BaB-ND: Long-Horizon Motion Planning with Branch-and-Bound and Neural Dynamics

Keyi Shen[1*], **Jiangwei Yu**[1*], **Huan Zhang**[1], **Yunzhu Li**[2]
[1]University of Illinois Urbana-Champaign [2]Columbia University

Abstract

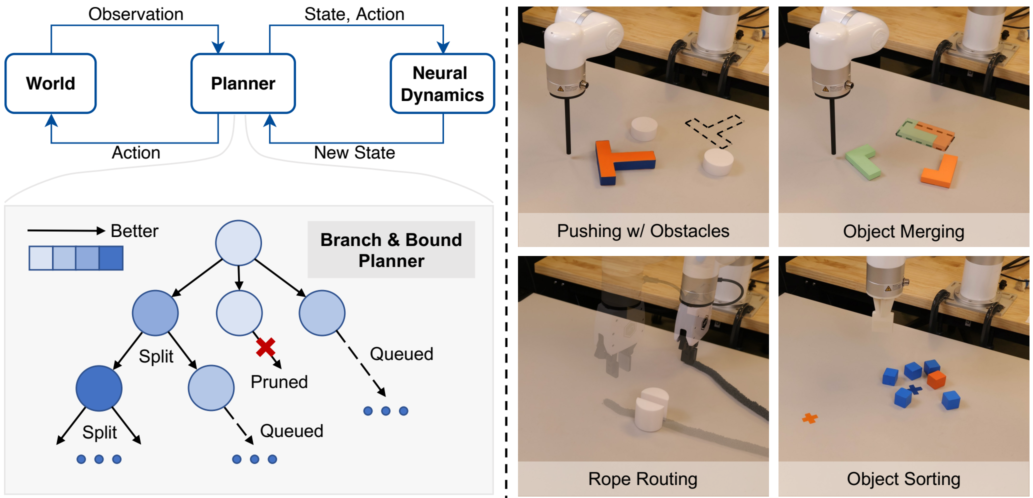
Neural-network-based dynamics models learned from observational data have shown strong predictive capabilities for scene dynamics in robotic manipulation tasks. However, their inherent non-linearity presents significant challenges for effective planning. Current planning methods, often dependent on extensive sampling or local gradient descent, struggle with long-horizon motion planning tasks involving complex contact events. In this paper, we present a GPU-accelerated branch-and-bound (BaB) framework for motion planning in manipulation tasks that require trajectory optimization over neural dynamics models. Our approach employs a specialized branching heuristic to divide the search space into sub-domains and applies a modified bound propagation method, inspired by the state-of-the-art neural network verifier α,β-CROWN, to efficiently estimate objective bounds within these sub-domains. The branching process guides planning effectively, while the bounding process strategically reduces the search space. Our framework achieves superior planning performance, generating high-quality state-action trajectories and surpassing existing methods in challenging, contact-rich manipulation tasks such as non-prehensile planar pushing with obstacles, object sorting, and rope routing in both simulated and real-world settings. Furthermore, our framework supports various neural network architectures, ranging from simple multilayer perceptrons to advanced graph neural dynamics models, and scales efficiently with different model sizes.

1 Introduction

Learning-based predictive models using neural networks reduce the need for full-state estimation and have proven effective across a variety of robotics-related planning tasks in both simulations (Li et al., 2018; Hafner et al., 2019c; Schrittwieser et al., 2020; Seo et al., 2023) and real-world settings (Lenz et al., 2015; Finn & Levine, 2017; Tian et al., 2019; Lee et al., 2020; Manuelli et al., 2020; Nagabandi et al., 2020; Lin et al., 2021; Huang et al., 2022; Driess et al., 2023; Wu et al., 2023; Shi et al., 2023). While neural dynamics models can effectively predict scene evolution under varying initial conditions and input actions, their inherent non-linearity presents challenges for traditional model-based planning algorithms, particularly in long-horizon scenarios.

To address these challenges, the community has developed a range of approaches. Sampling-based methods such as the Cross-Entropy Method (CEM) (Rubinstein & Kroese, 2013) and Model Predictive Path Integral (MPPI) (Williams et al., 2017) have gained popularity in manipulation tasks (Lowrey et al., 2018; Manuelli et al., 2020; Nagabandi et al., 2020; Wang et al., 2023) due to their flexibility, compatibility with neural dynamics models, and strong GPU support. However, their performance in more complex, higher-dimensional planning problems is limited and still requires further theoretical analysis (Yi et al., 2024). Alternatively, more principled optimization approaches, such as Mixed-Integer Programming (MIP), have been applied to planning problems using sparsified neural dynamics models with ReLU activations (Liu et al., 2023). Despite achieving global optimality and better closed-loop control performance, MIP is inefficient and struggles to scale to large neural networks, limiting its ability to handle larger-scale planning problems.

In this work, we introduce a branch-and-bound (BaB) based framework that achieves stronger performance on complex planning problems than sampling-based methods, while also scaling to large neural dynamics models that are intractable for MIP-based approaches. Our framework is inspired by the success of BaB in neural network verification (Bunel et al., 2018; 2020b; Palma et al.,



(a) Planning with BaB-ND (b) Benchmark tasks

Figure 1: **Framework overview.** (a) Our framework takes scene observations and applies a branch-and-bound (BaB) method to generate robot trajectories using the neural dynamics model (ND). The BaB-ND planner constructs a search tree by branching the problem into sub-domains and then systematically searches only in promising sub-domains by evaluating nodes with a bounding procedure. (b) BaB-ND demonstrates superior long-horizon planning performance compared to existing sampling-based methods and achieves better closed-loop control in real-world scenarios. We evaluate our framework on various complex planning tasks, including non-prehensile planar pushing with obstacles, object merging, rope routing, and object pile sorting.

2021), which tackles challenging optimization objectives involving neural networks. State-of-the-art neural network verifiers such as α,β-CROWN (Xu et al., 2021; Wang et al., 2021; Zhang et al., 2022a), utilize BaB alongside bound propagation methods (Zhang et al., 2018; Salman et al., 2019), demonstrating impressive strength and scalability in verification tasks, far surpassing MIP-based approaches (Tjeng et al., 2019; Anderson et al., 2020). However, unlike neural network verification, which only requires finding a lower bound of the objective, model-based planning demands high-quality feasible solutions (i.e., planned state-action trajectories). Thus, significant adaptation and specialization are necessary for BaB-based approaches to effectively solve planning problems.

Our framework, BaB-ND (Figure 1.a), divides the action space into smaller sub-domains through a novel branching heuristic (*branching*), estimates objective bounds using a modified bound propagation procedure to prune sub-domains that cannot yield better solutions (*bounding*), and focuses searches on the most promising sub-domains (*searching*). We evaluate our approach on contact-rich manipulation tasks that require long-horizon planning with non-smooth objectives, non-convex feasible regions (with obstacles), long action sequences, and diverse neural dynamics model architectures (Figure 1.b). Our results demonstrate that BaB-ND consistently outperforms existing sampling-based methods by systematically and strategically exploring the action space, while also being significantly more efficient and scalable than MIP-based approaches by leveraging the inherent structure of neural networks and GPU support.

We make three key contributions: (1) We propose a general, widely applicable BaB-based framework for effective long-horizon motion planning over neural dynamics models. (2) Our framework introduces novel branching, bounding, and searching procedures, inspired by neural network verification algorithms but specifically adapted for planning over neural dynamics models. (3) We demonstrate the effectiveness, applicability, and scalability of our framework across a range of complex planning problems, including contact-rich manipulation tasks, the handling of deformable objects, and object piles, using diverse model architectures such as multilayer perceptrons and graph neural networks.

2 Branch-and-Bound for Planning With Neural Dynamics Models

Formulation. We formulate the planning problem as an optimization problem in Eq. 1, where c is the cost function, t_0 is the current time step, and H is the planning horizon. $\hat{x}_t$ is the (predicted) state at time step t, and the current state $\hat{x}_{t_0} = x_{t_0}$ is known. $u_t \in \{u \mid \underline{u} \le u \le \overline{u}\} \subset \mathbb{R}^k$ is the robot's action at each step. f_{dyn} is the neural dynamics model, which takes state and action at time t and predicts the next state $\hat{x}_{t+1}$. **The goal of the planning problem** is to find a set of optimal actions u_t that minimize the predefined cost:

$$\min_{\{u_t \in \mathcal{U}\}} \sum_{t=t_0}^{t_0+H} c(\hat{x}_t, u_t) \quad \text{s.t.} \quad \hat{x}_{t+1} = f_{dyn}(\hat{x}_t, u_t) \qquad \Longrightarrow \qquad \min_{\boldsymbol{u} \in \mathcal{C}} f(\boldsymbol{u}) \tag{1}$$

This problem is challenging because it is a non-convex function involving the neural dynamics model f_{dyn}. Note that we assume the neural dynamic model f_{dyn} is known. Please refer to sections D and E.2 for details about learning the neural dynamic model.

To simplify notations, we can substitute all constraints on $\hat{x}_{t+1}$ into the summed cost recursively, and further simplify the problem as a constrained optimization problem $\min_{\boldsymbol{u} \in \mathcal{C}} f(\boldsymbol{u})$ (Eq. 1). Here f is our final objective, a scalar function that absorbs the neural network f_{dyn} and the cost function summed in all H steps. $\boldsymbol{u} = \{u_{t_0:t_0+H}\} \in \mathcal{C}$ is the action sequence and $\mathcal{C} \subset \mathbb{R}^d$ is the entire input space with dimension with $d = kH$. We also flatten $\boldsymbol{u}$ as a vector containing actions for all time steps, and use $\boldsymbol{u}_j$ to denote a specific dimension. Our goal is to then find the optimal objective value f^* and its corresponding optimal action sequence $\boldsymbol{u}^*$.

Branch-and-bound on a 1D toy example. Our work proposes to solve the planning problem Eq. 1 using branch-and-bound. Before diving into technical details, we first provide a toy case of a non-convex neural network function $f(\boldsymbol{u})$ in 1D space ($k = H = 1, \mathcal{C} = [-1, 1]$) and illustrate how to use branch-and-bound to find f^*.

In Figure 2.1, we visualize the landscape of $f(\boldsymbol{u})$ with its optimal value f^*. Initially, we don't know f^* but we can sample the function at a few different locations (organ points). Although sampling (*searching*) often fails to discover the optimal f^* over $\mathcal{C} = [-1, 1]$, it gives an upper bound of f^* since any orange point has an objective greater than or equal to f^*. We denote $\overline{f}^*$ as the current best upper bound (orange dotted line).

In Figure 2.2, we split $\mathcal{C}$ into two sub-domains $\mathcal{C}_1$ and $\mathcal{C}_2$ (*branching*) and then estimate the lower bound of the objective with a linear function in both $\mathcal{C}_1$ and $\mathcal{C}_2$ (*bounding*). The key insight is if the lower bound in one sub-domain is larger than $\overline{f}^*$, then sampling from that sub-domain will not yield any better objective than $\overline{f}^*$ and we may discard that sub-domain to reduce the search space. In the example, $\mathcal{C}_1$ is discarded in Figure 2.3.

Then, in Figure 2.4, we only perform sampling in $\mathcal{C}_2$ with the same number of samples. *Searching* in the reduced space is promising to obtain a better objective and therefore $\overline{f}^*$ can be improved.

