Supplementary: Modeling Multiple Normal Action Representations for Error Detection in Procedural Tasks

Overview

In this supplementary material, we provide the following sections:

- Sec. A: More details about our proposed Adaptive Multiple Normal Action Representation (AMNAR) framework.
- Sec. B: More details about the experiment setups.
- Sec. C: More visualizations for further demonstration on the effectiveness of the proposed AMNAR.

A. More Details about AMNAR

A.1. Potential Action Prediction Block

The Potential Action Prediction Block (PAPB) is a key component designed to predict all potential next actions based on the task graph G and the executed action sequence s. The variable-definition reference table and pseudocode for PAPB are shown in Tab. S1 and Algorithm 1, respectively.

Adjacency List Construction. PAPB begins by converting the task graph G into an adjacency list A, where each node in the graph links to its direct successors.

Longest Subsequence Identification. PAPB employs dynamic programming to find the longest subsequence s^* in s that adheres to the relationships defined by G. The algorithm maintains two tables: subseq$[i]$, which stores the longest non-branching subsequence ending at index i, and dp$[i]$, which stores the subseq$[i]$. A **non-branching subsequence** is defined as a sequence of nodes that form a continuous path in the task graph G, where all nodes are connected sequentially without any splits or branches (e.g., [0, 1, 2] in Fig. S1).

For each action y_i in s, the algorithm iterates over all previous actions y_j (where $j < i$) and checks whether y_i and y_j are connected in the task graph G. If this condition is met, dp$[i]$ and subseq$[i]$ are updated as follows:

$$\text{dp}[i] = \max(\text{dp}[i], \text{dp}[j] + 1), \tag{1}$$

$$\text{subseq}[i] = \begin{cases} \text{subseq}[j] \cup \{y_i\}, & \text{if dp}[j] + 1 > \text{dp}[i], \\ \text{subseq}[i] \cup (\text{subseq}[j] \cup \{y_i\}), & \text{if dp}[j] + 1 = \text{dp}[i]. \end{cases} \tag{2}$$

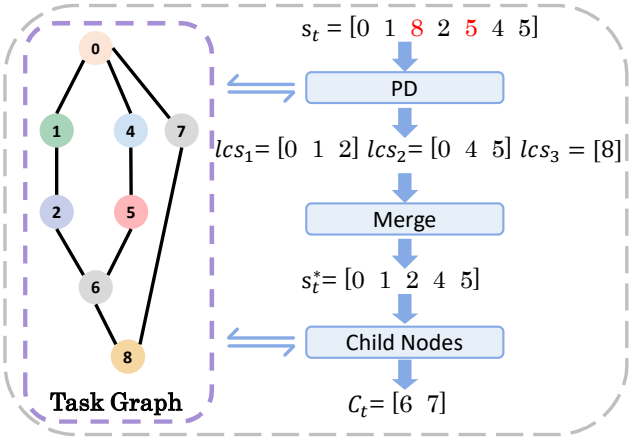

Figure S1. The Potential Action Prediction Block (PAPB) derives the longest matching subsequence from the executed sequence using the task graph. This subsequence is then used to identify all reachable nodes, representing valid next actions. This figure is reproduced from the main text for reference.

After processing s, the algorithm identifies the maximum value in dp, locating the index k with the longest non-branching subsequence L.

Merging Connected Nodes. While the longest subsequence identified in dynamic programming represents a non-branching path (e.g., [0, 1, 2] in Fig. S1), it may not capture all executed actions in scenarios where multiple branches exist in the task graph. To address this, PAPB iteratively examines each subsequence. For each subsequence, if any of its nodes matches a node in L, the subsequence is considered connected to L, and its nodes are merged into L. This merging process ensures that L includes all nodes relevant to the executed actions, resulting in the complete merged sequence s^*, which accurately reflects all executed actions within the task graph.

Next Action Prioritization. Based on s^*, PAPB computes the set of potential next actions PA as:

$$PA = (\bigcup_{a \in s^*} A[a]) \setminus s^*. \tag{3}$$

In this formula, $A[a]$ represents the set of direct successors of node a in the task graph G, as derived from the adjacency

list. By iterating over all nodes a in the longest merged subsequence s^*, the union $\bigcup_{a \in s^*} A[a]$ aggregates the successors of all nodes in s^*. The subtraction $\setminus s^*$ ensures that only actions not already included in s^* are retained in PA. This guarantees that PA contains all valid next actions that can logically follow the executed actions, without duplication.

PAPB efficiently combines dynamic programming and graph traversal to provide actionable insights from s and G. For detailed implementation, refer to Algorithm 1.

