Autoregressive Action Sequence Learning for Robotic Manipulation

Xinyu Zhang, Yuhan Liu, Haonan Chang, Liam Schramm and Abdeslam Boularias

***Abstract*—Designing a universal policy architecture that performs well across diverse robots and task configurations remains a key challenge. In this work, we address this by representing robot actions as sequential data and generating actions through autoregressive sequence modeling. Existing autoregressive architectures generate end-effector waypoints sequentially as word tokens in language modeling, which are limited to low-frequency control tasks. Unlike language, robot actions are heterogeneous and often include high-frequency continuous values—such as joint positions, 2D pixel coordinates, and end-effector poses—which are not easily suited for language-based modeling. Based on this insight, we extend causal transformers' single-token prediction to support predicting a variable number of tokens in a single step through our Chunking Causal Transformer (CCT). This enhancement enables robust performance across diverse tasks of various control frequencies, greater efficiency by having fewer autoregression steps, and lead to a hybrid action sequence design by mixing different types of actions and using a different chunk size for each action type. Based on CCT, we propose the Autoregressive Policy (ARP) architecture, which solves manipulation tasks by generating hybrid action sequences. We evaluate ARP across diverse robotic manipulation environments, including Push-T, ALOHA, and RLBench, and show that ARP, as a universal architecture, matches or outperforms the environment-specific state-of-the-art in all tested benchmarks, while being more efficient in computation and parameter sizes. Videos of our real robot demonstrations, all source code and the pretrained models of ARP can be found at `http://github.com/mlzxy/arp`.**

I. Introduction

Autoregressive models are the basis of recent breakthroughs in natural language processing [1]. These models predict the next token in a sequence based on the previous tokens. Autoregressive models are typically implemented as causal transformers, where each token attends only to preceding ones, and they are trained with the single objective of maximizing the conditional likelihood of each token. Despite their simplicity, autoregressive models such as GPTs [2] are shown to demonstrate a reasoning ability that can capture causal dependencies [3]. In this work, we present a new universal autoregressive architecture that can be used for various robot manipulation tasks in diverse environments.

Decision Transformer (DT) and Trajectory Transformer (TT) are two pioneering approaches that use autoregressive models to solve control tasks [4], [5]. These methods learn to generate trajectories as $(R_1, s_1, a_1, \ldots, R_T, s_T, a_T)$, where R_t, s_t, a_t respectively denote the reward-to-go [6], the state, and the action at time-step t. However, these methods are primarily applied to tasks with fully observed, low-dimensional states—which is rarely the case in robotics applications. Recent work focuses on applying autoregression only on action sequences, such as Gato [7], VIMA [8], ManipLLM [9]. Despite their impressive results, these methods remain limited to low-frequency control tasks, as they represent robot actions with key end-effector waypoints and generate one action at a time, similar to word generation in language modeling.

However, unlike language, robot actions are heterogeneous and include continuous values—such as joint positions, 2D pixel coordinates, and effector poses. Additionally, in high-frequency control tasks, continuous actions are expected to maintain temporal smoothness—a requirement absent in language modeling. To adapt autoregressive models for robotic tasks, we propose the Chunking Causal Transformer (CCT), an improved version of the causal transformer used in standard autoregressive models. CCT introduces an important modification: it predicts the future tokens (a chunk of actions) from empty tokens rather than from the original sequence, as illustrated in Figure 6. In doing so, CCT extends the next-single-token prediction of causal transformer to *chunking autoregression*—next-multi-token prediction in a single step. Despite its simplicity, chunking autoregression offers three key advantages:

1) Predicting multiple temporally correlated actions in a single step addresses the primary limitation of autoregressive models in high-frequency control tasks.
2) Chunking autoregression increases efficiency by reducing the number of inference passes that are required.
3) Variable chunk sizes enable a flexible action sequence design, such as mixing different types of actions and using a different chunk size for each type. For example, high-level actions, like sparse 2D waypoints, can be predicted first sequentially to guide the prediction of low-level actions, such as joint positions, in larger chunks. We show action sequence designs of our real robot task, Push-T, ALOHA, and RLBench in Figure 3.

While action chunking has already been introduced in the Action Chunking Transformer (ACT) by [10], ACT is a one-step prediction model with a fixed chunk size. Instead, our approach supports variable chunk sizes for generating hybrid action sequences. Figure 10 shows that our method outperforms ACT by a significant margin in all environments.

Manuscript received: November 18, 2024; Revised: January 23, 2025; Accepted: February 25, 2025.

This paper was recommended for publication by Editor Markus Vincze upon evaluation of the Associate Editor and Reviewers' comments. This work is partly supported by NSF awards 1846043 and 2132972.

The authors are with the Department of Computer Science, Rutgers University, xz653@rutgers.edu.

Digital Object Identifier (DOI): see top of this page.

Figure 9 shows that our chunking autoregression is the key factor behind this strong performance. We illustrate the essential difference between action chunking, standard autoregression, and our chunking autoregression in Figure 1. We illustrates why chunking autoregression works in Fig. A3.

To summarize, our contributions are threefold. (1) We propose the Chunking Causal Transformer (CCT), which extends the single-token prediction of causal transformer to multi-token prediction, and therefore enables chunking autoregression. We also design a novel attention interleaving strategy that allows CCT to be trained efficiently with teacher-forcing, as shown in Figure 6. (2) Based on our CCT, we present the Auto-regressive Policy (ARP), a model that learns to generate heterogeneous action sequences autoregressively for solving robotic manipulation tasks. The ARP architecture is summarized in Figure 4. (3) We evaluate the same ARP architecture across Push-T [13], ALOHA [10], and RLBench [14], three environments with diverse manipulation tasks and control modalities, as outlined in Figure 2. Our study shows that ARP matches or outperforms all environment-specific SoTAs, while being more efficient computationally and using smaller parameter sizes, as in Figure 8. In addition, we evaluate ARP with a real robot on a challenging, contact-rich nut-tightening task, as in Figure 12.

II. Related Work

Learning robotic manipulation from demonstrations. Imitation learning enables robots to learn to perform tasks demonstrated by experts [15], [16]. Recently, various methods have been developed for manipulation learning with different task constraints and control modalities. Notably, [13] proposed the Diffusion Policy (DP) method for solving the Push-T task. [10] proposed the Action Chunking Transformer (ACT) for bi-manual manipulation tasks in the ALOHA environment. [17] proposed RVT-2 for language-conditioned tasks in the RLBench environment [14]. We outline these environments and the corresponding state-of-the-art (SoTA) solutions in Figure 2 and Figure A1, respectively. In contrast, our proposed autoregressive policy is a universal architecture that outperforms each environment-specific SoTA on Push-T, ALOHA, and RLBench.

Autoregressive models for control tasks. Besides the pioneering Decision Transformer and Trajectory Transformer, recent works such as VIMA [8], Gato [7], GR1 [12], OpenVLA [11] and ManipLLM [9] have looked into designing autoregressive models for robotic tasks. Despite the impressive results, these approaches are limited to low-frequency control tasks that rely on end-effector waypoints [11]. GR1 and OpenVLA only use autoregression in their LLM backbones and do not apply autoregression to solve control tasks. Most of the existing works require fine-tuning a large language model (LLM) such as LLaMA [9] to include target end-effector poses within text-based responses or predict poses from LLM's hidden features. The reliance on resource-intensive LLMs leads to large computational overhead, even for tasks that could be addressed with lightweight models. Without these constraints, our autoregressive policy outperforms SoTAs in multiple environments while being more efficient in computation and parameter sizes.

Hierarchical policies. Planning actions on multiple levels of abstraction is an important ability [20]. Existing methods generally separate the designs of low-level and high-level policies, and uses different modules for the different levels of abstraction [20], [21]. This complicated procedure prohibits a wider application. In contrast, our autoregressive policy is able to capture the underlying causal dependencies in robotic tasks by predicting a sequence of actions of different levels of abstraction by using a single model, as shown by our action sequence designs for diverse environments in Figure 3.

III. Method

In this section, we present the Auto-regressive Policy (ARP), built upon our Chunking Causal Transformer (CCT), which generates robot action sequences autoregressively. We summarize the architecture in Figure 4.

Action sequence modeling. Unlike natural language, robot actions lack a universal vocabulary. As shown in Figure 2, different robot tasks may require drastically different types of actions. Therefore, we represent actions as structured sequences whose formats are tailored for each family of tasks. Figure 3 showcases the formats of the action sequences generated in our real robot experiment, Push-T, ALOHA, and RLBench tasks.

Embedding and decoding heterogeneous actions. Language models map each word to a continuous vector called word embedding. The word embeddings of the input sentences are fed into a causal transformer. The distribution of the next word is decoded from the output embedding of the last word with a linear layer. Figure 5 and 7 illustrate our embedding and decoding methods for robot actions. Discrete actions are embedded by a table lookup on a weight matrix and decoded into a categorical distribution with a linear layer, similar to words in language modeling. Continuous actions are embedded with a linear layer and decoded into the parameters of a Gaussian mixture distribution with another linear layer. Actions that are defined as pixel coordinates are embedded by retrieving the point-wise features at the coordinates on a visual feature map. The output spatial distribution is obtained by multiplying the output embedding with the visual feature map, and converting the result into a 2-d heatmap with the up-sampling operator from RAFT [22].