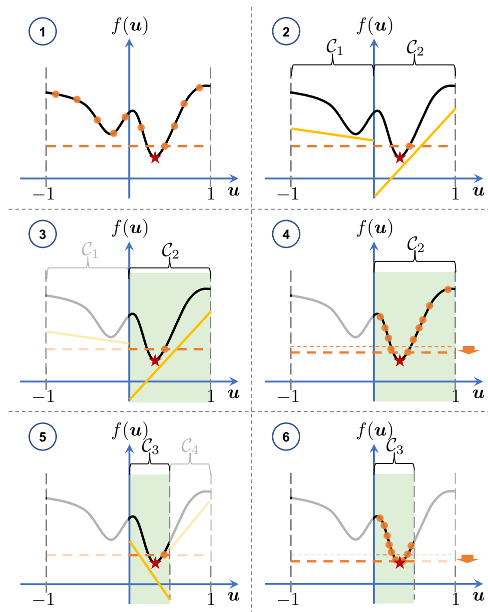


Figure 2: **Seeking f^* with Branch-and-Bound.** ① Sample on input space $\mathcal{C}$. ●: sampled points. ★: the optimal value f^*. – –: the current best upper bound of f^* from sampling. ② Branch $\mathcal{C}$ into $\mathcal{C}_1$ and $\mathcal{C}_2$. —: the linear lower bounds of f^* in sub-domains. ③ Discard $\mathcal{C}_1$ since its lower bound is larger than $\overline{f}^*$. ■: the remaining sub-domain to be searched. ④ Search on only $\mathcal{C}_2$ and upper bound of f^* is improved. - - - -: the previous upper bound. ⑤ Continue to branch $\mathcal{C}_2$ and bound on $\mathcal{C}_3$ and $\mathcal{C}_4$. ⑥ Search on $\mathcal{C}_3$. The upper bound approaches f^*.

We could repeat these procedures (*branching*, *bounding*, and *searching*) to reduce the sampling space and improve $\overline{f}^*$ as in Figure 2.5 and Figure 2.6. Finally, $\overline{f}^*$ will converge to f^*. This branch-and-bound method systematically partitions the input space and iteratively improves the objective. In practice, heuristics for branching, along with methods for bound estimation and solution search, are critical to the performance of branch and bound.

Methodology overview. We now discuss how to use the branch-and-bound (BaB) method to find high-quality actions for the neural dynamics planning problem presented as $\min_{\boldsymbol{u} \in \mathcal{C}} f(\boldsymbol{u})$ (Eq. 1). We define a *sub-problem* $\min_{\boldsymbol{u} \in \mathcal{C}_i} f(\boldsymbol{u})$ as minimizing $f(\boldsymbol{u})$ in a *sub-domain* $\mathcal{C}_i$, where $\mathcal{C}_i \subseteq \mathcal{C}$. Our algorithm, BaB-ND, involves three components: *branching* (Figure 3.b, Section 2.1), *bounding* (Figure 3.c, Section 2.2), and *searching* (Figure 3.d, Section 2.3).

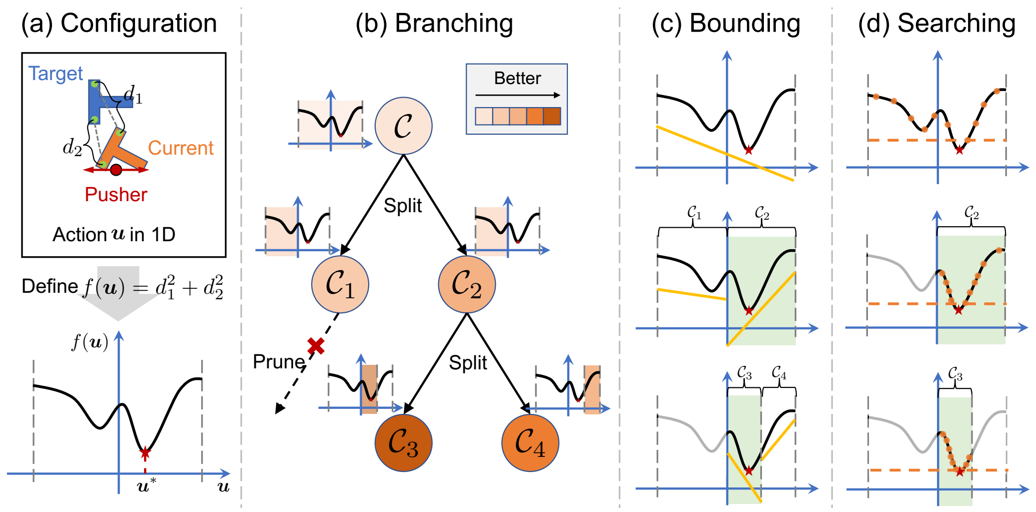


Figure 3: **Illustration of the branch and bound process.** (a) Configuration: we visualize a simplified case of pushing an object to approach the target with 1D action $\boldsymbol{u}$. We select two keypoints on the object and target and denote the distance as d_1 and d_2. Then we define our objective function $f(\boldsymbol{u})$ and seek $\boldsymbol{u}^*$ to minimize $f(\boldsymbol{u})$. (b) **Branching:** we iteratively construct the search tree by splitting, queuing, and even pruning nodes (sub-domains). In every iteration, only the most promising nodes are prioritized to split, cooperating with bounding and searching. (c) **Bounding:** In every sub-domain $\mathcal{C}_i$, we obtain the linear lower bound of $\underline{f}^*$ via bound propagation. (d) **Searching:** we search better solutions $\overline{f}^*$ on selected sub-domains. ▮ indicates the most promising sub-domain in every iteration. The search space becomes a smaller and smaller part of the original input domain $\mathcal{C}$ with better solutions found.

- *Branching* generates a partition $\{\mathcal{C}_i\}$ of some action space $\mathcal{C}$ such that $\bigcup_i \mathcal{C}_i = \mathcal{C}$, and it allows us to explore the solution space systematically.
- *Bounding* estimates the lower bounds of $f(\boldsymbol{u})$ on each sub-domains $\mathcal{C}_i$ (denoted as $\underline{f}^*_{\mathcal{C}_i}$). The lower bound can be used to prune useless domains and also guide the search for promising domains.
- *Searching* seeks good feasible solutions within each subdomain $\mathcal{C}_i$ and outputs the best objectives $\overline{f}^*_{\mathcal{C}_i}$. $\overline{f}^*_{\mathcal{C}_i}$ is an upper bound of f^*, as any feasible solution provides an upper bound for the optimal minimization objective f^*

We can always prune sub-domain $\mathcal{C}_j$ if its $\underline{f}^*_{\mathcal{C}_j} > \overline{f}^*$, where the best upper bound is defined as $\overline{f}^* := \min_i \overline{f}^*_{\mathcal{C}_i}$, since, in $\mathcal{C}_j$, there is no solution better than current best objective $\overline{f}^*$ among all sub-domains $\{\mathcal{C}_i\}$. The above procedure can be repeated many times, and each time during branching, a previously produced sub-domain $\mathcal{C}_i$ can be picked for further branching, bounding, and searching while the remaining sub-domains are stored in a set (denoted as $\mathbb{P}$). Our main algorithm is shown in Algorithm 1. Although this generic BaB framework has been used in neural network verifiers (Bunel et al., 2018; Wang et al., 2021), we now describe how to design the *branching*, *bounding*, and *searching* processes **specialized to the model-based planning setting**.

2.1 BRANCHING HEURISTICS FOR BAB-ND PLANNING

The efficiency of BaB heavily depends on the quality of branches. Hence, how to select promising sub-domains and how to split sub-domains are two essential questions in BaB, referring to `batch_pick_out`$(\mathbb{P}, n)$ and `batch_split` $(\{\mathcal{C}_i\})$ in Algorithm 1. Here we introduce our specialized branching heuristics to select and split sub-domains for seeking high-quality solutions.

Heuristic to select sub-domains to split. The function `batch_pick_out`$(\mathbb{P}, n)$ picks n most promising sub-domains for branching, based on their associated $\underline{f}^*_{\mathcal{C}_i}$ or $\overline{f}^*_{\mathcal{C}_i}$. The pickout process must balance exploitation (focusing on areas around good solutions) and exploration (investigating regions that have not been thoroughly explored). *First*, we sort sub-domains $\mathcal{C}_i$ by $\overline{f}^*_{\mathcal{C}_i}$ in ascending order and select the first n_1 sub-domains to form $\{\mathcal{C}^1_{\text{pick}}\}$. Sub-domains with smaller $\overline{f}^*_{\mathcal{C}_i}$ are prioritized, as good solutions have been found there. *Then*, we form another promising set $\{\mathcal{C}^2_{\text{pick}}\}$ by sampling

Algorithm 1 Branch and bound for planning. Comments are in brown.

1: **Function**: `bab_planning`
2: **Inputs**: $f, \mathcal{C}, n$ (batch size), `terminate` (Termination condition)
3: $\{(\overline{f}^*, \tilde{\boldsymbol{u}})\} \leftarrow$ `batch_search` $(f, \{\mathcal{C}\})$ ▷ Initially search on the whole $\mathcal{C}$
4: $\{\underline{f}^*\} \leftarrow$ `batch_bound` $(f, \{\mathcal{C}\})$ ▷ Initially bound on the whole $\mathcal{C}$
5: $\mathbb{P} \leftarrow \{(\mathcal{C}, \underline{f}^*, \overline{f}^*, \tilde{\boldsymbol{u}})\}$ ▷ $\mathbb{P}$ is the set of all candidate sub-domains
6: **while** `length`$(\mathbb{P}) > 0$ **and not** `terminate` **do**
7: $\quad\{(\mathcal{C}_i, \underline{f}^*_{\mathcal{C}_i}, \overline{f}^*_{\mathcal{C}_i}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i})\} \leftarrow$ `batch_pick_out` $(\mathbb{P}, n)$ ▷ Pick sub-domains to split and remove them from $\mathbb{P}$
8: $\quad\{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\} \leftarrow$ `batch_split` $(\{\mathcal{C}_i\})$ ▷ Each $\mathcal{C}_i$ splits into two sub-domains $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{up}}$
9: $\quad\{(\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i^{\text{lo}}}), (\overline{f}^*_{\mathcal{C}_i^{\text{up}}}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i^{\text{up}}})\} \leftarrow$ `batch_search` $(f, \{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\})$ ▷ Search new solutions
10: $\quad\{\underline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \underline{f}^*_{\mathcal{C}_i^{\text{up}}}\} \leftarrow$ `batch_bound` $(f, \{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\})$ ▷ Compute lower bounds on new sub-domains
11: $\quad$**if** $\min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right) < \overline{f}^*$ **then**
12: $\quad\quad\overline{f}^* \leftarrow \min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right), \tilde{\boldsymbol{u}} \leftarrow \arg\min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right)$ ▷ Update the best solution if needed
13: $\quad\mathbb{P} \leftarrow \mathbb{P} \bigcup$ `Pruner` $\left(\overline{f}^*, \{(\mathcal{C}_i^{\text{lo}}, \underline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{lo}}}), (\mathcal{C}_i^{\text{up}}, \underline{f}^*_{\mathcal{C}_i^{\text{up}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}})\}\right)$ ▷ Prune bad domains using $\overline{f}^*$
14: **Outputs:** $\overline{f}^*, \tilde{\boldsymbol{u}}$

$n - n_1$ sub-domains from the remaining N sub-domains, by $\mathrm{softmax}$ with the probability p_i defined in Eq. 2, where T is the temperature. A smaller $\underline{f}^*_{\mathcal{C}_i}$ may indicate some potentially better solutions in $\mathcal{C}_i$, which should be prioritized.

$$p_i = \frac{\exp(-T \cdot \underline{f}^*_{\mathcal{C}_i})}{\sum_{i=1}^{N} \exp(-T \cdot \underline{f}^*_{\mathcal{C}_i})} \tag{2}$$

Note that this heuristic was not discussed in neural network verification literature since, in the verification setting, all sub-domains must be verified, and thus, the order of which sub-domains to pick out first becomes less important.

Heuristic to split sub-domains. `batch_split` $(\{\mathcal{C}_i\})$ partitions every $\{\mathcal{C}_i\}$ to help search good solutions. For a box-constrained sub-domain $\mathcal{C}_i := \{\boldsymbol{u}_j \mid \underline{\boldsymbol{u}}_j \leq \boldsymbol{u}_j \leq \overline{\boldsymbol{u}}_j; j = 0, \ldots, d-1\}$, it is natural to split it into two sub-domains $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{up}}$ along a dimension j^* by bisection. Specifically, $\mathcal{C}_i^{\text{lo}} = \{\boldsymbol{u}_j \mid \underline{\boldsymbol{u}}_{j^*} \leq \boldsymbol{u}_{j^*} \leq \frac{\underline{\boldsymbol{u}}_{j^*} + \overline{\boldsymbol{u}}_{j^*}}{2}\}$, $\mathcal{C}_i^{\text{up}} = \{\boldsymbol{u}_j \mid \frac{\underline{\boldsymbol{u}}_{j^*} + \overline{\boldsymbol{u}}_{j^*}}{2} \leq \boldsymbol{u}_{j^*} \leq \overline{\boldsymbol{u}}_{j^*}\}$. In both $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{up}}$, $\underline{\boldsymbol{u}}_j \leq \boldsymbol{u}_j \leq \overline{\boldsymbol{u}}_j, \forall j \neq j^*$ holds.

