Spatial-Temporal Aware Visuomotor Diffusion Policy Learning

Supplementary Material

1. Video

The attached video demonstrates the application of our 4D Diffusion Policy, which is capable of handling complex tasks in real-world through high-level spatial-temporal awareness. We present an expert demonstration and evaluate the performance of DP4 after learning from this demonstration to complete similar tasks in other environments, thereby demonstrating the generalization ability of DP4. The following is a detailed explanation of the tasks that the 4D Diffusion Policy handles in the video:

Short-term Task: "Grasping Bottles" In the first scenario, the 4D Diffusion Policy demonstrates its ability to accurately grasp a bottle by leveraging its sophisticated 3D spatial awareness. This task involves precise control and interaction with a bottle placed within a dynamic environment. The policy processes sensory input and calculates the correct approach trajectory, ensuring the gripper is aligned with the bottle's position and orientation. Through this task, the 4D Diffusion Policy showcases its short-term task efficiency, where it must focus on accurately detecting and physically interacting with the object.

- The policy achieves this by continually adjusting its movements based on real-time feedback and predict the bottle's location, ensuring successful manipulation.
- The success of this task highlights the policy's ability to integrate real-time spatial information into a coherent action plan, achieving reliable object manipulation.

Long-term Task: "Stacking Cups" The second scenario involves a more complex, long-term task—stacking cups. The 4D Diffusion Policy exhibits its ability to grasp a cup and stack it onto another cup, all while understanding the high-level scene structures. This task requires not only precise interaction with the cups but also an awareness of the scene's dynamic configuration. The policy must predict and plan for the interaction between multiple objects (i.e., the cups) and their positions in a broader context.

- The 4D Diffusion Policy relies on a Gaussian world model to map the environment's layout, enabling it to recognize and process object placements and potential changes in the scene's configuration. The Gaussian world model allows for the modeling of uncertainties and complex interactions between objects, enhancing the policy's adaptability and accuracy in high-level scene tasks.
- In this case, the policy anticipates the need to stack the cups while considering factors like object stability and gravity, ensuring that the final configuration is stable.

Fluid Dynamics Task: "Pouring Water" The final task demonstrates the 4D Diffusion Policy's ability to handle complex fluid dynamics. In this scenario, the policy must interact with both a liquid (water) and a container, demonstrating its advanced understanding of fluid behavior and high-level scene dynamics. This task involves pouring water from one container to another while maintaining a steady stream and avoiding spills.

- The policy takes into account the fluid's interaction with the container and its movement through space. By understanding the physics of fluid dynamics, the 4D Diffusion Policy can anticipate the water's behavior and adjust its actions accordingly to successfully complete the task.
- The use of fluid dynamics modeling, along with high-level scene understanding, allows the policy to adjust its grasping strength and pouring angle, ensuring that the water is transferred without splashing or spillage.

2. Additional Implementation Details

The proposed 4D Diffusion Policy (DP4) comprises three key components: Multi-level 3D Spatial Awareness, 4D Spatiotemporal Awareness, and Diffusion-based Decision. Below, we provide a detailed description of the implementation of each component.

Multi-level 3D Spatial Awareness. To effectively capture complex 3D structures in the interactive physical world, this component primarily includes the 3D Local Encoder, 3D Global Encoder, and Generalizable Gaussian Regressor.

- The input to the 3D Local Encoder consists of a colorless cropped point cloud, downsampled from the raw point cloud using Farthest Point Sampling (FPS). We use a sample size of 512 in both simulated and real-world tasks. DP4 encodes the cropped point cloud into a compact 3D local representation using our designed DP4 3D Local Encoder. Below, we present a simple PyTorch implementation of the DP4 3D Local Encoder.

```
class DP4LocalEncoder(nn.Module):
    def __init__(self,
        observation_space: Dict,
        out_channel=256,
        state_mlp_size=(64, 64),
        voxel_mlp_size=(32, 32),
        use_pc_color=False,
        pointnet_type='pointnet'):
        super().__init__()

        self.n_output_channels = out_channel
        self.point_cloud_shape =
            ↪ observation_space['
            ↪ point_cloud']
        self.state_shape = observation_space
```

```
        ↪ ['agent_pos']

    self.use_pc_color = use_pc_color
    self.pointnet_type = pointnet_type

    if pointnet_type == "pointnet":
        self.extractor =
            ↪ PointNetEncoderXYZRGB() if
            ↪  use_pc_color else
            ↪ PointNetEncoderXYZ()

    # State MLP
    self.state_mlp = nn.Sequential(MLP(
        ↪ self.state_shape[0],
        ↪ state_mlp_size[-1],
        ↪ state_mlp_size[:-1], nn.ReLU)
        ↪ )

    self.n_output_channels +=
        ↪ state_mlp_size[-1]
```