Chunking causal transformer. Figure A4 illustrates the essential difference between a causal transformer and our CCT. A causal transformer modifies the token embedding with causal attention so that the last token becomes the next token. Our CCT modifies the token embedding with causal attention for the action tokens a_i and bidirectional attention for the empty tokens e_i (future actions). The empty tokens become the next tokens. This allows the prediction of a chunk of variable number of next tokens at once in a single forward pass by adding empty tokens, which enables action chunking during autoregressive generation. We study the impacts of our chunking autoregression in detail in Section IV. In ARP, CCT alternates between self-attention within the input embeddings

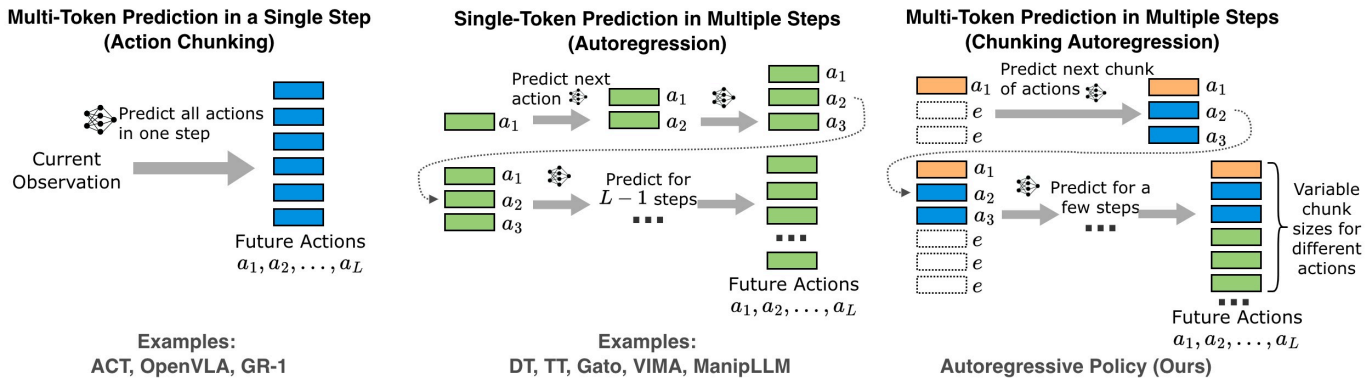

Fig. 1: **Existing Works versus Our Autoregressive Policy.** Action chunking models (left) predict all action tokens in a single step [10], [11], [12]. Standard autoregression models (middle) generate one action token in each step, which is inefficient and unsuitable for high-frequency control tasks [4], [5], [7], [8], [9]. Our proposed chunking autoregression (right) generates a chunk of variable number of action tokens per step, offering greater efficiency, strong performance across diverse tasks, and flexibility in designing hybrid action sequences. We compare the performance of these three action prediction strategies in Figure 9. Note all models use Model Predictive Control to predict L actions, execute them, update the observation, and then predict actions again. Autoregressive generation is performed without executing actions or changing the current observation.

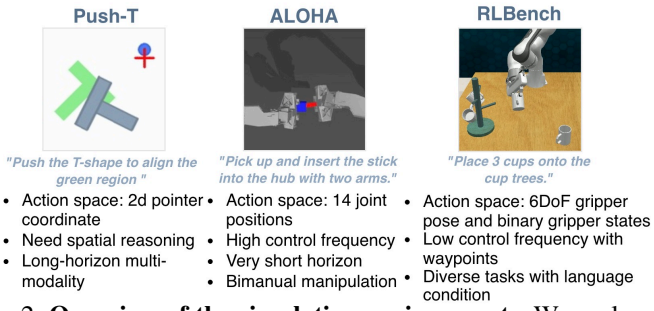

Fig. 2: **Overview of the simulation environments.** We evaluate our method on Push-T, ALOHA, and RLBench, three task suites with significantly different properties and requirements. Push-T [13] requires many steps to complete (long horizon) and where same sub-goals can be reached in various ways (multi-modality). ALOHA [10] has a high-dimensional action space (14 joints of two robot arms), a high control frequency (50Hz), and a short time limit (8 secs). RLBench [14] has only the gripper pose as action but contains 18 different language-conditioned tasks.

and cross-attention with vision features, as in Figure 4. We extract vision features from a standard backbone identical to the ones used in SoTA methods, as detailed in section IV.

Train-time attention interleaving. During training, a causal transformer is taught to predict each token in a given sequence by consuming all previous ground-truth tokens as input. This training strategy is named teacher-forcing [23]. As shown in Figure 6, only a single forward pass is required for training samples such as $a_1, a_2, a_3 \rightarrow a_4$ (predict a_4 from a_1, a_2, a_3), $a_1, a_2 \rightarrow a_3$, and $a_1 \rightarrow a_2$. Causal transformers are therefore efficiently trained with teacher-forcing. We follow this teacher-forcing strategy. However, training CCT yields separate forward passes per chunk. For example, the prediction of a_4 depends on a_2, a_3, as in $a_1, a_2, a_3, e_4 \rightarrow a_4$, but a_2, a_3 need to be replaced with e_2, e_3 to predict them from a_1, as in $a_1, e_2, e_3 \rightarrow a_2, a_3$. This prohibits the use of a single forward pass for both $a_1, a_2, a_3, e_4 \rightarrow a_4$ and $a_1, e_2, e_3 \rightarrow a_2, a_3$. Note a_i denotes the i-th action and e_i denotes the empty token for i-th action. This issue increases the training cost and complicates the training procedure.

To resolve this, we have all action tokens and their corresponding empty tokens as model input, as in ❶ ❷ of Figure A8. We then compute bidirectional attention within empty tokens and causal attention from empty tokens to action tokens using two masked attentions, as in ❸ ❹. Next, we compute causal attention within action tokens, as in ❺. This leverages the fact that action tokens are independent of future empty tokens. As a result, the updated action tokens are computed once and reused in the next layer, as in ❻. This enables a single forward pass of all tokens in three attention operations, regardless of the number of tokens or variable chunk configurations. We name this procedure *attention interleaving*. Figure A5 demonstrates the reduced MACs of training with attention interleaving. We implement attention interleaving as an internal acceleration mechanism of the transformer layer, which is transparent to other network modules. Note that attention interleaving is only used during training and incurs no additional inference cost.

Inference. During the test rollouts, we extract vision tokens from the current observation and provide them as input to ARP, which then generates actions autoregressively by sampling from the decoded action distribution and appending the selected actions to the existing action sequence. This process of generating and executing actions is repeated until episode termination (success, failure, or reaching the time limit). Actions are generated according to the sequence formats shown in Figure 3. We manually set the chunk size for each type of action, and total sequence length for each task. We provide more implementation details and hyper-parameter values in Section IV and Appendix B.

IV. Experiments

In this section, we investigate how the Auto-regressive Policy (ARP) performs compared to the existing methods

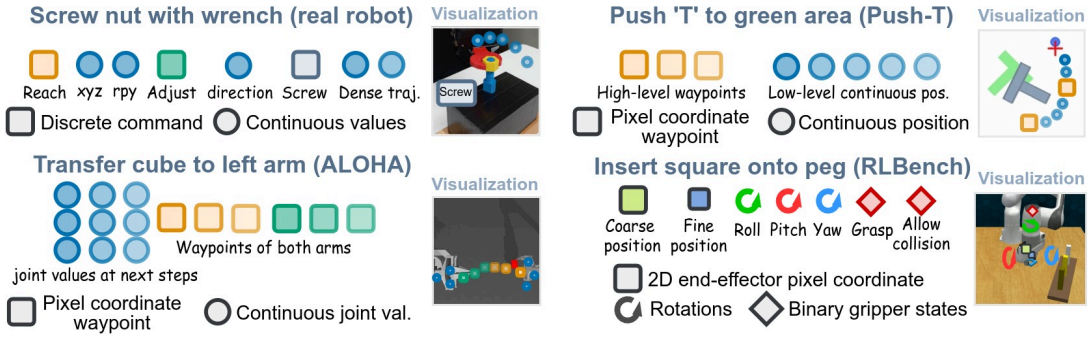

Fig. 3: **Learned Action Sequences.** In Push-T, our model predicts a sequence of high-level waypoints, followed by a sequence of low-level positions that connect the waypoints together and form the pushing trajectory, analogous to hierarchical planning [18]. In ALOHA, we predict the joint values and then end-effector waypoints conditioned on the joint values, a process akin to forward kinematics [19]. We bypass the waypoint generation during inference. In RLBench, we predict the target end-effector's position first, then gripper rotation and state in that position. For our real robot experiment, we define a set of primitive actions, as detailed in section IV-C. We predict the action type and then continuous values of that action.

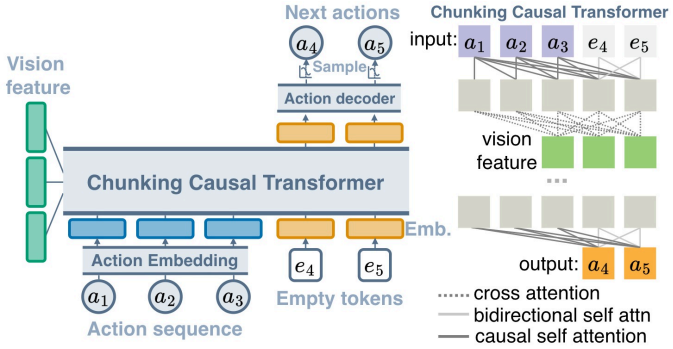

Fig. 4: **Autoregressive Policy Architecture.** A sequence of past actions and a chunk of empty tokens are concatenated and projected into embeddings. Empty tokens correspond to future actions, which are unknown and need to be predicted. These embeddings are fed into our Chunking Causal Transformer (CCT) along with the vision features of the current observation. CCT alternates between self-attention within the sequence embeddings and cross-attention with the vision features. Self-attention is causal for the input actions and bidirectional among the empty tokens. Distributions of future actions are decoded from the updated embeddings of the empty tokens.