One native way to select j^* is to choose the dimension with the largest input range $\overline{\boldsymbol{u}}_j - \underline{\boldsymbol{u}}_j$. This efficient strategy can help explore good solutions since dimensions with a larger range often indicate greater variability or uncertainty in f. However, it does not consider the specific landscape of f, which may imply more effective splitting dimensions.

We additionally consider the distribution of top $w\%$ samples with the best objectives from *searching* to partition $\mathcal{C}_i$ into promising sub-domains worth further searching. Specifically, for every dimension j, we record the number of top samples satisfying $\underline{\boldsymbol{u}}_j \leq \boldsymbol{u}_j \leq \frac{\underline{\boldsymbol{u}}_j + \overline{\boldsymbol{u}}_j}{2}$ and $\frac{\underline{\boldsymbol{u}}_j + \overline{\boldsymbol{u}}_j}{2} \leq \boldsymbol{u}_j \leq \overline{\boldsymbol{u}}_j$ as n_j^{lo} and n_j^{up}. Then, $|n_j^{\text{lo}} - n_j^{\text{up}}|$ indicates the distribution bias of top samples along a dimension j. A dimension with large $|n_j^{\text{lo}} - n_j^{\text{up}}|$ is critical to objective values in $\mathcal{C}_i$ and should be prioritized to split due to the imbalanced samples on two sides. In this case, it is often possible that one of the two subdomains $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{lo}}$ contains better solutions, and the other one has a larger lower bound of the objective to be pruned. This heuristic is also quite distinctive from the heuristic discussed in neural network verification literature (Bunel et al., 2018; 2020b), since we focus on finding better feasible solutions, not better lower bounds.

Based on the discussion above, we rank input dimensions descendingly by $(\overline{\boldsymbol{u}}_j - \underline{\boldsymbol{u}}_j) \cdot |n_j^{\text{lo}} - n_j^{\text{up}}|$, select the top dimension as j^*, and then split $\mathcal{C}_i$ into two subdomains by evenly split on dimension j^*.

2.2 Bounding Method for BaB-ND Planning

Our bounding procedure includes a few key modifications to improve the scalability and reduce the conservativeness of popular bound propagation-based algorithms like CROWN (Zhang et al., 2018). A crucial insight here is that in the planning problem, we don't require a strictly sound lower bound since our goal is to guide the searching of a high-quality feasible solution using the lower bound. This is distinct from neural network verification, where the goal is to prove a sound lower bound of $f(\boldsymbol{u})$. Based on this observation, we propose two approaches, *propagation early-stop* and *searching-integrated bounding*, to obtain an efficient estimation of the lower bound $\underline{f}^*_{\mathcal{C}_i}$.

Approach 1: Propagation early-stop. CROWN is a bound propagation algorithm that propagates a linear lower bound (inequality) through the neural network and has been successfully used in BaB-based neural network verifiers for the bounding step (Xu et al., 2021; Wang et al., 2021). The linear bound will be propagated backward from the output (in our case, $f(\boldsymbol{u})$) to the input of the network (in our case, $\boldsymbol{u}$), and be concretized to a concrete lower bound value using the constraints on inputs (in our case, $\mathcal{C}_i$). However, these linear bounds become increasingly loose when the network is deep and may produce vacuous lower bounds. In our neural dynamics model planning setting, due to the long time horizon H involved in Eq. 1, a neural dynamics model will be unrolled H times to form $f(\boldsymbol{u})$, leading to very loose bounds that are unhelpful for pruning useless domains during BaB.

To address this challenge, we stop the bound propagation process early to avoid the excessively loose bound when propagated through multiple layers to the input $\boldsymbol{u}$. The linear bound will be concretized using intermediate layer bounds (discussed in Approach 2 below) rather than the constraints on the inputs. A more formal description of this technique (with technical details on how CROWN is modified) is presented in Appendix C.2.

Approach 2: Search-integrated bounding. In CROWN, the propagation process requires recursively computing intermediate layer bounds (often referred to as *pre-activation bounds*) through bound propagation. These pre-activation bounds represent the lower and upper bounds for any intermediate layer that is followed by a nonlinear layer. The time complexity of this process is quadratic with respect to the number of layers. Directly applying the original CROWN-like bound propagation is both ineffective and inefficient for long-horizon planning, as the number of pre-activation bounds increases with the planning horizon. This results in overly loose lower bounds due to the accumulated relaxation errors and high execution times.

To quickly obtain the pre-activation bounds, we can utilize the by-product of extensive sampling during searching to form the empirical bounds instead of recursively using CROWN to calculate these bounds. Specifically, we denote the intermediate layer output for layer v as $\mathbf{g}_v(\boldsymbol{u})$, and assume we have M samples $\boldsymbol{u}^m$ ($m = 1, \ldots, M$) from the searching process. We calculate the preactivation lower and upper bounds as $\min_m \mathbf{g}_v(\boldsymbol{u}^m)$ and $\max_m \mathbf{g}_v(\boldsymbol{u}^m)$ dimension-wisely. Although these empirical bounds may underestimate the actual bounds, they are sufficient for CROWN to get a good estimation of $\underline{f}^*$ to guide searching.

2.3 Searching Approach for BaB-ND Planning

Given an objective function f and a batch of sub-domains $\{\mathcal{C}_i\}$, $\texttt{batch\_search}(f, \{\mathcal{C}_i\})$ seeks solutions in these sub-domains and outputs the best objectives and associated inputs $\{(\overline{f}^*_{\mathcal{C}_i}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i})\}$. A large variety of sampling-based methods can be utilized. We currently adapt MPPI as the underlying method. Other existing methods, such as CEM or projected Gradient Descent, can be alternatives. In typical neural network verification literature, searching is often ignored during BaB (Wang et al., 2021; Bunel et al., 2020b) since they do not aim to find high-quality feasible solutions during BaB.

To cooperate with the *bounding* component, we need to additionally record the output of any needed intermediate layer v, and obtain their bounds as described in Section 2.2. Since we require the lower bound of the optimal objective $\underline{f}^*_{\mathcal{C}_i}$ for every $\mathcal{C}_i$, the outputs of layer v are needed for every $\mathcal{C}_i$, calculated using the samples within the sub-domain $\mathcal{C}_i$.

Considering that the sub-domains $\{\mathcal{C}_i\}$ will become smaller and smaller, it is expected that sampling-based methods could provide good solutions. Moreover, since we always record $\overline{f}^*_{\mathcal{C}_i}$ and its associated $\tilde{\boldsymbol{u}}_{\mathcal{C}_i}$, they can initialize future searches on at least one of the split sub-domains $\{\mathcal{C}^l_i, \mathcal{C}^u_i\}$ from $\{\mathcal{C}_i\}$.

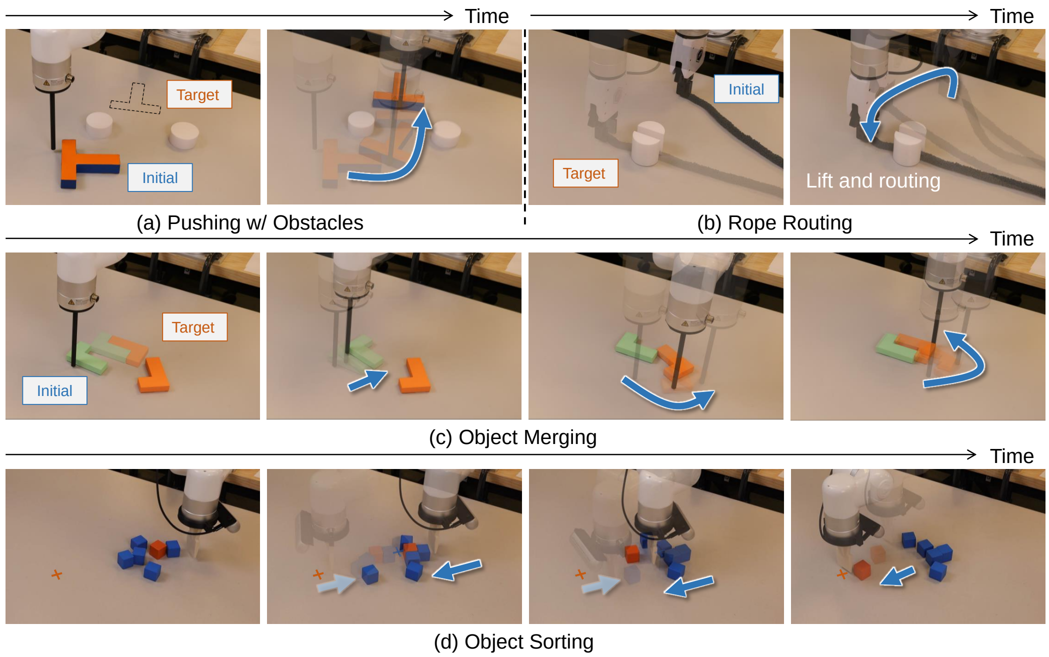


Figure 4: **Qualitative results on real-world manipulation tasks.** We evaluate our BaB-ND across four complex robotic manipulation tasks, involving non-convex feasible regions, requiring long-horizon planning, with interaction between multiple objects and the deformable rope. For every task, we visualize the initial and target configurations and one successful trajectory.

3 Experimental Results

In this section, we assess the performance of our BaB-ND across a variety of complex robotic manipulation tasks. Our primary objective is to address three key questions through experiments: *1)* How **effectively** does our BaB-ND perform long-horizon planning? *2)* Is our BaB-ND **applicable** to different manipulation scenarios with multi-object interactions and deformable objects? *3)* What is the **scalability** of our BaB-ND comparing to existing methods?

Experiment settings. We evaluate our BaB-ND on four complex robotic manipulation tasks involving non-smooth objectives, non-convex feasible regions and requiring long action sequences. Different architectures of neural dynamics like MLP and GNN are leveraged for different scenarios. Please refer to Section E for more details about tasks, dynamics models and cost functions.

- **Pushing with Obstacles.** In Figure 4.a, this task involves using a pusher to manipulate a "T"-shaped object to reach a target pose while avoiding collisions with obstacles. An MLP neural dynamics model is trained with interactions between the pusher and object without obstacles. Obstacles are modeled in the cost function, making non-smooth landscape and non-convex feasible regions.

- **Object Merging.** In Figure 4.c, two "L"-shaped objects are merged into a rectangle at a specific target pose, which requires a long action sequence with multiple contact mode switches.

- **Rope Routing.** As shown in Figure 4.b, the goal is to route a deformable rope into a tight-fitting slot (modeled in the cost function) in the 3D action space. Instead of greedily approaching to the target in initial steps, the robot needs to find the trajectory to finally reach the target.

- **Object Sorting.** In Figure 4.d, a pusher interacts with a cluster of objects to sort one outlier object out of the central zone to target while keeping others closely gathered. We use GNN to predict multi-object interactions. Every long-range action may significantly change the state. Additional constraints on actions are considered in the cost to avoid crashes between the robot and objects.

We compare our BaB-ND with three baselines: (1) **GD**: projected Gradient Descent on random samples with hyper-parameter searching on step size; (2) **MPPI**: Model Predictive Path Integral with hyper-parameter searching on noise level and reward temperature; (3) **CEM**: Decentralized Cross-Entropy Method (Zhang et al., 2022c) using an ensemble of CEM instances running independently performing local improvements of their sampling distributions.

