

Training-Time Action Conditioning for Efficient Real-Time Chunking

Kevin Black Allen Z. Ren Michael Equi Sergey Levine
Physical Intelligence

Abstract—Real-time chunking (RTC) enables vision-language-action models (VLAs) to generate smooth, reactive robot trajectories by asynchronously predicting action chunks and conditioning on previously committed actions via inference-time inpainting. However, this inpainting method introduces computational overhead that increases inference latency. In this work, we propose a simple alternative: simulating inference delay at training time and conditioning on action prefixes directly, eliminating any inference-time overhead. Our method requires no modifications to the model architecture or robot runtime, and can be implemented with only a few additional lines of code. In simulated experiments, we find that training-time RTC outperforms inference-time RTC at higher inference delays. In real-world experiments on box building and espresso making tasks with the $\pi_{0.6}$ VLA, we demonstrate that training-time RTC maintains both task performance and speed parity with inference-time RTC while being computationally cheaper. Our results suggest that training-time action conditioning is a practical drop-in replacement for inference-time inpainting in real-time robot control.

I. INTRODUCTION

Unlike chatbots or search engines, embodied agents must operate in real time. The feedback loop between an agent’s actions and its environment necessitates reactivity — like a human athlete, an agent cannot simply “stop and think” while the outside world changes. However, the ever-increasing size of frontier models makes this more and more difficult. Nowhere is this more evident than in the domain of robot learning, where vision-language-action models (VLAs) consisting of billions of parameters have increasingly been used to control robots at high frequencies to accomplish dexterous tasks. Producing smooth yet reactive trajectories when the model inference latency is in the tens to hundreds of milliseconds is no small challenge.

Real-time chunking (RTC; [5]) presents an approach to this problem that combines action chunking [9, 27], flow matching [13], and inference-time inpainting [18, 21]. In RTC, action chunks are predicted asynchronously — the next chunk is generated while the current one is still executing. To ensure continuity between chunks, each generation is conditioned on a frozen *prefix* of previously predicted actions, inpainting the rest. However, the inference-time inpainting method used by RTC introduces additional computational overhead — and hence latency — that somewhat defeats the purpose of a real-time execution framework. Empirically, we also find that inference-time inpainting is fundamentally limited in its ability to handle high inference delays.

In this work, we augment RTC with an inpainting method that simulates inference delay at training time and eliminates any inference-time computational overhead. Our method works

as a drop-in replacement for inference-time RTC: it requires no modifications to the model architecture or the robot runtime, and can be implemented with only a few additional lines of code. On simulated benchmarks, we find that training-time RTC outperforms inference-time RTC at higher inference delays. In the real world, we demonstrate that training-time RTC can be successfully added by fine-tuning a base model that was *not* pre-trained with action prefix conditioning. By applying training-time RTC to the $\pi_{0.6}$ VLA [24], we show improved performance over inference-time RTC on two highly complex tasks: box building and espresso making.

II. RELATED WORK

Action chunking and VLAs. Action chunking [9, 26] is the de facto standard in end-to-end imitation learning for visuomotor control. Recently, augmenting vision-language models (VLMs) to produce action chunks has demonstrated great success in robot manipulation, giving rise to vision-language-action models (VLAs) [4, 6–8, 10–12, 14, 17, 28, 29]. Subsequently, a plethora of methods have emerged to address the tension between large VLAs and high-frequency control. For example, Gemini Robotics [23] and GR0OT [3] employ hierarchical VLA designs where the model is split into a heavyweight System 2 (high-level planning) and lightweight System 1 (low-level action generation) component. MiniVLA [2] and SmolVLA [20] present VLA architectures that are altogether faster and more efficient than most designs, making inference at the edge more feasible. These contributions are orthogonal to ours, and come with their own tradeoffs (e.g., modified network architectures and training recipes).

Real-time execution of VLAs. The most closely related prior work is real-time chunking (RTC; [5]), which introduces an asynchronous execution framework that serves as a foundation for this work. Also related is SmolVLA [20], which presents an asynchronous execution algorithm that is similar to that of RTC; however, SmolVLA does not solve the inter-chunk discontinuity problem, which leads to out-of-distribution “jerks” between chunks. Concurrently to this work, A2C2 [19] and VLASH [22] both solve the discontinuity problem by adding a lightweight correction head and by conditioning on a single future action, respectively. In contrast to VLASH, we condition on a full prefix of future actions.

