Appendix

Contents

1 Implementation Details

1.1 Entity Encoding

To train our point cloud variational autoencoder [1], we normalize the point cloud of each entity P_i to be centered at the origin, i.e. $\bar{P}_i = P_i - t_i$, where t_i is the mean of all points in P_i. We then encode each centered $\bar{P}_i$ into a latent encoding: $z_i = \phi(\bar{P}_i)$. Our latent representation $u_i = [z_i, t_i]$ consists of the encoding of the point cloud's shape z_i and the position of the center of the point cloud t_i. We model the point cloud's position t_i explicitly such that the learned latent embedding z_i can focus on shape variation alone and the model that plans over u_i can still reason over point clouds at different spatial locations. During training, we record the 3D bounding box of all training data $t_{min}, t_{max} \in \mathbb{R}^3$, and we sample from $[t_{min}, t_{max}]$ during planning. We denote this combined distribution of $u = [z, t]$ as P_u.

1.2 Details on Training Set Feasibility Predictor

Hard Negative Samples Suppose skill k takes N_k latent vectors from observation $\hat{U}^o$ and M_k latent vectors from goal $\hat{U}^g$ as input. To generate random pairs of observations and goals as negative samples for the feasibility predictor, we can sample each latent point cloud representation u_i by sampling the shape z_i from the VAE prior p_z and sampling the position t_i from the distribution of positions in the training dataset. Such random negative samples are used similarly in DiffSkill [2]. However, as the combined dimension of the set representation becomes larger compared to a flat representation, we need a way to generate harder negative samples. To do so, for a positive pair of set representation $(\{u_i^o\}, \{u_j^g\})$, we randomly replace one of the entities u_i^o or u_j^g with a random sample in the latent space and use it as a negative sample. Our ablation results show that this way of generating hard negative samples is crucial for training our set feasibility predictor.

Noise on Latent Vectors During training of the feasibility predictor, for each of the input latent vector $u = [z, t]$, where $z \in \mathbb{R}^{D_z}$ is the latent encoding of the shape and $t \in R^3$ is the 3D position of the point cloud, we add a Gaussian noise to each part, i.e. $\hat{z} = z + \sigma_z \epsilon$ and $\hat{t} = t + \sigma_t \epsilon$, where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The amount of noise determines the smoothness of the feasibility landscape. Without any noise, planning with gradient descent with the feasibility function becomes much harder.

1.3 Details on the Attention Structure for Planning with Set Representation

Given the initial latent set observation U^{obs} with N_o components, and the skill sequences $k_1, \ldots, k_H$, in this section we describe how to generate the attention structure. We denote the latent set representation at step h as U^h, $h = 1 \ldots H$ and define $U^0 = U^{obs}$. As skill k_h takes in N_{k_h} components as input and M_{k_h} components as output, by calculation we know that U^h has $N_o + \sum_{i=1}^h M_{k_i} - N_{k_i}$ components. From now on, we denote $|U^h| = N_h$ and $U^h = \{u_{h,1}, \ldots, u_{h,N_h}\}$. As the skill k_h only applies to a subset of the input U^{h-1}, we now formally define the attention structure at step h to be I^h, which consists of a list of indices, each of length N_{k_h}, such that I^h selects a subset from U^{h-1} to be the input to the feasibility predictor, i.e.

$$\hat{U}^{h-1} = U_{I^h}^{h-1} \subseteq U^{h-1}.$$

However, enumerating all I^h is infeasible, as there are $C_{N_h}^{N_{k_h}}$ combinations for each step. Fortunately, we do not have to enumerate all different structures. The insight here is that, for each attention structure $I_1, \ldots I_H$, we will perform a low-level optimization. In this low-level optimization, we will first initialize all the latent vectors to be optimized from P_u and then perform gradient descent on them. As many of the attention structures yield topologically equivalent tree structures (An example of such a tree is illustrated in Figure 2c of the main paper), and each latent vector in the tree is sampled independently from the same distribution P_u, these topologically equivalent tree structures result in the same optimization process. As such, we do not need to exhaust all of such attention structures.

Instead of enumerating each topologically different structure and then sampling multiple initializations for the low-level optimization, we randomly sample sequences $(I_1, \dots I_H)$ and perform low-level gradient-descent optimization on all the samples. In this way, with enough samples, we will be able to cover all attention structures.

