

A Task Descriptions

Simulated tasks. We select 10 language-conditioned tasks from RLBench [14], all of which involve at least 2 variations. See Table 5 for an overview. Our task variations include randomly sampled colors, sizes, counts, placements, and categories of objects, totaling 166 different variations. The set of colors have 20 instances: red, maroon, lime, green, blue, navy, yellow, cyan, magenta, silver, gray, orange, olive, purple, teal, azure, violet, rose, black, and white. The set of sizes includes 2 types: short and tall. The set of counts has 3 instances: 1, 2, 3. The placements and object categories are specific to each task. For example, `open drawer` has 3 placement locations: top, middle and bottom. In addition to these semantic variations, objects are placed on the tabletop at random poses within a limited range.

Table 5: Language-conditioned tasks in RLBench [14].

Task	Variation Type	# of Variations	Avg. Keyframes	Language Template
<code>close jar</code>	color	20	6.0	"close the — jar"
<code>open drawer</code>	placement	3	3.0	"open the — drawer"
<code>sweep to dustpan</code>	size	2	4.6	"sweep dirt to the — dustpan"
<code>meat off grill</code>	category	2	5.0	"take the — off the grill"
<code>turn tap</code>	placement	2	2.0	"turn — tap"
<code>slide block</code>	color	4	4.7	"slide the block to — target"
<code>put in drawer</code>	placement	3	12.0	"put the item in the — drawer"
<code>drag stick</code>	color	20	6.0	"use the stick to drag the cube onto the — — target"
<code>push buttons</code>	color	50	3.8	"push the — button, [then the — button]"
<code>stack blocks</code>	color, count	60	14.6	"stack — — blocks"

Generalization tasks in simulation. We design 6 additional tasks where the scene is changed based on the original training environment, to test the generalization ability of GNFactor. Table 6 gives an overview of these tasks. Videos are also available on gnfactor-robot.github.io.

Table 6: Generalization tasks based on RLBench.

Task	Base	Change
<code>drag (D)</code>	<code>drag stick</code>	add two colorful buttons on the table
<code>slide (L)</code>	<code>slide block</code>	change the block size to a larger one
<code>slide (S)</code>	<code>slide block</code>	change the block size to a smaller one
<code>open (n)</code>	<code>open drawer</code>	change the position of the drawer
<code>turn (N)</code>	<code>turn tap</code>	change the position of the tap
<code>push (D)</code>	<code>push buttons</code>	add two colorful jar on the table

Real robot tasks. In the experiments, we perform three tasks along with three additional tasks where distracting objects are present. The *oven* task requires the agent to open the door on an oven, a task which poses challenges due to the precise coordination required. The *faucet* task requires the agent to rotate the faucet back to center position, which involves intricate motor control. Lastly, the *teapot* task requires the agent to locate the randomly placed teapot in the kitchen and move it on top of the stove with the correct pose. Among the three, the teapot task is considered the most challenging due to the random placement and the need for accurate location and rotation of the gripper. All 6 tasks are set up in two different kitchens, as visualized in Figure 6. The keyframes used in real robot tasks are given in Figure 7.

B Implementation Details

Voxel encoder. We use a lightweight 3D UNet (only 0.3M parameters) to encode the input voxel $100^3 \times 10$ (RGB features, coordinates, indices, and occupancy) into our deep 3D volumetric representation of size $100^3 \times 128$. Due to the cluttered output from directly printing the network, we

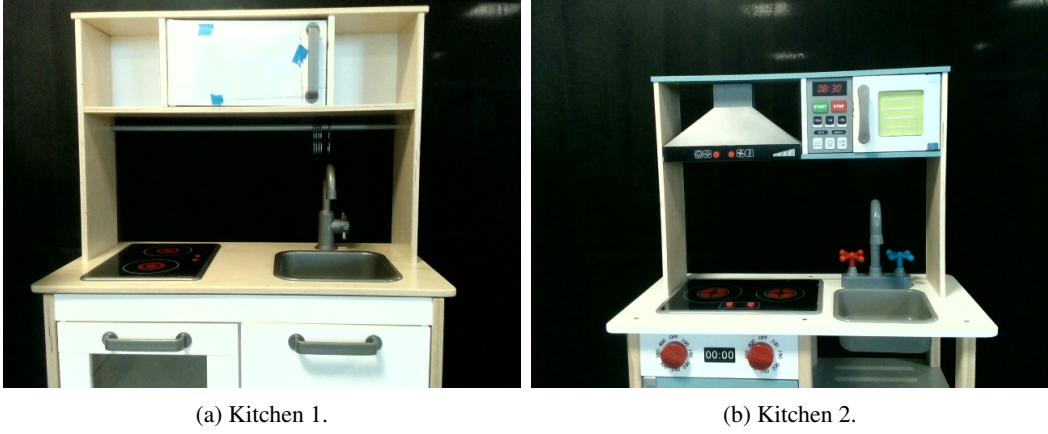


Figure 6: **Kitchens.** We give a closer view of our two kitchens for real robot experiments. The figures are captured in almost the same position to display the size difference between the two.