III. PRELIMINARIES

We use the same problem formulation as RTC [5]: we begin with an action chunking policy denoted by $p(\mathbf{A}_t|\mathbf{o}_t)$, where

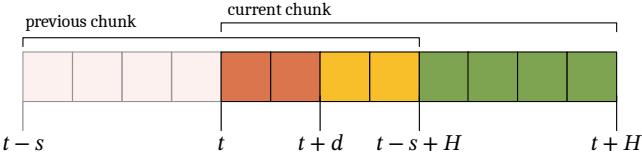


Fig. 1: A diagram illustrating two overlapping action chunks. The d actions between t and $t+d$, taken from the previous chunk, are the action prefix (red). From the diagram, we can easily see that we must satisfy the constraint $t+d \leq t-s+H \rightarrow d \leq H-s$ to have a valid action prefix. Note that inference-time RTC uses all $H-s$ overlapping actions (red and yellow) to guide the generation of the current chunk, whereas training-time RTC only uses the first d actions (red).

$\mathbf{A}_t = [\mathbf{a}_t, \mathbf{a}_{t+1}, \dots, \mathbf{a}_{t+H-1}]$ is a chunk of future actions, \mathbf{o}_t is an observation, and t indicates a controller timestep. We call H the *prediction horizon*, and at inference time, we roll out each chunk for $s \leq H$ timesteps, where s is the *execution horizon*.

To account for model inference, we define the quantity d to be the inference delay in units of controller timesteps. If inference begins at step t , then the resulting action chunk will not be available until step $t+d$, and so the first d actions cannot actually be executed. However, so long as $d \leq H-s$, these first d timesteps will correspond to actions from the previous chunk that can be executed in the meantime. We call these d actions from the previous chunk that overlap with the current chunk the *action prefix* (see Figure 1).

We consider policies trained with conditional flow matching [13], which minimizes the following loss:

$$\mathbf{A}_t^\tau = \tau \mathbf{A}_t + (1-\tau) \epsilon \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (1)$$

$$\mathcal{L}(\theta) = \mathbb{E} \|\mathbf{v}_\theta(\mathbf{A}_t^\tau, \mathbf{o}_t, \tau) - (\epsilon - \mathbf{A}_t)\|^2 \quad (2)$$

where \mathbf{v}_θ is a neural network and τ denotes the flow matching timestep. At inference time, \mathbf{v}_θ can be integrated from $\tau=0$ to 1 to produce samples from the dataset distribution $p(\mathbf{A}_t | \mathbf{o}_t)$.

IV. TRAINING-TIME ACTION CONDITIONING

Inference-time RTC [5] conditions the policy on the action prefix (Figure 1, red) using a inference-time inpainting method based on pseudoinverse guidance [18, 21]. For improved continuity between chunks, inference-time RTC additionally conditions on *all* overlapping actions, using exponentially decreasing weights for actions beyond the prefix (Figure 1, yellow). In RTC, this is referred to as “soft masking”. While pseudoinverse guidance affords great flexibility — enabling soft masking — it also requires computing a vector-Jacobian product (using backpropagation) during each denoising step.

The core insight of this work is that we can condition the policy on action prefixes at training time by simulating inference delay. While this does not afford the same flexibility as inference-time inpainting, it eliminates the computational overhead. Formally, we can learn $p(\mathbf{A}_{t+d:H} | \mathbf{o}_t, \mathbf{A}_{t:t+d})$, where $\mathbf{A}_{t:t+d}$ is an action prefix (Figure 1, red) and $\mathbf{A}_{t+d:H}$ is an action postfix (Figure 1, yellow and green), both taken from the same ground-truth action chunk. Implementing this for most standard policy architectures only requires 3 minimal changes:

