

A ASYMMETRIC SELF-PLAY GAME SETUP

A.1 GOAL VALIDATION

Some of the goals set via asymmetric self-play may not be useful or interesting enough to be included in the training distribution. For example, if Alice fails to touch any objects, Bob can declare a success without any action. A goal outside the table might be tricky to solve. We label such goals as *invalid* and make sure that Alice has generated a *valid* goal before starting Bob’s turn.

Even when Alice generates a valid goal, we can still penalize certain undesired goals. Specifically, if the visual perception is limited to a restricted area on the table due to the camera setting, we can penalize goals containing objects outside that range of view.

For goal validation, we check in the following order:

1. We check whether any object has moved. If not, the goal is considered *invalid* and the episode resets.
2. We check whether all the objects are on the table. If not, the goal is considered *invalid* and the episode resets.
3. We check whether a valid goal has objects outside the placement area, defined to be a 3D space that the robot end effector can reach and the robot camera can see. If any object is outside the area, the goal is deemed valid but obtains a *out-of-zone* penalty reward and the episode continues to switch to Bob’s turn.

A.2 REWARD STRUCTURE

Table 1 shows the reward structure for a single turn for goal setting and solving. The reward for Alice is based on whether it successfully generates a valid goal, and whether the generated goal is solved by Bob. Alice obtains 1 point for a valid goal, and obtains an additional 5 point game reward if Bob fails to achieve it. Additionally, a goal out of placement area triggers a -3 penalty. Rewarding Alice based only on Bob’s success or failure is simpler than the original reward from Sukhbaatar et al. (2018b), but we didn’t notice any degradation from this simplification (Appendix C.2).

Since Bob is a goal-conditioned policy, we provide *sparse* goal-conditioned rewards. Whenever one object is placed at its desired position and orientation, Bob obtains 1 point per-object reward. Bob obtains -1 reward such that the sum of per-object reward is at most 1 during a given turn. When all the objects are in the goal state, Bob obtains a 5 point success reward and its turn terminates. If Bob reaches a maximum number of allowed steps before achieving the goal, it is deemed a failure with 0 point reward.

When checking whether a goal has been achieved, we compare the position and orientation of each object with its goal position and orientation. For position, we compute the Euclidean distance between object centers. For rotation, we represent the orientation of an object by three Euler angles on three dimensions, roll, pitch, and yaw, respectively, and we compute the minimum angle needed to rotate the object into the goal orientation. If the distance and angle for all objects are less than a small error (0.04 meters and 0.2 radians respectively), we consider the goal achieved.

Table 1: Reward structure for a single goal.

Alice	Bob	Alice reward	Bob reward
Invalid goal	-	0	-
Out-of-zone goal	Failure	$1 - 3 + 5$	$0 + \text{per-object reward}$
Out-of-zone goal	Success	$1 - 3 + 0$	$5 + \text{per-object reward}$
Valid goal	Failure	$1 + 5$	$0 + \text{per-object reward}$
Valid goal	Success	$1 + 0$	$5 + \text{per-object reward}$

A.3 MULTI-GOAL GAME STRUCTURE

The overall flowchart of asymmetric self-play with a multi-goal structure is illustrated in Figure 10.

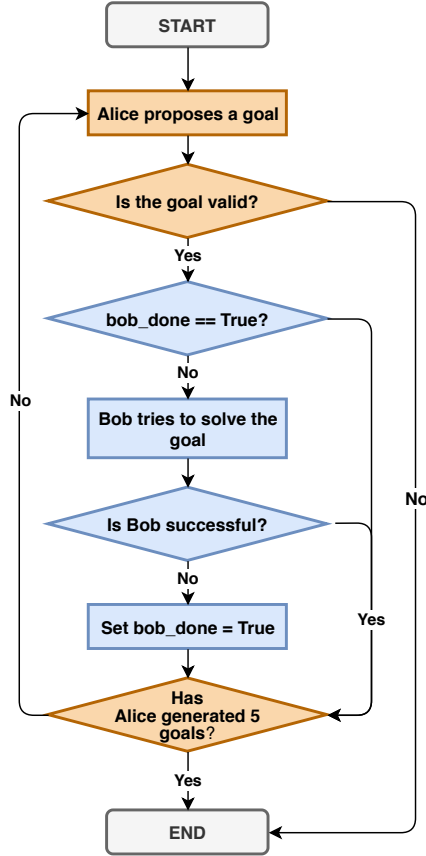


Figure 10: The flow chart of asymmetric self-play with a multi-goal game structure. The steps in orange belong to Alice while the blue ones belong to Bob.