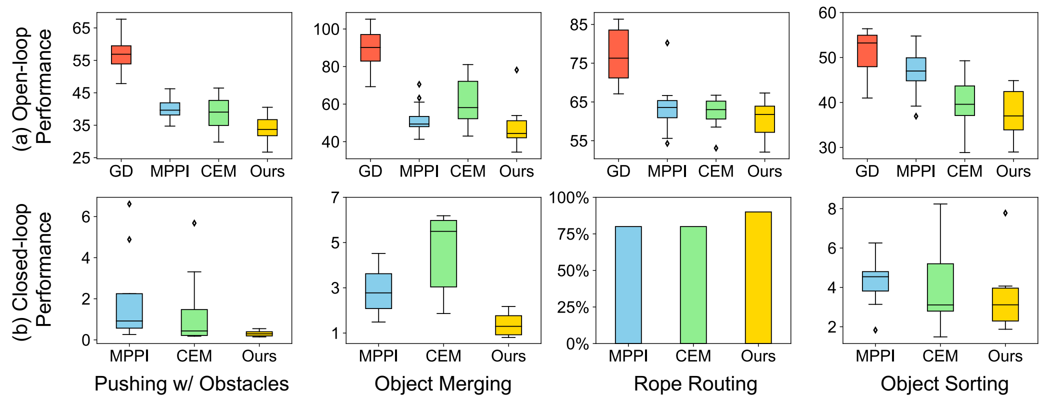


Figure 5: **Quantitative analysis of planning performance and execution performance in real world.** (a) The open-loop performance on all tasks. We report the best objective of Eq. 1 in different test case found by all methods. (b) The closed-loop performance of all tasks in real world. GD is not tested due to poor open-loop performance. We report the success rate for rope routing task and report the cost at the final step for other tasks. BaB-ND consistently outperforms baselines on open-loop performance leading better closed-loop performance.

We evaluate baselines and BaB-ND on the open-loop planning performance (the best objective of Eq. 1 found) in simulation and select the best two baselines to evaluate their real-world closed-loop control performance (the final cost or success rate of executions).

In real-world experiments, we first perform long-horizon planning to get reference trajectories of states and leverage MPC (Camacho & Bordons Alba, 2013) to efficiently track the trajectories in two tasks: *pushing with obstacles* and *object merging*. In the *rope routing* task, we directly execute the planned long-horizon action sequence due to its small sim-to-real gap. In the *object sorting* task, since the observations can change greatly after each push, we use MPC to re-plan after every action.

Effectiveness. We first evaluate the effectiveness of BaB-ND on *pushing with obstacles* and *object merging* tasks which are contact-rich and require strong long-horizon planning performance. The quantitative results of open-loop and closed loop performance for these tasks are presented in Figure 5.

In both tasks, our BaB-ND effectively optimizes the objective of Eq. 1 and gives better open-loop performance than all baselines. The better-planned trajectories can yield better closed-loop performance in the real world with efficient tracking. Specifically, in the pushing with obstacles task, GD offers much worse trajectories than others, often resulting in the T-shaped object stuck at one obstacle. MPPI and CEM can offer trajectories passing through the obstacles but with bad alignment with the target. In contrast, BaB-ND can not only pass through obstacles successfully, but also often perfectly align with the final target.

Applicability. We assess the applicability of BaB-ND on rope routing and object sorting tasks involving the manipulation of deformable objects and interactions between multiple objects modeled by GNNs. The quantitative results in Figure 5 demonstrate our applicability on these tasks.

In the rope routing task, MPPI, CEM and ours achieve similar open-loop performance while GD may struggle at sub-optimal trajectories, routing the rope horizontally and getting stuck outside the slot. In the object sorting task, CEM can outperform MPPI in simulation and real-world since MPPI is more suitable for planning continuous action sequences while actions are discrete in the task. Ours outperforms CEM with similar median and smaller variance.

Scalability. We evaluate the scalability of our BaB-ND comparing with MIP (Liu et al., 2023) and evaluate the runtime of each primary component of BaB-ND. Please refer to Section F for results.

4 Conclusion

In this paper, we propose a branch-and-bound-based framework for long-horizon motion planning in robotic manipulation tasks. We leverage specialized branching heuristics for systematical search and adapt the bound propagation algorithm from neural network verification to estimate tight bounds of objectives efficiently. Our framework demonstrates superior planning performance in complex, contact-rich manipulation tasks and is scalable and adaptable to various model architectures.

References

Pulkit Agrawal, Ashvin Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. *arXiv preprint arXiv:1606.07419*, 2016.

Ross Anderson, Joey Huchette, Will Ma, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming*, 183(1):3–39, 2020.

Stanley Bak, Changliu Liu, and Taylor Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016.

Victor Blomqvist. Pymunk. `https://pymunk.org`, November 2022.

Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Lagrangian decomposition for neural network verification. *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2020a.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification, 2020b.

Eduardo F Camacho and Carlos Bordons Alba. *Model Predictive Control*. Springer Science & Business Media, 2013.

Alessandro De Palma, Harkirat Singh Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. Scaling the convex barrier with active sets. *International Conference on Learning Representations (ICLR)*, 2021.

Danny Driess, Zhiao Huang, Yunzhu Li, Russ Tedrake, and Marc Toussaint. Learning multi-object dynamics with compositional neural radiance fields. In *Conference on robot learning*, pp. 1755–1768. PMLR, 2023.

Frederik Ebert, Chelsea Finn, Alex X Lee, and Sergey Levine. Self-supervised visual planning with temporal skip connections. In *CoRL*, pp. 344–356, 2017.

Frederik Ebert, Chelsea Finn, Sudeep Dasari, Annie Xie, Alex Lee, and Sergey Levine. Visual foresight: Model-based deep reinforcement learning for vision-based robotic control. *arXiv preprint arXiv:1812.00568*, 2018.

J. Zico Kolter Eric Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning (ICML)*, 2018.

Claudio Ferrari, Mark Niklas Muller, Nikola Jovanovic, and Martin Vechev. Complete verification via multi-neuron relaxation guided branch-and-bound. *arXiv preprint arXiv:2205.00263*, 2022.

Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2786–2793. IEEE, 2017.

Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *arXiv preprint arXiv:1605.07157*, 2016.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2019.

Bernhard Paus Graesdal, Shao Yuan Chew Chia, Tobia Marcucci, Savva Morozov, Alexandre Amice, Pablo A. Parrilo, and Russ Tedrake. Towards tight convex relaxations for contact-rich manipulation, 2024. URL https://arxiv.org/abs/2402.10312.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pp. 2555–2565. PMLR, 2019b.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pp. 2555–2565, 2019c.

Tyler Han, Alex Liu, Anqi Li, Alex Spitzer, Guanya Shi, and Byron Boots. Model predictive control for aggressive driving over uneven terrain, 2024.

Hanjiang Hu, Jianglin Lan, and Changliu Liu. Real-time safe control of neural network dynamic models with sound approximation, 2024a.

Hanjiang Hu, Yujie Yang, Tianhao Wei, and Changliu Liu. Verification of neural control barrier functions with symbolic derivative bounds propagation. In *8th Annual Conference on Robot Learning*, 2024b. URL https://openreview.net/forum?id=jnubz7wB2w.

Zixuan Huang, Xingyu Lin, and David Held. Mesh-based dynamics model with occlusion reasoning for cloth manipulation. In *Robotics: Science and Systems (RSS)*, 2022.

Panagiotis Kouvaros and Alessio Lomuscio. Towards scalable complete verification of relu neural networks via dependency-based branching. In *IJCAI*, pp. 2643–2650, 2021.

Tejas D Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Malcolm Reynolds, Andrew Zisserman, and Volodymyr Mnih. Unsupervised learning of object keypoints for perception and control. *Advances in neural information processing systems*, 32:10724–10734, 2019.

Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47), 2020.

Ian Lenz, Ross A Knepper, and Ashutosh Saxena. Deepmpc: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*, volume 10, pp. 25. Rome, Italy, 2015.

Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B Tenenbaum, and Antonio Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*, 2018.

Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 1205–1211. IEEE, 2019.

Yunzhu Li, Antonio Torralba, Anima Anandkumar, Dieter Fox, and Animesh Garg. Causal discovery in physical systems from videos. *Advances in Neural Information Processing Systems*, 33, 2020.

Xingyu Lin, Yufei Wang, Zixuan Huang, and David Held. Learning visible connectivity dynamics for cloth smoothing. In *Conference on Robot Learning*, 2021.

Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

Ziang Liu, Genggeng Zhou, Jeff He, Tobia Marcucci, Li Fei-Fei, Jiajun Wu, and Yunzhu Li. Model-based control with sparse neural dynamics. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=ymBG2xs9Zf.

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. *arXiv preprint arXiv:1811.01848*, 2018.

Jingyue Lu and M. Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations (ICLR)*, 2020.

Lucas Manuelli, Yunzhu Li, Pete Florence, and Russ Tedrake. Keypoints into the future: Self-supervised correspondence in model-based reinforcement learning. *arXiv preprint arXiv:2009.05085*, 2020.

Tobia Marcucci. *Graphs of Convex Sets with Applications to Optimal Control and Motion Planning*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2024.

Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. The third international verification of neural networks competition (vnn-comp 2022): Summary and results. *arXiv preprint arXiv:2212.10376*, 2022.

Anusha Nagabandi, Kurt Konolige, Sergey Levine, and Vikash Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pp. 1101–1112. PMLR, 2020.

Alessandro De Palma, Rudy Bunel, Aymeric Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

Jacob Sacks, Rwik Rana, Kevin Huang, Alex Spitzer, Guanya Shi, and Byron Boots. Deep model predictive optimization, 2023.

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A convex relaxation barrier to tight robustness verification of neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Younggyo Seo, Danijar Hafner, Hao Liu, Fangchen Liu, Stephen James, Kimin Lee, and Pieter Abbeel. Masked world models for visual control. In *Conference on Robot Learning*, pp. 1332–1344. PMLR, 2023.

Haochen Shi, Huazhe Xu, Zhiao Huang, Yunzhu Li, and Jiajun Wu. Robocraft: Learning to see, simulate, and shape elasto-plastic objects with graph networks. *arXiv preprint arXiv:2205.02909*, 2022.

Haochen Shi, Huazhe Xu, Samuel Clarke, Yunzhu Li, and Jiajun Wu. Robocook: Long-horizon elasto-plastic object manipulation with diverse tools, 2023.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages (POPL)*, 2019.

HJ Suh and Russ Tedrake. The surprising effectiveness of linear models for visual foresight in object pile manipulation. *arXiv preprint arXiv:2002.09093*, 2020.

Stephen Tian, Frederik Ebert, Dinesh Jayaraman, Mayur Mudigonda, Chelsea Finn, Roberto Calandra, and Sergey Levine. Manipulation by feel: Touch-based control with deep predictive models. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 818–824. IEEE, 2019.

Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming, 2019.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Shiqi Wang, Huan Zhang, Kaidi Xu, Suman Jana, Xue Lin, Cho-Jui Hsieh, and Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network robustness verification. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Yixuan Wang, Yunzhu Li, Katherine Driggs-Campbell, Li Fei-Fei, and Jiajun Wu. Dynamic-Resolution Model Learning for Object Pile Manipulation. In *Proceedings of Robotics: Science and Systems*, Daegu, Republic of Korea, July 2023. doi: 10.15607/RSS.2023.XIX.047.

Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. *arXiv preprint arXiv:1506.07365*, 2015.

Tianhao Wei and Changliu Liu. Safe control with neural network dynamic models, 2022.

Grady Williams, Andrew Aldrich, and Evangelos A Theodorou. Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics*, 40(2): 344–357, 2017.

Junlin Wu, Huan Zhang, and Yevgeniy Vorobeychik. Verified safe reinforcement learning for neural network dynamic models, 2024. URL https://arxiv.org/abs/2405.15994.

Philipp Wu, Alejandro Escontrela, Danijar Hafner, Pieter Abbeel, and Ken Goldberg. Daydreamer: World models for physical robot learning. In *Conference on Robot Learning*, pp. 2226–2240. PMLR, 2023.

Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *International Conference on Learning Representations (ICLR)*, 2021.

Lin Yen-Chen, Maria Bauza, and Phillip Isola. Experience-embedded visual foresight. In *Conference on Robot Learning*, pp. 1015–1024. PMLR, 2020.

Zeji Yi, Chaoyi Pan, Guanqi He, Guannan Qu, and Guanya Shi. Covo-mpc: Theoretical analysis of sampling-based mpc and optimal covariance design, 2024.