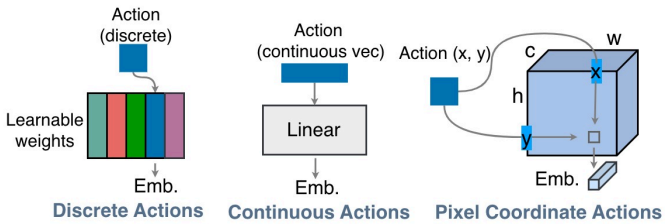

Fig. 5: **Embeddings for Discrete, Continuous, and Pixel-coordinate Actions.** Discrete actions are embedded by a simple table lookup on a weight matrix. Continuous actions are embedded with a linear layer. Pixel-coordinate actions are embedded by retrieving the point-wise features at the coordinates on the visual feature maps.

that were designed specifically for each environment. In addition, we examine whether auto-regression and action chunking are the primary contributors to the performance gains and evaluate how well existing methods perform across different environments. Further, we verify ARP on a challenging nut-tightening task with a real robot. Finally, we demonstrate that ARP can estimate the likelihood of robot actions and predict actions based on human inputs. All of our source code and the pre-trained models can be found at `http://github.com/mlzxy/arp`. A single-file implementation of our ARP can be found at `arp.py`.

A. Comparison with State-of-the-Art

Setup. We compare the autoregressive policy (ARP) against the SoTA solutions in Push-T, ALOHA, and RLBench environments. Push-T is a single task. ALOHA consists of two tasks: insertion and cube transfer. RLBench includes 18 tasks, each with multiple language variants. These environments are illustrated in Figure 2 and Figure A7. For Push-T and ALOHA, we train a separate policy for each task. For RLBench, a single policy is trained for all 18 tasks. In Push-T, the policy observes the last two 96×96 RGB frames, and predicts a window of future 2-d pointer positions as actions. In ALOHA, the policy observes the current 480×640 RGB frame and predicts a window of future 14-dimensional joint positions. In RLBench, the policy observes four RGBD 128×128 images and predicts the next target end-effector pose and gripper states. Existing SoTA techniques in these environments are outlined in Figure A1. We use the same vision backbones as the SoTA solutions to extract vision tokens, namely ResNet50 for Push-T and ALOHA, and Multi-View Transformer [17] for RLBench. We use the same training data, number of episodes, optimizer configuration, and evaluation frequency as the SoTA solutions. We detail the full list of hyper-parameters, such as the number of layers, sequence lengths, chunk sizes, and optimizer setups in Appendix B. Success rates for Push-T and RLBench are averaged over three independent runs. ALOHA's results are averaged over five runs.

Results. Figure 8 shows that our autoregressive policy (ARP) matches or outperforms environment-specific SoTAs while being more computationally efficient. Figure A6 compares the per-task success rates of our ARP and RVT-2 [17].

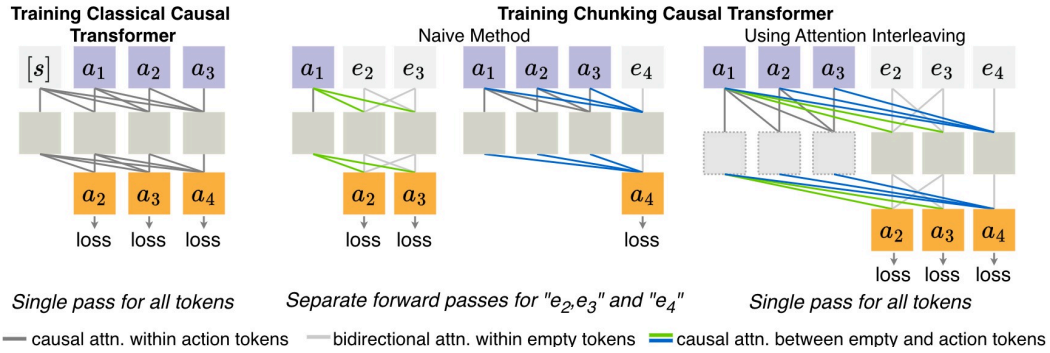

Fig. 6: **Training Chunking Causal Transformer (CCT) with Teacher-forcing.** Causal transformers are trained efficiently with only a single forward pass for all tokens in a given sequence. However, suppose a_2, a_3 and a_4 are in separate chunks, the CCT forward passes of predicting a_2, a_3 and a_4 cannot be merged directly. Naively running separate passes significantly increases computation costs, as in Figure A5. With the proposed attention interleaving, we can update all the empty tokens in a single pass, regardless of the number of tokens or variable chunk size configurations. The key idea is to update empty tokens and action tokens separately and reuse the causally attended action tokens to update empty tokens. A step-by-step example is provided in Figure A8 and `Video/attention-interleaving-tour.mp4` in the supplementary.

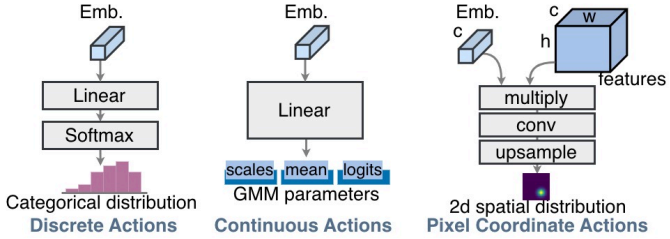

Fig. 7: **Decoders for Discrete, Continuous, and Pixel-coordinate Actions.** For discrete actions, we decode the action embeddings into a categorical distribution with a linear layer followed by a softmax operation. For continuous actions, we decode the embeddings into the parameters of a Gaussian mixture distribution with a linear layer. For the pixel-coordinate actions, we multiply the embedding with a visual feature map or a weight tensor, and convert the result into a 2-d heatmap.

We present two variants of ARP: both share the same autoregressive policy architecture, but the second variant has more CCT layers. The first ARP model matches the performance of RVT-2 while being more efficient, whereas the second one outperforms RVT-2. Notably, RVT-2 requires the current timestep as an input, whereas ARP relies solely on visual and language inputs. Further discussion on the architectural differences between RVT-2 and ARP and an analysis of the impact of timesteps, are provided in Appendix D.

B. Analysis

Does the performance gain come from chunking autoregression? Our action sequence design incorporates additional inputs for Push-T and ALOHA, as shown in Figure 3. These inputs are automatically extracted from the demonstration trajectories. In Push-T, the high-level waypoints are simply uniformly sampled down from the low-level trajectories and then discretized. In ALOHA, the pixel coordinates of the end-effector are computed from the joint values with the robot's forward kinematics and the camera parameters. It is possible that the performance gain of ARP originates from this extra information instead of our proposed architecture.

Environment	Method	Success Rate		#MACs	#Params
Push-T	Diffusion Policy	78.8		6.8G	25.5M
	ARP (Ours)	**87.1**		**3.7G**	**23.5M**
ALOHA	ACT	20.8	80.8	17.8G	50.9M
	ARP (Ours)	**24.8**	**94**	17.8G	**47.6M**
RLBench	RVT-2	81.4		57.1G	72.1M
	ARP (Ours)	81.6		**56.2G**	**71.9M**
		84.9		57.4G	73.8M

Fig. 8: **Comparing our Autoregressive Policy to the SoTA of each environment.** Our autoregressive policy (ARP) outperforms environment-specific SoTA and is more efficient in MACs (number of multiply-accumulates) and parameter sizes. We report results of the transformer version of the diffusion policy because of its overall better performance. The RVT-2 [17] results are reported from the original paper. All MACs and parameter sizes are measured with THOP [24]. We list the success rates of both the insertion (left) and cube transfer (right) tasks in Gym-ALOHA [25]. Per task success rates of RLBench are summarized in Figure A6.

Generation Mode	Push-T	ALOHA	
		Cube Transfer	Insertion
SoTA	78.8	80.8	20.8
Action Chunking	77.6	81.2	21.2
Single-token Autoregression	82.4	46	6.8
Chunking Autoregression (Ours)	**87.1**	**94**	**24.8**

Fig. 9: **Comparison of Action Prediction Strategies.** We compare the success rates (%) of action prediction strategies shown in Figure 1. Action chunking and next-token autoregression are implemented by simply setting the chunk size to a constant of full sequence length or 1 in our ARP, while keeping other settings unchanged, such as action sequence design in Figure 3. Our chunking autoregression supports variable chunk sizes and uses a different chunk size for each type of action token during generation.

Method	PushT	ALOHA		RLBench
		Cube Transfer	Insertion	
Diffusion Policy	78.8	10	1.6	-
ACT	77.5	80.8	20.8	69.8
ARP (Ours)	**87.1**	**94**	**24.8**	**81.2**

Fig. 10: **Evaluation of existing methods on various environments.** ACT, a VAE-based method, performs competitively across all environments, whereas Diffusion Policy struggles in ALOHA and RLBench. While we believe stronger diffusion-based methods can be developed in the future, our results suggest simpler architectures are more robust across diverse tasks.

Figure 9 compares the success rates of action chunking, standard single-token autoregression, and our chunking autoregression in Push-T and ALOHA. They share the same implementation, with action chunking models generating the entire sequence at once by setting the CCT chunk size to the full sequence length, and single-token autoregression setting the chunk size to 1. In contrast, our chunking autoregression uses different chunk sizes for each type of action. The results clearly show that our proposed chunking autoregression is the key factor behind the better performance.