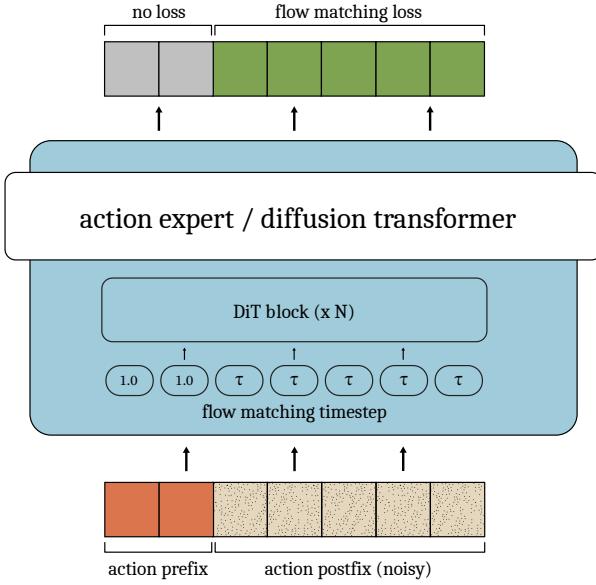


Fig. 2: An illustration of our conditioning architecture, as applied to a standard diffusion transformer such as the $\pi_{0.6}$ action expert. We always feed in ground-truth, non-noisy prefix actions, while learning to denoise the postfix actions. The flow matching timestep differs between tokens, which indicates the inference delay to the model.

- 1) Modify the model architecture to allow for a different flow matching timestep for each action timestep. For a diffusion-transformer-like architecture [16], which uses adaLN-zero conditioning for the flow matching timestep, this is trivial — simply allow the scale, shift, and gate to differ between tokens. This does not change the number of learnable parameters.
- 2) Use ground-truth, non-noisy actions for the prefix, and set the corresponding flow matching timesteps to 1. Do not change anything for the postfix. This conditions the model on the ground-truth action prefix while using it to denoise only the postfix.
- 3) Mask the loss function so that loss is only computed on outputs corresponding to the postfix.

See Figure 2 for an illustration of this conditioning scheme as applied to a standard diffusion-transformer-like architecture (e.g., the $\pi_{0.6}$ action expert). See Algorithm 1 for Python code fully implementing loss calculation and action generation. In practice, since we do not know the exact inference delay ahead of time (and inference delays in the real world may vary), we sample d randomly during training.

With these modifications, action generation takes as input an action prefix $\mathbf{A}_{t:t+d}$ and the delay itself d and produces as output an action postfix $\mathbf{A}_{t+d:H}$. As such, it adheres to the same interface as the action generation component of inference-time RTC (see [5], Algorithm 1) and thus acts as a seamless drop-in replacement.

V. EXPERIMENTS

In our experiments, we aim to compare training-time RTC to inference-time RTC, as well as to naive synchronous and

asynchronous baselines. Our simulated experiments use the same dynamic Kinetix [15] benchmark as RTC (see [5] for details). Our real-world experiments build on the $\pi_{0.6}$ base model [24], and include two precise and challenging tasks: box building and espresso making. We use the same experimental setup as $\pi_{0.6}^*$ [1].

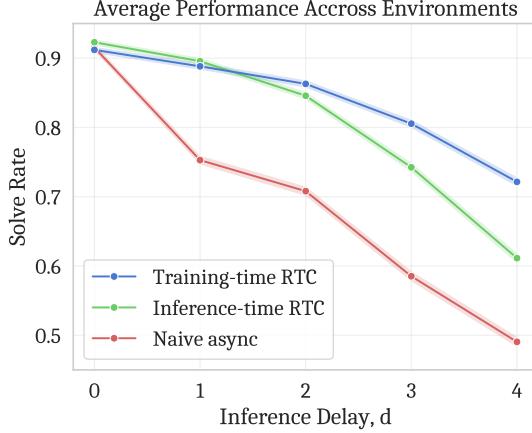


Fig. 3: Simulated results: inference delay vs. solve rate with a fixed execution horizon of $s = \max(d, 1)$. Training-time RTC performs better than inference-time RTC at inference delays of 2 or higher. Each data point represents 2048 trials, and 95% Wilson score intervals are shaded in.

A. Simulated Results

In the dynamic Kinetix benchmark, following RTC [5], we train action chunking flow policies with a prediction horizon of $H = 8$ and a 4-layer MLP-Mixer [25] architecture for 32 epochs on data generated by a mixture of expert policies. We report binary success rates with 2048 rollouts per data point and test delays between 0 (fully closed-loop) and 4 (the maximum supported when $H = 8$). Naive asynchronous and inference-time RTC both use the same checkpoint, which is trained normally *without* action prefix conditioning for 32 epochs.