Now, we can sequentially build up the subgoals latent set representation during planning. Specifically, assuming that we have constructed the previous latent set representation U^{h-1}, we will now describe the procedure for constructing U^h, as well as the predicted feasibility for the current skill k_h, i.e. $f_{k_h}(\hat{U}^{o,h}, \hat{U}^{g,h})$, where $\hat{U}^{o,h}, \hat{U}^{g,h}$ are the subset of U^{h-1} and U^h attended by the feasibility predictor. First, we generate I_h by randomly choosing the index of a subset from U_{h-1}. U^h are composed of two parts: The first part is the latent vectors generated by applying the skill k_h. For this part, we will create a set of new vectors $u_{h,0}, \dots u_{h,M_{h_k}}$. This part of the latent vectors will be attended by the feasibility predictor as $\hat{U}^{g,h} = \{u_{h,0}, \dots u_{h,M_{h_k}}\}$. The second part of U^h comes from the previous latent set vectors that are not modified by the skill, i.e. $U^{h-1} \setminus U^{h-1}_{I_h}$, and U^h is the addition of both parts, i.e.

$$U^h = \hat{U}^{g,h} \cup (U^{h-1} \setminus U^{h-1}_{I_h})$$

In this way, we can sequentially build up U^h from U^{h-1}, and U^0 is simply U^{obs}. At the same time, we have determined our attention structure and the feasibility prediction. Our objective can thus be written as

$$\arg\min_{\mathbf{k},\mathbf{I},\mathbf{U}} J(\mathbf{k}, \mathbf{I}, \mathbf{U}) = \prod_{h=1}^{H} f_{k_h}(\hat{U}^{o,h}, \hat{U}^{g,h}) \exp(-C(U^H, U^g)), \tag{1}$$

where $\mathbf{U}$ is the set union of all latent vectors to be optimized.

1.4 Network Architectures

Set Feasibility Predictor We use a Multi-Layer Perceptron (MLP) with ReLU activations for our feasibility predictor. We apply max-pooling to the transformed latent vectors of $\hat{U}^o$ and $\hat{U}^g$ to achieve permutation invariance. Below is our architecture:

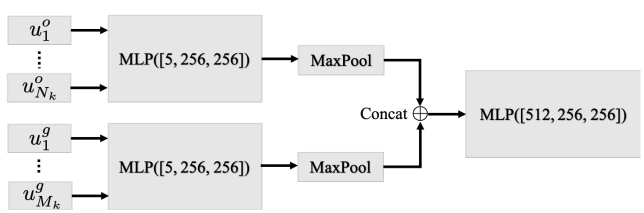


Figure 1: Architecture for the set feasibility predictor

Set Cost Predictor We use a 3-layer MLP with a hidden dimension of 1024 and ReLU activations.

Set Policy The set point cloud policy for the k^{th} skill π_k takes in an observed point cloud P^{obs}, a goal point cloud P^{goal}, and a tool point cloud P_k^{tool} and outputs an action at each timestep to control the tool directly. The tool point cloud P^{tool} is obtained by sampling points on the mesh surface of the tool and then transforming these points to the same camera frame as the P^{obs} and P^{goal}, assuming the pose of the tool is known from the robot state. Instead of taking latent vectors as input, the policy functions directly in point cloud space, which allows it to handle times when spatial abstraction is ambiguous. For instance, during cutting and merging, the number of dough components gradually increases or decrease, during which the latent set representation is not changing smoothly while the point clouds change smoothly during the process. We later show the advantage of using point clouds directly as the policy input. We concatenate each point's (x, y, z) coordinates with a one-hot encoding to indicate whether the point belongs to the observation, tool, or goal, and we input the points into a PointNet++ [3] encoder followed by an MLP which outputs the action. We use a point

	LiftSpread	GatherMove	CutRearrange	CRS + CRS-Twice
# of initial configurations	200	200	1500	1200
# of target configurations	200	200	1500	1200
# of training trajectories	1800	1800	1350	1080
# of testing trajectories	200	200	150	120
# of total trajectories	2000	2000	1500	1200
# of total transitions	1e5	1e5	7.5e4	6e4

Table 1: Summary of training/testing data

cloud for the tool to allow the PointNet encoder to reason about the interaction between the tool and the dough in the same space. We use PyTorch Geometric's [4] implementation of PointNet++ and with the following list of modules in our encoder.

```
SAModule(0.5,0.05,MLP([3+3,64,64,128]))
SAModule(0.25,0.1,MLP([128+3,128,128,256]))
GlobalSAModule(MLP([256+3,256,128,512,1024]))
```

The MLP following the encoder consists of hidden dimensions $[1024, 512, 256]$ and ReLU activations.