Our approach innovates action chunking to support variable chunk sizes in sequence generation. Without our improvement, the standard next-token autoregression performs poorly at ALOHA, a task suites that requires fine-grained control inputs. We discuss why chunking autoregression matters in Fig. A3 and Appendix C. Compared to one-step action chunking models, our intuition can be explained through an example: imagine task B is difficult, but solving task A first, followed by solving task $B|A$ (task B given the result of task A), is much easier. An autoregressive model follows this sequential process, solving task A first and then leveraging the result to make task B more feasible. In contrast, a one-step model attempts to predict both tasks simultaneously, treating A and B as independent problems. While the one-step model may solve task A implicitly as part of solving task B, it does not explicitly take advantage of the problem structure and is therefore prone to shortcuts. This phenomenon has been explored in more depth for NLP tasks by [3].

Do existing methods work in different environments? Figure 10 shows how existing methods perform in different environments. When testing in a new environment, we keep the same architecture but adapt the vision backbone and optimizer to the environment's established setup. RVT-2 was not implemented for Push-T and ALOHA, as it is designed for sparse waypoint predictions, which are incompatible with the high-frequency actions required in these tasks. We did not implement the diffusion policy for RLBench, as it refines actions from gaussian noise, which conflicts with the common practice in RLBench of predicting actions in discrete spaces. While 3D Diffuser Actor [26] reports competitive results on RLBench, it uses a completely different architecture.

Figure 10 reveals that ACT, a VAE-based method, performs competitively across all environments, whereas the diffusion policy struggles to deliver meaningful performance

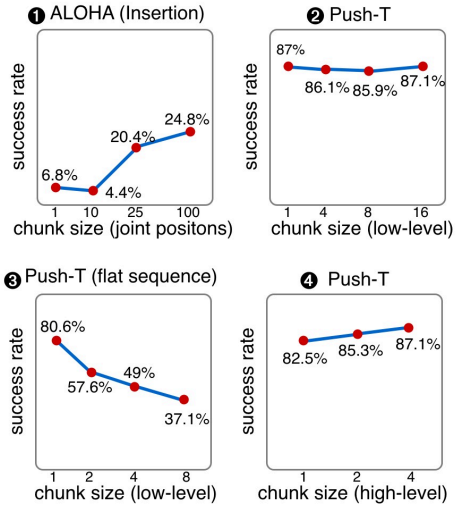

Fig. 11: **Impact of chunk size on performance.** Our results suggest that the optimal chunk size depends on both the task and the action sequence design. Therefore, the ability of our chunking causal transformer to support variable chunk sizes is essential for maximizing performance.

in ALOHA. This outcome is surprising, given the recent popularity of diffusion-based techniques. While we believe a strong diffusion architecture, like 3D Diffuser Actor on RLBench, could be developed for ALOHA, this suggests that simpler architectures could be more robust across a wider range of tasks and environments. Our auto-regressive policy is trained with a single objective: to maximize the conditional likelihood of each action in a sequence. We believe this simplicity contributes to its robust performance across diverse environments.

How does chunk size affect performance? Instead of predicting only the next token, our chunking causal transformer (CCT) is able to predict a variable number of next tokens, that is, a chunk of actions. Figure 11 illustrates the relationship between chunk size and success rate. The first plot shows that larger chunks significantly improve policy performance, a trend also observed by ACT [10]. This advantage of chunking actions seems generalizable to high-frequency control in short-horizon tasks. Interestingly, while larger chunk sizes for joint positions improve performance, action chunking without autoregression, where both end-effector waypoints and joint positions are predicted simultaneously, yields inferior results, as in Figure 9.

The second plot indicates that for Push-T, policy performance is largely insensitive to the chunk size of low-level trajectories because the standard deviation of the success rate ranges between 1 and 2. In this case, a moderate chunk size can be a better choice, given the common practice of executing only the first few predicted actions and then rerunning the policy, a test-time technique reduces error accumulation. This technique benefits from a moderate chunk size through early termination of autoregressive generation without sacrificing performance or computational efficiency.

In the third plot, we explore a different action sequence format for Push-T, where we remove high-level waypoints and flatten the trajectories into a vector, as detailed in Figure A2.

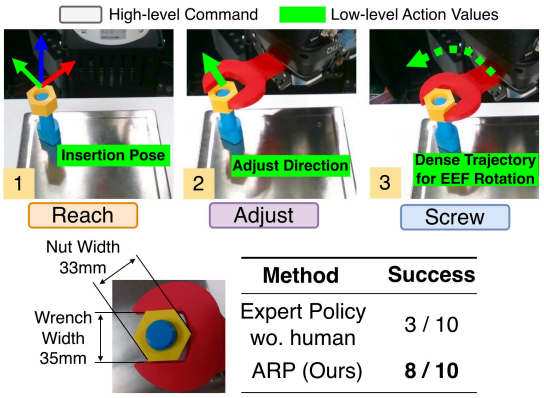

Fig. 12: **Real robot experiment.** Our ARP learns to adaptively select high-level commands and generate low-level action values, including position adjustment after unsuccessful insertion. We achieve a success rate of 8/10 in this nut-screwing task that requires a precise tool alignment. The bolt's position (blue) and nut's height (yellow) are randomized at every episode.

This design yields a completely different trend, with the policy performing well only when the chunk size is 1. The fourth plot shows that increasing the chunk size for high-level waypoints improves policy performance. These findings demonstrate that the optimal chunk size depends on both the task and the action sequence format. As a result, CCT's ability to flexibly adjust variable chunk sizes is essential for maximizing performance.

C. Real Robot Experiment

Setup. We evaluate ARP on a challenging tight nut-screwing task using a real robot, which requires precise alignment between the nut and wrench with a tolerance of 2mm, as shown in Figure 12. In each episode, the bolt (blue) is randomly placed on a 20×20 cm^2 table, while the height of the nut (yellow) is randomized along the 6cm tall bolt. The orientations of both the bolt and nut are also randomized per episode. We define three primitive actions: reach, adjust, and bolt. At each step, our ARP predicts a high-level command to select the action and then generates corresponding low-level action values. For example, ARP first predicts the reach command and an insertion pose. Next, the robot attempts to insert the wrench. After every unsuccessful attempt, the policy predicts the adjustment direction to adjust the wrench's position and reattempt insertion. Once the insertion succeeds, the policy switches to the screw command and predicts a dense trajectory to follow in order to rotate the end-effector around the wrench. All commands are automatically predicted by the autonomous model instead of being manually specified. An impedance controller stops unsuccessful insertions based on force feedback. We deploy this model on a Kuka LBR iiwa robot. We use 480×640 RBG-D observations from a single RealSense D415 camera. We use MVT as the vision backbone. To simplify the task, we assume the wrench is already grasped by the robot in a pre-defined position. An episode is considered successful if a screw action is completed after no more than three attempts to align the wrench on the nut. We trained ARP using 70 demonstrations collected from an expert policy. The expert policy uses Foundation Pose [27] to estimate insertion pose, with human operators providing fine-grained adjustments.

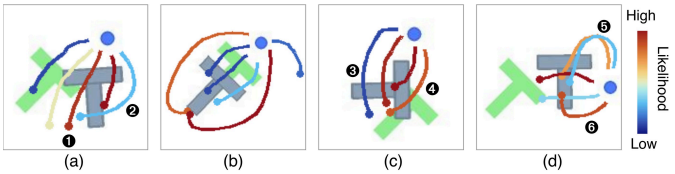

Fig. 13: **Trajectory Likelihood Estimation.** ARP generally assigns higher likelihoods to effective trajectories over futile ones, and demonstrates its understanding of action multi-modality as in subfigure (b). ARP's likelihood inference ability can identify model weaknesses and find defective demonstrations. All trajectories are human-drawn and are not seen in training.

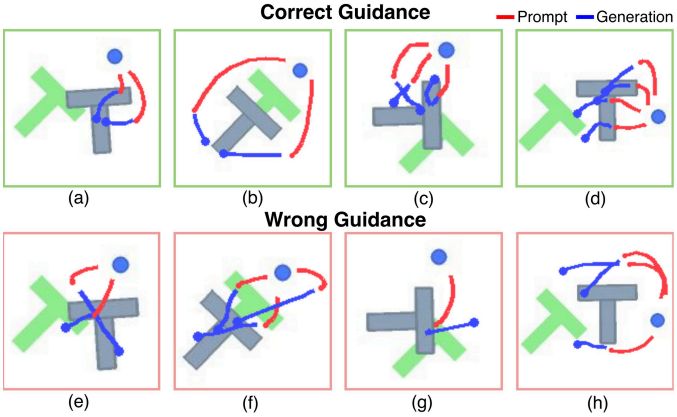

Fig. 14: **Trajectory Prediction based on Human Guidance.** We show predicted trajectories of ARP (blue), conditioned on human-drawn trajectories (red). The correct guidance is given with the intention of completing the task, and the wrong guidance is aimed at failing the task. ARP performs as expected under correct guidance. Under wrong guidance, ARP recovers from failure in subfigure (g), avoids further mistakes in subfigure (h), and amplifies the errors in subfigure (e) and (f), which reflects out-of-distribution behavior, as the training set consists only of successful demonstrations.

Results. Figure 12 shows that ARP screws nuts successfully in 8 out of 10 episodes, while the expert policy only has 3 successes out of 10 without human interventions. Most episodes succeeded without any adjustments because we used the adjusted successful insertion pose as the label for the reach command during training. To test ARP's adaptive adjustment ability, we add a uniform noise ranging from -5mm and 5mm along the normal plane of the insertion pose. Despite the noise, our ARP still succeeds in 6 out of 10 trials, with an average number of 1.6 adjustments per trial.

D. Qualitative Visualization

We showcase all the evaluation tasks in Figure A7. Video demonstrations of ARP in simulation and in the real world are available in the supplementary material. In this section, we show two key advantages of ARP: (1) estimating the likelihood of given robot actions, (2) and predicting actions conditioned on human input.