Algorithm 1 Potential Action Prediction Block (PAPB)

Input: Task graph G, Executed action sequence s
Output: Prioritized list of next actions PA
 # Build Adjacency Lists:
Initialize $A[u] = \emptyset$ for all $u \in G$
for each edge (u, v) in G **do**
  $A[u] \leftarrow A[u] \cup \{v\}$
end for
 # DP Process:
Initialize $\text{dp}[i] \leftarrow 1$ and $\text{subseq}[i] \leftarrow \{y_i\}$ for all i
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to $i-1$ **do**
    if $y_i \in A[y_j]$ **or** $y_j \in A[y_i]$ **then**
      if $\text{dp}[j] + 1 > \text{dp}[i]$ **then**
        $\text{dp}[i] \leftarrow \text{dp}[j] + 1$
        $\text{subseq}[i] \leftarrow \text{subseq}[j] \cup \{y_i\}$
      else if $\text{dp}[j] + 1 == \text{dp}[i]$ **then**
        $\text{subseq}[i] \leftarrow \text{subseq}[i] \cup \text{subseq}[j] \cup \{y_i\}$
      end if
    end if
  end for
end for
 # Collect Max-Length Subsequences:
$k \leftarrow \max(\text{dp}[1], \text{dp}[2], \ldots, \text{dp}[n])$
$L \leftarrow \bigcup\{\text{subseq}[i] \mid \text{dp}[i] = k\}$
 # Merge Connected Nodes in L:
Initialize $s^* \leftarrow L$
for each $node$ in L **do**
  for each $neighbor \in A[node]$ **do**
    if $neighbor \in L$ **then**
      $s^* \leftarrow s^* \cup \{neighbor\}$
    end if
  end for
end for
 # Collect Potential Next Actions:
$PA \leftarrow (\bigcup_{a \in s^*} A[a]) \setminus s^*$
Return PA

A.2. Representation Reconstruction Block

The Representation Reconstruction Block (RRB) is designed to reconstruct multiple normal action representations at time t using the frame-wise features of executed actions and the embedding of the t-th action. The RRB consists of two key components: a dilated convolutional layer and a local cross-attention module, as illustrated in Fig. S2.

Table S1. Variable Definitions of PAPB

Variable	Definition
G	Task graph
s	Executed action sequence
s^*	The longest matching subsequence
A	Adjacency list of G
$A[a]$	The set of direct successors of node a
$\text{subseq}[i]$	Longest non-branching subsequence ending at index i
$\text{dp}[i]$	Length of $\text{subseq}[i]$
k	Index with the maximum $\text{dp}[k]$
L	Longest non-branching subsequence
PA	Final potential next actions

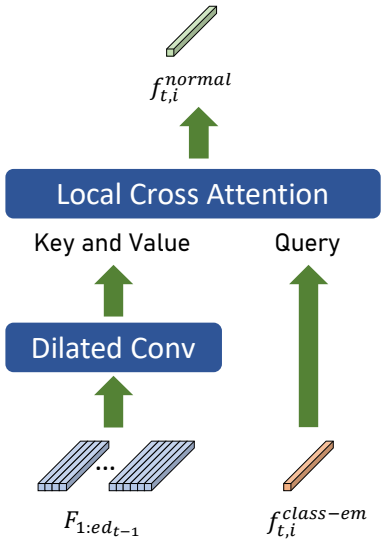

Figure S2. Architecture of the Representation Reconstruction Block (RRB). The RRB reconstructs the i-th normal action representation $f_{t,i}^{\text{normal}}$ for time t by combining the frame-wise refined features $F_{1:ed_{t-1}}$ (key and value) and the action class embedding $f_{t,i}^{\text{class-emb}}$ (query).

To ensure temporal causality, all modules within the RRB are implemented in a causal manner. Specifically, when reconstructing the normal action representations at time t, the frame-wise features corresponding to time t and any future frames are not accessible, thereby adhering to the sequential nature of the task.

Dilated Convolutional Layer. The dilated convolutional layer employs a kernel size of 3 and consists of 5 layers. The dilation rate of the first layer is set to 1, while the subsequent layers follow an exponential growth pattern. Specifi-

cally, the dilation rate d_i for the i-th layer is defined as:

$$d_i = 3^i. \tag{4}$$

This design allows the receptive field to expand exponentially with depth.

Local Cross Attention. The local cross attention module consists of a single attention layer with a local window length of 32 and 2 attention heads. Depthwise convolutions project the query, key, and value features, with causal padding ensuring only past and current time steps are accessible, preserving temporal causality.