Ji Yin, Zhiyuan Zhang, Evangelos Theodorou, and Panagiotis Tsiotras. Trajectory distribution control for model predictive path integral control using covariance steering. In *2022 International Conference on Robotics and Automation (ICRA)*, pp. 1478–1484, 2022. doi: 10.1109/ICRA46639.2022.9811615.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. General cutting planes for bound-propagation-based neural network verification. *Advances in Neural Information Processing Systems*, 2022a.

Huan Zhang, Shiqi Wang, Kaidi Xu, Yihan Wang, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. A branch and bound framework for stronger adversarial attacks of ReLU networks. In *International Conference on Machine Learning (ICML)*, pp. 26591–26604. PMLR, 2022b.

Kaifeng Zhang, Baoyu Li, Kris Hauser, and Yunzhu Li. Adaptigraph: Material-adaptive graph-based neural dynamics for robotic manipulation. In *Proceedings of Robotics: Science and Systems (RSS)*, 2024.

Zichen Zhang, Jun Jin, Martin Jagersand, Jun Luo, and Dale Schuurmans. A simple decentralized cross-entropy method, 2022c. URL https://arxiv.org/abs/2212.08235.

A RELATED WORKS

Neural dynamics model learning in manipulation. Dynamics models learned from observations in simulation or the real world using deep neural networks (DNNs) have been widely and successfully applied to robotic manipulation tasks (Shi et al., 2023; Wang et al., 2023). Neural dynamics models can be learned directly from pixel space (Finn et al., 2016; Ebert et al., 2017; 2018; Yen-Chen et al., 2020; Suh & Tedrake, 2020) or low-dimensional latent space (Watter et al., 2015; Agrawal et al., 2016; Hafner et al., 2019b;a; Schrittwieser et al., 2020; Wu et al., 2023). Other approaches use more structured scene representations, such as keypoints (Kulkarni et al., 2019; Manuelli et al., 2020; Li et al., 2020), particles (Li et al., 2018; Shi et al., 2022; Zhang et al., 2024), and meshes (Huang et al., 2022). Our work employs keypoint or object-centric representations, and the proposed BaB-ND framework is compatible with various architectures, ranging from multilayer perceptrons (MLPs) to graph neural networks (GNNs) (Battaglia et al., 2016; Li et al., 2019).

Model-based planning with neural dynamics models. The highly non-linear and non-convex nature of neural dynamics models hinders the effective optimization of model-based planning problems. Previous works (Yen-Chen et al., 2020; Ebert et al., 2017; Nagabandi et al., 2020; Finn & Levine, 2017; Manuelli et al., 2020; Sacks et al., 2023; Han et al., 2024) utilize sampling-based algorithms like CEM (Rubinstein & Kroese, 2013) and MPPI (Williams et al., 2017) for online planning. Despite their flexibility and ability to leverage GPU support, these methods struggle with large input dimensions due to the exponential growth in required samples. Previous work (Yin et al., 2022) improved MPPI by introducing dynamics model linearization and covariance control techniques, but their effectiveness on neural dynamics models remains unclear. Other approaches (Li et al., 2018; 2019) have used gradient descent to optimize action sequences but encounter challenges with local optima and non-smooth objective landscapes. Recently, methods inspired by neural network verification have been developed to achieve safe control and robust planning over systems involving neural networks (Wei & Liu, 2022; Liu et al., 2023; Hu et al., 2024a; Wu et al., 2024; Hu et al., 2024b), but their scalability to more complex real-world manipulation tasks is still uncertain. Moreover, researchers are also exploring the promising direction of performing planning over graphs of convex sets (GCSs) for contact-rich manipulation tasks Marcucci (2024); Graesdal et al. (2024). However, these approaches do not incorporate neural networks.

Neural network verification. Neural network verification ensures the reliability and safety of neural networks (NNs) by formally proving their output properties. This process can be formulated as finding the *lower bound* of a minimization problem involving NNs, with early verifiers utilizing MIP (Tjeng et al., 2019) or linear programming (LP) (Bunel et al., 2018; Lu & Kumar., 2020). These approaches suffer from scalability issues (Salman et al., 2019; Zhang et al., 2022b; Liu et al., 2021) because they have limited parallelization capabilities and fail to fully exploit GPU resources. On the other hand, bound propagation methods such as CROWN (Zhang et al., 2018) can efficiently propagate bounds on NNs (Eric Wong, 2018; Singh et al., 2019; Wang et al., 2018; Gowal et al., 2019) in a layer-by-layer manner and can be accelerated on GPUs. Combining bound propagation with BaB leads to successful approaches in NN verification (Bunel et al., 2020a; De Palma et al., 2021; Kouvaros & Lomuscio, 2021; Ferrari et al., 2022), and notably, the α,β-CROWN framework (Xu et al., 2021; Wang et al., 2021; Zhang et al., 2022a) achieved strong verification performance on large NNs (Bak et al., 2021; Müller et al., 2022). In our model-based planning setting, we utilize the lower bounds from verification, with modification and specializations, to guide our systematic search procedure to find high-quality feasible solutions.

B ALGORITHM OF BAB-ND

The BaB-ND algorithm Algorithm 2 takes an objective function f with neural networks, a domain $\mathcal{C}$ as input space and a termination condition if necessary. The sub-procedure `batch_search` seeks better solutions on domains $\{\mathcal{C}_i\}$. It returns the best objectives $\{\overline{f}^*_{\mathcal{C}_i}\}$ and corresponding solution $\{\tilde{\boldsymbol{u}}_{\mathcal{C}_i}\}$ for n selected subdomains simultaneously. The sub-procedure `batch_bound` computes the lower bounds of f^* on domains $\{\mathcal{C}_i\}$ in the way described in. It operates in a batch and returns the lower bounds $\{\underline{f}^*_{\mathcal{C}_i}\}$.

In the algorithm, we maintain $\overline{f}^*$ and $\tilde{\boldsymbol{u}}$ as the best objective and solution we can find. We also maintain a global set $\mathbb{P}$ storing all the candidate sub-domains which $\underline{f}^*_{\mathcal{C}_i} \geq \overline{f}^*$. Initially, we only

Algorithm 2 Branch and bound for planning. Comments are in brown.

1: **Inputs**: $f, \mathcal{C}, n$ (batch size), `terminate` (Termination condition)
2: $\{(\overline{f}^*, \tilde{\boldsymbol{u}})\} \leftarrow$ `batch_search` $(f, \{\mathcal{C}\})$ ▷ Initially search on the whole $\mathcal{C}$
3: $\{\underline{f}^*\} \leftarrow$ `batch_bound` $(f, \{\mathcal{C}\})$ ▷ Initially bound on the whole $\mathcal{C}$
4: $\mathbb{P} \leftarrow \{(\mathcal{C}, \underline{f}^*, \overline{f}^*, \tilde{\boldsymbol{u}})\}$ ▷ $\mathbb{P}$ is the set of all candidate sub-domains
5: **while** `length`$(\mathbb{P}) > 0$ **and not** `terminate` **do**
6: $\quad\{(\mathcal{C}_i, \underline{f}^*_{\mathcal{C}_i}, \overline{f}^*_{\mathcal{C}_i}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i})\} \leftarrow$ `batch_pick_out` $(\mathbb{P}, n)$ ▷ Pick sub-domains to split and remove them from $\mathbb{P}$
7: $\quad\{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\} \leftarrow$ `batch_split` $(\{\mathcal{C}_i\})$ ▷ Each $\mathcal{C}_i$ splits into two sub-domains $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{up}}$
8: $\quad\{(\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i^{\text{lo}}}), (\overline{f}^*_{\mathcal{C}_i^{\text{up}}}, \tilde{\boldsymbol{u}}_{\mathcal{C}_i^{\text{up}}})\} \leftarrow$ `batch_search` $(f, \{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\})$ ▷ Search new solutions
9: $\quad\{\underline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \underline{f}^*_{\mathcal{C}_i^{\text{up}}}\} \leftarrow$ `batch_bound` $(f, \{\mathcal{C}_i^{\text{lo}}, \mathcal{C}_i^{\text{up}}\})$ ▷ Compute lower bounds on new sub-domains
10: $\quad$**if** $\min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right) < \overline{f}^*$ **then**
11: $\quad\quad\overline{f}^* \leftarrow \min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right), \tilde{\boldsymbol{u}} \leftarrow \arg\min\left(\{\overline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}}\}\right)$ ▷ Update the best solution if needed
12: $\quad\mathbb{P} \leftarrow \mathbb{P} \bigcup$ `Pruner` $\left(\overline{f}^*, \{(\mathcal{C}_i^{\text{lo}}, \underline{f}^*_{\mathcal{C}_i^{\text{lo}}}, \overline{f}^*_{\mathcal{C}_i^{\text{lo}}}), (\mathcal{C}_i^{\text{up}}, \underline{f}^*_{\mathcal{C}_i^{\text{up}}}, \overline{f}^*_{\mathcal{C}_i^{\text{up}}})\}\right)$ ▷ Filter out bad sub-domains using $\overline{f}^*$, insert the left domains back to $\mathbb{P}$
13: **Outputs:** $\overline{f}^*, \tilde{\boldsymbol{u}}$

have the whole input domain $\mathcal{C}$, so we perform `batch_search` and `batch_bound` on $\mathcal{C}$ and initialize current $\overline{f}^*$, $\tilde{\boldsymbol{u}}$ and $\mathbb{P}$ (Line 2-4).

Then we utilize the power of GPUs to split, search and bound sub-domains in parallel and always maintain $\mathbb{P}$ (Line 6-11). Specifically, `batch_pick_out` selects n (batch size) promising sub-domains from $\mathbb{P}$. If the length of $\mathbb{P}$ is less than n, then we reduce n to the length of $\mathbb{P}$. `batch_split` splits each selected $\mathcal{C}_i$ to two sub-domains $\mathcal{C}_i^{\text{lo}}$ and $\mathcal{C}_i^{\text{up}}$ according to a branch heuristic in parallel. `Pruner` filters out bad sub-domains (proved with $\underline{f}^*_{\mathcal{C}_i} > \overline{f}^*$) and we insert the remaining ones to $\mathbb{P}$.

The loop breaks if there is no sub-domain left in $\mathbb{P}$ or some other pre-defined termination conditions such as timeout and find good enough objective $\overline{f}^* \leq f_{th}$, are satisfied (Line 5). We finally return the best objective $\overline{f}^*$ and corresponding solution $\tilde{\boldsymbol{u}}$.

C More details about bounding

C.1 Proofs of CROWN bounding

In this section, we first share the background of neural network verification including its formulation and a efficient linear bound propagation method CROWN (Zhang et al., 2018) to calculate bounds over neural networks. We take the Multilayer perceptron (MLP) with ReLU activation as the example and CROWN is a general framework which is suitable to different activations and model architectures.

Definition. We define the input of a neural network as $x \in \mathbb{R}^{d_0}$, and define the weights and biases of an L-layer neural network as $\mathbf{W}^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ and $\mathbf{b}^{(i)} \in \mathbb{R}^{d_i}$ ($i \in \{1, \cdots, L\}$) respectively. The neural network function $f : \mathbb{R}^{d_0} \to \mathbb{R}$ is defined as $f(x) = z^{(L)}(x)$, where $z^{(i)}(x) = \mathbf{W}^{(i)}\hat{z}^{(i-1)}(x) + \mathbf{b}^{(i)}$, $\hat{z}^{(i)}(x) = \sigma(z^{(i)}(x))$ and $\hat{z}^{(0)}(x) = x$. σ is the activation function and we use ReLU throughout this paper. When the context is clear, we omit $\cdot(x)$ and use $z_j^{(i)}$ and $\hat{z}_j^{(i)}$ to represent the *pre-activation* and *post-activation* values of the j-th neuron in the i-th layer. Neural network verification seeks the solution of the optimization problem in Eq. 3:

$$\min f(x) := z^{(L)} \quad \text{s.t. } z^{(i)} = \mathbf{W}^{(i)}\hat{z}^{(i-1)} + \mathbf{b}^{(i)}, \hat{z}^{(i)} = \sigma(z^{(i)}), x \in \mathcal{C}, i \in \{1, \cdots, L-1\} \tag{3}$$

The set $\mathcal{C}$ defines the allowed input region and our aim is to find the minimum of $f(x)$ for $x \in \mathcal{C}$, and throughout this paper we consider $\mathcal{C}$ as an ℓ_p ball around a data example x_0: $\mathcal{C} = \{x \mid \|x - x_0\|_p \leq \epsilon\}$.

