EDGI: Equivariant Diffusion for Planning with Embodied Agents

Supplementary Material

Anonymous Author(s)
Affiliation
Address
email

A Architecture details

On a high level, EDGI follows Diffuser [1]. In the following, we will describe the key difference: our $\mathrm{SE}(3) \times \mathbb{Z} \times \mathrm{S}_n$-equivariant architecture for the diffusion model.

Overall architecture. We illustrate the architecture in Fig. 2 in the main paper. After converting the input data in our internal representation (see Sec. 3.2 in the main paper), the data is processed with an equivariant U-net with four levels. At each level, we process the hidden state with two residual standard blocks, before downsampling (in the downward pass) or upsampling (in the upward pass).

Residual standard block. The main processing unit of our architecture processes the current hidden state with an equivariant block consisting of a temporal layer, an object layer, a normalization layer, and a geometric layer. In parallel, the context information (an embedding of diffusion time and a conditioning mask) is processed with a context block. The hidden state is added to the output of the context block and processes with another equivariant block. Finally, we process the data with a linear attention layer over time. This whole pipeline consists of an equivariant block, a context block, and another equivariant block is residual (the inputs are added to the outputs).

Temporal layers. Temporal layers consist of one-dimensional convolutions without bias along the time dimension. We use a kernel size of 5.

Normalization layers. We use a simple equivariant normalization layer that for each batch element rescales the entire tensor w_{toc} to unit variance. This is essentially an equivariant version of LayerNorm. The difference is that our normalization layer does not shift the inputs to zero means, as that would break equivariance with respect to $\mathrm{SO}(3)$.

Geometric layers. In the geometric layers, the input state is split into scalar and vector components. The vector components are linearly transformed to reduce the number of channels to 16. We then construct all $\mathrm{SO}(3)$ invariants from these 16 vectors by taking pairwise inner products and concatenating them with the scalar inputs. This set of scalars is processed with two MLPs, each consisting of two hidden layers and ReLU nonlinearities. The MLPs output the scalar outputs and coefficients for a linear map between the vector inputs and the vector outputs, respectively. Finally, there is a residual connection that adds the scalar and vector inputs to the outputs.

Linear attention over time. To match the architecture used by Janner et al. [1] as closely as possible, we follow their choice of adding another residual linear attention over time at the end of each level in the U-net. We make the linear attention mechanism equivariant by computing the attention weights as

Context blocks. The embeddings of diffusion time and conditioning information are processed with a Mish nonlinearity and a linear layer, like in Janner et al. [1]. Finally, we embed them in our internal representation by zero-padding the resulting tensor.

Upsampling and downsampling. During the downsampling path, there is a final temporal layer that implements temporal downsampling and increases the number of channels by a factor of two. Conversely, during the upsampling path, we use a temporal layer for temporal upsampling and a reduction of the number of channels.

B Navigation experiments

We introduce a new navigation environment. The scene consists of a spherical agent navigating a plane populated with a goal state and $n = 10$ spherical obstacles. At the beginning of every episode, the agent position, agent velocity, obstacle positions, and goal position are initialized randomly (in a rotation-invariant way). We simulate the environment dynamics with PyBullet [2].

Offline dataset. To obtain expert trajectories, we train a TD3 [3] agent in the implementation by Raffin et al. [4] for 10^7 steps with default hyperparameters on this environment. We generate 10^5 trajectories for our offline dataset.

State. The state contains the agent position, agent velocity, goal position, and obstacle positions.

Actions. The action space is two-dimensional and specifies a force acting on the agent.

Rewards. At each time step, the agent receives a reward equal to the negative Euclidean distance to the goal state. In addition, a penalty of -0.1 is added to the reward if the agent touches any of the obstacles. Finally, there is an additional control cost equal to -10^3 times the force acting on the agent. We affinely normalize the rewards such that a normalized reward of 0 corresponds to that achieved by a random policy and a normalized reward of 100 corresponds to the expert policy.

