Diffusion Policies with Multimodal Conditioning

Maurice Rahme

March 18, 2025

1 Introduction

This document discusses a diffusion policy architecture that integrates multimodal conditioning. The model transforms a noisy action sequence into a clean output (e.g., end-effector poses or motor commands).

Conditioning Signal Processing: Visual inputs (e.g., images) and non-visual inputs (e.g., state or pose) are processed via dedicated encoders. Their outputs, along with a time embedding from a diffusion step encoder, are concatenated into a conditioning vector that is injected into the U-Net via FiLM modules.

Conditioning

```
    +------------------------+     +------------------------+
    | Visual Input (Image)   |     | Non-Visual Input       |
    +------------------------+     | (State, Pose, etc.)    |


    +------------------------+     +------------------------+
    | Visual Encoder (CNN)   |     | Non-Visual Encoder     |
    +------------------------+     | (MLP, etc.)            |


    +------------------------+     +------------------------+
    | Visual Features        |     | Non-Visual Features    |
    +------------------------+     +------------------------+
             \____________________________/


              +-------------------------+
              | Global Conditioning     | <-- Concatenation of visual and non-visual features
              +-------------------------+


              +-------------------------+
              | Time Embedding          | <-- Diffusion Step Encoder
              +-------------------------+


              +-------------------------+
              | Final Conditioning      | <-- Global + Time embedding
              +-------------------------+


         (Injected into U-Net via FiLM modules)
```

1.1 U-Net

Here's the outline for a 2-stage U-Net.

```
ENCODER 1 --------------[Skip 1]---------------> DECODER 2
    |                                               ↑
    ↓                                               |
ENCODER 2 --------------[Skip 2]---------------> DECODER 1
                \                      /
                 \     BOTTLENECK     /
                  \__________________/
```

Encoders

The encoder blocks extract hierarchical features from the noised action input. They progressively reduce the resolution, capturing more abstract representations as the network goes deeper. This dimensionality reduction helps the network focus on global structures while filtering out noise.

Residual Connections

Residual or skip connections ensure that fine-grained details, which might be lost during the downsampling in the encoders, are directly transferred to the decoders. This direct path helps in reconstructing the output with greater fidelity by reintroducing local details.

Bottleneck

The bottleneck consolidates global context and deep features extracted from the input. By processing the compressed representation further, the bottleneck forms a context-aware distillation that guides the upsampling process in the decoders. This is critical for ensuring that the denoised output maintains consistency with the overall structure of the original action.

The bottleneck is typically formed by applying one or more conditional residual blocks (similar to the ones described above) without additional downsampling. This deep processing helps in integrating information from the entire input before reconstruction.

Decoders

The decoder blocks upsample the processed representation back to the original resolution. They integrate the high-level context from the bottleneck with the detailed information provided by the skip connections. This combination enables the network to generate a refined, denoised action that accurately reflects both global and local features.

1.1.1 Integration of Conditioning via FiLM

Each encoder and decoder block internally uses conditional residual blocks to process the noised input. In these blocks, the FiLM module injects the conditioning signal (derived from visual inputs, non-visual inputs, and a time embedding) by computing per-channel scale and bias parameters. These parameters modulate the convolutional feature maps without directly mixing the conditioning data into the convolutional operations. This approach ensures that the external context influences the learning process without compromising the spatial and temporal structure of the noised action.

2 The Diffusion Process

2.1 Forward Process

Starting with a clean target x_0, noise is added progressively:

$$x_t = \sqrt{\alpha_t}\, x_{t-1} + \sqrt{1-\alpha_t}\, \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

This process corrupts the action until it is nearly Gaussian.

2.2 Reverse Process

The reverse process is modeled as:

$$p_\theta(x_{t-1} \mid x_t, c) = \mathcal{N}\Big(x_{t-1}; \mu_\theta(x_t, t, c),\, \Sigma_\theta(x_t, t, c)\Big),$$

where c is the conditioning signal (global conditioning $\oplus$ time embedding). The network learns to predict the noise, thereby refining the action.

2.3 Extensions

Goal conditioning (e.g goal pose for PushT) is critical for the policy to learn multi-goal scenarios.

Self-attention can improve temporal consistency for highly complex and/or long-horizon tasks.