1.5 Training Details

Training data. We inherit the data generation procedure from DiffSkill [2]: first, we randomly generate initial and target configurations. The variations in these configurations include the location, shape, and size of the dough and the location of the tool. We then sample a specific initial configuration and a target configuration and perform gradient-based trajectory optimization to obtain demonstration data. For each task, the demonstration data consists of all the transitions from executing the actions outputted by the trajectory optimizer. We perform a train/test split on the dataset and select 5 configurations in the test split for evaluating the performance for all the methods. More information about training and testing data can be found in Table 1.

Point cloud VAE. We train our point cloud VAE by maximizing the evidence lower bound (ELBO). For a dataset of observations $P(X)$, which consists of the segmented point cloud of each entity in the scene, we optimize the following objective:

$$\mathcal{L}_{VAE} = \mathbb{E}_{Q_\phi(z|x)}\left[\log P_\psi(X|z)\right] - D_{KL}(Q_\phi(z|X)||p(z)) \tag{2}$$

where $Q_\phi(z|X)$ is the encoder modeled as a diagonal Gaussian, $P_\psi(X|z)$ is the decoder, and $p(z)$ is standard Gaussian. The VAE is pretrained, and we fix its weights prior to training the other modules.

Point cloud policy. We train our point cloud policy with standard behavioral cloning (BC) loss, i.e. for the k-th skill, we sample a transition from the demonstration data, which contains the observed point clouds $\{P_i^o\}$, goal point clouds $\{P_j^g\}$, a tool point cloud P_k^{tool}, and the action of the tool a. Then, we match point clouds in the observation set to those in the goal set by finding the pairs of point clouds that are within a Chamfer Distance of ϵ: $\{(P_i^o, P_j^g) \mid D_{Chamfer}(P_i^o, P_j^g) < \epsilon\}$ and filter out the non-relevant point clouds. Last, we pass the filtered point clouds into the policy and minimize the following loss:

$$\mathcal{L}_{\pi_k} = \mathbb{E}\left[\|a - \pi_k(\{P_i^o\}, P_k^{tool}, \{P_j^g\})\|^2\right] \tag{3}$$

Feasibility predictor. We train the feasibility predictor for the k-th skill by regressing to the ground-truth feasibility label using mean squared error (MSE) as loss, i.e.

$$\mathcal{L}_{f_k} = \mathbb{E}\left[\left(f_k(\hat{U}^o, \hat{U}^g) - \mathbb{1}\{\hat{U}^o, \hat{U}^g \text{ is a positive pair}\}\right)^2\right] \tag{4}$$

During training, we obtain positive pairs for the feasibility predictor by sampling two point clouds (P^{obs}, P^{goal}) from the same trajectory in the demonstration set. To find $\hat{U}^o, \hat{U}^g$, we first cluster the

Parameter	LiftSpread	GatherMove	CutRearrange	CRS + CRS-Twice
Yield stress	200	200	150	150
Ground friction	1.5	1.5	0.5	0.5
Young's modulus (E)	5e3	5e3	5e3	5e3
Poisson's ratio (ν)	0.15	0.15	0.15	0.15

Table 2: Parameters for simulation dough

observation and goal point clouds into two sets $\{P_i^o\}, \{P_j^g\}$ respectively. Then, we match point clouds in the observation set to those in the goal set by finding the pairs of point clouds that are within a Chamfer Distance of ϵ: $\{(P_i^o, P_j^g) \mid D_{Chamfer}(P_i^o, P_j^g) < \epsilon\}$. We then remove these point clouds from the corresponding set, since these are the point clouds that have already been moved to the target location in the goal. We can then encode the remaining point clouds into $\hat{U}^o, \hat{U}^g$ using our VAE.

Cost predictor. We train the cost predictor by simply regressing to the Chamfer Distance (CD) between two entities represented by their latent vectors, i.e.

$$\mathcal{L}_c = \mathbb{E}\left[\left(c(\phi(P_i), \phi(P_j)) - D_{Chamfer}(P_i, P_j)\right)^2\right] \tag{5}$$

where P_i and P_j are point clouds of a single entity sampled from the dataset and ϕ is the encoder. There are two reasons that we train a cost predictor on latent vectors instead of directly computing the Chamfer Distance between two point clouds. For one, decoding each latent vector would greatly bottleneck the planning speed. Experiments on CutRearrange show that with our learned cost predictor, the planning takes 35s; on the other hand, if we decode the latent vectors and use the Chamfer Distance, even with a subsampled point cloud of 200 points, the planning takes 37200s (around 10 hours), which is impractical to use. Moreover, using a cost predictor can also offer us the flexibility to incorporate more complex reward functions in the future.