C Kuka experiments

We use the object manipulation environments and tasks from Janner et al. [1], please see that work for details on the environment. In our experiments, we consider three tasks: unconditional stacking, conditional stacking, and block rearrangement. For a fair comparison, we re-implement the Diffuser algorithm while making bug fixes in the codebase of Janner et al. [1], which mainly included properly resetting the environment.

State. We experiment with two parameterizations of the Kuka environment state. For the Diffuser baseline, we use the original 39-dimensional parameterization from Janner et al. [1].

For our EDGI, we need to parameterize the system in terms of $\mathrm{SE}(3) \times \mathbb{Z} \times \mathrm{S}_n$ representations. We, therefore, describe the robot and block orientations with $\mathrm{SO}(3)$ vectors as follows. Originally, the robot state is specified through a collection of joint angles. One of these encodes the rotation of the base along the vertical z-axis. We choose to represent this angle as a ρ_1 vector in the xy-plane. In addition, we add the gravity direction (the z-axis itself) as another ρ_1 vector, which is also the normal direction of the table on which the objects rest. Combined, these vectors define the pose of the base of the robot arm. Rotating gravity direction, and the robot and object pose by $\mathrm{SO}(3)$ can be interpreted as a passive coordinate transformation, or as an active rotation of the entire scene, including gravity. As the laws of physics are invariant to this transformation, this is a valid symmetry of the problem.

The n objects can be translated and rotated. Their pose is thus given by a translation $t \in \mathbb{R}^3$ and rotation in $r \in \mathrm{SO}(3)$ relative to a reference pose. The translation transforms by a global rotation $g \in \mathrm{SO}(3)$ as a vector via representation ρ_1. The rotational pose transforms by left multiplication $r \mapsto gr$. The $\mathrm{SO}(3)$ pose is not a Euclidean space, but a non-trivial manifold. Even though diffusion on manifolds is possible [5, 6], we simplify the problem by embedding the pose in a Euclidean space. This is done by picking the first two columns of the pose rotation matrix $r \in \mathrm{SO}(3)$. These columns each transform again as a vector with representation ρ_1. This forms an equivariant embedding $\iota : \mathrm{SO}(3) \hookrightarrow \mathbb{R}^{2\times 3}$, whose image is two orthogonal 3-vectors of unit norm. Via the Gram-Schmidt procedure, we can define an equivariant map $\pi : \mathbb{R}^{2\times 3} \to \mathrm{SO}(3)$ (defined almost everywhere), that is a left inverse to the embedding: $\pi \circ \iota = \mathrm{id}_{\mathrm{SO}(3)}$. Combining with the translation, the roto-translational pose of each object is thus embedded as three ρ_1 vectors.

We also tested the performance of the baseline Diffuser method on this reparameterization of the state but found worse results.

Hyperparameters. We also follow the choices of Janner et al. [1], except that we experiment with a linear noise schedule as an alternative to the cosine schedule they use. For each model and each dataset, we train the diffusion model with both noise schedules and report the better of the two results.

References

[1] Michael Janner, Yilun Du, Joshua B Tenenbaum, and Sergey Levine. Planning with diffusion for flexible behavior synthesis. *arXiv preprint arXiv:2205.09991*, 2022. (Cited on pages 1, 2, and 3)

[2] Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2019. (Cited on page 2)

[3] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018. (Cited on page 2)

[4] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html. (Cited on page 2)

[5] Valentin De Bortoli, Emile Mathieu, Michael John Hutchinson, James Thornton, Yee Whye Teh, and Arnaud Doucet. Riemannian Score-Based generative modelling. October 2022. URL https://openreview.net/pdf?id=oDRQGo8I7P. (Cited on page 2)

[6] Chin-Wei Huang, Milad Aghajohari, Avishek Joey Bose, Prakash Panangaden, and Aaron Courville. Riemannian diffusion models. August 2022. URL http://arxiv.org/abs/2208.07949. (Cited on page 2)