We expect a multi-goal game structure to induce a more complicated goal distribution, as goals can be built on top of each other. For example, in order to stack 3 blocks, you might need to stack 2 blocks first as a subgoal at the first step. A multi-goal structure also encourages Bob to internalize environmental information during multiple trials of goal solving. Many aspects of the environment, such as the simulator’s physical dynamics and properties of objects, stay constant within one episode, so Bob can systematically investigate these constant properties and exploit them to adapt its goal solving strategy accordingly. Similar behavior with multi-goal setting was observed by [OpenAI et al., \(2019a\)](#).

In our experiments, when we report the success rate from multi-goal episodes, we run many episodes with a maximum of 5 goals each and compute the success rate as

$$\text{success_rate} = \frac{\text{total_successes}}{\text{total_goals}}.$$

Note that because each episode terminates after one failure, at the end of one episode, we would have either $\text{total_goals} = \text{total_successes}$ if Bob succeeded at every goal, or $\text{total_goals} = \text{total_successes} + 1$ if Bob fails in the middle.

A.4 TRAINING ALGORITHM

Algorithm 1 and 2 describe pseudocode for the training algorithm using asymmetric self-play. Both policies are optimized via Proximal Policy Optimization (PPO) ([Schulman et al., 2017](#)). Additionally, Bob optimizes the Alice behavioral cloning (ABC) loss using Alice’s demonstrations collected during the interplay between two agents. In the algorithm, \mathcal{L}_{RL} denotes a loss function for PPO and \mathcal{L}_{ABC} denotes a loss function for ABC. A trajectory τ contains a list of (state, action, reward) tuples,

Algorithm 1 Asymmetric self-play

Require: θ_A, θ_B ▷ Initial parameters for Alice and Bob
Require: η ▷ RL learning rate
Require: β ▷ weight of BC loss
for training steps = 1, 2, ... **do**
 $\theta_A^{\text{old}} \leftarrow \theta_A, \theta_B^{\text{old}} \leftarrow \theta_B$ ▷ initialize behavior policy parameters
 for each rollout worker **do** ▷ parallel data collection
 $\mathcal{D}_A, \mathcal{D}_B, \mathcal{D}_{BC} \leftarrow \text{CollectRolloutData}(\theta_A^{\text{old}}, \theta_B^{\text{old}})$ ▷ replay buffers for Alice, Bob and ABC
 end for
 $\theta_A \leftarrow \theta_A - \eta \nabla_{\theta_A} \mathcal{L}_{\text{RL}}$ ▷ optimize PPO loss with data popped from \mathcal{D}_A
 $\theta_B \leftarrow \theta_B - \eta \nabla_{\theta_B} [\mathcal{L}_{\text{RL}} + \beta \mathcal{L}_{\text{ABC}}]$ ▷ optimize RL loss with \mathcal{D}_B and ABC loss with \mathcal{D}_{BC}
end for

Algorithm 2 CollectRolloutData

Require: $\theta_A^{\text{old}}, \theta_B^{\text{old}}$ ▷ behavior policy parameters for Alice and Bob
Require: $\pi_A(a|s; \theta_A^{\text{old}}), \pi_B(a|s, g; \theta_B^{\text{old}})$ ▷ policies for Alice and Bob
Require: ξ ▷ whether Bob succeeded to achieve a goal
 $\mathcal{D}_A \leftarrow \emptyset, \mathcal{D}_B \leftarrow \emptyset, \mathcal{D}_{BC} \leftarrow \emptyset$ ▷ Initialize empty replay buffers.
 $\xi \leftarrow \text{True}$ ▷ initialize to True (success)
 for number of goals = 1, ..., 5 **do**
 $\tau_A, g \leftarrow \text{GenerateAliceTrajectory}(\pi_A, \theta_A^{\text{old}})$ ▷ generates a trajectory τ_A and a goal g
 if goal g is invalid **then**
 break
 end if
 if ξ is True **then**
 $\tau_B, \xi \leftarrow \text{GenerateBobTrajectory}(\pi_B, \theta_B^{\text{old}}, g)$ ▷ generate a trajectory τ_B and update ξ
 $\mathcal{D}_B \leftarrow \mathcal{D}_B \cup \{\tau_B\}$ ▷ update replay buffer for Bob
 end if
 $r_A \leftarrow \text{ComputeAliceReward}(\xi, g)$
 $\tau_A[-1][2] \leftarrow r_A$ ▷ overwrite the last reward in trajectory τ_A with r_A
 $\mathcal{D}_A \leftarrow \mathcal{D}_A \cup \{\tau_A\}$ ▷ update replay buffer for Alice
 if ξ is False **then**
 $\tau_{BC} \leftarrow \text{RelabelDemonstration}(\tau_A, g, \pi_B, \theta_B^{\text{old}})$ ▷ relabeled to be goal-augmented
 $\mathcal{D}_{BC} \leftarrow \mathcal{D}_{BC} \cup \{\tau_{BC}\}$ ▷ update replay buffer for ABC
 end if
 end for
return $\mathcal{D}_A, \mathcal{D}_B, \mathcal{D}_{BC}$