Finally, We train our policy, feasibility predictor, and cost predictor with the following loss:

$$\mathcal{L}_{PASTA} = \sum_{k=1}^{K} \mathbb{E}\left[\lambda_\pi \mathcal{L}_{\pi_k} + \lambda_f \mathcal{L}_{f_k} + \lambda_c \mathcal{L}_c\right] \tag{6}$$

We use $\lambda_\pi = 1$, $\lambda_f = 10$, and $\lambda_c = 1$ for all of our experiments.

2 Details on Simulation Experiments

2.1 Hyperparameters for Simulation Dough

We use PlasticineLab [5] for evaluating our simulation experiments. We provide the hyperparameters that are relevant to the properties of the dough in simulation to enhance the replicability of our results. See Table 2 for details.

2.2 Hyperparameters for DBSCAN

To cluster a point cloud, we use Scikit-learn's [6] implementation of DBSCAN [7] with `eps=0.03, min_samples=6, min_points=10` for all of our environments. Further, we assign each noise point identified by DBSCAN to its closest cluster.

2.3 Hyperparameters for PASTA

Table 3 shows the hyperparameters used for PASTA in our simulation tasks. Planning for CRS-Twice requires a large amount of samples. Therefore, we modify the planner to improve sample efficiency. See Sec. 2.4 for details.

Training parameters	LiftSpread	GatherMove	CutRearrange	CRS	CRS-Twice
Point Cloud VAE					
learning rate	2e-3	2e-3	2e-3	2e-3	2e-3
latent dimension	2	2	2	2	2
Feasibility predictor					
learning rate	1e-4	1e-3	1e-4	1e-4	1e-4
batch size	256	256	256	256	256
noise on shape encoding σ_z	0	0	0	0.02	0.02
noise on position σ_t	0.01	0.01	0.005	0.01	0.01
Cost predictor					
learning rate	1e-4	1e-3	1e-4	1e-4	1e-4
batch size	256	256	256	256	256
Policy					
learning rate	1e-4	1e-3	1e-4	1e-4	1e-4
batch size	10	10	10	10	10
noise on point cloud	0.005	0.005	0.005	0.005	0.005
Planning parameters	**LiftSpread**	**GatherMove**	**CutRearrange**	**CRS**	**CRS-Twice**
learning rate	0.01	0.01	0.01	0.01	0.01
number of iterations	200	100	100	200	300
number of samples	5000	5000	5000	50000	500000

Table 3: Summary of hyperparameters used in PASTA. For CRS-Twice, we use the same model as CRS but modify the planner to have better sample efficiency.

2.4 Receding Horizon Planning for CRS-Twice

As the planning horizon increases, the number of possible skill sequences as well as the number of possible attention structures increases exponentially. The task of CRS-Twice has a planning horizon of 6 and is a much more difficult task to solve. As such, for this task, we specify the skill sequences and use Receding Horizon Planning (RHP). Starting from the first time step, we follow the procedure in Algorithm 1 but only optimize for H_{RHP} steps into the future and compare the achieved subgoal with the final target to compute the planning loss. After optimization, we take the first subgoal from the plan and discard the rest of the plan. We then repeat this process until we reach the overall planning horizon H. In our experiments, we use $H_{RHP} = 3$. While we can perform model predictive control and execute the first step before planning for the second step, we find this open-loop planning and execution to be sufficient for the task.

3 Details on Real World Experiments

3.1 Heuristic Policies

Transferring the learned policies from simulation to the real world can be more difficult than transferring the planner itself, as the policies are affected more by the sim2real gap, such as the difference in friction and properties of dough in the real world. To sidestep this challenge, for our real world experiments, we design three heuristic policies: cut, push, and roll to execute the generated plans in the real world.

Just like our learned policies in simulation, each heuristic policy takes in the current observation and the generated subgoal in point clouds and outputs a sequence of desired end effector positions used for impedance control. In addition, each policy takes in the attention mask provided by the planner indicating the components of interest. The same DBSCAN procedure is used for this. The cut policy first calculates the cutting point by computing the length ratio of the resulting components. Then it cuts the dough and separates it such that the center of mass of each resulting component matches the one in the subgoal. For the push policy, given a component and a goal component, the policy pushes the dough in the direction that connects the two components' center of mass. The roll policy first moves the roller down to make contact with the dough. Then, based on the goal component's length, the policy calculates the distance it needs to move the roller back and forth when making contact with the dough.

3.2 Procedure for Resetting the Dough

To compare different methods with the same initial and target configurations, we first use a 3D-printed mold to fit the dough to the same initial shape. We then overlay the desired initial location on the image captured by the top-down camera and place the dough at the corresponding location in the workspace to ensure different methods start from the same initial location.

3.3 Procedure for the Human Baseline

Following the same procedure in section 3.2, we first reset the dough to the initial configuration. Then, we overlay the goal point cloud on the image captured by the top-down camera. The overlay image is shown on a screen and presented to the human in real-time when the human is completing the task.