Likelihood inference. To generate the next token a_n, an auto-regressive model estimates the conditional probability $P(a_n|a_1,...,a_{n-1})$. Using the product rule, the model can estimate the joint probability $P(a_1,...,a_n) = \prod_{i=2}^{n} P(a_i|a_1,\ldots,a_{i-1})P(a_1)$ for any given sequences, a capability that more advanced generative frameworks such as VAE and diffusion lack. Figure 13 shows for different trajectories the likelihood estimated by ARP. All these trajectories are human demonstrations. ARP generally assigns higher likelihoods to effective trajectories and lower likelihoods to futile ones. For instance, in sub-figure (b), ARP assigns high likelihoods to two symmetrical trajectories around the T object, demonstrating its understanding of action multi-modality. However, some likelihood assignments are less intuitive. For example, trajectories ❶, ❹, and ❻ receive moderately high likelihoods, yet they may not bring the T-shape object closer to the green target, at least not better than the low-likelihood trajectories ❷ and ❸. ❺ marks two similar trajectories, yet they have different likelihoods. We believe that this type of likelihood inference can help identify the model's weaknesses and eliminate defective demonstrations from the training data.

Prediction with human guidance. Auto-regressive models generate future tokens conditioned on the previous sequence. In Figure 14, we illustrate examples of trajectories of ARP (blue) in Push-T, predicted conditionally on human-drawn initial trajectories (red). The first row (green) shows predictions under correct guidance, where the intention is to complete the task successfully. The second row (pink) is based on a wrong guidance with the intention of failing the task. ARP completes the trajectory correctly given a correct initial part. Given a wrong initiation, sub-figure (g) shows ARP's recovery from failure by correcting its initial trajectory. In sub-figures (e) and (f), however, ARP amplifies the initial error by pushing further in the wrong direction. This behavior reflects ARP's out-of-distribution response, as the training set consists only of successful demonstrations.

V. Discussion

We have shown that ARP is a strong and universal architecture that can be trained to perform diverse manipulation tasks. Here we discuss its limitations and potential future directions.

Learning to plan. Planning is a key ability of intelligent agents. It requires the agent to reason not only about its actions but also their impacts on its environment. Motivated by the reasoning capacity of auto-regressive models in NLP, a promising direction is to incorporate planning into ARP. One possible solution is to predict sequences of both states and actions. States in robotics are typically high-dimensional, such as images or point clouds. To solve this problem, ARP can be extended to generate future states by using recent hybrid architectures of autoregression [28].

Interactive robot learning. Human-Robot collaboration improves efficiency by allowing the robot to recover from its errors [29]. One possible future direction is to integrate active learning techniques into ARP to learn from immediate human feedback. The auto-regressive mechanism naturally supports conditioning action prediction on human input. Moreover, ARP can estimate the likelihood of action sequences. Likelihood is a common measure for identifying the most informative samples in active learning [30]. This can be used, for example, to prioritize demonstrations of tasks where the robot encounters more difficulties.

Adaptive action sequence learning. Despite ARP's impressive performance, it still requires a manual design of action sequence formats and chunk sizes for each environment. Unlike natural language, robot actions lack a universal vocabulary. A promising direction is to design a universal robot action language applicable across various environments [31], [32], which reduces the cost of defining new actions, unifies training datasets, and improves generalization.

References

[1] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023. 1

[2] B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, vol. 1, 2020. 1

[3] B. Prystawski, M. Li, and N. Goodman, "Why think step by step? reasoning emerges from the locality of experience," *Advances in Neural Information Processing Systems*, vol. 36, 2024. 1, 6

[4] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," *Advances in neural information processing systems*, vol. 34, pp. 15 084–15 097, 2021. 1, 3

[5] M. Janner, Q. Li, and S. Levine, "Offline reinforcement learning as one big sequence modeling problem," *Advances in neural information processing systems*, vol. 34, pp. 1273–1286, 2021. 1, 3

[6] A. Tamar, D. Di Castro, and S. Mannor, "Learning the variance of the reward-to-go," *Journal of Machine Learning Research*, vol. 17, no. 13, pp. 1–36, 2016. 1

[7] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, *et al.*, "A generalist agent," *arXiv preprint arXiv:2205.06175*, 2022. 1, 2, 3

[8] Y. Jiang, A. Gupta, Z. Zhang, G. Wang, Y. Dou, Y. Chen, L. Fei-Fei, A. Anandkumar, Y. Zhu, and L. Fan, "Vima: General robot manipulation with multimodal prompts," *arXiv preprint arXiv:2210.03094*, vol. 2, no. 3, p. 6, 2022. 1, 2, 3

[9] X. Li, M. Zhang, Y. Geng, H. Geng, Y. Long, Y. Shen, R. Zhang, J. Liu, and H. Dong, "Manipllm: Embodied multimodal large language model for object-centric robotic manipulation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 18 061–18 070. 1, 2, 3

[10] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn, "Learning fine-grained bimanual manipulation with low-cost hardware," *arXiv preprint arXiv:2304.13705*, 2023. 1, 2, 3, 6, 11

[11] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, *et al.*, "Openvla: An open-source vision-language-action model," *arXiv preprint arXiv:2406.09246*, 2024. 2, 3

[12] H. Wu, Y. Jing, C. Cheang, G. Chen, J. Xu, X. Li, M. Liu, H. Li, and T. Kong, "Unleashing large-scale video generative pre-training for visual robot manipulation," *arXiv preprint arXiv:2312.13139*, 2023. 2, 3

[13] C. Chi, S. Feng, Y. Du, Z. Xu, E. Cousineau, B. Burchfiel, and S. Song, "Diffusion policy: Visuomotor policy learning via action diffusion," *arXiv preprint arXiv:2303.04137*, 2023. 2, 3, 11

[14] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "Rlbench: The robot learning benchmark & learning environment," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3019–3026, 2020. 2, 3, 14

[15] X. Zhang and A. Boularias, "One-shot imitation learning with invariance matching for robotic manipulation," in *Proceedings of Robotics: Science and Systems*, Delft, Netherlands, 2024. 2

[16] M. Zare, P. M. Kebria, A. Khosravi, and S. Nahavandi, "A survey of imitation learning: Algorithms, recent developments, and challenges," *IEEE Transactions on Cybernetics*, 2024. 2

[17] A. Goyal, V. Blukis, J. Xu, Y. Guo, Y.-W. Chao, and D. Fox, "Rvt-2: Learning precise manipulation from few demonstrations," *arXiv preprint arXiv:2406.08545*, 2024. 2, 4, 5, 11

[18] D. Hafner, K.-H. Lee, I. Fischer, and P. Abbeel, "Deep hierarchical planning from pixels," *Advances in Neural Information Processing Systems*, vol. 35, pp. 26 091–26 104, 2022. 4

[19] S. Kucuk and Z. Bingul, *Robot kinematics: Forward and inverse kinematics*. INTECH Open Access Publisher London, UK, 2006. 4

[20] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, "Hierarchical reinforcement learning: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–35, 2021. 2

[21] S. Belkhale, Y. Cui, and D. Sadigh, "Hydra: Hybrid robot actions for imitation learning," in *Conference on Robot Learning*. PMLR, 2023, pp. 2113–2133. 2

[22] Z. Teed and J. Deng, "Raft: Recurrent all-pairs field transforms for optical flow," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*. Springer, 2020, pp. 402–419. 2

[23] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989. 3

[24] L. Zhu, "Lyken17/pytorch-opcounter: Count the macs / flops of your pytorch model." https://github.com/Lyken17/pytorch-OpCounter, (Accessed on 09/16/2024). 5

[25] R. Cadene, S. Alibert, A. Soare, Q. Gallouedec, A. Zouitine, and T. Wolf, "Lerobot: State-of-the-art machine learning for real-world robotics in pytorch," https://github.com/huggingface/lerobot, 2024. 5

[26] T.-W. Ke, N. Gkanatsios, and K. Fragkiadaki, "3d diffuser actor: Policy diffusion with 3d scene representations," *arXiv preprint arXiv:2402.10885*, 2024. 6

[27] B. Wen, W. Yang, J. Kautz, and S. Birchfield, "Foundationpose: Unified 6d pose estimation and tracking of novel objects," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 17 868–17 879. 7

[28] B. Chen, D. M. Monso, Y. Du, M. Simchowitz, R. Tedrake, and V. Sitzmann, "Diffusion forcing: Next-token prediction meets full-sequence diffusion," *arXiv preprint arXiv:2407.01392*, 2024. 8

[29] H. Liu, A. Chen, Y. Zhu, A. Swaminathan, A. Kolobov, and C.-A. Cheng, "Interactive robot learning from verbal correction," *arXiv preprint arXiv:2310.17555*, 2023. 8

[30] A. T. Taylor, T. A. Berrueta, and T. D. Murphey, "Active learning in robotics: A review of control principles," *Mechatronics*, vol. 77, p. 102576, 2021. 8

[31] X. Zhang, Y. Liu, H. Chang, and A. Boularias, "Scaling manipulation learning with visual kinematic chain prediction," 2024. [Online]. Available: https://arxiv.org/abs/2406.07837 8

[32] J. Zheng, J. Li, D. Liu, Y. Zheng, Z. Wang, Z. Ou, Y. Liu, J. Liu, Y.-Q. Zhang, and X. Zhan, "Universal actions for enhanced embodied foundation models," *arXiv preprint arXiv:2501.10105*, 2025. 8

[33] S. James, K. Wada, T. Laidlow, and A. J. Davison, "Coarse-to-fine q-attention: Efficient learning for visual robotic manipulation via discretisation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 13 739–13 748. 13

APPENDIX

A. Code and Pretrained Models

The source code of our autoregressive policy is included in both the supplementary folder `Code` and https://github.com/mlzxy/arp. Please check the `README.md` for instructions on installation, dataset setup, downloading pretrained models, and documentation.

