6 Appendix

6.1 Extended related works

There are also many works on neural CBFs, but they target problems unrelated to input saturation. Some examples: learning an unknown safety criterion from safe expert trajectories [33, 34], jointly learning a safe policy and safety certificate via reinforcement learning [35, 36], optimizing the task performance of a CBF-based controller [37], and learning dynamics under model uncertainty [38].

6.2 Appendix for preliminaries

Defining the safe sets: we define the safe sets corresponding to the limit-blind CBF and modified CBF. We need to first define the following functions, for all j in $[1, r-1]$ where r is the relative degree betweeen safety specification $\rho(x)$ and input u:

$$\phi_j = \left[\prod_{i=1}^{j}\left(1 + c_i \frac{\partial}{\partial t}\right)\right]\rho \quad , \forall j \in [1, r-1] \tag{9}$$

and then from [5], we have

$$\mathcal{S} = \mathcal{A} \cap \left[\bigcap_{j=1}^{r-1}\{\phi_j\}_{\leq 0}\right] \qquad \text{(limit-blind safe set)}$$

$$\mathcal{S}^* = \mathcal{S} \cap \{\phi^*\}_{\leq 0} \qquad \text{(modified safe set)}$$

For this paper, it is also convenient to indicate the function $\mathrm{ss}^*(x)$ that implicitly defines the safe set $\mathcal{S}^*$ as its 0-sublevel set.

$$\mathcal{S}^* = \{\mathrm{ss}^*(x)\}_{\leq 0} \tag{10}$$

$$\mathrm{ss}^*(x) = \max_{\forall j}(\rho(x), \phi_j(x), \phi^*(x)) \tag{11}$$

Comparison to the class-κ CBF formulation: there is a different CBF formulation that requires

$$\dot{\phi}(x) \leq -\alpha(\phi(x)) \quad , \quad \forall x \in \mathcal{D} \tag{12}$$

for a class-κ function $\alpha : \mathbb{R} \to \mathbb{R}$ ($\alpha(0) = 0$, α is continuous and monotonically increasing). While this is a stricter constraint than ours (it constrains the inputs at all states, not just boundary states), the benefit is that it produces a smooth control signal. It could be possible to extend our method to CBFs of this variety. The only major change is that the critic would search for counterexamples in the domain $\mathcal{D}$, rather than just along the safe set boundary, $\partial\mathcal{S}^*$. One could learn the $\alpha(\cdot)$ function as well, parametrizing as a monotonic NN [39].

6.3 Appendix for methodology

More recommendations for the design of $\rho^*(x)$
If it is very awkward to choose an x_e in Eqn. 5, then another design can be used. For example, $\rho^*(x)$ can be:

$$\rho^*(x) = \mathrm{softplus}(\mathrm{nn}(x)) + \rho(x) \tag{13}$$

The disadvantage of such a design is that the safe set might be empty (Contraint 3 is violated). It might be fine to starting training with no safe set, since the volume regularization term in the objective may slowly create a safe set.

Helper functions for training algorithm

Algorithm 2 Sampling uniformly on a boundary (MSample from [40])

function SAMPLEBOUNDARY(θ, N_{samp}) ▷ Note that θ defines the boundary, $\partial\mathcal{S}^*$
  Set error parameter $\epsilon \in (0, 1]$ to 0.01, boundary attribute τ to 0.25, n to state space dim.
  $\mathbb{X}_{samp} \leftarrow \{\}$
  $\sigma \leftarrow 2\left(\tau\sqrt{\epsilon}/4(n + 2\ln(1/\epsilon))\right)^2$ ▷ Set hyperparam. to meet sampling guarantees
  While size of $\mathbb{X}_{samp} < N_{samp}$:
    $p \leftarrow$ sample uniformly inside $\mathcal{S}^*$
    $q \leftarrow$ sample from Gaussian$(p, \sigma \cdot I_{n\times n})$
    $x \leftarrow$ attempt to intersect segment $\overline{pq}$ with boundary
    If x is not none:
      Add x to $\mathbb{X}_{samp}$
  return $\mathbb{X}_{samp}$
end function

Algorithm 3 Projecting to a boundary

function PROJTOBOUNDARY($\mathbb{X}, \theta$) ▷ Project set $\mathbb{X}$ to boundary defined by θ
  Set learning rate $\gamma = 0.01$
  $\text{ss}^*_\theta \leftarrow$ function that defines the boundary implicitly ($\partial\mathcal{S}^* \triangleq \{\text{ss}^*_\theta\}_{=0}$)
  Repeat:
    $\mathbb{X} \leftarrow \mathbb{X} - \gamma \cdot \nabla_{\mathbb{X}} |\text{ss}^*_\theta(\mathbb{X})|$ ▷ Batch GD
  Until convergence
  return $\mathbb{X}$
end function