3.4 Procedure for Making the Real Dough

Material	Quantity(g)	Baker's percentage(%)
Flour	300	100
Water	180	60
Yeast	3	1

Table 4: All-purpose dough recipe

We follow the recipe shown in Table 4 to make the real dough. Following the tradition of baking, we use the backer's percentage, so that each ingredient in a formula is expressed as a percentage of the flour weight, and the flour weight is always expressed as 100%. First, we take 300 grams of flour, 3 grams of yeast, and 180 grams of water into a basin. Then, we mix the ingredients and knead the dough for a few minutes. Next, we use a food warp to seal the dough in the basin and put them in the refrigerator to let the dough rest for 4-5 hours. Finally, we take out the dough from the refrigerator and reheat it with a microwave for 30-60 seconds to soften it.

4 Additional Experiments

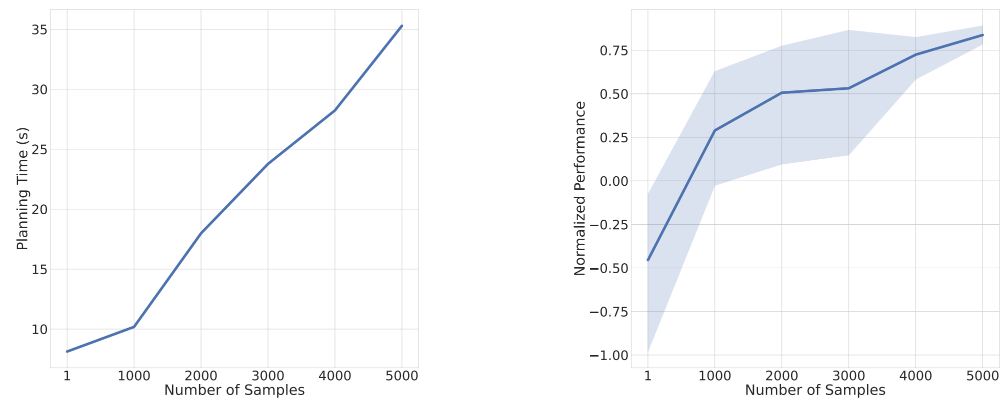
4.1 Ablation Studies

Ablations on feasibility predictor. Following the discussions in Sec 1.2, we train a feasibility predictor without adding any noise to show that adding noise helps with the optimization landscape during planning. We call this ablation *No Smoothing Feasibility*. As shown in Table 5, this variant only achieves half of the success rate of PASTA, suggesting the importance of noise during training.

Ablation Method	Performance / Success
No Smoothing Feasibility	0.744 / 40%
Shared Encoder Policy	-0.304 / 0%
Tool Concat Policy	0.516 / 60%
Set without Filtering	0.360 / 20%
PASTA (Ours)	**0.837 / 80%**

Table 5: Additional ablation results from CutRearrange.

Ablations on policy. We consider two ablation methods for our set policy. First, we consider a *Shared Encoder Policy* that takes in the latent vectors from the encoder and uses a max pooling layer followed by an MLP to produce the action. The architecture is very similar to our Set Feasibility Predictor. Our results in Table 5 show that this architecture has zero success in our task. We hypothesize that this is because the entity encoding can



(a) Planning time v.s. number of samples

(b) Planning performance v.s. number of samples

Figure 2: Planning time and performance with varying number of samples in CutRearrange.We show the mean and standard deviation of performance over 5 runs.

be unstable during the skill execution. For example, during cutting, the dough slowly transitions from one piece to two pieces, making the input to the policy unstable.

Second, we compare with a *Tool Concat Policy* that takes in the observation and goal point cloud of the dough, passes them through a PointNet++ [3] encoder to produce a feature, and then concatenates the tool state to the feature. The concatenated feature is passed through a final MLP to output the action. In comparison, the set policy in PASTA takes the point cloud of the tool and concatenates it with the dough in the point cloud space before passing it to the PointNet. We hypothesize that this way allows PointNet to reason more easily about the spatial relationships between the tool point cloud and the dough point cloud. Results in Table 5 highlight the advantage of using a point cloud to represent the tool.

Ablation on set representation. We consider a variant of PASTA *Set without Filtering*, which uses the same set representation as PASTA, but does not filter entities that are approximately the same both during training and testing. This filtering is only possible with a set representation and we want to show the advantage of this filtering. For this ablation, during training, the feasibility predictor takes in all the entities in the scene in set representation, and the policy takes in the concatenation of point clouds from each entity. During planning, we do not enumerate attention structures but instead optimize for all the entities. As shown in Table 5, without filtering, this ablation performs significantly worse than PASTA, showing that filtering is an important advantage enabled by our set representation.

