
- RL may give a better performance than traditional methods on sequential decision making problems.

Related Works

- Ranked Reward(R2)[1] algorithm computes ranked rewards by comparing the terminal reward of the agent against its previous performance, which is then used to update the neural network.
- Deep RL has been used in designing a bin with least surface area that could pack all the items , which uses policy-based RL (Reinforce) with a 2-step Neural Network (Ptr-Net) consisting of RNNs and has shown to achieve 5% improvement over the heuristic methods[2].

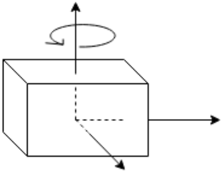
[1]Laterre et al., “Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization”.

[2]Hu et al., “Solving a New 3D Bin Packing Problem with Deep Reinforcement Learning Method”.

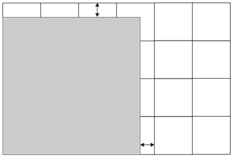
We make the following assumptions:

- Boxes are *cuboidal* in shape
- Order of boxes is not known a priori.
- Once a box is placed, the agent cannot shift it.
- Boxes can be rotated by $\pm 90^\circ$ only along three axes.
- The dimensions are integer (smallest scale 1cm).
- The agent can see n upcoming boxes.
- Boxes can be placed at a location only if all the corners have same level.(called feasibility)
- A box cannot be placed below an already placed box.

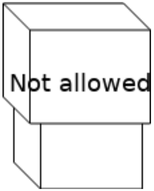
Assumptions



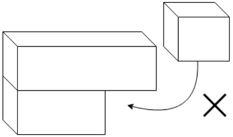
(a) Rotation is allowed



(b) Discrete Space



(c) All corners are not at the same level



(d) Cannot place box below an already placed one

Background

We denote some basic reinforcement learning terminology used.

- $\mathcal{S}$: State
- $\mathcal{A}$: Action
- $\mathcal{R}$: Reward
- $\pi(s)$ or π : Function which maps states to action
- $Q(s, a)$: Q-value function or action-value function, which is the expected reward if agent is in state(s) and takes action(a)
- We try to maximise the expected reward.

State Representation

We denote each container with a 2D grid of shape 45×80.
The values in the cell denote the height upto which that cell is filled.

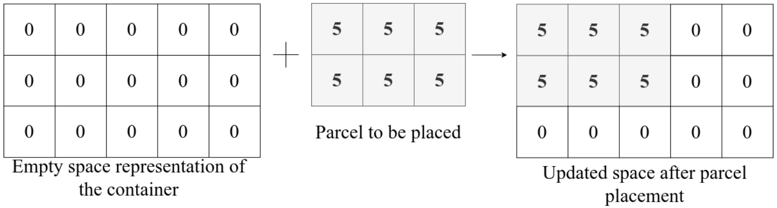
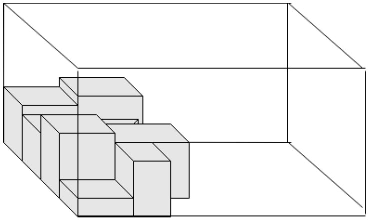


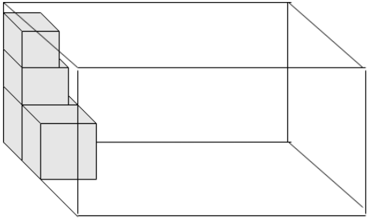
Figure: Space Representation. We will refer this type of state representation as *top-view* of container

Baselines

- **First Fit:** Place boxes in the first found feasible location, scanning from the top left of the container. If no feasible locations are found in the currently used containers, the orientation of the box is changed and the search is executed again.
- **Floor Building:** attempts to pack the container layer-by-layer, from the floor up. So effectively searches for positions with lowest height h possible.
- **Column Building:** attempts to build towers of boxes with the highest feasible h coordinate in the container. But not stable from a robot packability perspective.