For the critic, the first step to computing boundary counterexamples is sampling on the boundary. Alg. 3 from [40] provides a method to uniformly sample on manifolds with bounded absolute curvature and diameter. The algorithm finds points on the boundary by sampling line segments and checking if they intersect the boundary. An important trait of the algorithm is that it is efficient: the number of evaluations of the membership function ss^*_θ does not depend on the state space dimension, n. It only depends on the curvature of the boundary (captured by an inversely proportional "condition number", τ) and the error threshold, ϵ, which bounds the total variation distance between the sampling distribution and a true uniform distribution. We do not measure or estimate τ; for our purposes, it is enough to set it sufficiently small. Another essential helper routine (Alg. 5) projects states onto the boundary. The boundary $\partial\mathcal{S}^*$ is implicitly defined as the 0-level set of a function ss^*_θ. Thus, we can simply apply gradient descent to minimize $|\text{ss}^*_\theta(x)|$ toward 0, which is a "good enough" approximate projection scheme.

6.4 Appendix for experiments

System model for quadcopter-pendulum

In our 10D state and 4D input model, the states are roll-pitch-yaw quadcopter orientation (γ, α, β) and roll-pitch pendulum orientation (ϕ_p, θ_p), as well as the first derivatives of these states. The inputs are thrust and torque $(F, \tau_\gamma, \tau_\beta, \tau_\alpha)$, which are limited to a bounded convex polytope set. The quadcopter dynamics are from [41], which models the inputs realistically, and the pendulum dynamics are from [31]:

$$\begin{bmatrix}\ddot{\gamma}\\ \ddot{\beta}\\ \ddot{\alpha}\end{bmatrix} = R(\gamma,\beta,\alpha)J^{-1}\begin{bmatrix}\tau_\gamma\\ \tau_\beta\\ \tau_\alpha\end{bmatrix} \tag{14}$$

$$\begin{bmatrix}\ddot{\phi}\\ \ddot{\theta}\end{bmatrix} = \begin{bmatrix}\frac{3}{2mL_p\cos\theta}(k_y(\gamma,\beta,\alpha)\cos\phi + k_z(\gamma,\beta,\alpha)\sin\phi)\\ \frac{3}{2mL_p}(-k_x(\gamma,\beta,\alpha)\cos\theta - k_y(\gamma,\beta,\alpha)\sin\phi\sin\theta + k_z(\gamma,\beta,\alpha)\cos\phi\sin\theta)\end{bmatrix}(F+mg) + \begin{bmatrix}2\dot{\theta}\dot{\phi}\tan\theta\\ -\dot{\phi}^2\sin\theta\cos\theta\end{bmatrix} \tag{15}$$

where $R(\gamma, \beta, \alpha)$ rotates between the quadcopter and world frame and is computed as the composition of the rotations about the X, Y, Z axes of the world frame. Also, the variables $k_x(\gamma, \beta, \alpha), k_y(\gamma, \beta, \alpha), k_z(\gamma, \beta, \alpha)$ are defined as:

$$R(\gamma, \beta, \alpha) \triangleq R_z(\alpha) R_y(\beta) R_x(\gamma) \tag{16}$$

$$k_x(\gamma, \beta, \alpha) \triangleq (\cos\alpha \sin\beta \cos\gamma + \sin\alpha \sin\gamma) \tag{17}$$

$$k_y(\gamma, \beta, \alpha) \triangleq (\sin\alpha \sin\beta \cos\gamma - \cos\alpha \sin\gamma) \tag{18}$$

$$k_z(\gamma, \beta, \alpha) \triangleq (\cos\beta \cos\gamma) \tag{19}$$

and $J = \text{diag}(J_x, J_y, J_z) = \text{diag}(0.005, 0.005, 0.009)\ \text{kg} \cdot \text{m}^2$ contains the moments of inertia of the quadcopter, $m = 0.84$ kg is the mass of the combined system, $L_p = 0.03$ m is the length of the pendulum. The values of these physical parameters are taken from the default values in a high-fidelity quadcopter simulator, jMAVSim [42] and also extrapolated from the real-world experiments in [31]. Our control inputs are limited to a convex polyhedral set defined by:

$$\mathcal{U} \triangleq \{u \mid u + [mg, 0, 0, 0] = Mv, \text{for some } v \in [\vec{0}, \vec{1}]\} \tag{20}$$

$$M \triangleq \begin{bmatrix} k_1 & k_1 & k_1 & k_1 \\ 0 & -\ell k_1 & 0 & \ell k_1 \\ \ell k_1 & 0 & -\ell k_1 & 0 \\ -k_2 & k_2 & -k_2 & k_2 \end{bmatrix} \tag{21}$$

with $\ell = \frac{0.3}{2}, k_1 = 4.0, k_2 = 0.05$ from jMAVSim. The interpretation of this is that v contains the low-level motor command signals at each rotor, which are limited between 0 to 1, and we linearly transform them to thrusts and torques using the *mixer matrix* M, which is derived using first principles [41]. Finally, we perform a change of variables on the input by adding mg to the thrust so that the origin is an equilibrium ($\dot{x}|_{x=0} = 0$).