First, let we consider the neural network with only linear layers. in this case, it is easily to get a linear relationship between x and $f(x)$ that $f(x) = \mathbf{W}x + \mathbf{b}$ no matter what is the value of L and derive the closed form of $f^* = \min f(x)$ for $x \in \mathcal{C}$. With this idea in our mind, for neural networks with non-linear activation layers, if we could bound them with some linear functions, then it is still possible to bound $f(x)$ with linear functions.

Then, we show that the non-linear activation ReLU layer $\hat{z} = \mathrm{ReLU}(z)$ can be bounded by two linear functions in three cases according to the range of pre-activation bounds $\mathbf{l} \leq z \leq \mathbf{u}$: active ($\mathbf{l} \geq 0$), inactive ($\mathbf{u} \leq 0$) and unstable ($\mathbf{l} < 0 < \mathbf{u}$) in Lemma C.1.

Lemma C.1 (Relaxation of a ReLU layer in CROWN)**.** *Given pre-activation vector $z \in \mathbb{R}^d, \mathbf{l} \leq z \leq \mathbf{u}$ (element-wise), $\hat{z} = \mathrm{ReLU}(z)$, we have*

$$\underline{\mathbf{D}} z + \underline{\mathbf{b}} \leq \hat{z} \leq \overline{\mathbf{D}} z + \overline{\mathbf{b}},$$

where $\underline{\mathbf{D}}, \overline{\mathbf{D}} \in \mathbb{R}^{d \times d}$ are diagonal matrices defined as:

$$\underline{\mathbf{D}}_{j,j} = \begin{cases} 1, & \text{if } \mathbf{l}_j \geq 0 \\ 0, & \text{if } \mathbf{u}_j \leq 0 \\ \boldsymbol{\alpha}_j, & \text{if } \mathbf{u}_j > 0 > \mathbf{l}_j \end{cases} \qquad \overline{\mathbf{D}}_{j,j} = \begin{cases} 1, & \text{if } \mathbf{l}_j \geq 0 \\ 0, & \text{if } \mathbf{u}_j \leq 0 \\ \frac{\mathbf{u}_j}{\mathbf{u}_j - \mathbf{l}_j}, & \text{if } \mathbf{u}_j > 0 > \mathbf{l}_j \end{cases} \tag{4}$$

$\boldsymbol{\alpha} \in \mathbb{R}^d$ is a free vector s.t., $0 \leq \boldsymbol{\alpha} \leq 1$. $\underline{\mathbf{b}}, \overline{\mathbf{b}} \in \mathbb{R}^d$ are defined as

$$\underline{\mathbf{b}}_j = \begin{cases} 0, & \text{if } \mathbf{l}_j > 0 \text{ or } \mathbf{u}_j \leq 0 \\ 0, & \text{if } \mathbf{u}_j > 0 > \mathbf{l}_j. \end{cases} \qquad \underline{\mathbf{b}}_j = \begin{cases} 0, & \text{if } \mathbf{l}_j > 0 \text{ or } \mathbf{u}_j \leq 0 \\ -\frac{\mathbf{u}_j \mathbf{l}_j}{\mathbf{u}_j - \mathbf{l}_j}, & \text{if } \mathbf{u}_j > 0 > \mathbf{l}_j. \end{cases} \tag{5}$$

Proof. For the j-th ReLU neuron, if $\mathbf{l}_j \geq 0$, then $\mathrm{ReLU}(z_j) = z_j$; if $\mathbf{u}_j < 0$, then $\mathrm{ReLU}(z_j) = 0$. For the case of $\mathbf{l}_j < 0 < \mathbf{u}_j$, the ReLU function can be linearly upper and lower bounded within this range:

$$\boldsymbol{\alpha}_j z_j \leq \mathrm{ReLU}(z_j) \leq \frac{\mathbf{u}_j}{\mathbf{u}_j - \mathbf{l}_j}(z_j - \mathbf{l}_j) \quad \forall\, \mathbf{l}_j \leq z_j \leq \mathbf{u}_j$$

where $0 \leq \boldsymbol{\alpha}_j \leq 1$ is a free variable - any value between 0 and 1 produces a valid lower bound. □

Next we apply the linear relaxation of ReLU to the L-layer neural network $f(x)$ to further derive the linear lower bound of $f(x)$. The idea is to propagate a weight matrix $\underline{\widetilde{\mathbf{W}}}$ and bias vector $\underline{\widetilde{\mathbf{b}}}$ from the L-th layer to 1-th layer. Specifically, when propagate through ReLU layer, we should greedily select upper bound of $\hat{z}_j$ when $\underline{\widetilde{\mathbf{W}}}_{i,j}$ is negative and select lower bound of $\hat{z}_j$ when $\underline{\widetilde{\mathbf{W}}}_{i,j}$ is positive to calculate the lower bound of $f(x)$. When propagate through linear layer, we do not need to do such selection since there is no relaxation on linear layer.

Theorem C.2 (CROWN bound propagation on neural network)**.** *Given the L-layer neural network $f(x)$ as defined in Eq. 3, we could find a linear function with respect to input x.*

$$f(x) := z^{(L)} \geq \underline{\widetilde{\mathbf{W}}}^{(1)} x + \underline{\widetilde{\mathbf{b}}}^{(1)} \tag{6}$$

where $\underline{\widetilde{\mathbf{W}}}$ and $\underline{\widetilde{\mathbf{b}}}$ are recursively defined as following:

$$\underline{\widetilde{\mathbf{W}}}^{(l)} = \underline{\mathbf{A}}^{(l)} \mathbf{W}^{(l)}, \underline{\widetilde{\mathbf{b}}}^{(l)} = \underline{\mathbf{A}}^{(l)} \mathbf{b}^{(l)} + \underline{\mathbf{d}}^{(l)}, \forall l = 1 \dots L \tag{7}$$

$$\underline{\mathbf{A}}^{(L)} = \mathbf{I} \in \mathbb{R}^{d_L \times d_L}, \underline{\widetilde{\mathbf{b}}}^{(L)} = 0 \tag{8}$$

$$\underline{\mathbf{A}}^{(l)} = \underline{\widetilde{\mathbf{W}}}^{(l+1)}_{\geq 0} \underline{\mathbf{D}}^{(l)} + \underline{\widetilde{\mathbf{W}}}^{(l+1)}_{<0} \overline{\mathbf{D}}^{(l)} \in \mathbb{R}^{d_{l+1} \times d_l}, \forall l = 1 \dots L-1 \tag{9}$$

$$\underline{\mathbf{d}}^{(l)} = \underline{\widetilde{\mathbf{W}}}^{(l+1)}_{\geq 0} \underline{\mathbf{b}}^{(l)} + \underline{\widetilde{\mathbf{W}}}^{(l+1)}_{<0} \overline{\mathbf{b}}^{(l)} + \underline{\widetilde{\mathbf{b}}}^{(l)}, \forall l = 1 \dots L-1 \tag{10}$$

where $\forall l = 1 \dots L-1, \underline{\mathbf{D}}^{(l)}, \overline{\mathbf{D}}^{(l)} \in \mathbb{R}^{d_l \times d_l}$ and $\underline{\mathbf{b}}^{(l)}, \overline{\mathbf{b}}^{(l)} \in \mathbb{R}^{d_l}$ are defined as in Lemma C.1. And subscript "≥ 0" stands for taking positive elements from the matrix while setting other elements to zero, and vice versa for subscript "< 0".

Proof. First we have

$$\begin{aligned} f(x) := z^{(L)} &= \underline{\mathbf{A}}^{(L)} z^{(L)} + \underline{\mathbf{d}}^{(L)} && (11) \\ &= \underline{\mathbf{A}}^{(L)} \mathbf{W}^{(L)} \hat{z}^{(L-1)} + \underline{\mathbf{A}}^{(L)} \mathbf{b}^{(L)} + \underline{\mathbf{d}}^{(L)} && (12) \\ &= \underline{\widetilde{\mathbf{W}}}^{(L)} \hat{z}^{(L-1)} + \underline{\widetilde{\mathbf{b}}}^{(L)} && (13) \end{aligned}$$

Refer to Lemma C.1, we have

$$\underline{\mathbf{D}}^{(L-1)}z^{(L-1)}+\underline{\mathbf{b}}^{(L-1)} \le \hat{z}^{(L-1)} \le \overline{\mathbf{D}}^{(L-1)}z^{(L-1)}+\overline{\mathbf{b}}^{(L-1)} \tag{14}$$

Then we can form the lower bound of $z^{(L)}$ element by element: we greedily select the upper bound $\hat{z}_j^{(L-1)} \le \overline{\mathbf{D}}_{j,j}^{(L-1)}z_j^{(L-1)}+\overline{\mathbf{b}}_j^{(L-1)}$ when $\widetilde{\underline{\mathbf{W}}}_{i,j}^{(L)}$ is negative, and select the lower bound $\hat{z}_j^{(L-1)} \ge \underline{\mathbf{D}}_{j,j}^{(L-1)}z_j^{(L-1)}+\underline{\mathbf{b}}_j^{(L-1)}$ otherwise. It can be formatted as

$$\widetilde{\underline{\mathbf{W}}}^{(L)}\hat{z}^{(L-1)}+\widetilde{\underline{\mathbf{b}}}^{(L)} \ge \underline{\mathbf{A}}^{(L-1)}z^{(L-1)}+\underline{\mathbf{d}}^{(L-1)} \tag{15}$$

where $\underline{\mathbf{A}}^{(L-1)} \in \mathbb{R}^{d_L \times d_{L-1}}$ is defined as

$$\underline{\mathbf{A}}_{i,j}^{(L-1)} = \begin{cases} \widetilde{\underline{\mathbf{W}}}_{i,j}^{(L)}\overline{\mathbf{D}}_{j,j}^{(L-1)}, & \text{if } \widetilde{\underline{\mathbf{W}}}_{i,j}^{(L)} < 0 \\ \widetilde{\underline{\mathbf{W}}}_{i,j}^{(L)}\underline{\mathbf{D}}_{j,j}^{(L-1)}, & \text{if } \widetilde{\underline{\mathbf{W}}}_{i,j}^{(L)} \ge 0 \end{cases} \tag{16}$$

for simplicity, we rewrite it in matrix form as

$$\underline{\mathbf{A}}^{(L-1)} = \widetilde{\underline{\mathbf{W}}}_{\ge 0}^{(L)}\underline{\mathbf{D}}^{(L-1)}+\widetilde{\underline{\mathbf{W}}}_{<0}^{(L)}\overline{\mathbf{D}}^{(L-1)} \tag{17}$$

And $\underline{\mathbf{d}}^{(L-1)} \in \mathbb{R}^{d_L}$ is similarly defined as

$$\underline{\mathbf{d}}^{(L-1)} = \widetilde{\underline{\mathbf{W}}}_{\ge 0}^{(L)}\underline{\mathbf{b}}^{(L-1)}+\widetilde{\underline{\mathbf{W}}}_{<0}^{(L)}\overline{\mathbf{b}}^{(L-1)}+\widetilde{\underline{\mathbf{b}}}^{(L)} \tag{18}$$

Then we continue to replace $z^{(L-1)}$ in Equation 15 as $\mathbf{W}^{(L-1)}\hat{z}^{(L-2)}+\mathbf{b}^{(L-1)}$

$$\begin{aligned}\widetilde{\underline{\mathbf{W}}}^{(L)}\hat{z}^{(L-1)}+\widetilde{\underline{\mathbf{b}}}^{(L)} &\ge (\underline{\mathbf{A}}^{(L-1)}\mathbf{W}^{(L-1)})\hat{z}^{(L-2)}+\underline{\mathbf{A}}^{(L-1)}\mathbf{b}^{(L-1)}+\underline{\mathbf{d}}^{(L-1)} \\ &= \widetilde{\underline{\mathbf{W}}}^{(L-1)}\hat{z}^{(L-2)}+\widetilde{\underline{\mathbf{b}}}^{(L-1)}\end{aligned} \tag{19}$$

By continuing to propagate the linear inequality to the first layer, we get

$$f(x) \ge \widetilde{\underline{\mathbf{W}}}^{(1)}\hat{z}^{(0)}+\widetilde{\underline{\mathbf{b}}}^{(1)} = \widetilde{\underline{\mathbf{W}}}^{(1)}x+\widetilde{\underline{\mathbf{b}}}^{(1)} \tag{20}$$

□

After getting the linear lower bound of $f(x)$, and given $x \in \mathcal{C}$, we could solve the linear lower bound in closed form as in Theorem C.3. It is given by the Hölder's inequality.