For training-time RTC, we resume training from the 24th epoch and fine-tune for 8 epochs with action prefix conditioning. We do this so that all methods are matched in training compute. We sample delays from $\{0, 1, 2, 3, 4\}$ with exponentially decreasing weights, as we found that higher delays need less training supervision. Better results could likely be obtained by spending more training compute training individual checkpoints for each delay.

The results are presented in Figure 3. We find that training-time RTC outperforms inference-time RTC at inference delays of 2 and higher — with the gap significantly widening as the delay increases. This is likely because, as the size of the prefix grows, the inpainting algorithm has to “work harder” to produce a consistent postfix. In these cases, the training-time algorithm is more robust than the pure inference-time algorithm, which relies on a linearization obtained from the Jacobian of the model. Training-time RTC performs very marginally worse at delays of 1 and 0, likely because training-time RTC does not always receive training supervision for every action — i.e.,

slightly less training compute is spent learning to generate the first and second actions.

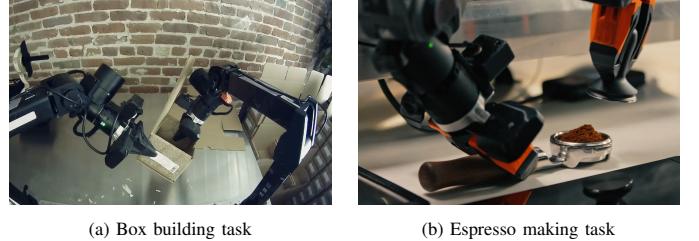


Fig. 4: Real-world evaluation tasks: building a cardboard box and making espresso (including grinding, tamping, extracting, and pouring).

B. Real-World Results

In our real-world experiments, we use the $\pi_{0.6}$ base model [24] and test on the espresso making and box building tasks from $\pi_{0.6}^*$ [1]; see Figure 4 for an illustration. As in the simulated experiments, we use the same checkpoint for the synchronous baseline and inference-time RTC, and train a second checkpoint with action prefix conditioning for training-time RTC. Both checkpoints are fine-tuned from the base model on the target task for 8,000 gradient steps with a batch size of 512. We sample delays uniformly between 0 and 10 during training, which supports a maximum latency of 200ms on a 50Hz robot. During evaluations, we perform inference on a remote H100 server with 5 denoising steps, averaging 108ms of end-to-end latency for training-time RTC ($d \approx 5$) and 135ms for inference-time RTC ($d \approx 7$).

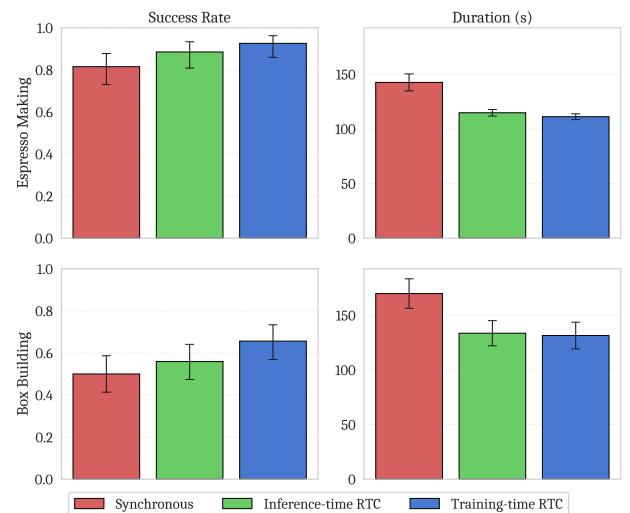


Fig. 5: Real-world results: success rate and duration for espresso making and box building. Training-time and inference-time RTC perform similarly, while both improving speed over synchronous inference. Error bars represent 68% Wilson score intervals for success rate and ± 1 SEM for duration.

The results are presented in Figure 5. We find that training-time RTC maintains both performance and speed parity with inference-time RTC without any computational overhead. Both variants of RTC clearly improve speed over the synchronous

inference baseline, which exhibits visible pauses in between chunks.

VI. DISCUSSION AND FUTURE WORK

In this work, we have presented a simple and effective drop-in replacement for real-time chunking (RTC) that elides any inference-time computational overhead by adding a small amount of additional training compute. Our method requires no modifications to the model architecture or the robot runtime, and can be implemented with only a few additional lines of code. Our simulated experiments show that training-time RTC outperforms inference-time RTC at higher inference delays, while our real-world experiments show that training-time RTC maintains both performance and speed parity with inference-time RTC without any computational overhead.

