G3Flow: Generative 3D Semantic Flow for Pose-aware and Generalizable Object Manipulation

Supplementary Material

A. Simulation Tasks

We provide detailed descriptions of all simulation tasks, as shown in Table 6, totaling 5 tasks.

Task	**Description**
Bottle Adjust	A bottle is placed horizontally on the table. The bottle's design is random and does not repeat in the training and testing sets. When the bottle's head is facing left, pick up the bottle with the right robot arm so that the bottle's head is facing up; otherwise, do the opposite.
Tool Adjust	A tool is placed horizontally on the table. The tool's design is random and does not repeat in the training and testing sets. When the tool's head is facing left, pick up the tool with the right robot arm so that the tool's head is facing up; otherwise, do the opposite.
Diverse Bottles Pick	A random bottle is placed on the left and right sides of the table. The bottles' designs are random and do not repeat in the training and testing sets. Both left and right arms are used to lift the two bottles to a designated location.
Shoe Place	Shoes are placed randomly on the table, with random designs that do not repeat in the training and testing sets. The robotic arm moves the shoes to a blue area in the center of the table, with the shoe head facing the left side of the table.
Dual Shoes Place	One shoe is placed randomly on the left and right sides of the table. The shoes are the same pair with random designs that do not repeat in the training and testing sets. Both left and right arms are used to pick up the shoes and place them in the blue area, with the shoe heads facing the left side of the table.

Table 6. Benchmark Task Descriptions.

B. Implementation Details

This section will provide a detailed introduction to the implementation details of G3Flow as described in the paper, including the setup of the experiments.

B.1. Structure Details

Vision Foundation Model. We utilize the ViT-S/14 variant and transform all images to a resolution of 420 by 420 pixels. These are then fed into the model to obtain feature maps of size 30 by 30, where each pixel has a 384-dimensional feature representation. Subsequently, these features are transformed back to the original image dimensions. The PyTorch implementation is as follow:

```python
def get_dino_feature(image, transform_size=420, model=None):
    img, H, W = transform_np_image_to_torch(image, transform_size=transform_size)
    res = model(img) # torch.Size([1, 384, 30, 30])
    feature = np.array(res.cpu().unsqueeze(0))
    new_order = (0, 1, 3, 4, 2) # torch.Size([1, 30, 30, 384])
    feature = np.transpose(feature, new_order)
    orig_shape_feature = transform_shape(torch.Tensor(np.transpose(feature[0], (0, 3, 1, 2)))
        ↪ , H, W)
    orig_shape_feature_line = orig_shape_feature.reshape(-1, orig_shape_feature.shape[-1])
    return orig_shape_feature, orig_shape_feature_line
```

PCA. We employ Principal Component Analysis (PCA) to reduce the feature dimensionality from 384 to 5.

Perception. For image observations, we uniformly employ a camera setup with a resolution of 320 by 240 pixels and a field of view (fovy) of 45 degrees. We apply Farthest Point Sampling (FPS) to both the feature point cloud and the real observation point cloud, downsampling them to 1024 points. We provide a simple PyTorch implementation of our Feature Pointcloud Encoder as follows:

```
class PointNetFeaturePCDEncoder(nn.Module):
   def __init__(self,
             in_channels,
             out_channels,
             use_layernorm,
             final_norm,
             use_projection,
             **kwargs
             ):
      super().__init__()
      block_channel = [512, 512, 256]

      self.mlp = nn.Sequential(
         nn.Linear(in_channels, block_channel[0]),
         nn.LayerNorm(block_channel[0]) if use_layernorm else nn.Identity(),
         nn.ReLU(),
         nn.Linear(block_channel[0], block_channel[1]),
         nn.LayerNorm(block_channel[1]) if use_layernorm else nn.Identity(),
         nn.ReLU(),
         nn.Linear(block_channel[1], block_channel[2]),
         nn.LayerNorm(block_channel[2]) if use_layernorm else nn.Identity(),
      )

      self.final_projection = nn.Sequential(
         nn.Linear(block_channel[-1], out_channels),
         nn.LayerNorm(out_channels)
      )

      self.use_projection = use_projection

   def forward(self, x):
      x = self.mlp(x)
      x = torch.max(x, 1)[0]
      x = self.final_projection(x)
      return x
```

B.2. Parameter Details

Training Setup. The training setup for the Diffusion Policy based on G3Flow is shown in Tab. 7.

Parameter	Value
horizon	8
n_obs_steps	3
n_action_steps	6
num_inference_steps	10
dataloader.batch_size	256
dataloader.num_workers	8
dataloader.shuffle	True
dataloader.pin_memory	True
dataloader.persistent_workers	False
optimizer._target_	torch.optim.AdamW
optimizer.lr	1.0e-4
optimizer.betas	[0.95, 0.999]
optimizer.eps	1.0e-8
optimizer.weight_decay	1.0e-6
training.lr_scheduler	cosine
training.lr_warmup_steps	500
training.num_epochs	3000
training.gradient_accumulate_every	1

Table 7. Model Training Settings. Hyper-parameter Settings for Training and Deployment of G3Flow-empowered DP.

Baselines Setup. We outline the key training settings for the baseline in Tab. 8.

Parameter	DP	DP3
horizon	8	8
n_obs_steps	3	3
n_action_steps	6	6
num_inference_steps	100	10
dataloader.batch_size	128	256
dataloader.num_workers	0	8
dataloader.shuffle	True	True
dataloader.pin_memory	True	True
dataloader.persistent_workers	False	False
optimizer._target_	torch.optim.AdamW	torch.optim.AdamW
optimizer.lr	1.0e-4	1.0e-4
optimizer.betas	[0.95, 0.999]	[0.95, 0.999]
optimizer.eps	1.0e-8	1.0e-8
optimizer.weight_decay	1.0e-6	1.0e-6
training.lr_scheduler	cosine	cosine
training.lr_warmup_steps	500	500
training.num_epochs	300	3000
training.gradient_accumulate_every	1	1
training.use_ema	True	True

Table 8. Baselines Settings. Hyper-parameter Settings for Training and Deployment of DP and DP3 Algorithms.