Baseline details

Hand-designed CBF: the system was not very intuitive to reason about, so we picked a simple and general function form from [8, 15]:

$$\rho^* = (\gamma^2 + \beta^2 + \delta_p^2)^{a_1} - (\pi/4)^{2 \cdot a_1} + a_2 \tag{22}$$

where $a_1 > 0, a_2 \geq 0$ are the parameters. The parameters were optimized with an evolutionary algorithm, Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [43] using the same objective function from our method for fairness. After tuning the hyperparameters of CMA-ES, the best parametrization we found was:

$$\rho^* = (\gamma^2 + \beta^2 + \delta_p^2)^{3.76} - ((\pi/4)^2)^{3.76} \tag{23}$$

$$\phi^* = \rho^* + 0.01 \cdot \dot{\rho}^* \tag{24}$$

Safe MPC: the MPC formulation was kept similar to CBF-QP. The objective here is also to minimize modification to $k_{nom}(x)$ while keeping the trajectory safe and forward invariant.

$$\min_{u(t) \in \mathcal{U}} \int_0^T \|u(t) - k_{nom}(x(t))\|_2^2 \, \partial t \tag{25}$$

$$x(0) = x_0 \tag{26}$$

$$\dot{x}(t) = f(x(t)) + g(x(t))u(t) \tag{27}$$

$$\rho(x(t)) \leq 0, \forall t \in [0, T] \tag{28}$$

$$\rho_b(x(T)) \leq 0 \qquad \text{(terminal constraint)}$$

The terminal constraint ensures invariance (safety for all time) of the MPC solution by enforcing the last predicted state $x(T)$ to lie in an invariant set defined by $\rho_b(x)$. We set $\rho_b(x)$ to be the approximated region of attraction of an LQR stabilizing controller: $\rho_b(x) = \|x\|_2^2 - 0.1$.

Training details

Random seeds: We trained a neural CBF for the quadcopter-pendulum problem on 5 different random seeds. The random seed affects the neural CBF initialization, the critic's counterexamples, etc.

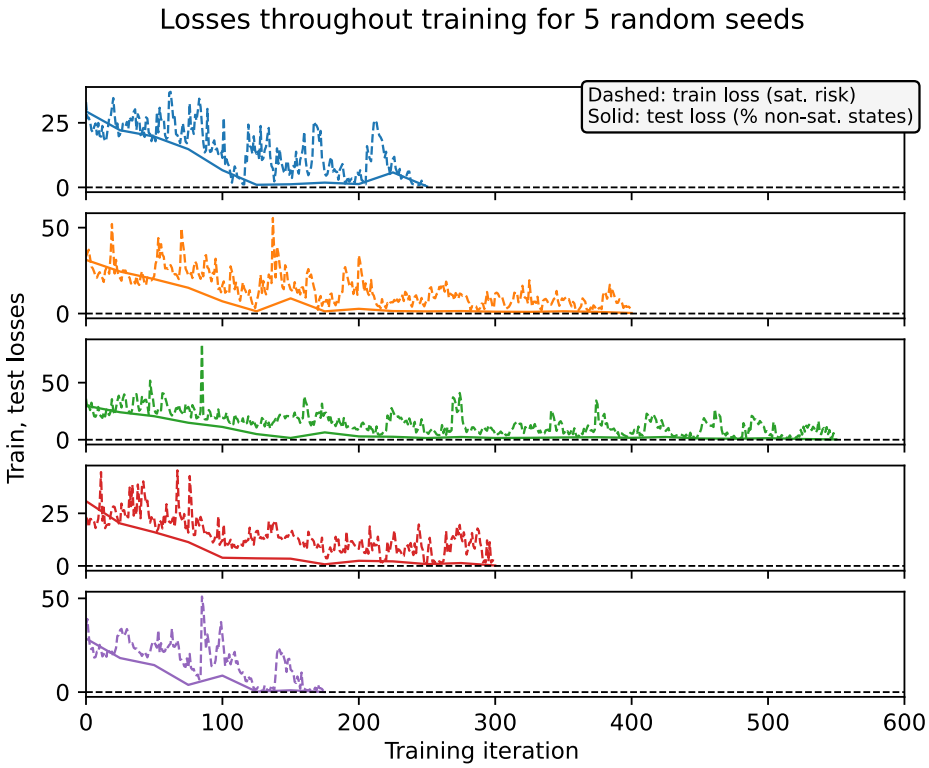
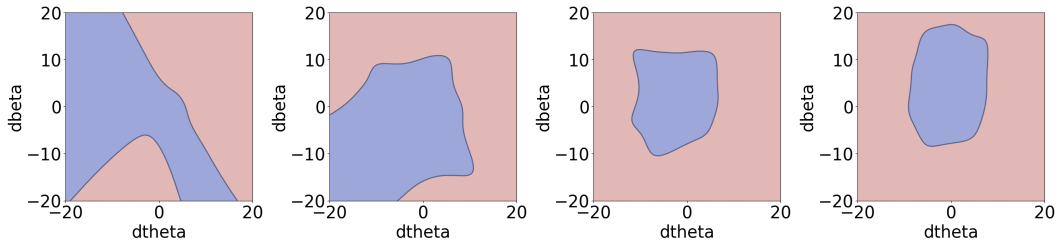