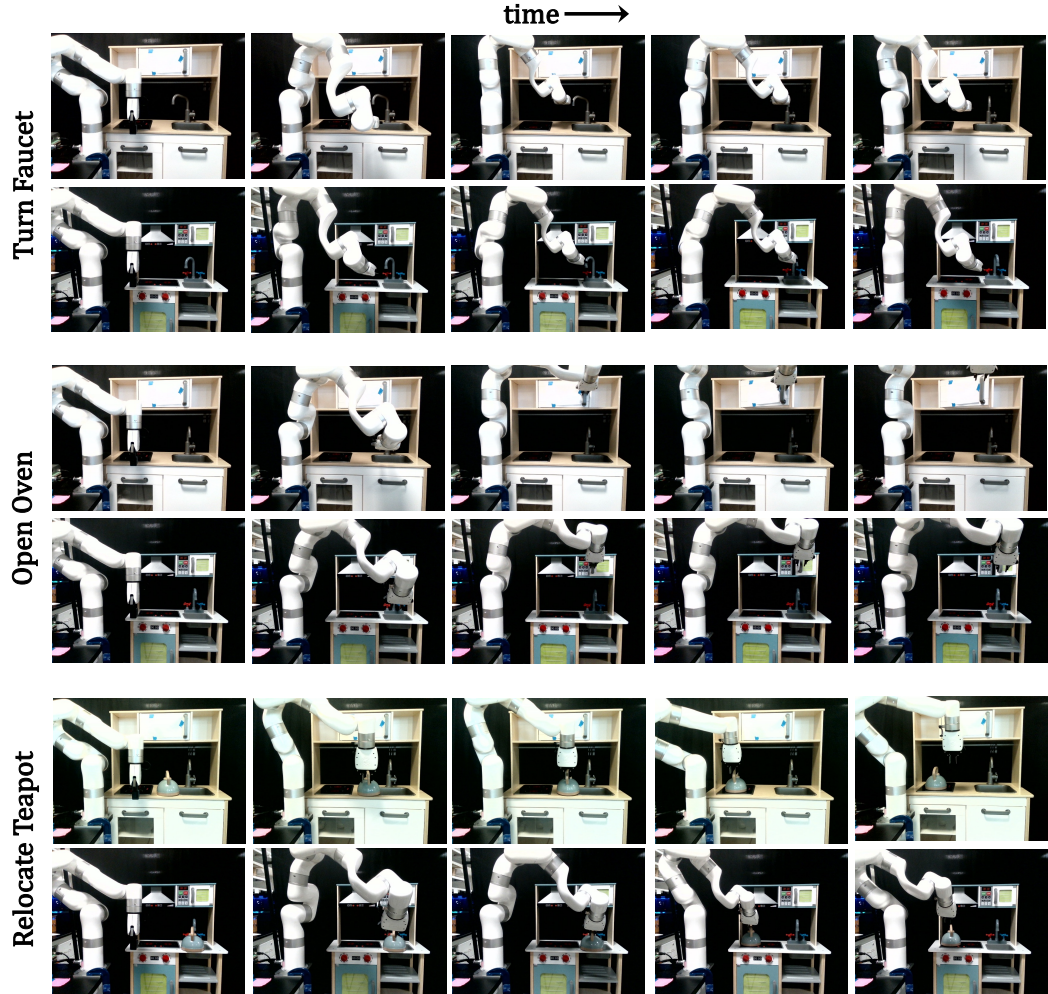


Figure 7: **Keyframes for real robot tasks.** We give the keyframes used in our 3 real robot tasks across 2 kitchens.

provide the PyTorch-Style pseudo-code for the forward process as follows. For each block, we use a cascading of one Convolutional Layer, one BatchNorm Layer, and one LeakyReLU layer, which is common practice in the vision community.

```

443 def forward(self, x):
444     conv0 = self.conv0(x) # 100^3x8
445     conv2 = self.conv2(self.conv1(conv0)) # 50^3x16
446     conv4 = self.conv4(self.conv3(conv2)) # 25^3x32
447
448     x = self.conv6(self.conv5(conv4)) # 13^3x64
449     x = conv4 + self.conv7(x) # 25^3x32
450     x = conv2 + self.conv9(x) # 50^3x16
451     x = self.conv_out(conv0 + self.conv11(x)) # 100^3x128
452     return x

```

Neural Radiance Field. The overall network architecture of our GNF is close to the original NeRF [30] implementation. It mainly consists of 5 ResnetFCBlocks, in which a skip connection is used. The input feature is first projected to 512 with a linear layer and fed into these blocks, and then projected to the output dimension 516 (RGB, density, and Diffusion feature) with a cascading of one ReLU function and one linear layer. We provide the PyTorch-Style pseudo-code for the networks as follows.

```

459 GNF(
460     Linear(in_features=170, out_features=512, bias=True),
461     (0-4): 5 x ResnetFCBlocks(
462         (fc_0): Linear(in_features=512, out_features=512, bias=True)
463         (fc_1): Linear(in_features=512, out_features=512, bias=True)
464         (activation): ReLU()
465     ),
466     ReLU(),
467     Linear(in_features=512, out_features=516, bias=True)
468 )