4.2 Visualization of the Latent Space

We visualize the latent space of PASTA in CutRearrange in Figure 3 and visualize the latent space of Flat 3D baseline in Figure 4 for comparison. Since we use a latent dimension of 2 for all of our environments, we can visualize the original latent space without applying any dimensionality reduction techniques. PASTA only encodes the shape of each entity and thus can better model the variations in shapes. On the other hand, Flat 3D couples the shape variation with the relative position of two entities. This makes a flat representation difficult to generalize compositionally to scenes with different numbers of entities or scenes with entities that have novel relative spatial locations to each other.

4.3 Runtime of PASTA

We implement the planning in the latent set representation in an efficient way, which can plan with multiple different structures in parallel on a GPU. To demonstrate the efficiency of PASTA, we vary the number of samples used for planning and record the planning time and final performance. We conduct the experiments in CutRearrange. Figure 2a shows that the planning time scales approximately linearly with the number of samples, and Figure 2b shows the planning performance versus the number of samples. As the result suggests, PASTA can achieve its optimal performance with a very short amount of planning time (under 1 minute) for the majority of our tasks. Finally, we summarize the planning time for all of our tasks in simulation in Table 6.

	LiftSpread	GatherMove	CutRearrange	CRS	CRS-Twice*
Planning time (seconds)	58	35	35	307	7810

Table 6: Summary of planning time of PASTA in all of the simulation tasks. CRS-Twice uses Receding Horizon Planning, which results in an increase in planning time.

4.4 Additional metrics for real world experiments

We also quantitatively computed the action error v.s. subgoal error for our real world trajectories. The results are shown in Table 7. From the results in the table, our planned goal is closer to the ground truth goal than the achieved goal, measured by the Earth Mover's Distance (EMD), which shows that the controller does not compensate for the error of the planner.

	CutRearrange	CRS	CRS-Twice
EMD(planned goal, ground-truth goal)	0.038 ± 0.004	0.027 ± 0.004	0.029 ± 0.002
EMD(reached goal, ground-truth goal)	0.056 ± 0.007	0.044 ± 0.006	0.054 ± 0.016

Table 7: Action error v.s. subgoal error for real world experiments. For each task, the mean $\pm$ std for 4 trajectories are shown.

4.5 Robustness of PASTA

We show that PASTA is robust to two types of variations and can retain high performance.

Robust to planning horizon First, we increase the planning horizon from the minimal length for the task (3) to twice the minimal length (6), and we observe that PASTA retains a high performance across all horizons. The results are shown in Table 8. This suggests that in practice, one can specify a maximum planning horizon for PASTA when the exact horizon is unknown.

Robust to distractors Second, we show that PASTA is robust to distractors in the scene. We add 2 distractor objects in CRS (which makes the scene have 4 objects in total). We observe that PASTA retains a normalized performance of 0.879 and 100% success rate (without distractor: 0.896/100%) using the same amount of samples to plan. Our planner is able to ignore the distractors using our attention structure at every step to only attend to the relevant components in the scene. We also added an example trajectory with distractor dough pieces to our website under "CRS with distractors".

Planning Horizon	3	4	5	6
Performance	0.896	0.866	0.90	0.878
Success Rate	5/5	4/5	5/5	4/5

Table 8: PASTA's performance v.s. varying numbers of planning horizon in CRS.

5 Further Discussion on Limitations and Future Work

More Efficient Planning Planning skill sequences with a large search space is a challenging problem by itself but much progress has been made by the task and motion planning community to obtain a plan skeleton [8, 9, 10]. For example, Caelan et al. [8] propose two methods, the first one is to interleave searching the skill sequence with lower-level optimization and the second one is to have lazy placeholders for some skills. Danny et al. [10] propose to predict skill sequences from visual observation. Recent works have also explored finding skill sequences using pre-trained language models [11, 12].

Sim2Real Transfer for Real Dough One possible approach is to train with domain randomization to make the policy more robust to changing dynamics (e.g. stickiness) of dough. Another option is to perform online system identification of the dough dynamics parameters [13, 14] or real2sim methods [15, 16]. In future work, we can also integrate our method with other works that perform low-level dough manipulation in the real world, such as recent work from Qi et al. [17].

Goal Specification Our planner requires specifying the goal with a point cloud and coming up with a point cloud goal is not always easy. However, rapid progress are being made with language-conditioned manipulation and future work can combine language to specify more diverse tasks.