B. Hyper-parameters and Implementation Details

In this section, we provide a full list of hyper-parameters in Figure A1, Figure A2, and Figure A3 for Push-T, ALOHA, and RLBench, respectively, along with comments on selected hyper-parameters to provide additional implementation details.

Model. The mlp size denotes the hidden feature dimension of the MLP network within the standard multi-head attention operation. The number of latents refers to the number of Gaussians for the Gaussian mixture distribution used to decode continuous actions. The backbone denotes the network used to extract the vision features. We use the ResNet50 for Push-T and ALOHA, and Multi-View Transformer (MVT) for RLBench, identical to the ones used in Diffusion Policy, ACT, and RVT2.

Action Sequence. The horizon refers to the number of actions predicted at each step, while the number of action steps indicates how many actions are actually executed, with the remainder discarded. We adopt the same horizon and action steps as state-of-the-art methods. In Push-T, the chunk size for both high- and low-level actions matches the horizon, meaning all high-level points are predicted in one chunk, followed by all low-level points. Yet, interestingly, as shown in Figure 9, combining these two chunks into a single-step prediction degrades performance. For RLBench, which uses the next key end-effector pose as the control interface, there is no need for high-frequency actions, so neither the horizon nor action steps apply. Instead, low-level robot movements are generated using RLBench's built-in RRT planner. We use a chunk size of 2 for binary gripper states and a chunk size of 1 for end-effector positions and rotations. For example, ARP first predicts the roll, followed by pitch and yaw of the rotation Euler angle. We follow the strategy of RVT-2 to predict coarse positions and then refine them by zooming into the images (with updated vision features) to obtain more accurate positions. The end-effector positions are predicted in 2-d, and the 3-d positions are derived from the 2-d coordinates of each viewpoint.

Train & Eval. The observation $2 \times 96 \times 96 \times 3$ represents 2 frames of RGB images, each with a resolution of 96x96 pixels. For RLBench, the observation $4 \times 128 \times 128 \times 4$ refers to RGBD images (with one depth channel) at 128x128 resolution, captured from 4 cameras. In ALOHA, the maximum evaluation steps of 400 and control frequency of 50Hz indicate an evaluation time limit of 8 seconds. LAMB refers to the large batch optimizer. We use the same number of training steps, evaluation frequency, optimizer, learning rate, and learning rate scheduler as used by the SoTA solutions.

TABLE A1: Hyperparameters used in our experiments on Push-T.

Hyperparameter	Value
Model	
number of layers	30
embedding size	64
mlp size	256
number of latents (gmm)	4
backbone	RN50
Action Sequence	
horizon (low-level)	16
horizon (high-level)	4
number of action steps	8
chunk size (low-level)	16
chunk size (high-level)	4
Train & Eval	
observation	RGB $2 \times 96 \times 96 \times 3$
control frequency	10
maximum evaluation steps	300
train epochs	2000
eval frequency	50
batch size	128
learning rate	0.0001
learning rate scheduler	cosine with restart
optimizer	AdamW

TABLE A2: Hyperparameters used in our experiments on ALOHA

Hyperparameter	Value
Model	
number of layers	4
embedding size	512
mlp size	2048
number of latents (gmm)	1
backbone	RN50
Action Sequence	
horizon (joints)	100
horizon (waypoints)	10
number of action steps	100
chunk size (joints)	100
chunk size (waypoints)	1
Train & Eval	
observation	RGB $1 \times 480 \times 640 \times 3$
control frequency	50
maximum evaluation steps	400
train steps	100000
eval frequency	10000
batch size	8
learning rate	1.00e-5
learning rate scheduler	none
optimizer	AdamW

C. Discussion on Action Chunking

Action chunking has a clear downside – when predicting multiple actions at a time, the agent doesn't receive information about what state was observed after the first action. This means that the agent is operating with less information than if a single-step prediction was used. At the same time, in a MDP the state is guaranteed to be a sufficient statistic for the optimal policy. Given this information, why should action chucking be useful?

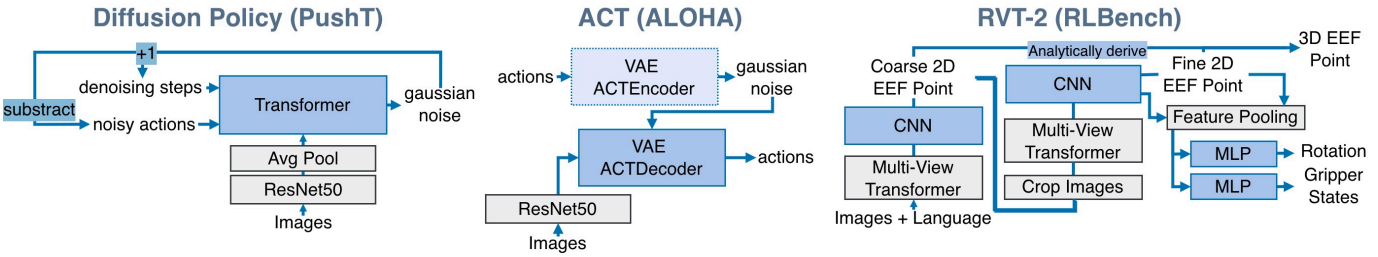

Fig. A1: **Overview of SoTA solutions on Push-T, ALOHA, and RLBench.** Diffusion Policy (DP) [13] iteratively subtracts Gaussian noises from noisy actions. The transformer network predicts the Guassian noise at each step. Action Chunking Transformer (ACT) [10] is a VAE architecture that predicts actions directly from images and Gaussian noises. RVT-2 [17] is a hybrid and more complex model, but it is trained directly with behavior cloning and it does not require a generative framework such as diffusion or VAE.

TABLE A3: Hyperparameters used in our experiments on RLBench.

Hyperparameter	**Value**
Model	
number of layers	8
embedding size	128
mlp size	512
backbone	MVT
Action Sequence	
chunk size	mix of 2 and 1
Train & Eval	
observation	RGBD $4 \times 128 \times 128 \times 4$
maximum evaluation steps	25
train epochs	80000
eval frequency	10000
batch size	96
learning rate	1.25e-5
learning rate scheduler	cosine
optimizer	LAMB

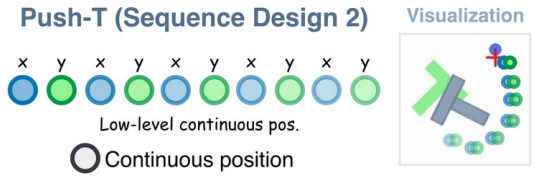

Fig. A2: **Flattened Action Sequence for Push-T.** Based on the action sequence in Figure 3, we remove the high-level waypoints and flatten the 2D coordinates into a single vector. For example, a trajectory of $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ is transformed into vector $(x_1, y_1, x_2, y_2, x_3, y_3)$. The policy is trained to predict first the x-coordinate of the initial point, then the y-coordinate, followed by the x- and y-coordinates of subsequent points.

We propose two main reasons. First, as has been explored in other imitation learning works, using expert data means that the dataset often lacks information on how to recover from errors, which means that predictions grow worse over time. Using longer action chunks effectively shortens the time horizon. However, we find that action chunking still has noticeable benefits even when the state is well-covered,

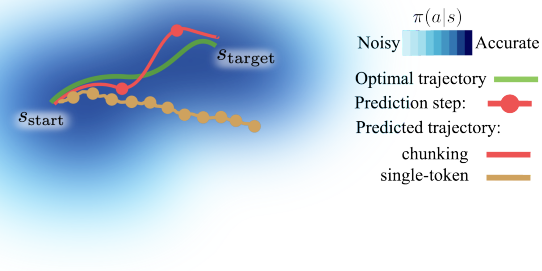

Fig. A3: **Why Chunking Autoregression Works.** Consider a robot navigating from state s_{state} to s_{target} in configuration space, where policy accuracy is indicated by color intensity (darker = higher accuracy). The green line denotes the optimal trajectory. Chunking autoregression has smaller accumulated error by having fewer prediction steps, keeping the trajectory within high-accuracy regions and converging near s_{target}. In contrast, single-step autoregression suffers from error compounding—each step introduces noise, progressively pushing predictions into out-of-distribution regions. Crucially, this divergence occurs regardless execution, as autoregressive generation conditions on previous predictions: noisy input from last step lead to increasingly unstable outputs.

such as in the Push-T environment. Additionally, this problem becomes less severe as the dataset grows – when the prediction error goes to zero, so does the effect of error recovery.

The second and perhaps stronger explanation is that if the demonstrations are non-Markov, the Markov policy that maximizes single-step accuracy is *not necessarily the optimal policy*. This is true even even if the demonstration policies are optimal, and even in the limit as data and model capacity become infinite. This is because the state occupancy measure is not convex with respect to the policy, so linear combinations of policies can lead to state distributions that are not linear combinations of the demonstration state distributions. This can be address either by learning a non-Markov policy, or by learning a Markov policy that imitates the desired state

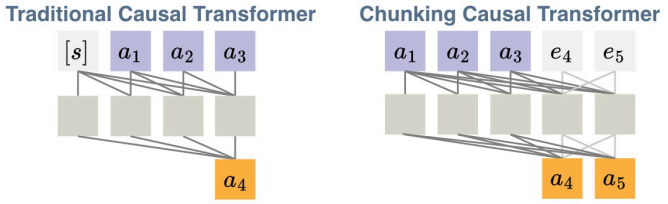

Fig. A4: **Causal Transformer versus Chunking Causal Transformer.** Causal transformer prepends the input sequence with a "start" token $[s]$ and modifies the token embedding with causal attention so that the last token a_3 becomes the next token a_4. Chunking Causal Transformer (CCT) appends the input sequence with a chunk of empty tokens, for example, e_4, e_5. CCT modifies the token embedding with causal attention for the action tokens a_1, a_2, a_3 and bidirectional attention for the empty tokens e_4, e_5. The empty tokens e_4, e_5 become the next tokens a_4, a_5. CCT can predict a variable amount of next tokens by configuring the number of empty tokens.

distribution rather than the demonstrations.