```

Perceiver Transformer. Our usage of Perceiver Transformer is close to PerAct [3]. We use 6 attention blocks to process the sequence from multi-modalities (3D volume, language token, and robot proprioception) and output a sequence also. The usage of Perceiver Transformer enables us to process the long sequence with computational efficiency, by only utilizing a small set of latents to attend the input. The output sequence is then reshaped back to a voxel to predict the robot action. The Q-function for translation is predicted by a 3D convolutional layer, and for the prediction of openness, collision avoidance, and rotation, we use global max pooling and spatial softmax operation to aggregate 3D volume features and project the resulting feature to the output dimension with a multi-layer perception. We could clarify that the design for the policy module is not our contribution; for more details please refer to PerAct [3] and its official implementation on <https://github.com/peract/peract>.

480 C Demonstration Collection for Real Robot Tasks

For the collection of real robot demonstrations, we utilize the HTC VIVE controller and basestation to track the 6-DOF poses of human hand movements. We then use triad-openvr package¹ to employ SteamVR and accurately map human operations onto the xArm robot, enabling it to interact with objects in the real kitchen. We record the real-time pose of xArm and 640×480 RGB-D observations with the pyrealsense2². Though the image size is different from our simulation setup, we use the same shape of the input voxel, thus ensuring the same algorithm is used across the simulation and the real world. The downscaled images (80×60) are used for neural rendering.

¹https://github.com/TriadSemi/triad_openvr

²<https://pypi.org/project/pyrealsense2/>

D Detailed Data

Besides reporting the final success rates in our main paper, we give the success rates for the best single checkpoint (*i.e.*, evaluating all saved checkpoints and selecting the one with the highest success rates), as shown in Table 7. Under this setting GNFactor outperforms PerAct with a larger margin. However, we do not use the best checkpoint in the main results for fairness.

We also give the detailed number of success in Table 8 for reference in addition to the success rates computed in Table 2.

Table 7: **Multi-task test results on RLBench.** We report the success rates for the best single checkpoint for reference. We could observe GNFactor surpasses PerAct by a large margin.

Method / Task	close jar	open drawer	sweep to dustpan	meat off grill	turn tap	Average
PerAct	22.7 \pm 5.0	62.7 \pm 13.2	0.0 \pm 0.0	46.7 \pm 14.7	36.0 \pm 9.8	
GNFactor	40.0\pm5.7	77.3\pm7.5	40.0\pm11.8	66.7\pm8.2	45.3\pm3.8	
Method / Task	slide block	put in drawer	drag stick	push buttons	stack blocks	
PerAct	22.7\pm6.8	9.3 \pm 5.0	12.0 \pm 6.5	18.7 \pm 6.8	5.3 \pm 1.9	23.6
GNFactor	18.7 \pm 10.5	10.7\pm12.4	73.3\pm13.6	20.0\pm3.3	8.0\pm0.0	40.0

Table 8: **Detailed data for generalization to novel tasks.** We evaluate 20 episodes, each across 3 seeds, for the final checkpoint and report the number of successful trajectories here.

Generalization	PerAct	GNFactor w/o. Diffusion	GNFactor
drag (D)	2, 0, 2	15, 2, 5	18, 5, 5
slide (L)	6, 6, 8	1, 10, 10	6, 5, 4
slide (S)	0, 2, 1	6, 1, 5	0, 3, 1
push (D)	6, 3, 3	4, 4, 5	7, 6, 6
open (N)	6, 2, 7	5, 2, 9	8, 5, 6
turn (N)	4, 5, 2	2, 7, 2	6, 6, 5

E Hyperparameters

We give the hyperparameters used in GNFactor as shown in Table 9. **We are committed to releasing the code for further details.** For the GNF training, we use a ray batch size $b_{\text{ray}} = 512$, corresponding to 512 pixels to reconstruct, and use $\lambda_{\text{feat}} = 0.01$ and $\lambda_{\text{recon}} = 0.01$ to maintain major focus on the action prediction. We uniformly sample 64 points along the ray for the “coarse” network and sample 32 points with depth-guided sampling and 32 points with uniform sampling for the “fine” network.

Table 9: **Hyperparameters** used in GNFactor.

Variable Name	Value
training iteration	100k
image size	$128 \times 128 \times 3$
input voxel size	$100 \times 100 \times 100$
batch size	2
optimizer	LAMB [50]
learning rate	0.0005
ray batch size b_{ray}	512
weight for reconstruction loss λ_{recon}	0.01
weight for embedding loss λ_{feat}	0.01
number of transformer blocks	6
number of sampled points for GNF	64
number of latents in Perceiver Transformer	2048
dimension of Stable Diffusion features	512
dimension of CLIP language features	512
hidden dimension of NeRF blocks	512