$\tau = \{(s_0, a_0, r_0), (s_1, a_1, r_1), \dots\}$. A goal-augmented trajectory τ_{BC} contains a list of (state, goal, action, reward) tuples, $\tau_{BC} = \{(s_0, g, a_0, r_0), (s_1, g, a_1, r_1), \dots\}$.

B TRAINING SETUP**B.1 SIMULATION SETUP**

We utilize the MuJoCo physics engine (Todorov et al., 2012) to simulate our robot environment and render vision observations and goals. We model a UR16e robotic arm equipped with a RobotIQ 2F-85 parallel gripper end effector. The robot arm is controlled via its tool center point (TCP) pose that is actuated via MuJoCo constraints. Additionally, we use a PID controller to actuate the parallel gripper using position control.

B.2 ACTION SPACE

We define a 6-dimensional action space consisting of 3D relative gripper position, 2D relative gripper rotation, and a 1D desired relative gripper finger position output that is applied symmetrically

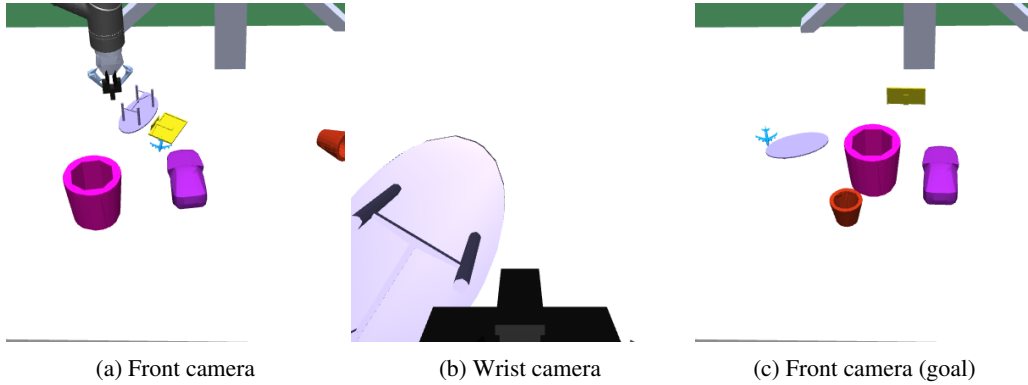


Figure 11: Example vision observations from our camera setup. (a) observation from a camera mounted in front of the table (the front camera). (b) observation from the mobile camera mounted on the gripper wrist. (c) goal observation from the front camera.

to the two gripper pinchers. The two rotational degrees of freedom correspond to yaw and pitch axes (wrist rotation and wrist tilt) respectively, with respect to the gripper base. We use a discretized action space with 11 bins per dimension and learn a multi-categorical distribution.

B.3 OBSERVATION SPACE

We feed observations of robot arm position, gripper position, object state, and goal state into the policy. The object state observation contains each object’s position, rotation, velocity, rotational velocity, the distance between the object and the gripper, as well as whether this object has contacts with the gripper. The goal state observation includes each object’s desired position and rotation, as well as the relative distance between the current object state and the desired state.

In the hybrid policy for the ShapeNet training environment, we additionally feed three camera images into the policy: an image of the current state captured by a fixed camera in front of the table, an image of the current state from a camera mounted on the gripper wrist, and an image of the goal state from the fixed camera. Figure 11 illustrates the example observations from our camera setup. Both Alice and Bob take robot and object state observations as inputs, but Alice does not take goal state inputs since it is not goal-conditioned.

B.4 MODEL ARCHITECTURE

We use independent policy and value networks in the PPO policy. Both have the same observation inputs and network architecture, as illustrated in Figure 13. The permutation invariant embedding module concatenates all the observations per object, learns an embedding vector per object and then does max pooling in the object dimension. The vision module uses the same model architecture as in IMPALA (Espeholt et al. 2018). For all experiments, we use completely separate parameters for the policy and the value network except the vision module, which is shared between them.