D. Comparison with RVT-2

Architectural difference between RVT-1 and ARP. Our ARP shares the same visual-language encoder, i.e., Multi-View Transformer (MVT) with RVT-2. Three notable architectural differences between ARP and RVT-2 are:

1) RVT-2 employs a two-stage approach. In the first stage, RVT-2 predicts a coarse end-effector pose from images of the entire workspace. New images are then captured at the predicted coarse pose location, providing finer visual details of the surrounding area. In the second stage, RVT-2 uses these detailed images to predict a fine-grained end-effector pose, which serves as the final output. To implement this, RVT-2 utilizes two MVT encoders—one for coarse inputs and another for fine inputs—along with two separate policy networks for coarse and fine predictions. Each policy network is composed of CNNs and MLPs.
   Similarly, our ARP adopts this two-stage approach for RLBench tasks. We also employ two MVT encoders for coarse and fine inputs, respectively. However, unlike RVT-2, ARP uses a single autoregressive policy network. During the coarse stage, the input visual tokens to this network are from the coarse MVT encoder, while during the fine stage, the input visual tokens come from the fine MVT encoder.
2) RVT-2 uses "Location Conditioned Rotation", a hand-crafted MLP that predicts gripper rotation conditioned on the gripper's translation. In contrast, ARP achieves a similar effect through autoregression, eliminating the need for a manually designed component.
3) RVT-2 directly upscales the visual features from the MVT encoder to predict pixel coordinates. In contrast, ARP upscales the multiplication of the predicted token (generated through autoregression) with the MVT visual features. Our approach is more aligned with sequence learning, as shown in Figure 7.

We remark that all three architectural differences pertain specifically to the autoregressive policy and are not part of the visual-language encoder.

Why not use timestep as model input. In Figure 8, the RVT-2 result is obtained with the current timestep as input, while our ARP models do not include timestep in their input. We do not use timestep as input for two reasons. First, during training, RVT-2 does not utilize the correct timestep due to an implementation bug in its data loader. This issue can be traced down in the RVT-2 codebase at L239, L265, L392 of `rvt/utils/dataset.py`. Specifically, the timestep of a frame varies depending on when it is inserted into the replay buffer, which we believe is an unintended behavior by the authors of RVT-2. Accurately emulating this behavior, or migrating our implementation to use this data loader would require substantial engineering effort. Moreover, this data-loader has been noted by other researchers for being confusing, as highlighted in this GitHub discussion (link) and even the code comments from RVT (link). Therefore, we have opted to train ARP without using time-step information.

Second, and more importantly, the timestep in RLBench does not correspond to physical time but rather to the number of macro-steps executed so far in the current episode. Each macro-step consists of three stages: (1) predicting the next end-effector pose and gripper action using a policy (e.g., RVT-2 or ARP), (2) planning a trajectory to reach the predicted pose using RRT, and (3) executing the trajectory and gripper action. As noted by the authors of RVT-2 and PerAct in these GitHub discussions (link1, link2), "time could matter as it informs the network of the current stage of the task". However, in the real world, the autonomous robot must learn to infer the task stage solely from visual inputs, as there will be no oracle that will be telling the robot the current stage of the task. This capability is critical for scenarios where the robot must interact with the physical world continuously without requiring controlled resets to an initial stage with a timestep of 0, such as placing objects in predefined regions. For these reasons, we have chosen to exclude timestep information when training ARP models.

Dive deeper into timestep and sampling strategy. As we mention above, existing methods for RLBench, such as PerAct, RVT, and ARM are impacted by a flawed training data-loader implementation. The behavior of this data-loader is two-fold:

1) The sampling rate of keyframes gets increased. Note that keyframes are the frames where the gripper stopped, which are provided by RLBench.
2) The timestep of a frame depends on when it is inserted into the replay buffer (randomized).

As a result of its unclear behavior, it is difficult to study the impact of timestep or sampling strategy in training RLBench models. To resolve this issue, we implemented our RLBench dataloader from scratch. Our dataloader has a simple implementation and straightforward behaviors:

1) It only loads keyframes.

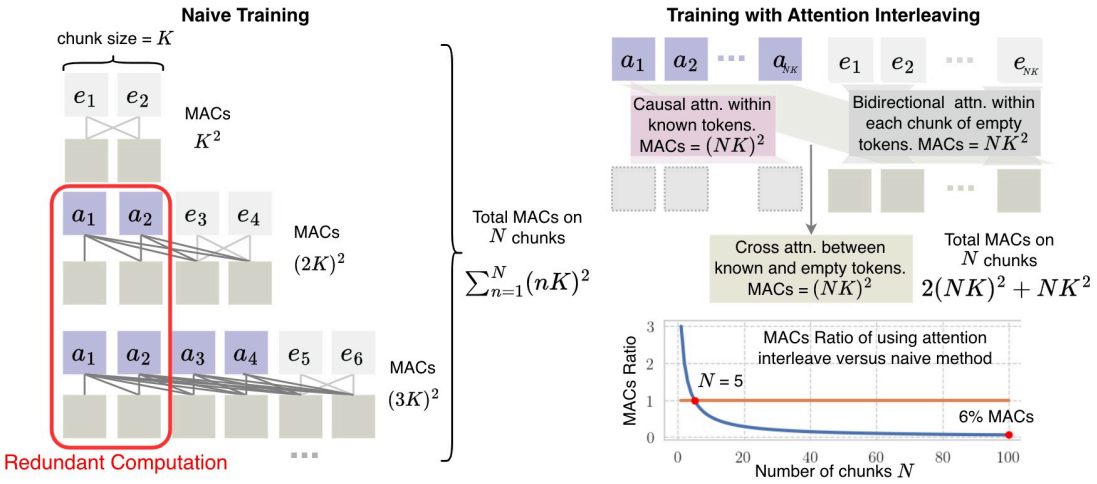

Fig. A5: **Naive Training versus Training with Attention Interleaving.** The left figure demonstrates that the causal attention within a_1, a_2 is computed twice, when inputs are a_1, a_2, e_3, e_4 and $a_1, a_2, a_3, a_4, e_5, e_6$. This redundancy can be reduced by precomputing the causal attention of all action tokens and caching the results. In doing so, the MACs are reduced from $\sum_{n=1}^{N}(nK)^2$ to $2(NK)^2 + NK^2$, where N, K are chunk number and chunk size. For simplicity, we count the MACs as the number of attention entries. In addition to the reduced MACs, we find that having a single forward pass for all tokens yields a much cleaner training procedure, a benefit that is not quantified by the raw number of multiply-accumulate operations.

Method	Avg. Success	Avg. Rank	Close Jar	Drag Stick	Insert Peg	Meat off Grill	Open Drawer	Place Cups	Place Wine	Push Buttons
RVT2	81.4	2.22	**100.0**	99.0	40.0	**99.0**	74.0	38.0	95.0	**100**
ARP (Ours)	81.6	1.89	97.6	88.0	53.2	96.0	90.4	48	92.0	**100.0**
ARP$^+$ (Ours)	**84.9**	**1.61**	95.2	**99.2**	**78.4**	97.6	**92.8**	**48.8**	**96**	**100.0**
	Put in Cupboard	Put in Drawer	Put in Safe	Screw Bulb	Slide Block	Sort Shape	Stack Blocks	Stack Cups	Sweep to Dustpan	Turn Tap
RVT2	66.0	96.0	**96.0**	88.0	92.0	35.0	**80.0**	69.0	**100.0**	99.0
ARP (Ours)	68.0	**99.2**	94.4	85.6	**98.4**	35.2	55.2	76.8	90.4	**100.0**
ARP$^+$ (Ours)	**69.6**	98.4	86.4	**89.6**	92.8	**46.4**	63.2	**80.0**	97.6	96.0

Fig. A6: **Performance on RLBench.** We report the success rate for each task, and measure the average success rate and rank across all tasks. ARP$^+$ shares the same network definition with ARP but has more layers. The MACs / parameter sizes of RVT-2, ARP, ARP$^+$ are 72.1M/57.1G, 71.9M/56.2G, and 73.8M/57.4G, respectively. ARP performs comparably or outperforms RVT-2 on all tasks. Note that RVT-2 requires current timestep as input, and ARP models do not use timestep.

2) It provides a correct, non-randomized timestep, if timestep is required as input.

Table A4 compares the results of RVT-2 and ARP on different frame sampling strategies and timestep configurations. The "original sampling + randomized timestep" represents the official implementation of RVT-2. We have some surprising observations:

1) If we remove the randomized timestep, then the performance of RVT-2 drops significantly (81.4 to 77.0). This indicates that the timesteps, due to the unintended randomization, may serve as a regularization instead of providing extra information.
2) The original implementation of RVT-2 requires the timestep as an input. However, we find that by updating the dataloader, we can achieve comparable results without it (81.6 vs. 81.4). This indicates the importance of keyframes in RLBench training, and also verifies the effectiveness of our implementation.
3) If correct timesteps are provided during training, the performance of both RVT-2 and ARP drops drastically (81.4 to 74.1, 81.6 to 77.8). This indicates that the correct timestep is actually harmful to RLBench model trainings, contrary to previous believes. This can be explained by seeing timestep as an information leakage for task stage.