Baselines


(a) Floor Building


(b) Column Building

Stability Scores

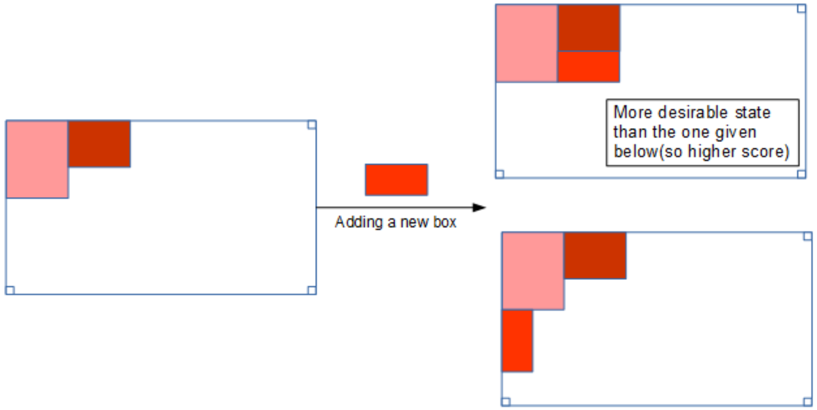
- We assign scores for each position for a given box, which we call **stability scores**, because they represent the stability of the overall structure once the box has been placed.
- Higher the stability score for a particular position, more favourable is that position to place the box.
- But we don't want to be greedy with this score and place the boxes just according to the scores. This is where the RL agent will choose a position from the top n stability score positions.

Stability Scores Equation

$$S = -\alpha_1\, G_{\mathrm{var}} + \alpha_2\, G_{\mathrm{high}} + \alpha_3\, G_{\mathrm{flush}} - \alpha_4\,(i+j) - \alpha_5\, h_{i,j} \tag{1}$$

- G_{var}: Sum of absolute differences of heights with neighbouring cells around the box. The walls do not contribute to this term.
- G_{high}: We count the number of bordering cells that are higher than the height of the proposed location after loading.
- G_{flush}: Denotes how smooth the surface will be after placing the box. We count the number of bordering cells that would be exactly level with the top surface of the box, if it was placed in the proposed location.
- $(i+j)$: We want the agent to start placing the boxes closer to the top-left of the box.
- $h_{i,j}$: We want the agent to place the boxes in a layer-wise manner. i.e. start a new layer(floor) only if there are no positions for the current box.

Stability scores


More desirable state than the one given below(so higher score)
Adding a new box

Search Space

- Once a box is placed at a position (i, j), we append the the tuple (i, j) in the search space.
- If we want to to place a box of dimensions (l, b, h), while evaluating the stability scores at (i, j) we also evaluate the stability scores at $(i, j \pm l), (i \pm b, j), (i \pm b, j \pm l)$.
- Once box has been placed, we remove all the points from the search space which come under the recently placed box.

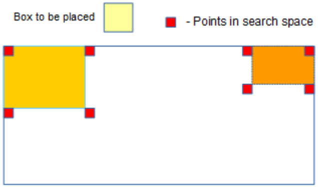


Figure: The search space evolution

WallE - the heuristic algorithm

- We use the stability score to guide the agent to place the box.
- For each box we find the stability score at all the positions in the search space, for all orientations.
- Then we place the box at the position which has the highest stability score.
- Performs slightly better than the baselines!

PackMan - Heuristic + RL

- Use the stability scores to give choices to the RL agent.
- Give the top-K (positions,orientations) to the agent.
- State for the agent is the current top-view of the container and the projected top-view of the container if the box is placed at the proposed location.
- The forward pass for all the K-(positions,orientations) is given as a batch.