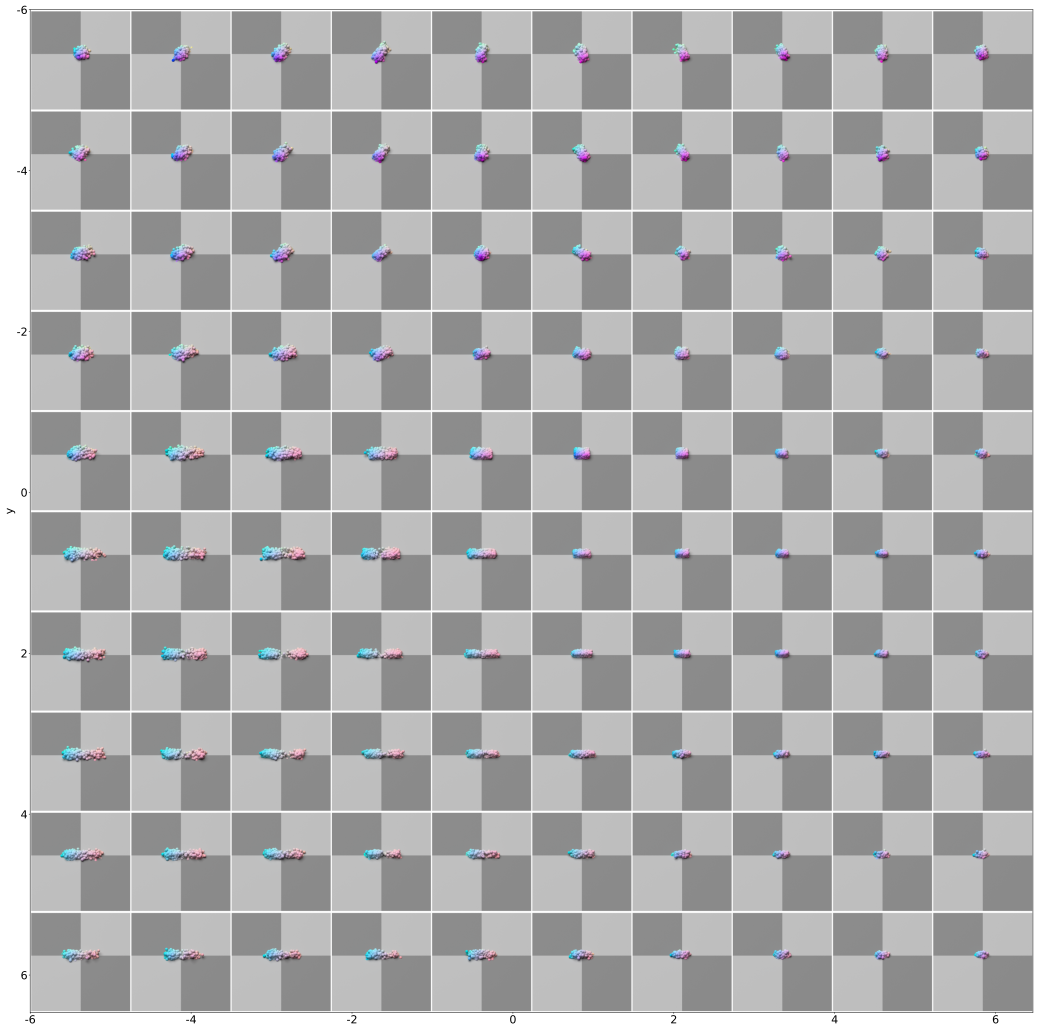


Figure 3: Latent space of PASTA in CutRearrange. We sample coordinates on a grid from the 2D latent space encoding the shapes and then decoding each latent vector into a point cloud. We then rearrange the decoded point cloud into the grid based on the corresponding coordinates in the latent space.