It is important to note that the potentially flawed design of this data-loader originates from the early work, C2FARM [33], rather than RVT-2. This legacy design was subsequently adopted by PerAct, RVT, and RVT-2 to ensure fair comparisons by maintaining consistency in the training data distribu-

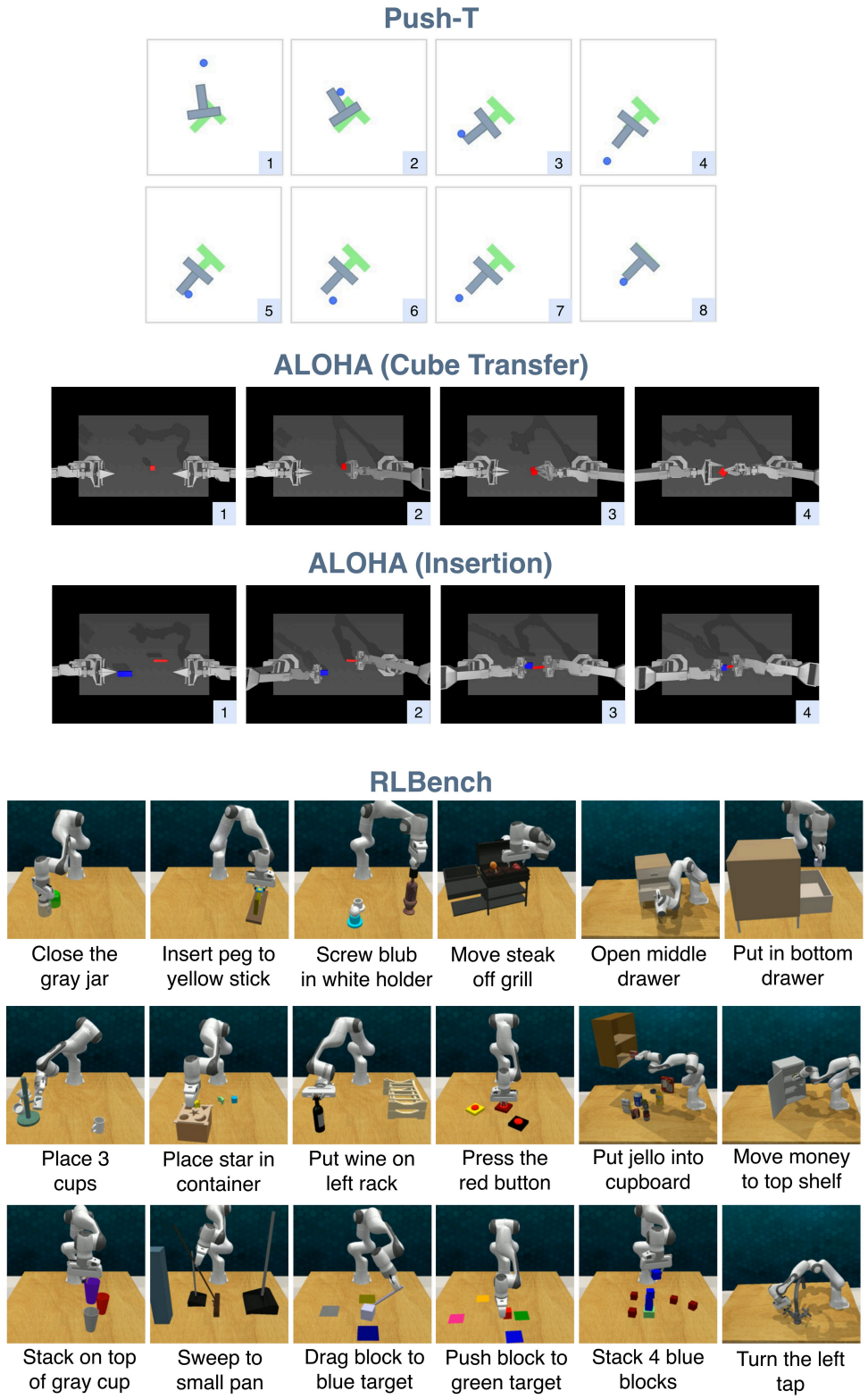

Fig. A7: **Demonstrations of all tasks in Push-T, ALOHA, and RLBench.** We provide visualizations of key frames from a single episode of Push-T and ALOHA, with the frame order indicated at the bottom right. For RLBench, we visualize one language variant for each task. RLBench features over 100 task variants specified through natural language commands [14], such as "`open [pos] drawer`" where `pos` is selected from `top, middle, bottom`, and "`stack [num] [color] blocks`", where `num` ranges from `2, 3, 4`, and `color` is chosen from a palette of 20 colors.

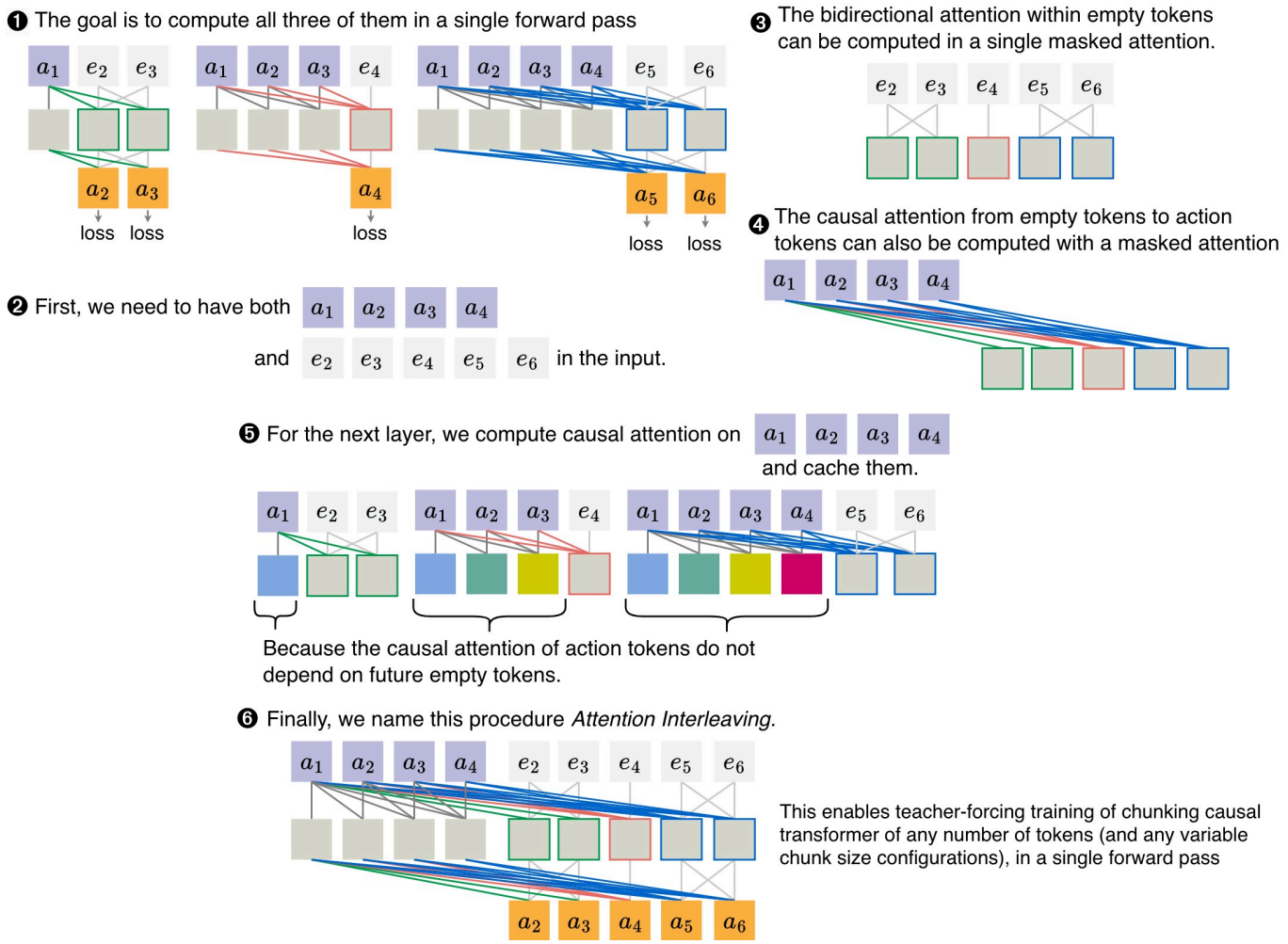

Fig. A8: **Step-by-Step Explanation of Attention Interleaving.** We provide a video version of this figure `Video/attention-interleaving-tour.mp4` in the supplementary.

tion. We hope our released implementation of this simplified data-loader and ARP can be helpful for future research in RLBench or similar robotics environments.

TABLE A4: Impacts of sampling strategy and timestep on RLBench models. The original data-loader of RVT-2 is adopted from prior works, including RVT, PerAct, and ARM. However, this implementation has been noted by researchers as being confusing (discussion, comment). Moreover, as detailed in Appendix D, the original data-loader randomizes the timestep of training samples, an unintended behavior by the previous authors. In contrast, we propose a simplified yet equally effective approach that only samples keyframes, which enables the study of the impacts of timestep and provides a solid foundation for future research.

Sampling strategy	Timestep (train)	Success rate	Model
Original	Randomized	81.4	RVT-2
	None	77	
Keyframes Only (Ours)	None	81.6	
	Correct	74.1	
	None	81.6	ARP
	Correct	77.8	