B.5 HYPERPARAMETERS

Hyperparameters used in our PPO policy and asymmetric self-play setup are listed in Table 2 and Table 3.

The maximum goal solving steps for Bob reported in Table 3 is the number of steps allowed per object within one episode. If Bob has spent all these time steps but still cannot solve the goal, it deems a failure and the episode terminates for Bob.

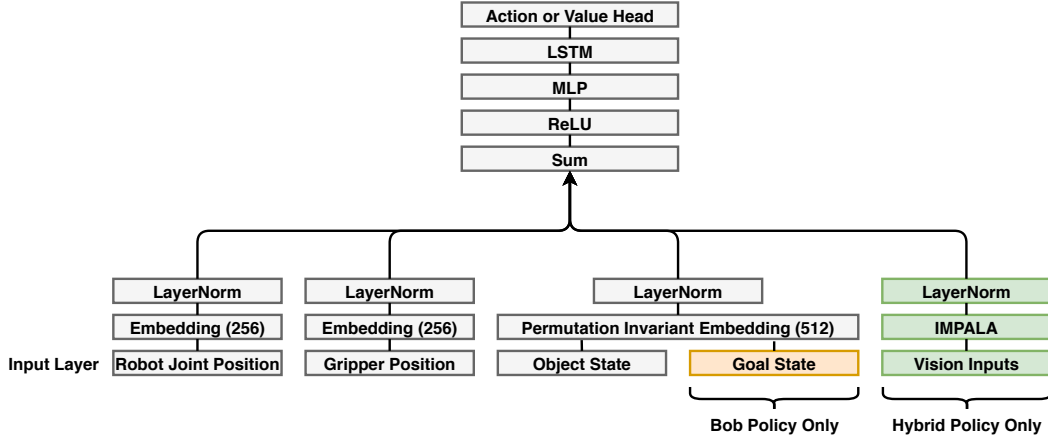


Figure 12: Network architecture of value/policy network.

Table 2: Hyperparameters used for PPO.

Hyperparameter	Value
discount factor γ	0.998
Generalized Advantage Estimation (GAE) λ	0.95
entropy regularization coefficient	0.01
PPO clipping parameter ϵ_{ppo}	0.2
ABC clipping parameter ϵ	0.2
optimizer	Adam (Kingma & Ba, 2014)
learning rate η	3×10^{-4}
sample reuse (experience replay)	3
value loss weight	1.0
ABC loss weight	0.5

Table 3: Hyperparameters used for hardware configuration, batch size and self-play episode length.

Hyperparameter	1-2 Block manipulation (state)	ShapeNet object rearrangement (hybrid)
GPUs per policy	1	32×8
rollout worker CPUs	64×29	576×29
batch size	4096	$55 \times 32 \times 8$
Alice’s goal setting steps T	100	250
Bob’s maximum goal solving steps	200	600

B.6 HOLDOUT TASKS

Here are a list of tasks for evaluating zero-shot generalization capability of the hybrid policy. Some of them are visualized in Figure 8. Note that none of the objects here appear in the training data.

- Table setting: arrange a table setting consisting of a plate, spoon, knife, salad fork, and main course fork.
- Mini chess: place four chess pieces next to a chess board.
- Rainbow (2-6 pieces): build a rainbow out of colored, wooden pieces by placing the half-circle shapes together so they resemble a rainbow. We have 5 variations of the rainbow tasks by taking different numbers of pieces which are indexed from the outer circle to inner one.

- Ball-capture: capture two, red field-hockey balls by placing four (two blue and two green) cylinders at so their rotational axes intersect rays from the spheres at roughly 90, 240, and 300 degree angles about the same axial (Z) direction.
- Tangram Puzzle: move blue pieces to form the standard, seven piece tangram square solution.
- Domino: stand up 5 wooden domino pieces in a curved layout.
- Block push (1–8 objects): push blocks to match position and orientation of goal configurations. All goal objects are on the surface of the table.
- Block pick-and-place (1–3 objects): push blocks, and lift up one block in the air. One goal object is in the air and all other goal objects are on the table surface.
- Block stacking (2–4 objects): stack blocks to form a tower in a specific location and specific rotation.
- YCB object push (1–8 objects): push YCB objects⁵ to match position and orientation of goal configurations. All goal objects are on the surface of the table.
- YCB pick-and-place (1–3 objects): push YCB objects and lift up one YCB object in the air. One goal object is in the air and all other goal objects are on the table surface.

By default, each holdout task presents 5 goals per episode and terminates episodes upon a failure. Exceptions are Table setting, Mini chess, Rainbow (2-6 pieces), Ball-capture, and Tangram, which only present a single goal per episode because only a single fixed goal configuration is available for each holdout task.