- To comprehensively capture global 3D information, the 3D Global Encoder receives input in the form of the global voxel derived from the full point cloud and the robot's pose. DP4 processes this voxel to produce a compact global 3D representation through our custom-designed DP4 3D Global Encoder. A simple PyTorch implementation of the DP4 3D Global Encoder is presented:

```
class DP4GlobalEncoder(nn.Module):
    def __init__(self, in_channels=10,
        ↪ out_channels=64, norm_act=
        ↪ InPlaceABN):
        super().__init__()
        CHANNELS = [8, 16, 32, 64]
        # Define the convolution layers
        self.conv0 = ConvBnReLU3D(
            ↪ in_channels, CHANNELS[0],
            ↪ norm_act=norm_act)
        self.conv1 = ConvBnReLU3D(CHANNELS
            ↪ [0], CHANNELS[1], stride=2,
            ↪ norm_act=norm_act)
        self.conv2 = ConvBnReLU3D(CHANNELS
            ↪ [1], CHANNELS[1], norm_act=
            ↪ norm_act)
        self.conv3 = ConvBnReLU3D(CHANNELS
            ↪ [1], CHANNELS[2], stride=2,
            ↪ norm_act=norm_act)
        self.conv4 = ConvBnReLU3D(CHANNELS
            ↪ [2], CHANNELS[2], norm_act=
            ↪ norm_act)
        self.conv5 = ConvBnReLU3D(CHANNELS
            ↪ [2], CHANNELS[3], stride=2,
            ↪ norm_act=norm_act)
        self.conv6 = ConvBnReLU3D(CHANNELS
            ↪ [3], CHANNELS[3], norm_act=
            ↪ norm_act)
        # Transpose convolutions for
            ↪ upsampling
        self.conv7 = nn.Sequential(
            nn.ConvTranspose3d(CHANNELS[3],
                ↪ CHANNELS[2], 3, padding=1,
                ↪  stride=2, bias=False),
            norm_act(CHANNELS[2]))
        self.conv9 = nn.Sequential(
            nn.ConvTranspose3d(CHANNELS[2],
                ↪ CHANNELS[1], 3, padding=1,
                ↪  output_padding=1, stride
                ↪ =2, bias=False),
            norm_act(CHANNELS[1])
        )
        self.conv11 = nn.Sequential(
            nn.ConvTranspose3d(CHANNELS[1],
                ↪ CHANNELS[0], 3, padding=1,
                ↪  output_padding=1, stride
                ↪ =2, bias=False),
            norm_act(CHANNELS[0]) )

        # Final convolution for output
        self.conv_out = nn.Conv3d(CHANNELS
            ↪ [0], out_channels, 1, stride
            ↪ =1, padding=0, bias=True)
```