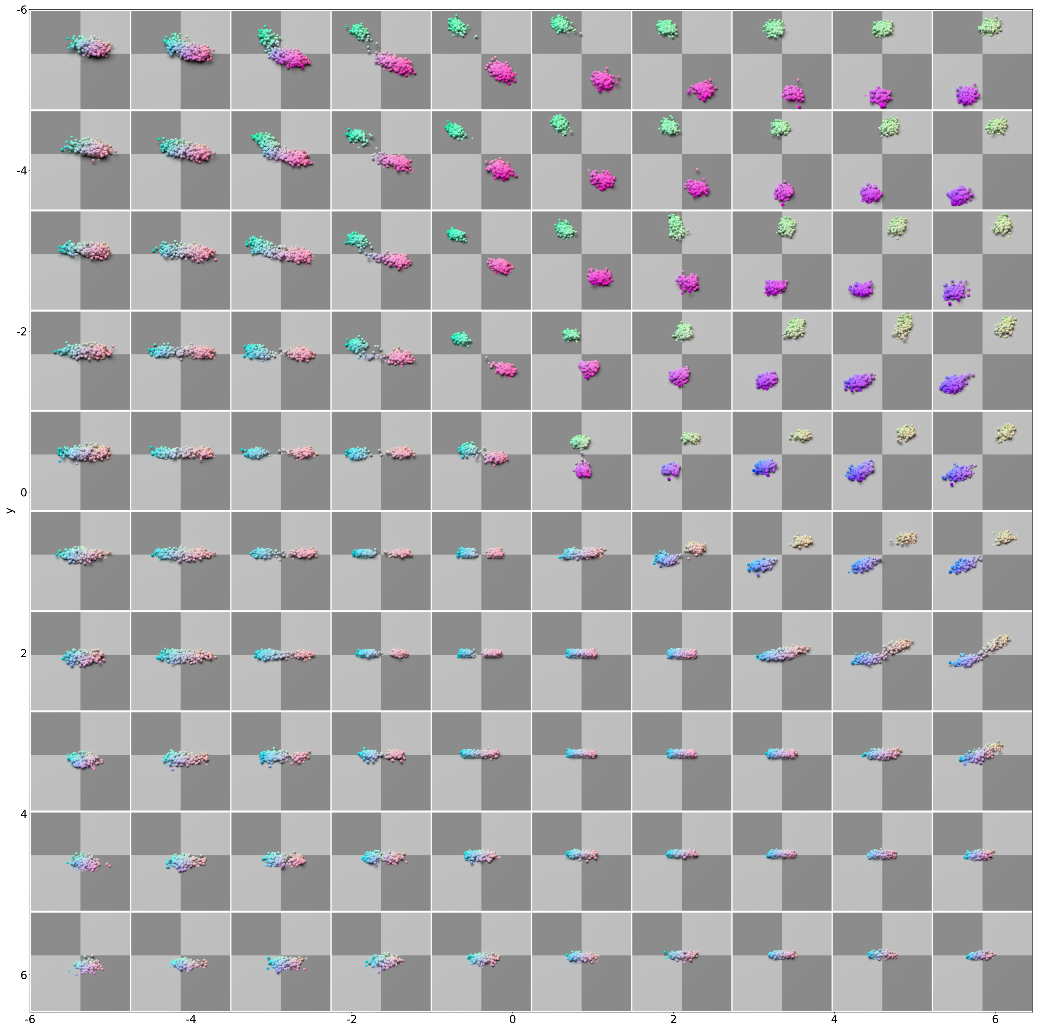


Figure 4: Latent space of Flat 3D in CutRearrange. We sample coordinates on a grid from the 2D latent space encoding the shapes and then decoding each latent vector into a point cloud. We then rearrange the decoded point cloud into the grid based on the corresponding coordinates in the latent space.

References

[1] G. Yang, X. Huang, Z. Hao, M.-Y. Liu, S. Belongie, and B. Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. *ICCV*, 2019.

[2] X. Lin, Z. Huang, Y. Li, J. B. Tenenbaum, D. Held, and C. Gan. Diffskill: Skill abstraction from differentiable physics for deformable object manipulations with tools. *ICLR*, 2022.

[3] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, 2017.

[4] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[5] Z. Huang, Y. Hu, T. Du, S. Zhou, H. Su, J. B. Tenenbaum, and C. Gan. Plasticinelab: A soft-body manipulation benchmark with differentiable physics. In *International Conference on Learning Representations*, 2020.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[8] C. Garrett, T. Lozano-Pérez, and L. Kaelbling. Sample-based methods for factored task and motion planning. 2017.

[9] B. Kim and L. Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. In *Conference on Robot Learning*, pages 955–968. PMLR, 2020.

[10] D. Driess, J.-S. Ha, and M. Toussaint. Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. *arXiv preprint arXiv:2006.05398*, 2020.

[11] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[12] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207*, 2022.

[13] W. Yu, J. Tan, C. K. Liu, and G. Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv preprint arXiv:1702.02453*, 2017.

[14] A. Kumar, Z. Fu, D. Pathak, and J. Malik. Rma: Rapid motor adaptation for legged robots. *arXiv preprint arXiv:2107.04034*, 2021.

[15] F. Ramos, R. C. Possas, and D. Fox. Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators. *arXiv preprint arXiv:1906.01728*, 2019.

[16] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[17] C. Qi, X. Lin, and D. Held. Learning closed-loop dough manipulation using a differentiable reset module. *IEEE Robotics and Automation Letters*, 7(4):9857–9864, 2022.