3 Pseudocode

```
Algorithm DiffusionPolicy:
Input:
    - NoisedActionSequence X
    - Diffusion Time Step t
    - VisualInput (e.g., Image)
    - NonVisualInput (e.g., State, Pose)

Process:
    // Process conditioning inputs:
    1. VisualFeatures  <- VisualEncoder(VisualInput)
    2. NonVisualFeatures <- NonVisualEncoder(NonVisualInput)
    3. GlobalCondition <- Concatenate(VisualFeatures, NonVisualFeatures)
    4. TimeEmbed <- DiffusionStepEncoder(t)
    5. Conditioning <- Concatenate(TimeEmbed, GlobalCondition)

    // Process the noised action:
    6. PredictedNoise <- U-Net(
           InputSignal = X,
           Conditioning = Conditioning
       )

Output:
    - DenoisedAction or PredictedNoise


Module: U-Net (with Conditional Residual Blocks)
Input:
    - InputSignal: NoisedActionSequence X
    - Conditioning: Combined conditioning signal

Process:
    a. Apply Initial Convolution to project InputSignal.
    b. For each Downsampling Stage:
         i. Apply a ConditionalResidualBlock with FiLM modulation
            (using Conditioning).
         ii. Save skip connection.
         iii. Downsample via Conv1D.
    c. Apply Bottleneck Residual Blocks.
    d. For each Upsampling Stage:
         i. Upsample the current representation.
         ii. Concatenate with corresponding Skip Connection.
         iii. Apply a ConditionalResidualBlock with FiLM modulation.
    e. Apply Final Convolution to produce OutputSignal.

Output:
    - OutputSignal (Predicted Noise / Denoised Action)
```

4 Data Loader and Data Format

Data organization is critical for success. My training was failing until I wrote unit tests that allowed me to identify issues in my PolicyDataset code. The data is organized into episodes and is processed using a sliding window approach with appropriate padding to ensure fixed-length sequences.

Dataset Construction

The `PolicyDataset` The class loads training samples and applies a temporal organization to them. For each episode, samples are grouped by an `episode_index` and sorted by `frame_index`. The accumulated data includes:

- **Images:** stored as a tensor of shape $(N, 3, 96, 96)$.
- **Agent Positions:** extracted from the observation state and stored as a tensor of shape $(N, 2)$.
- **Actions:** stored as a tensor of shape $(N, 2)$.
- **Future:** extend to include goal conditioning via pose.

Sliding Window Sampling

A sliding window approach is employed to extract fixed-length sequences:

- **Window Parameters:**
  - `pred_horizon`: Total length of the sequence to extract.
  - `obs_horizon`: Number of timesteps used as the observation window.
  - `action_horizon`: Horizon for action data, used to determine padding at the sequence end.
- **Index Computation:** The helper function `create_sample_indices` computes indices in the form:

  $$[\texttt{buffer\_start\_idx}, \texttt{buffer\_end\_idx}, \texttt{sample\_start\_idx}, \texttt{sample\_end\_idx}]$$

  These indices indicate the segment of the data to be extracted and specify the insertion positions within a fixed-length sample. Padding is applied by repeating the edge values when necessary.

Normalization and Data Formatting

Before a sample is returned:

- **Normalization:**
  - Agent positions and actions are normalized using a dedicated normalization strategy (scaling agent positions to $[-1, 1]$, and applying a similar scheme to actions).
  - Images are preprocessed and normalized to $[0, 1]$ as part of an image transformation pipeline.
- **Data Output:** Each sample is a dictionary containing:
  - `'image'`: A sequence of images with shape $(\texttt{obs\_horizon}, 3, 96, 96)$.
  - `'agent_pos'`: A flattened vector of agent positions of shape $(\texttt{obs\_horizon} \times 2,)$.
  - `'action'`: A sequence of actions with shape $(\texttt{pred\_horizon}, 2)$.

Processing Pipeline Overview

```
1. Group samples by episode and sort by frame index.
2. Accumulate data into contiguous tensors for images, agent positions, and actions.
3. Compute episode boundaries and generate sliding window indices.
4. Extract fixed-length sequences using sample_sequence(), applying edge padding as needed.
5. Normalize agent positions and actions using a normalization routine.
6. Return the final sample:
    - Observations: First obs_horizon timesteps for images and agent positions.
    - Actions: Full pred_horizon timesteps.
```

This structured approach ensures that the diffusion policy receives temporally consistent and properly formatted multimodal data, facilitating effective training and inference.