Action Class Embedding. As mentioned in Section 3.3 of the main text, $F_{1:ed_{t-1}}$ represents the frame-wise refined visual features extracted from the Action Segmentation Model up to frame ed_{t-1}. The $f^{\text{class-emb}}(y)$ represents the class embedding for action class y, computed as the mean feature of all action samples belonging to this class. Formally, it is defined as:

$$f^{\text{class-emb}}(y) = \frac{\sum_{t \in \mathcal{I}_y} f_t^{\text{action}}}{N_y}, \tag{5}$$

where $\mathcal{I}_y$ is the set of indices for samples belonging to class y, $N_y = |\mathcal{I}_y|$ is the total number of samples in this class, and f_t^{action} represents the feature of the t-th action sample. This class embedding serves as a representative feature for action class y.

The $f_{t,i}^{\text{class-emb}}$ represents the class embedding for the i-th potential action class corresponding to the t-th action. It is used as the query input in the Local Cross Attention module (see Fig. S2), where it interacts with the key and value features derived from the frame-wise refined features $F_{1:ed_{t-1}}$ after processing through the dilated convolution layer.

B. More Experimental Setups

In this section, we provide comprehensive details about the experimental setup to complement the descriptions in the main text. Specifically, we elaborate on the preprocessing and usage of the HoloAssist[3] datasets, frequency analysis of multiple valid next actions, as well as the experimental environment and hyperparameter settings.

B.1. HoloAssist Dataset

Since the official release of the HoloAssist dataset lacks a designated test set, we train our AMNAR and EgoPED [1] frameworks on the training set, compute thresholds using the training set, and evaluate performance on the validation set. The tasks used for training and validation, along with their respective durations, are summarized in Table S2. To train the Action Segmentation Model (ASM), we utilize the fine-grained action annotations, specifically either verb or noun labels, as segment labels.

Table S2. Duration of Training and Validation Sets for HoloAssist Tasks (in minutes)

Task Name	Train (min)	Val (min)
atv	84.63	12.37
circuitbreaker	45.30	8.62
coffee	137.17	16.38
computer	226.43	38.95
dslr	289.22	38.15
gladom_assemble	320.95	50.60
gladom_disassemble	211.03	29.02
gopro	561.58	78.18
knarrevik_assemble	843.08	114.08
knarrevik_disassemble	465.00	71.63
marius_assemble	357.58	52.28
marius_disassemble	208.38	36.83
navvis	122.65	21.25
nespresso	225.47	28.47
printer_big	162.15	26.87
printer_small	295.05	42.32
rashult_assemble	942.42	128.90
rashult_disassemble	545.65	68.47
switch	469.07	70.82

The HoloAssist training set includes both normal and erroneous actions. To ensure accurate learning of normal action representations, we train AMNAR exclusively on normal actions, excluding erroneous ones during training. For HoloAssist experiments, due to the absence of an official test set, we follow a standard split by training on the provided training set (approximately 166 hours of video from 350 instructor-performer pairs) and evaluating on the validation set. Additionally, we exclude the "Belt" task from final evaluations, as it contains only one error-free sample, which could skew performance metrics.

Moreover, some action classes appear only in the validation set and are absent from the training set. To maintain consistency during inference, we classify these unseen classes as background actions. For task graph construction, since HoloAssist lacks predefined task graphs, we generate them by analyzing all training sequences.

We also introduce a random baseline for HoloAssist experiments. This baseline employs the same ASM trained with the aforementioned strategy and, during inference, randomly classifies each action segment as either normal or erroneous.

B.2. CaptainCook4D Dataset

The CaptainCook4D dataset [2] is a large-scale egocentric 4D dataset designed for understanding errors in procedural cooking activities. It comprises 384 recordings (94.5 hours) of individuals performing 24 different recipes in real

kitchen environments. The dataset includes videos of participants correctly following recipe instructions as well as instances where they deviate and introduce errors. It provides 5.3K step annotations and 10K fine-grained action annotations, with errors categorized into seven distinct types. Data modalities include RGB video, depth, 3D hand joint tracking, and IMU data, captured using a head-mounted GoPro and HoloLens2.

For our experiments, since CaptainCook4D lacks predefined task graphs, we generate them by analyzing all training sequences, similar to the approach used for HoloAssist. To focus on execution-related errors, we exclude the "Missing Step" and "Ordering" error types during evaluation, as these sequence-level anomalies are beyond the primary scope of AMNAR.

B.3. Task Graph Generation for Procedural Task Modeling

To better model procedural tasks in both HoloAssist and CaptainCook4D, we derive task graphs from action sequences, as these datasets do not provide predefined graphs. Each task graph is represented as a Directed Acyclic Graph (DAG) that captures valid action transitions based on observed sequences.