Theorem C.3 (Bound Concretization under ℓ_p ball Perturbations)**.** *Given the L-layer neural network $f(x)$ as defined in Eq. 3, and input $x \in \mathcal{C} = \mathbb{B}_p(x_0,\epsilon) = \{x \mid \|x-x_0\|_p \le \epsilon\}$, we could find concrete lower bound of $f(x)$ by solving the optimization problem $\min_{x\in\mathcal{C}} \widetilde{\underline{\mathbf{W}}}^{(1)}x+\widetilde{\underline{\mathbf{b}}}^{(1)}$ and its solution gives*

$$\min_{x\in\mathcal{C}} f(x) \ge \min_{x\in\mathcal{C}} \widetilde{\underline{\mathbf{W}}}^{(1)}x+\widetilde{\underline{\mathbf{b}}}^{(1)} \ge -\epsilon\|\widetilde{\underline{\mathbf{W}}}^{(1)}\|_q+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} \tag{21}$$

where $\frac{1}{p}+\frac{1}{q}=1$ and $\|\cdot\|_q$ denotes taking ℓ_q-norm for each row in the matrix and the result makes up a vector.

Proof.

$$\begin{aligned}&\min_{x\in\mathcal{C}} \widetilde{\underline{\mathbf{W}}}^{(1)}x+\widetilde{\underline{\mathbf{b}}}^{(1)} && (22)\\
=&\min_{\lambda\in\mathbb{B}_p(0,1)} \widetilde{\underline{\mathbf{W}}}^{(1)}(x_0+\epsilon\lambda)+\widetilde{\underline{\mathbf{b}}}^{(1)} && (23)\\
=&\epsilon(\min_{\lambda\in\mathbb{B}_p(0,1)} \widetilde{\underline{\mathbf{W}}}^{(1)}\lambda)+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} && (24)\\
=&-\epsilon(\max_{\lambda\in\mathbb{B}_p(0,1)} -\widetilde{\underline{\mathbf{W}}}^{(1)}\lambda)+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} && (25)\\
\ge&-\epsilon(\max_{\lambda\in\mathbb{B}_p(0,1)} |\widetilde{\underline{\mathbf{W}}}^{(1)}\lambda|)+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} && (26)\\
\ge&-\epsilon(\max_{\lambda\in\mathbb{B}_p(0,1)} \|\widetilde{\underline{\mathbf{W}}}^{(1)}\|_q\|\lambda\|_p)+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} \text{(Hölder's inequality)} && (27)\\
=&-\epsilon\|\widetilde{\underline{\mathbf{W}}}^{(1)}\|_q+\widetilde{\underline{\mathbf{W}}}^{(1)}x_0+\widetilde{\underline{\mathbf{b}}}^{(1)} && (28)\end{aligned}$$

□

C.2 Details about bound propagation early-stop

We parse the objective function f into a computational graph $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ in PyTorch $\mathbf{V}$ and $\mathbf{E}$ are the set of nodes and edges and perform bounding on $\mathcal{G}$. The input $\boldsymbol{u}$, constant values like x_{t_0} and model parameters are the input nodes of $\mathcal{G}$ in the input set of $\mathcal{G}$, $\mathcal{I} = \{v \mid \text{In}(v) = \emptyset\}$ where $\text{In}(v) = \{w \mid (w, v) \in \mathbf{E}\}$ is the set of input nodes of node v. Any arithmetical operations like ReLU requiring input operands are node in $\mathcal{G}$ while their input sets are non-empty. o is the only output node of $\mathcal{G}$ which gives the scale objective value. Our method (Algorithm 3) takes graph $\mathcal{G}$ of f, the output node o to bound, and a set of early-stop nodes $\mathcal{S} \subset \mathbf{V}$ as the input and outputs the lower bounds of the value of o i.e., $\underline{f}^*$.

Algorithm 3 Bound Propagation w/ Early-stop.

1: **Function**: `compute_bound`
2: **Inputs**: computational graph $\mathcal{G}$, output node o, early-stop set $\mathcal{S}$
3: $\texttt{CROWN\_init}(\mathcal{G}, o)$
4: $Q \leftarrow \texttt{Queue}(), Q.\texttt{push}(o)$
5: **while** $\texttt{length}(Q) > 0$ **do**
6:     $v \leftarrow Q.\texttt{pop}()$
7:     **for** $w \in \text{In}(v)$ **do**
8:         $d_w \mathrel{-}= 1$
9:         **if** $d_w = 0$ and $w \notin \mathcal{I}$ **then**
10:             $Q.\texttt{push}(w)$
11:     **if** $v \in \mathcal{S}$ **then**
12:         **continue**
13:     $\texttt{CROWN\_prop}(v)$
14: $\underline{f}^* \leftarrow \texttt{CROWN\_concretize}(\mathcal{I}, \mathcal{S})$
15: **Outputs:** $\underline{f}^*$

It first perform `CROWN_init` to initialize d_v for all nodes v, the number of output nodes of v that have not been visited. It maintains a queue Q of nodes to visit and performs Breadth First Search on $\mathcal{G}$ starting from o. When it visits a node v, it traverses all input nodes w of v, decrementing d_w. When all its output nodes are visited and it is not an input node of $\mathcal{G}$, w is added to Q for propagation (Lines 7-10). The key modification lies in Lines 11-12, where it stops bound propagation from v to all input nodes if $v \in \mathcal{S}$. Finally, it concretizes the output bound $\underline{f}^*$ at nodes $v \in \mathcal{I} \cup \mathcal{S}$ based on their lower and upper bounds $\mathbf{l}_v$ and $\mathbf{u}_v$. We assume $\mathbf{l}_v$ and $\mathbf{u}_v$ are known for $v \in \mathcal{I}$ since we know the input range of $\boldsymbol{u}$ and all constant values and model parameters.

An illustrative example for bounding. We provide a step-by-step visualization of bounding on $f(\boldsymbol{u}) = \boldsymbol{u}_1 + (\boldsymbol{u}_1 - \boldsymbol{u}_2)$ with early-stop set $\mathcal{S} = \{-\}$ in Figure A6.

In **Step 1**, we initialize CROWN and the queue Q for traversal with the output node f. In **Step 2**, we update the out-degree of $+$, which is the input of f, and propagate from f to $+$. As $d_+ = 0$ indicates that all its outputs (here only f) have been visited, the node $+$ is added to Q. In **Step 3**, we continue propagation to the input nodes of $+$, which are $\boldsymbol{u}_1$ and $-$. Here only $-$ is added to Q, and only one of $\boldsymbol{u}_1$'s outputs $(+)$ is visited. In **Step 4**, we visit $-$, which is defined as an early-stop node; the backward flow stops propagating to its input nodes $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$. Also, $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are not added to Q since they do not input nodes.

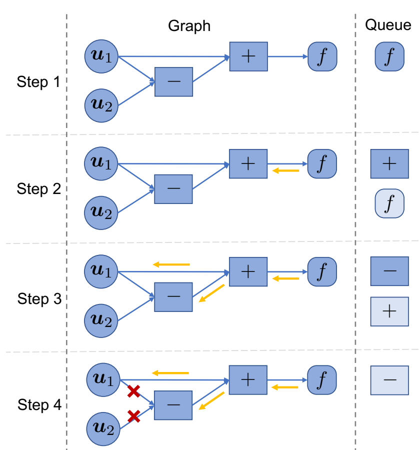


Figure A6: **Bound propagation with early-stop on $f(\boldsymbol{u}) = \boldsymbol{u}_1 + (\boldsymbol{u}_1 - \boldsymbol{u}_2)$.** In every node, the symbol is in black. The forward edges are in blue and backward flows are in yellow. In the queue, blue nodes are nodes popped and lightblue nodes are nodes added.

Hence, we perform the backward propagation from o to $\boldsymbol{u}_1$ and $+$. Without early-stop, we will continue to propagate $-$ to $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ after **Step 4** and use the bounds of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ to calculate $\underline{f}^*$. The more propagation through nodes may introduce more relaxations and make the final bound looser. While with early-stop, we do not propagate the bounds to $\boldsymbol{u}_1$ again and to $\boldsymbol{u}_2$.

Finally, we require the lower and upper bounds of $\boldsymbol{u}_1$ and $+$ to bound f. In CROWN, bound of $+$ can be obtained by recursively calling `compute_bound` with $o = +$ which propagates the bound of $+$ to $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ introduce extra looseness in bounding. While in our approach, we obtain the bound of $+$ empirically from samples during *searching*.

D NEURAL DYNAMICS MODEL LEARNING

We learn the neural dynamics model from the state-action pairs collected from interactions with the environment. Let the state and action at time t be denoted as x_t and u_t. Our goal is to learn a predictive model f_{dyn}, instantiated as a neural network, that takes a short sequence of states and actions with l-step history and predicts the next state at time $t+1$:

$$\hat{x}_{t+1} = f_{dyn}(x_t, u_t). \tag{29}$$

To train the dynamics model for better long-term prediction, we iteratively predict future states over a time horizon T_h and optimize the neural network parameters by minimizing the mean squared error (MSE) between the predictions and the ground truth future states:

$$\mathcal{L} = \frac{1}{T_h} \sum_{t=l+1}^{l+T_h} \|x_{t+1} - f_{dyn}(\hat{x}_t, u_t)\|_2^2. \tag{30}$$

More training details about model learning for every task will be given in Section E.2.

E EXPERIMENT DETAILS

E.1 DATA COLLECTION

For training the dynamics model, we randomly collect interaction data from simulators. For Pushing with Obstacles, Object Merging, and Object Sorting tasks, we use Pymunk (Blomqvist, 2022) to collect data, and for the Rope Routing task, we use FleX to generate data. In the following paragraphs, we will introduce the data generation process for different tasks in detail.

Pushing w/ Obstacles. As shown in Figure A7.a, the pusher is simulated as a 5mm cylinder. The stem of the T has a length of 90mm and a width of 30mm, while the bar has a length of 120mm and a width of 30mm. The pushing action along the x-y axis is limited to 30mm. We don't add explicit obstacles in the data generation process, while the obstacles are added as penalty terms during planning. We generated 32,000 episodes, each containing 30 pushing actions between the pusher and the T.

Object Merging. As shown in Figure A7.b, the pusher is simulated as a 5mm cylinder. The leg of the L has a length of 30mm and a width of 30mm, while the foot has a length of 90mm and a width of 30mm. The pushing action along the x-y axis is limited to 30mm. We generated 64,000 episodes, each containing 40 pushing actions between the pusher and the two Ls.

Object Sorting. As shown in Figure A7.c, the pusher is simulated as a rectangle measuring 45mm by 3.5mm. The radius of the object pieces is set to 15mm. For this task, we use long push as our action representation, which generates the start position and pushing action length along the x-y axis. The pushing action length is bounded between -100mm and 100mm. We generated 32,000 episodes, each containing 12 pushing actions between the pusher and the object pieces.

Rope Routing. As shown in Figure A7.d, we use a xArm6 robot with gripper to interact with the rope. The rope has a length of 30cm and a radius of 0.03cm. One end of the rope is fixed while the gripper grasps the other end. We randomly sample actions in 3D space, with the action bound set to 30cm. The constraint is that the distance between the gripper position and the fixed end of the rope cannot exceed the rope length. We generated 15,000 episodes, each containing 6 random actions. For this task, we will post-process the dataset and split each action into 2cm sections.