Rewards for PackMan

At each loading step t, the environment returns a step reward given by,

$$r_t = \beta_1 \frac{v_{box,t}}{T \cdot L \cdot B \cdot H} - \beta_2 \frac{h_{max} - h_{min}}{H} - \beta_3 f_{new\_edges} + \beta_4 S_{i,j}, \tag{2}$$

- T : total number of containers used.
- L : length of the container
- B : width of the container
- H : height of the container
- $v_{box,t}$: volume of the current box
- $h_{max} - h_{min}$: variation in height across all occupied containers
- f_{new_edges} : count of new edges getting created due the loading of the box
- $S_{i,j}$: stability score at position (i, j)

Terminal Reward

$$\zeta = \frac{V_{packed}}{T \cdot L \cdot B \cdot H} - \tau \tag{3}$$

- τ : Average packing fraction over all episodes since the start of training.

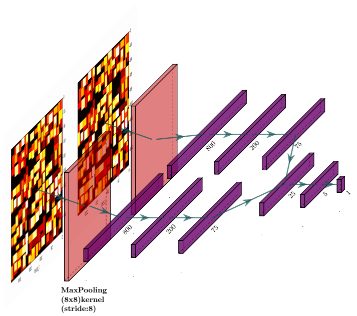
The terminal reward is back-propagated through all steps in an episode with discount factor $\rho = 0.99$.

Experiment Details

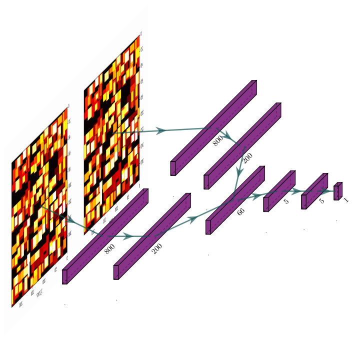
- We use 16 containers each of size 45×80 all lined one after another. Thus the container top-view is of size 45×1280
- We make a dataset[3] of boxes of random sizes which would ideally fit with 100% packing efficiency in 10 containers if placed *perfectly*.
- We use two different methods to reduce the dimension size of the input to the network, although both of them give similar results.
    - MaxPooling
    - Aggregate States
- We give top-10 positions to the RL agent.

[3]This artificial dataset was made to simulate the episodes and should not be confused with *datasets* used in supervised learning.

Networks Used



(a) MaxPooling



(b) Aggregate State

Networks Used

- **MaxPooling:** We use a maxpooling layer of kernel size 8×8 before linearising them. So the maxpooling converts the 45×1280 input to 5×160, which after linearisation is an $800-$dimensional vector
- **Aggregate State:** We use the aggregate function where the input state image is divided into sub-sections of size 9×8. Each of these sub-section($\mathbf{I}$) is then aggregated using the following formula:

$$\frac{mean(\mathbf{I}) - min(\mathbf{I})}{max(\mathbf{I}) - min(\mathbf{I})} \tag{4}$$

  So the input image is resized from 45×1280 input to 5×160 which is linearised to an 800-dimensional vector.

Advantage of maxPooling over Aggregate state: The maxPooling operation can be done in a GPU with much higher speed as compared to the aggregate function, which is evaluated in a CPU.

```
Result: Q-Network
Initialize replay buffer $\mathcal{M}$ to capacity $N$ ;
Initialize action-value function $Q$ with random weights;
while episode_num < tot_episode do
    Initialise state $s_1$ to the current container state;
    for $t = 1, T$ do
        Compute top-K box locations for each orientation ordered by stability
          score;
        Compute future states for each location;
        With probability $\epsilon$ select a random action (location) $a_t$;
        else select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$;
        Execute action $a_t$ in environment and observe reward $r_t$ and state $s_{t+1}$;
        Store transition $(s_t, r_t, s_{t+1})$ in $\mathcal{M}$;
        Sample random minibatch of transitions $(s_j, r_j, s_{j+1})$ from $\mathcal{M}$;
        Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & ; \\ \text{for non-terminal } s_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end
    episode$+ = 1$;
end
```