The graph construction consists of three steps: 1. **Extract Action Sequences**: Identify non-background action sequences from the recordings and insert a start state (e.g., background) at the beginning of each sequence. 2. **Compute Transition Weights**: Measure the co-occurrence frequency of each action pair across all sequences to form a weighted transition matrix. 3. **Build a Maximum-Weight DAG**: Use a greedy algorithm to select the highest-weight edges while disallowing cycles, preserving only acyclic paths.

This procedure ensures that frequent, logically coherent transitions are included in the final task graph, providing a reliable structure for analyzing procedural tasks. For the complete pseudocode of this task graph generation process, please refer to Algorithm 2.

This approach ensures the task graph reflects frequent, logical action transitions while maintaining an acyclic structure, suitable for procedural task analysis.

Algorithm 2 Task Graph Generation

Input: Action sequences S
Output: Task graph G as a list of edges
 # Compute Transition Weights:
Initialize $T[(u, v)] \leftarrow 0$ for all possible (u, v)
for each $seq \in S$ **do**
  for $i \leftarrow 0$ to len$(seq) - 2$ **do**
    for $j \leftarrow i + 1$ to len$(seq) - 1$ **do**
      $T[(\text{seq}[i], \text{seq}[j])] \leftarrow T[(\text{seq}[i], \text{seq}[j])] + 1$
    end for
  end for
end for
 # Sort Transitions by Weight:
$P \leftarrow \text{sort}(T.\text{items}(), \text{key} = \text{weight}, \text{descending})$
 # Build Maximum-Weight DAG:
Initialize $G \leftarrow \emptyset$
for (u, v) in P **do**
  if adding (u, v) to G keeps G acyclic **then**
    $G \leftarrow G \cup \{(u, v)\}$
  end if
end for
Return G

B.4. Frequency Analysis of Multiple Valid Next Actions

In Section 4.4 of the main text, we compare average improvements across tasks, noting that the *coffee* task has the highest occurrence of multiple valid next actions. This observation stems from a frequency analysis of multiple valid next actions using the following metrics: **non-deterministic action ratio**, **average number of valid next actions** and **average maximum transfer probability**.

A **non-deterministic action** is defined as an action whose preceding action has more than one potential next action. As illustrated in Figure S1, consider action a_1, which follows action a_0. Since action a_0 has multiple potential next actions (actions a_1, a_4, a_7), action a_1 is considered a non-deterministic action (as are a_4 and a_7).

The **non-deterministic action ratio** refers to the proportion of non-deterministic actions among all actions within a task. A higher ratio indicates a greater prevalence of multiple valid next actions, contributing to task complexity. As shown in Table S3, the tasks *tea*, *coffee*, and *oatmeal* have notably high non-deterministic action ratios of 75.00%, 70.59%, and 69.23%, respectively.

The **average number of valid next actions** represents the mean count of potential valid next actions for each action in a task. For instance, if action a_0 has potential next actions a_1, a_2, and a_3, the number of valid next actions is 3. A higher average indicates that actions generally have more possible subsequent actions, increasing the task's complexity. In terms of this metric, the *coffee* task stands out with a value of 2.82, higher than those of other tasks.

The **average maximum transfer probability** is the average of the highest probabilities with which actions transition to their next actions. For example, if action a_0 transitions to a_1, a_2, and a_3 with probabilities of 20.00%, 25.00%, and 55.00%, the maximum transfer probability for a_0 is 55.00%. A lower average maximum transfer probability indicates greater uncertainty in transitioning to a specific next action, reflecting higher diversity in valid next steps. As shown in Table S3, the *coffee* and *oatmeal* tasks have

lower average maximum transfer probabilities of 67.27% and 67.09%, respectively.

The *coffee* task stands out across all three metrics, indicating a high frequency of multiple valid next actions. This complexity makes it the most suitable task for demonstrating the effectiveness of our Adaptive Multiple Normal Action Representation (AMNAR) framework. Consistent with our frequency analysis, AMNAR achieves the most substantial improvement in error detection accuracy for the *coffee* task, as evidenced in Table 1 of the main text. This correlation underscores the advantage of AMNAR in handling tasks with diverse and multiple valid action sequences.

Table S3. Metrics for Task Transition Matrices Across Five Tasks. Higher non-deterministic ratios (↑) indicate greater complexity due to multiple valid next actions. Higher average numbers of valid next actions (↑) suggest increased complexity of a task. Lower average maximum transfer probabilities (↓) indicate greater uncertainty in action transitions.