- To improve the representation with global structural and texture data, we employ a Generalizable Gaussian Regressor. This regressor uses the deep volume as a scene representation to derive the Gaussian world model via Gaussian Splatting. The model is trained by rendering both RGB and depth images based on the generated Gaussian world model. Below, we present a simple PyTorch implementation of Generalizable Gaussian Regressor.

```
class GeneralizableGSRegressor(nn.Module):
    def __init__(self, cfg, with_gs_render=
        ↪ True):
        super().__init__()
        self.cfg = cfg
        self.with_gs_render = with_gs_render

        self.use_xyz = cfg.use_xyz
        self.d_in = 3 if self.use_xyz else 1
        self.use_code = cfg.use_code

        if self.use_code:
            self.code = PositionalEncoding.
                ↪ from_conf(cfg["code"],
                ↪ d_in=self.d_in)
            self.d_in = self.code.d_out

        self.d_latent = cfg.d_latent
        self.d_lang = cfg.d_lang
        self.d_out = sum(self.
            ↪ _get_splits_and_inits(cfg)
            ↪ [0])

        self.encoder = ResnetFC(
            d_in=self.d_in, d_latent=self.
```

```
            ↪ d_latent, d_lang=self.
            ↪ d_lang, d_out=self.d_out,
        d_hidden=cfg.mlp.d_hidden,
            ↪ n_blocks=cfg.mlp.n_blocks,
            ↪  combine_layer=cfg.mlp.
            ↪ combine_layer,
        beta=cfg.mlp.beta, use_spade=cfg.
            ↪ mlp.use_spade,
    )
    self.gs_parm_regresser =
        ↪ GSPointCloudRegresser(cfg,
        ↪ self._get_splits_and_inits(
        ↪ cfg)[0])

    self.scaling_activation = torch.exp
    self.opacity_activation = torch.
        ↪ sigmoid
    self.rotation_activation = torch.nn.
        ↪ functional.normalize
```

4D Spatiotemporal Awareness. To highlight 4D spatiotemporal awareness within the diffusion policy, our DP4 incorporates dynamics into the Gaussian world model. Expanding on this model, DP4 utilizes a deformable MLP to track changes in Gaussian parameters over time. Below, we present a simple PyTorch implementation of the DP4 deformable MLP.

```
class DeformableNet(nn.Module):
    def __init__(self, d_in, d_out=4,
        ↪ n_blocks=5, d_latent=0, d_hidden
        ↪ =128, beta=0.0, combine_layer
        ↪ =1000, use_spade=False):
        super().__init__()

        self.lin_in = nn.Linear(d_in,
            ↪ d_hidden) if d_in > 0 else None
        self.lin_out = nn.Linear(d_hidden,
            ↪ d_out)
        self.blocks = nn.ModuleList([
            ↪ ResnetBlockFC(d_hidden, beta=
            ↪ beta) for _ in range(n_blocks)
            ↪ ])

        self.d_latent = d_latent
        self.use_spade = use_spade
        self.combine_layer = combine_layer

        # Initialize the layers
        if self.lin_in:
            nn.init.kaiming_normal_(self.
                ↪ lin_in.weight, a=0, mode="
                ↪ fan_in")
            nn.init.constant_(self.lin_in.bias
                ↪ , 0.0)
        nn.init.kaiming_normal_(self.lin_out.
            ↪ weight, a=0, mode="fan_in")
        nn.init.constant_(self.lin_out.bias,
            ↪ 0.0)

        # Handle latent layers
        self.lin_z = nn.ModuleList([nn.Linear
            ↪ (d_latent, d_hidden) for _ in
            ↪ range(min(combine_layer,
            ↪ n_blocks))])
        self.scale_z = nn.ModuleList([nn.
            ↪ Linear(d_latent, d_hidden) for
            ↪ _ in range(min(combine_layer,
            ↪ n_blocks))]) if use_spade

        # Activation function
        self.activation = nn.Softplus(beta=
            ↪ beta) if beta > 0 else nn.ReLU
            ↪ ()
```

Diffusion-based Decision. The Diffusion-based Decision backbone is a convolutional network-based diffusion policy that converts random Gaussian noise into a coherent sequence of actions. For implementation, we employ the official PyTorch framework provided by 3D Diffusion Policy (DP3) [45]. In practice, the model is designed to predict a series of H actions based on N_{obs} observed timesteps, but it executes only the last N_{act} actions during inference. We set $H = 4$, $N_{obs} = 2$, and $N_{act} = 3$ for the diffusion-based baselines, which are consistent with the original DP3 configuration. We independently scale the minimum and maximum values of each action and observation dimension to $[-1, 1]$. Normalizing actions to $[-1, 1]$ is essential for DDPM and DDIM predictions, as these models clip the predictions to $[-1, 1]$ for stability.