Algorithm 0: PackMan Algorithm

Results

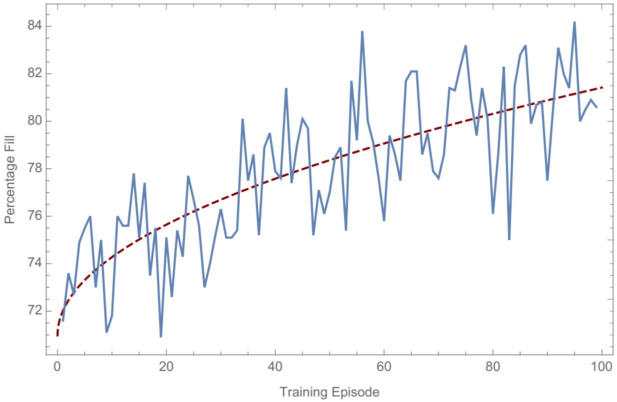


Figure: Improvement is there in performance, but need to train more and show saturation!

Speed comparison

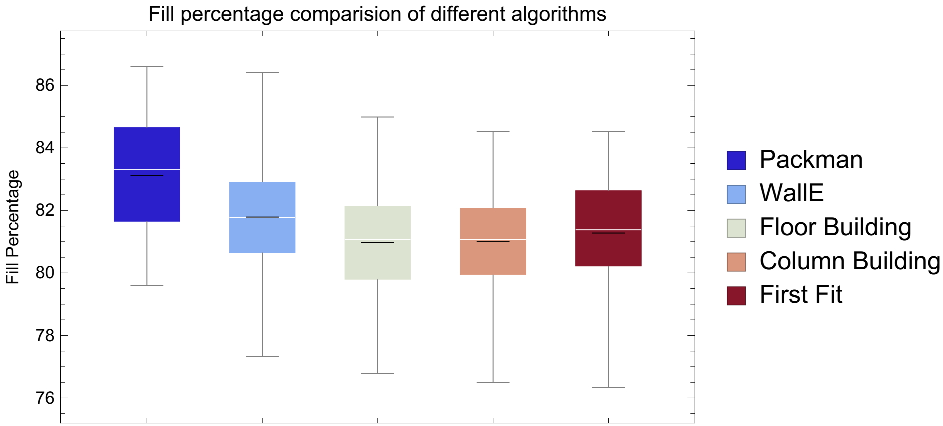


Figure: PackMan performs better than all others!

Advantages and Disadvantages of using Stability Scores

- **Advantages**
  - Helping the RL agent in choosing a small set of options.
  - Guarantee of all the boxes getting placed in the containers.
  - Sample efficient as most of the actions are *good enough* for achieving the goal.
- **Disadvantages**
  - Cost(time) of evaluating stability scores is very high. Especially when the search space has increased in size.

What if we *learnt* a function to get stability scores?

The main disadvantage of *PackMan* is the computations required for evaluating the *stability scores*.

- **Idea:** Learn a network which will give approximate stability score values for a given top-view state and the current box dimension.
    - Use *SegNet*[4] a network used for semantic segmentation of images.
    - The SegNet will classify each pixel in the top-view state representation with one of the [-10,-5,-4,-3,-2,-1,0,1,2,3,4,5] stability scores.

[4]Badrinarayanan, Kendall, and Cipolla, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation".

SegNet for our problem

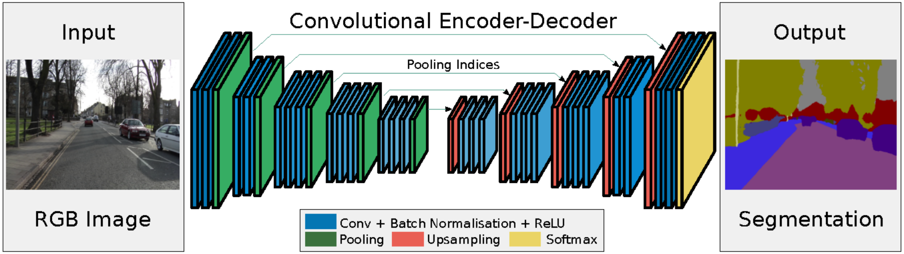


Figure: SegNet[5]: Input will be the top-view of the container and top-view of current box (2 channels). Output will be an array with stability scores at each position.