Metric	Tea	Coffee	Pinwheels	Oatmeal	Quesadilla
Non-deterministic Ratio (%) ↑	75.00	70.59	26.67	69.23	40.00
Avg. Valid Next Actions ↑	1.75	2.82	1.13	1.85	1.20
Avg. Max Transfer Prob. (%) ↓	74.02	67.27	88.15	67.09	75.00

B.5. EDA of non-deterministic actions

In Sec. 4.4 of the main paper, we evaluate the Error Detection Accuracy (EDA) of non-deterministic actions. This experiment measures the average frame-wise accuracy of non-deterministic actions in error detection.

B.6. Experimental Environment and Hyperparameters

All experiments are conducted on an Nvidia Tesla V100 GPU with 32GB of VRAM. The training process uses a batch size of 8 and runs for 200 epochs. The learning rate is initialized to 0.001 and adjusted dynamically using a cosine annealing schedule.

C. Visualization Examples

Fig. S3 presents additional visualization examples of error detection using the AMNAR framework on the EgoPER dataset [1]. These examples highlight how AMNAR effectively identifies various types of errors in procedural tasks, demonstrating its robustness and adaptability in complex scenarios.

As shown in Fig. S4, the AMNAR framework accurately detects errors even when they occur within actions sharing the same label, effectively distinguishing between normal and erroneous executions.

References

[1] Shih-Po Lee, Zijia Lu, Zekun Zhang, Minh Hoai, and Ehsan Elhamifar. Error detection in egocentric procedural task videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18655–18666, 2024. 3, 5

[2] Rohith Peddi, Shivvrat Arya, Bharath Challa, Likhitha Pallapothula, Akshay Vyas, Jikai Wang, Qifan Zhang, Vasundhara Komaragiri, Eric Ragan, Nicholas Ruozzi, et al. Captaincook4d: A dataset for understanding errors in procedural activities. *arXiv preprint arXiv:2312.14556*, 2023. 3

[3] Xin Wang, Taein Kwon, Mahdi Rad, Bowen Pan, Ishani Chakraborty, Sean Andrist, Dan Bohus, Ashley Feniello, Bugra Tekin, Felipe Vieira Frujeri, et al. Holoassist: an egocentric human interaction dataset for interactive ai assistants in the real world. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 20270–20281, 2023. 3

Figure S3. Visualization examples from the EgoPER dataset using the AMNAR framework. In the **top sample**, two errors are detected: a misoperation—quick oats are poured directly into the bowl without measuring, and an additional action. The **middle sample** also contains three errors: a misoperation, an additional action, and using the wrong tool—stirring with a knife instead of a spoon. The **bottom sample** illustrates a sequence of errors: an accidental error—dropping the tortilla to the ground, followed by a corrective action—replacing the dropped tortilla with a new one, and finally an additional action—adding an incorrect ingredient.

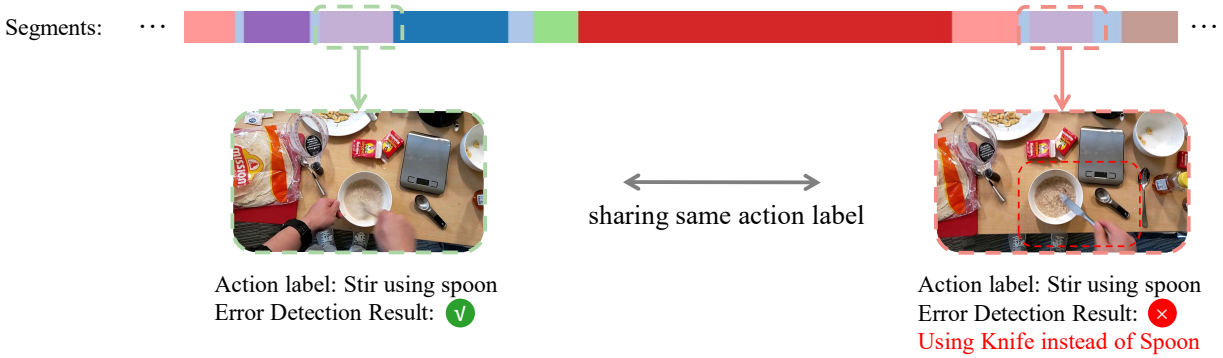

Figure S4. In this example, the AMNAR framework encounters two actions sharing the same label: one is a correctly executed action (normal), and the other is an erroneous action. Despite the shared label, AMNAR successfully detects the error in the second action (using a knife instead of a spoon) while correctly identifying the first action as normal, avoiding any false positives.