However, training-time RTC is fundamentally less flexible than inference-time RTC; it only supports conditioning on a “hard” action prefix corresponding to the inference delay, whereas inference-time RTC can “softly” incorporate additional actions beyond the prefix. Additionally, training-time RTC requires carefully choosing the distribution of delays to simulate at training time based on the expected inference latency. We look forward to future work that can address these limitations and incorporate the best of both worlds.

VII. ACKNOWLEDGMENTS

We thank Laura Smith for developing the espresso making tasks, and helping with early evaluations. We thank Brian Ichter for feedback on the manuscript. As always, we thank our entire team of robot operators for their contributions to data collection and evaluations.

REFERENCES

- [1] Ali Amin, Raichel Aniceto, Ashwin Balakrishna, Kevin Black, Ken Conley, Grace Connors, James Darpinian, Karan Dhabalia, Jared DiCarlo, Danny Driess, et al. $\pi_{0.6}^*$: a vla that learns from experience. *arXiv preprint arXiv:2511.14759*, 2025.
- [2] Suneel Belkhale and Dorsa Sadigh. Minivila: A better vla with a smaller footprint, 2024. URL <https://github.com/Stanford-ILIA/opencvla-mini>.
- [3] Johan Bjorck, Fernando Castañeda, Nikita Cherniadov, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.
- [4] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolò Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0 : A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.
- [5] Kevin Black, Manuel Y Galliker, and Sergey Levine. Real-time execution of action chunking flow policies. *arXiv preprint arXiv:2506.07339*, 2025.
- [6] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alex Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Lisa Lee, Tsang-Wei Edward Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspair Singh, Anikait Singh, Radu Soricu, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *arXiv preprint arXiv:2307.15818*, 2023.
- [7] Chi-Lam Cheang, Guangzeng Chen, Ya Jing, Tao Kong, Hang Li, Yifeng Li, Yuxiao Liu, Hongtao Wu, Jiafeng Xu, Yichu Yang, Hanbo Zhang, and Minzhao Zhu. Gr-2: A generative video-language-action model with web-scale knowledge for robot manipulation. *arXiv preprint arXiv:2410.06158*, 2024.
- [8] An-Chieh Cheng, Yandong Ji, Zhaojing Yang, Xueyan Zou, Jan Kautz, Erdem Biyik, Hongxu Yin, Sifei Liu, and Xiaolong Wang. NaVILA: Legged Robot Vision-Language-Action Model for Navigation. *arXiv preprint arXiv:2412.04453*, 2024.
- [9] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, page 02783649241273668, 2023.
- [10] OX-Embodiment Collaboration, A Padalkar, A Pooley, A Jain, A Bewley, A Herzog, A Irpan, A Khazatsky, A Rai, A Singh, et al. Open X-Embodiment: Robotic learning datasets and RT-X models. *arXiv preprint arXiv:2310.08864*, 1(2), 2023.
- [11] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolò Fusai, et al. $\pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.
- [12] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvila: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.
- [13] Yaron Lipman, Ricky TQ Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. *arXiv preprint arXiv:2210.02747*, 2022.
- [14] Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. Rdt-1b: a diffusion foundation model for bimanual manipulation. *arXiv preprint arXiv:2410.07864*, 2024.
- [15] Michael Matthews, Michael Beukman, Chris Lu, and Jakob Foerster. Kinetix: Investigating the training of