3. Details of Simulation Tasks

For the simulation experiments, we selected tasks from various domains, covering a broad spectrum of robotic skills. These tasks include both challenging scenarios, such as bimanual manipulation, deformable object manipulation, and articulated object manipulation, and simpler tasks like parallel gripper manipulation. The tasks consist of Adroit, DexArt, and RLBench tasks. This section provides a detailed overview of these simulation tasks.

Adroit. This task employs a simulated version of the highly dexterous manipulator ADROIT, a 24-DoF anthropomorphic platform designed to tackle challenges in dynamic and precise manipulation. The first, middle, and ring fingers have 4 DoF, while the little finger and thumb have 5 DoF, and the wrist has 2 DoF. Each DoF is controlled by position control and is equipped with a joint angle sensor. The experimental setup of ADROIT utilizes the MuJoCo physics simulator. MuJoCo's stable contact dynamics make it well-suited for contact-rich hand manipulation tasks. The kinematics, dynamics, and sensing details of the physical hardware were meticulously modeled to enhance physical realism. In addition to dry friction in the joints, all hand-object

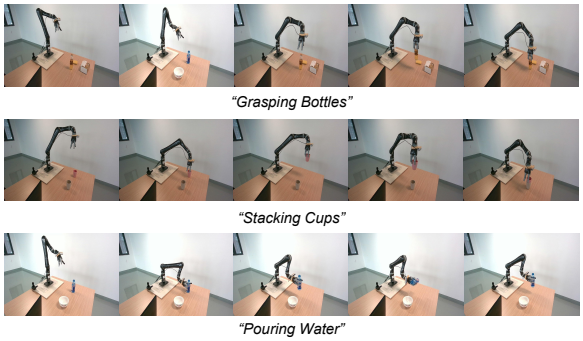

Figure 8. **Keyframes for real robot tasks.** We give the keyframes used in our 3 real robot tasks.

interactions involve planar friction. Object-fingertip interactions support torsion and rolling friction.

DexArt. This simulation task includes four dexterous manipulation tasks: Faucet, Bucket, Laptop, and Toilet, each involving a mix of seen and unseen objects. The Faucet task evaluates the coordination between the dexterous hand and arm motions. Although a 2-jaw parallel gripper could perform this task, it heavily depends on precise arm motion due to its low DoF end-effector. In the Bucket task, the robot must lift a bucket. To ensure stability, it should extend its hand under the handle and grip it to achieve form closure. In the Laptop task, the robot grasps the middle of the screen and opens the laptop lid, a task well-suited for dexterous hands. A parallel gripper can achieve this by precisely gripping the lid between its jaws. The Toilet task is similar to the Laptop task, but requires the robot to open a larger, more irregular toilet lid. This task is more challenging due to the lid's irregular and diverse geometry.

RLBench. We selected 10 tasks from RLBench [17], each with a minimum of two variations. These variations include random changes in color, size, count, placement, and object category, resulting in 166 unique combinations. The set of colors includes 20 options: red, maroon, lime, green, blue, navy, yellow, cyan, magenta, silver, gray, orange, olive, purple, teal, azure, violet, rose, black, and white. The size variations are categorized as short and tall, while the count options are 1, 2, and 3. The placement and object categories differ depending on the task. For instance, in the "open drawer" task, objects can be placed in one of three locations: top, middle, or bottom. Additionally, objects are randomly positioned on the tabletop within a constrained pose range. We also created six extra tasks where the scene is modified from the original training environment to assess the system's generalization capability.

4. Details of Real-Robot Tasks

In the experiments, we conduct three primary tasks, along with three additional ones that include distracting objects. The "*grasping bottles*" task requires the agent to pick up a bottle from a table, a challenge due to the precise coordination and 3D spatial awareness needed. The "*pouring water*" task involves transferring water from a bottle to a bowl, requiring the agent to manage the complex 4D dynamics of the water in the physical world. The "*stacking cups*" task tasks the agent with finding a randomly placed cup on the table and understanding its 3D spatial structure. Among these tasks, the "*pouring water*" task is considered the most difficult, as it demands accurate placement and rotation of the gripper in response to the positioning. The keyframes for the real robot tasks are shown in Figure 8.