[5]Image for illustrative purposes only

What if we *learnt* a function to get stability scores?

- **Advantages:**
  - Lesser computation time for stability scores while training the RL agent.
- **Problems:**
  - The state representations have to be *good enough*, so that SegNet can learn co-relations between the state representations and stability scores.
  - Even small errors in classifying the pixels can cause the agent to place the box in an unstable place rendering the whole structure to be unstable.
- **Results:**
  - The network could not learn the co-relations as the loss saturated at a very high value.
  - Haven't experimented with this enough, so a viable option to try!

Policy Based methods

- The first thing which comes to one's mind when he/she is told to use RL for this problem would probably be this!

State(s)	Top view of container + dimensions of box
Action(a)	Let the agent give (i, j) to place the box

- We try to find a policy π_θ which maximises the expected discounted-reward

$$J(\theta) = \mathop{\mathbb{E}}_{s_0, a_0, \dots s_T} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right] \tag{5}$$

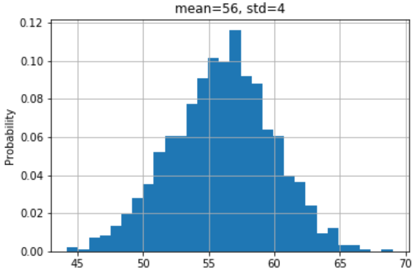
Deep Deterministic Policy Gradient(DDPG)[6]

- Used when action space is continuous.
- Maps states to actions directly.
- Is deterministic.
- In our case gives (i,j): the co-ordinates to place the current box.

[6] Lillicrap et al., "Continuous control with deep reinforcement learning".

Soft Actor Critic(SAC)[7]

- Used when action space is continuous.
- Gives a mean and standard deviation for each state to model a Gaussian Distribution over the actions.
- We sample from the distribution to get the co-ordinates(i, j) to place the box.



[7]Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor".

Disadvantages of using SAC and DDPG for our problem

- During training, if we use a randomly initialised policy, it may not be feasible to place the box at the position given by the agent. So we have to give the same state once again(for stochastic policies) discard the box completely.
- Sample inefficient, as a lot of the positions given by the agent are not feasible.
- Even if there is a small difference in the predicted position, the further placements of the boxes would be hampered.

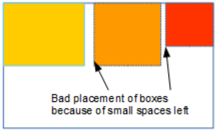
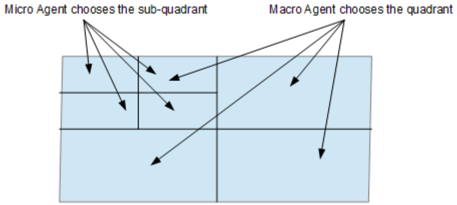


Figure: Bad placement of boxes due to small errors in co-ordinate values

Hierarchical Model

Here we have two agents: Macro and Micro

- Macro: Chooses the quadrant.
- Micro: Chooses the sub-quadrant in the quadrant given by macro.
- Now in the given sub-quadrant, choose position as: $(i, j) = \text{argmax}_{i,j}\, \mathrm{score}(\text{sub-quadrant})$


Micro Agent chooses the sub-quadrant
Macro Agent chooses the quadrant

Hierarchical Model

	State	Action
Macro	Top view of container + dimensions of box	choose quadrant
Micro	Top view of quadrant chosen by macro + dimensions of box	choose sub-quadrant
Final		choose (i, j) according to stability score