C NON SELF-PLAY BASELINES

C.1 BASELINES FOR CURRICULUM

We compared asymmetric self-play with several baselines incorporating hand-designed curricula in Sec. 5.2.

All the baselines are trained on a mixture of push, flip, pick-and-place, and stacking tasks as the goal distribution. The initial state of objects is generated by randomly placing objects within the placement area of the table without overlaps. The number of objects is sampled from $\{1, 2\}$ with equal probability.

Factorized Automatic Domain Randomization (FADR) (OpenAI et al., 2019a) is applied to grow curriculum parameters described below. Precisely, for each parameter we track a list of performance scores when the parameter is configured at current maximum and other parameters are randomly sampled. The value of this parameter will be increased if the tracked score rises above a threshold.

1. The no curriculum baseline trains the goal-conditioned policy directly on a fixed goal distribution and environment parameters. Precisely there are 50% goals for push and flip, 35% for pick-and-place, and 15% for stacking.
2. The curriculum:distance baseline uses a hand-designed curriculum over (1) goal_distance_ratio: the Euclidean distance between the initial position of the objects and their goal positions, and (2) goal_rotation_weight: the weight applied on the object rotation distance for goal state matching. At the beginning of the episode, given an initial position x_0 and a goal position x_g , we artificially make the goal easier according to goal_distance_ratio by resetting the goal position to $x'_g = x_0 + (x_g - x_0) \times \text{goal_distance_ratio}$. A small ratio reduces the distance and thus makes the task less difficult. The parameter goal_rotation_weight controls how much we care about a good match between object rotation. Given the current object rotation r_t and a desired rotation r_g , we consider them as a valid match if $|r_t - r_g| \times \text{goal_rotation_weight} < r_{\text{threshold}}$, where $r_{\text{threshold}} = 0.2$ (radians) is the success threshold for rotational matching. In other words, a small goal rotation weight creates a less strict success threshold. Both parameters range between $[0, 1]$ and gradually increase from 0 to 1 as training progresses.

⁵<https://www.ycbbenchmarks.com/object-models/>

3. The curriculum:distribution controls the proportion of pick-and-place and stacking goals via two ADR parameters, pickup_proba and stack_proba. When sampling new goals, with probability pickup_proba, a random object is moved up to the air and with probability stack_proba, we consider a small 2-block tower as the goal. Both parameters range between $[0, 0.5]$ and gradually increase from 0 to 0.5 as training progresses.
4. The curriculum:full baseline adopts all the ADR parameters described so far, goal_distance_ratio, goal_rotation_weight, pickup_proba and stack_proba. When setting up a pick-and-place goal, the height above the table surface is also interpolated according to goal_distance_ratio.

C.2 COMPARISON WITH TIMESTEP-BASED REWARD

Contrary to timestep-based reward originally proposed by [Sukhbaatar et al. \(2018b\)](#), we reward Alice simply based on the success or failure of Bob’s goal solving attempt. We compare our simplified reward structure with timestep-based reward for Alice as described in [Sukhbaatar et al. \(2018b\)](#) with time reward scale factor 0.01 ($\gamma = 0.01$ based on the notation from [Sukhbaatar et al. \(2018b\)](#)). The performance is very similar. Note that this result is not a direct comparison to [Sukhbaatar et al. \(2018b\)](#), but an ablation study of two different reward functions based on our best asymmetric self-play configuration.

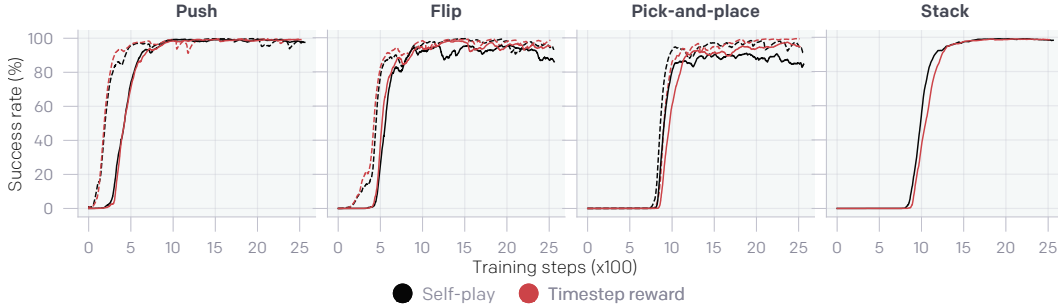


Figure 13: The comparison of our asymmetric self-play reward with the timestep-based reward.