E.2 NEURAL DYNAMICS MODEL LEARNING

For different tasks, we choose different types of model architecture and design different input outputs. For Pushing with Obstacles, Object Merging, and Rope routing tasks, we use MLP as our dynamics model; And for the Object Sorting task, we utilize GNN as the dynamics model, since the pieces are naturally modeled by Graph. Below is the detailed information for each task.

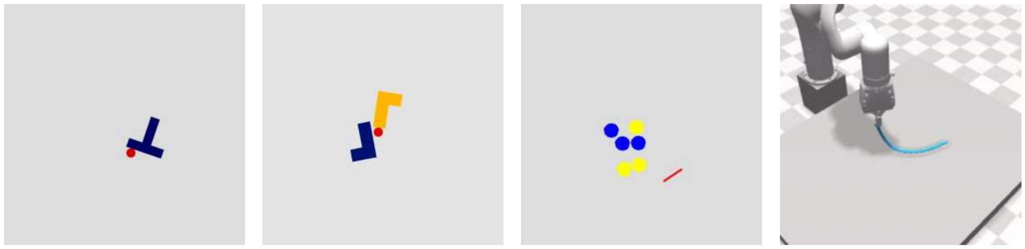


Figure A7: **Simulation environments visualization.** We use Pymunk to simulate environments involving only rigid body interactions. For manipulating deformable objects: ropes, we utilize NVIDIA Flex to simulate the interactions between the rope and the robot gripper.

Pushing w/ Obstacles. We use a four-layer MLP with [128, 256, 256, 128] neurons in each respective layer. The model is trained with an Adam optimizer for 7 epochs, using a learning rate of 0.001. A cosine learning rate scheduler is applied to regularize the learning rate. For the model input, we select four key points as shown in Figure A7.a, and calculate their relative coordinates to the current and next pusher positions. These coordinates are concatenated (resulting in a state dimension of 16) and input into the model. For the loss function, given the current state and action sequence, the model predicts the next 6 states, and we compute the MSE loss with the ground truth.

Object Merging. We use the same architecture, optimizer, training epochs, and learning rate scheduler as in the Pushing w/ Obstacles setup. For the model input, we select three key points for each L, as shown in Figure A7.b, and calculate their relative coordinates to both the current and next pusher positions. These coordinates are then concatenated (resulting in a state dimension of 12) and input into the model. We also use the same loss function as in the Pushing with Obstacles setup.

Object Sorting. We use the same architecture as DPI-Net (Li et al., 2018). The model is trained with an Adam optimizer for 15 epochs, with a learning rate of 0.001, and a cosine learning rate scheduler to adjust the learning rate. For the model input, we construct a fully connected graph neural network using the center position of each piece. We then calculate their relative coordinates to the current and next pusher positions. These coordinates are concatenated as the node embedding and input into the model. For the loss function, given the current state and action sequence, the model predicts the next 6 states, and we compute the MSE loss with the ground truth.

Rope Routing. We use a two-layer MLP with 128 neurons in each layer. The model is trained with an Adam optimizer for 50 epochs, with a learning rate of 0.001, and a cosine learning rate scheduler to adjust the learning rate. For the model input, we use farthest point sampling to select 10 points on the rope, reordered from closest to farthest from the gripper. We then calculate their relative coordinates to both the current and next gripper positions, concatenate these coordinates, and input them into the model. For the loss function, given the current state and action sequence, the model predicts the next 8 states, and we compute the MSE loss with the ground truth.

E.3 Model-Based Planning

In this section, we will introduce our cost functions for model-based planning Eq. 1 across different tasks. For every task, we assume the initial and target state x_0 and x_{target} are given. We denote the position of the end-effector at time t as p_t. In tasks involving continuous actions like Pushing w/ Obstacles, Object Merging, and Rope Routing, action u_t is defined as the movement of end-effector, $p_t = p_{t-1} + u_t$ and p_0 is given by initial configuration. In the task of Object Sorting involving discrete pushing, p_t is given by the action a_i as the pusher position before pushing. In settings with obstacles, we set the set of obstacles as O. Every $o \in O$ has its associated static position and size as p_o and s_o.

Pushing w/ Obstacles. As introduced before, we formalize the obstacles as a penalty term rather than explicitly introducing them in the dynamics model. Our cost function is defined by a cost to the goal position plus a penalty cost indicating whether the object or pusher collides with the obstacle. The detailed cost is listed in Equation 31. Ideally, c_T can be optimized to 0 by a strong planner with the proper problem configuration.

$$\begin{aligned} c_t = c(x_t, u_t) = w_t \left\| x_t - x_{\text{target}} \right\| \\ + \lambda \sum_{o \in O} (\text{ReLU}(s_o - \|p_t - p_o\|) + \text{ReLU}(s_o - \|x_t - p_o\|)) \end{aligned} \tag{31}$$

where $\|x_t - x_{\text{target}}\|$ gives the difference between the state at time t and the target. $\|p_t - p_o\|$ and $\|x_t - p_o\|$ give the distance between the obstacle o and the end-effector and the object. Two ReLU items yield positive values (penalties) when the pusher or object are located within the obstacle o. w_t is the weight increasing with time t to encourage the alignment to the target. λ is the large constant value to avoid any collision. In implementation, x_t is a concatenation of positions of keypoints, $\|x_t - p_o\|$ is calculated keypoint-wisely. Ideally, c_T can be optimized to 0 by a strong planner with the proper problem configuration.

Object Merging. In this task requiring long horizon planning to manipulate two objects, we don't set obstacles and only consider the different between state at every time step and the target. The cost is shown in Equation 32.

$$c_t = w_t \left\| x_t - x_{\text{target}} \right\| \tag{32}$$

Object Sorting. In this task, a pusher interacts with a cluster of object pieces belonging to different classes. We set y_goal as the target position for every class. Additionally, for safety concerns to prevent the pusher from pressing on the object pieces, we introduce obstacles defined as the object pieces in the cost Equation 33. For every object piece o, its size s_o is set as larger than the actual size in the cost and its position p_o is given by x_t. with the sizes larger than that of objects. The definition of the penalty is similar to that in Pushing w/ Obstacles.

$$c_t = w_t \left\| x_t - x_{\text{target}} \right\| + \lambda \sum_{o \in O} \text{ReLU}(s_o - \|p_t - p_o\|) \tag{33}$$

Rope Routing. In this task containing the deformable rope, we sample some keypoints by Farthest Point Sampling (FPS). x_{target} is defined as the target positions of sampled keypoints. The cost is defined in Equation 34 which is similar to the one in pushing w/ obstacles. Here, two obstacles are introduced to form the tight-fitting slot. In implementation, naively applying such cost does not always achieve our target routing the rope into the slot since a trajectory greedily translating in z-direction without lift maybe achieve optimum. Hence, we additionally modify the formulation by assigning different weights for different directions (x, y, z) when calculating $\|x_t - x_{\text{target}}\|$ to make sure the desirable trajectory yields the lowest cost.

$$c_t = w_t \left\| x_t - x_{\text{target}} \right\| + \lambda \sum_{o \in O} (\text{ReLU}(s_o - \|p_t - p_o\|) + \text{ReLU}(s_o - \|x_t - p_o\|)) \tag{34}$$

F Experimental Results

Synthetic example. Before deploying our BaB-ND on robotic manipulation tasks, we create a synthetic function to test its capability to find optimal solutions in a highly non-convex problem. We define $f(\boldsymbol{u}) = \Sigma_{i=0}^{d-1} 5\boldsymbol{u}_i^2 + \cos 50\boldsymbol{u}_i,\ \boldsymbol{u} \in [-1, 1]^d$ where d is the input dimension. The optimal solution $f^* \approx -1.9803d$ and $f(\boldsymbol{u})$ has 16 local optima with two global optima on every dimension. Hence, optimizing $f(\boldsymbol{u})$ can be challenging when d increases.

In Figure A8, we visualize the best objective values found by different methods over different input dimensions up to $d = 100$. BaB-ND consistently outperforms all baselines which converge to non-ideal sub-optimal values. For $d = 100$, BaB-ND can achieve optimality on 98 to 100 dimensions. This synthetic experiment demonstrate the potential of BaB-ND on neural dynamics planning tasks.

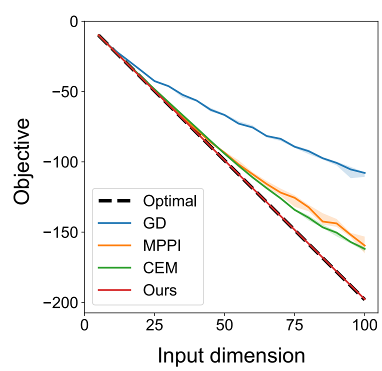


Figure A8: **Optimization result on a synthetic $f(\boldsymbol{u})$ over increasing dimensions d.** BaB-ND outperforms all baselines on the optimized objective.

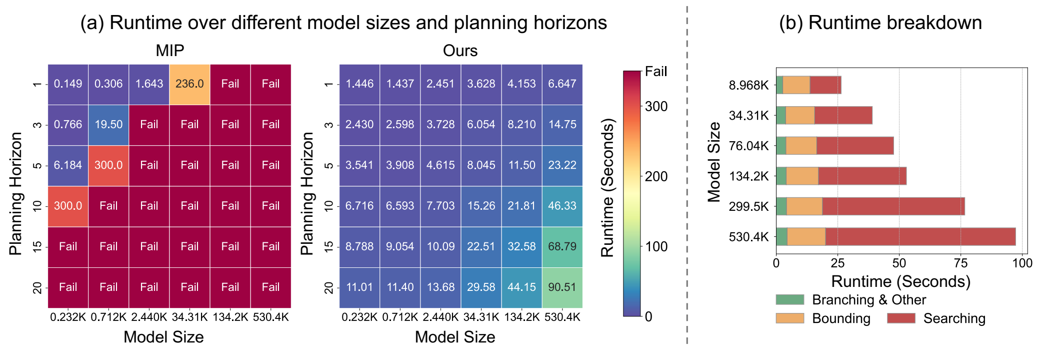


Figure A9: **Quantitative analysis of runtime and scalability.** (a) The runtime of MIP and ours on solving simple planning problems with different model sizes and planning horizons. BaB-ND can handle much larger problems than MIP. (“Fail” indicates MIP fails to find any feasible solution within 300 seconds.) (b) Runtime breakdown of our components on large and complex planning problems with $H = 20$. Runtimes on components except searching increase a little with increasing of model size, indicating the excellent scalability of our approach.

Scalability. We evaluate the scalability of our BaB-ND on object pushing task with different model sizes and different planning horizons on multiple test cases comparing with MIP (Liu et al., 2023). We train the neural dynamics models with different sizes and the same architecture and use the number of parameters in the single neural dynamics model f_{dyn} to indicate the model size.

In Figure A9 (a), we visualize the average runtime of MIP and ours on test cases with different model sizes and planning horizons. To be friendly to MIP, we remove all items about the obstacles and define the objective as the step cost after planning horizon H, $c(s_{t_0+T}, x_{t_0+T}, s_{\text{goal}})$ instead of the accumulated cost. However, MIP still only handles small problems. Among all 36 settings, it gives optimal solutions on 6 settings, gives sub-optimal solutions on 3 settings, and fails to find any solution on all remaining settings within 300 seconds. On the contrary, our BaB-ND scales up well to large problems with planning horizon $H = 20$ and a model containing over 500K parameters.

In Figure A9 (b), we evaluate the runtime of each primary component of our BaB-ND across various model sizes, ranging from approximately 9K to over 500K, in the context of an original objective for the pushing w/ obstacles tasks (containing items to model obstacles and accumulated cost among all steps) over a planning horizon of $H = 20$. The breakdown bar chart illustrates that the runtimes for the *branching* and *bounding* components grow relatively slowly across model sizes, which increase by over 50-fold. Our improved bounding procedure, as discussed in Section 2.2, scales well with growing model size. In addition, the *searching* runtime scales in proportion to neural network size since the majority of searching time is spent on sampling the model with a large batch size on GPUs.