- **Advantage:** Search Space for evaluating stability scores is small and constant.
- **Disadvantage:** The final action is not taken by the agent but by the heuristic, so the network did not learn the co-relations of the state and the action quickly.

```
Result: Macro Policy $\pi_M$, Micro Policy $\pi_m$
random initialization of $\pi_M, \pi_m$ ;
replayBufferMacro,replayBufferMicro;
while $episode\_num < tot\_episode$ do
    $a_M = epsGreedy(\pi(s_M))$;
    $s_m = getMicroState(s_M, a_M)$;
    $a_m = epsGreedy(\pi(s_m))$;
    $(i, j) = chooseLocation(a_m, a_M)$ # returns argmax of stability scores in
     sub-quadrant;
    if feasible action then
        $reward = +1$,;
    else
        $reward = -3$;
    end
    store the tuple $< s_m, a_m, reward, next\_s_m >$ in replayBufferMicro;
    store the tuple $< s_M, a_M, reward, next\_s_M >$ in replayBufferMacro;
    Update policy according to DDPG or SAC;
    episode$+ = 1$;
end
```

Behavioural Cloning

- **Behavioural Cloning**(BC) is learning a function which maps states to actions from an *expert policy*.
- Expert policies can be either a table which contains $\pi^*(s)$ or some kind of heuristic which gives $\pi(s)$(not the *best* one but *good enough*)
- Can use these (imitated)policies as an initialisation for policy-based methods.

Behavioural Cloning

- This can be a fairly good initialisation for the policy-based agents to start learning.

```
**Result:** Policy $\pi$
random initialization of $\pi$ ;
**while** *episode_num* $<$ *tot_episode* **do**
    generate episode($\mathcal{S}, \mathcal{A}^*$) with the optimal actions for 100% packing ;
    get action predictions: $\mathcal{A} = \pi(\mathcal{S})$ for states in episode;
    $\mathcal{L} = \|\mathcal{A} - \mathcal{A}^*\|_2^2$;
    backprop;
    episode$+ = 1$;
**end**
```

States for Imitation Learning

- We used two different state representations:
  - Top-view of container + box dimensions.
    - The top-view of the box is linearised.
    - The box-dimensions are concatenated just before the final layer.
  - Top-view of past four frames of container + dimensions of past four boxes added.
    - The top-view of the past four frames are concatenated as channels of an image. So the top-view input shape is $45 \times 80 \times 4$.
    - The top-view is **not** linearised.
    - The dimensions of the past four boxes are concatenated just before the final layer.

Behavioural Cloning

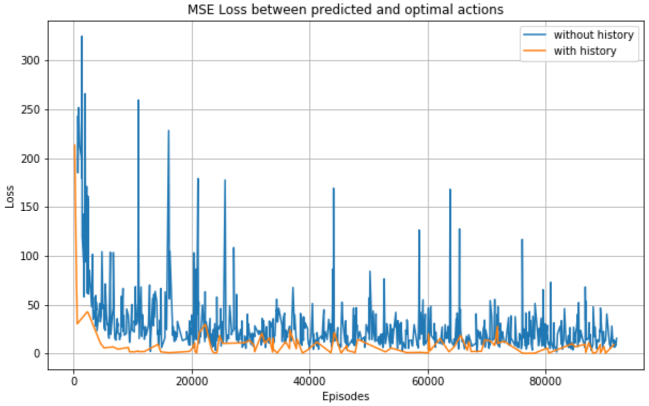


Figure: The MSE loss for the experiment with history saturates at a lower value than the experiment without history

step	pred_x	pred_y	opt_x	opt_y
0	0	0	0	0
1	2	1	0	0
2	0	0	0	0
3	7	8	0	26
4	0	25	0	26
5	0	25	0	26
6	0	45	0	46
7	0	45	0	46
8	0	45	0	46
9	0	67	0	66
10	0	66	0	66

Table: Most of the predicted actions differ from the optimal actions by a very small value.