- general agents through open-ended physics-based control tasks. *arXiv preprint arXiv:2410.23208*, 2024.
- [16] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4195–4205, 2023.
- [17] Karl Pertsch, Kyle Stachowicz, Brian Ichter, Danny Driess, Suraj Nair, Quan Vuong, Oier Mees, Chelsea Finn, and Sergey Levine. Fast: Efficient action tokenization for vision-language-action models. *arXiv preprint arXiv:2501.09747*, 2025.
- [18] Ashwini Pokle, Matthew J Muckley, Ricky TQ Chen, and Brian Karrer. Training-free linear image inverses via flows. *arXiv preprint arXiv:2310.04432*, 2023.
- [19] Kohei Sendai, Maxime Alvarez, Tatsuya Matsushima, Yutaka Matsuo, and Yusuke Iwasawa. Leave no observation behind: Real-time correction for vla action chunks. *arXiv preprint arXiv:2509.23224*, 2025.
- [20] Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. Smolvla: A vision-language-action model for affordable and efficient robotics. *arXiv preprint arXiv:2506.01844*, 2025.
- [21] Jiaming Song, Arash Vahdat, Morteza Mardani, and Jan Kautz. Pseudoinverse-guided diffusion models for inverse problems. In *International Conference on Learning Representations*, 2023.
- [22] Jiaming Tang, Yufei Sun, Yilong Zhao, Shang Yang, Yujun Lin, Zhuoyang Zhang, James Hou, Yao Lu, Zhijian Liu, and Song Han. Vlash: Real-time vlas via future-state-aware asynchronous inference. *arXiv preprint arXiv:2512.01031*, 2025.
- [23] Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, et al. Gemini robotics: Bringing ai into the physical world. *arXiv preprint arXiv:2503.20020*, 2025.
- [24] Physical Intelligence team. $\pi_{0.6}$ model card. 2025.
- [25] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems*, 34: 24261–24272, 2021.
- [26] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023.
- [27] Tony Z Zhao, Jonathan Tompson, Danny Driess, Pete Florence, Kamyar Ghasemipour, Chelsea Finn, and Ayzaan Wahid. Aloha unleashed: A simple recipe for robot dexterity. *arXiv preprint arXiv:2410.13126*, 2024.
- [28] Haoyu Zhen, Xiaowen Qiu, Peihao Chen, Jincheng Yang, Xin Yan, Yilun Du, Yining Hong, and Chuang Gan. 3d-vla: 3d vision-language-action generative world model. *arXiv preprint arXiv:2403.09631*, 2024.
- [29] Ruijie Zheng, Yongyuan Liang, Shuaiyi Huang, Jianfeng Gao, Hal Daumé III, Andrey Kolobov, Furong Huang, and Jianwei Yang. Tracevla: Visual trace prompting enhances spatial-temporal awareness for generalist robotic policies. *arXiv preprint arXiv:2412.10345*, 2024.

Algorithm 1 Python code implementing the loss and sampling functions for training-time action conditioning. Differences from standard flow matching code are highlighted in red.

```
import jax
import jax.numpy as jnp

def compute_loss(rng, model, observation, action_chunk, max_delay):
    b, ah, ad = action_chunk.shape # (batch_size, action_horizon, action_dim)
    noise_rng, time_rng, delay_rng = jax.random.split(rng)
    time = jax.random.uniform(time_rng, (b,))
    noise = jax.random.normal(noise_rng, (b, ah, ad))
    # sample delays from some distribution of choice:
    # here, we use Unif[0, max_delay), as in our real-world experiments
    delay = jax.random.randint(delay_rng, (b,), 0, max_delay)

    # set time to 1.0 for the action prefix
    # time becomes shape (batch_size, action_horizon)
    prefix_mask = jnp.arange(ah)[None, :] < delay[:, None]
    time = jnp.where(prefix_mask, 1.0, time[:, None])

    # compute the noisy action postfix and run the model
    x_t = time[:, :, None] * action_chunk + (1 - time[:, :, None]) * noise
    pred_v_t = model(observation, x_t, time)
    loss = (pred_v_t - (action_chunk - noise))**2

    # compute the loss on the postfix only
    postfix_mask = jnp.logical_not(prefix_mask)[:, :, None]
    loss = jnp.sum(loss * postfix_mask) / (jnp.sum(postfix_mask) + 1e-8)
    return loss

def sample_actions(rng, model, observation, action_prefix, delay, num_steps):
    # assume action_prefix is padded to (batch_size, action_horizon, action_dim),
    # but only the first delay actions are valid
    b, ah, ad = action_prefix.shape
    x_t = jax.random.normal(rng, (b, ah, ad))
    time = 0.0
    dt = 1 / num_steps
    prefix_mask = jnp.arange(ah)[None, :] < delay

    for _ in range(num_steps):
        x_t = jnp.where(prefix_mask[:, :, None], action_prefix, x_t)
        time_masked = jnp.where(prefix_mask, 1.0, time)
        v_t = model(observation, x_t, time_masked)
        x_t = x_t + dt * v_t
        time = time + dt

    return x_t
```