Behavioural Cloning

- Using the imitated policy directly can be hard, as optimal action and the predicted action can differ by a very small value, but the consequences of the placement would hamper the further placement of boxes as shown in Fig.10.
- **Solution:**
  - Search in the neighbouring co-ordinates for the *desirability of positions* and place at the best position according to the scores to solve the problem of small discrepencies in the actions.
  - If action given by RL-agent is infeasible, then let the heuristic search for a position in the search space.
  - If no position is found by the heuristic too, then RL-agent was correct and hence no penalty.
  - If a feasible position is found by the heuristic, place the box at that position and penalise the RL-agent.

```
Result: Policy $\pi$
Input: Policy $\pi$ learned from Behavioural Cloning;
Initialise: empty replayBuffer, searchSpace while $episode\_num < tot\_episode$ do
    while episode not done do
        get $a_t = (i, j) = epsGreedy(\pi(s_t))$;
        search in a square region(searchSquare) of $(i \pm 5, j \pm 5)$ for maximum stability score;
        $(x, y) =$ argmax $stabilityScore(searchSquare)$;
        if $(i, j)$ is feasible then
            ;
        else if elseif condition then
            something elseif ;
        else
            something else ;
        end
    end
    episode$+ = 1$;
end
```

Conclusions

- Heuristics + RL works very well.
- State Representation choice is very important so that the agent can learn co-relations between:
  - State and actions taken.
  - Top-view of container and box dimensions.
- Could use the *'Prediction of Stability Scores'* method in future work for improving speed.

Possible Ideas to try further!

One of the biggest problems is to make the agent learn co-relations between the state and the action taken. So making the state representation as informative as possible is important.

- Different State Representations:
  - Top-view of container and top-view of box (CNN)
    - For top-view of the box, use some kind of attention mechanism to neglect the zeros in the array.
    - Use past four frames of the top-view and past four boxes added.
    - Use the difference of past four consecutive frames so that the agent can co-relate between actions and states.
  - Top-view of container and (l, b, h).
    - Concatenate (l, b, h) in some layer but with batch-norm for both channels (top-view and dimensions) before concatenation.
  - Autoencoder:
    - Instead of maxPooling or Aggregate States function to reduce the dimension, can train a convolutional autoencoder and take the latent space representation as the input to the policy networks.

Another problem with policy based methods was that the agent could not learn co-relations between the top-view of the container and the current box dimensions.

- Instead of just getting the co-ordinates of the position (i, j) to place the box, let the agent learn to give $(i, j), (i + b, j), (i, j + l), (i + b, j + l)$ so that the agent is forced to learn co-relations between the top-view of the container and the current box dimensions.

Hierarchy Model:

- Initially just train *Macro* so that it learns to choose quadrants.
- Then train *Micro* and *Macro* by updating *Micro* much more frequently than *Macro*.
- Keep on adding agents till the action of the last agent is down to the smallest resolution possible.
- Can use Option-Critic Algorithm[8] for the hierarchical model, because of the temporal-abstraction provided in it.
- Also, can try using the algorithm in the *Growing Action Spaces*[9] paper.

[8] Bacon, Harb, and Precup, “The Option-Critic Architecture”.
[9] Farquhar et al., “Growing Action Spaces”.

Behavioural Cloning

- Instead of just cloning actions, we can clone actions and stability scores so that the agent can learn that even with small differences in (i, j) the stability scores vary a lot.

```
Result: Policy $\pi$
random initialization of $\pi$ ;
while $episode\_num < tot\_episode$ do
    generate episode$(s, a^*)$ with the optimal actions for 100% packing ;
    get action and stability score predictions: $a, \mathcal{S} = \pi(s)$ for states in
     episode;
    get actual stability score $\mathcal{S}^*$ at co-ordinates $s$;
    $\mathcal{L} = \|A - A^*\|_2^2 + \|\mathcal{S} - \mathcal{S}^*\|_2^2$;
    backprop;
    episode$+= 1$;
end
```

Acknowledgements

- Harshad, Ansuma, Richa and Hardik for all the help and suggestions.
- RISE Lab, Indian Institute of Technology Madras for the compute resources.

The End