Figure 3: Plot of train and test loss over training for 5 runs with different random seeds. The black dashed line marks 0, the target loss. As we can see, the runs finish training in different lengths of time, but they all ultimately train successfully (reach ≈ 0 loss).



(a) Iter. 0 (b) Iter. 100 (c) Iter. 120 (d) Iter. 250

Figure 4: An axis-aligned 2D slice (depicting $\dot{\theta}$ (pendulum pitch velocity) vs. $\dot{\beta}$ (quadcopter pitch velocity) of the 10D safe set, at four points during training. The safe set being learned is in blue.

The test loss (% non-saturating states) consistently reaches ≈ 0 across seeds; the seeds only affect how long it takes to reach this loss (14 ± 4 hours, on average). For Table 1, we chose the run that yielded a CBF that balances performance and a large safe set volume.

Training hyperparameters: (1) Critic: takes 20 gradient steps with learning rate $1\mathrm{e}{-3}$ to optimize batch of size 500. To initialize the batch, uses 50% uniform random samples, 50% warmstarted from the previous critic call. (2) Neural CBF: $\mathrm{nn} : \mathbb{R}^n \to \mathbb{R}$ is a multilayer perceptron with 2 hidden layers (sizes 64, 64) and $\tanh, \tanh, \mathrm{softplus}$ activations. It uses the default Xavier random initialization [44]; the c_i coefficients are initialized uniformly in $[0, 0.01]$. (3) Regularization: we used regularization weight 150.0 and 250 state samples in $\mathcal{D}$ to compute the regularization term. (4) Learner: used learning rate $1\mathrm{e}{-3}$.

Ablation study: We conduct ablation to analyze the effect of two key design choices: (1) our regularization term and (2) batch computing counterexamples.

Reg weight	Volume (as % of domain volume)
0	5.72e−3
10	8.36e−3
50	1.34e−2
200	1.91e−2

Batch size	Training time (h)
1	-
10	2.50
50	1.88
100	1.88
500	11.00

Table 2: (Left) Demonstrating how increasing the regularization weight effectively increases the volume of the learned safe set. (Right) Demonstrating how using a medium-sized *batch* of counterexamples can provide significant speed gains. Batch size 1 didn't finish.

For the regularization term, we measure the impact of the regularization weight on the volume of the learned safe set. We varied the weight between 0 (no regularization) and 200 and chose learned safe sets that attained a similarly low loss (within 0.05 of each other). Next, we approximated the safe set volume by sampling: we took 2.5 million uniformly random samples in the state domain and checked whether they belonged to the safe set. We see in Table 2 that increasing the regularization weight effectively increases the volume.

For batch computing counterexamples, we measure the effect of batch size on the training time. To compute training time, we consider training finished when the loss drops below a certain threshold (note that batch size 1 didn't finish). We might expect that for medium-sized batches improve the quality of the counterexamples (since there are more counterexample options in a batch), resulting in more efficient training. On the other hand, we also expect that for larger batch sizes, the overhead of creating the batch (mainly, sampling a large number of points on the boundary) exceeds any speed gains. In fact, this is what we observe in Table 2: as we increase the batch size from 10 to 50, the training speed improves by 25%. But a further increase from 100 to 500 sees the speed drop due to the aforementioned overhead.

Testing details

Implementing safe control in discrete time: Our CBF has a continuous-time formulation, and moreover, yields discontinuous control which is abruptly activated at the boundary. This means that in discrete time, where the system state is sampled at some frequency, the system might reach the boundary *in between* samples and the safe control may not kick in to prevent exiting from $\mathcal{S}^*$. In this case, we should have safe control kick in slightly before the boundary. This can be at a fixed distance from the boundary (at $\mathrm{ss}^*(x) = -\epsilon$ for small $\epsilon > 0$) or we can leverage the known dynamics to apply safe control when the boundary would otherwise be crossed in the next time step. We use the latter approach when simulating rollouts for Table 1. Another way to address this could be to seek finite-time or asymptotic convergence guarantees, in addition to forward invariance guarantees. If we had them, the system would be returned quickly to $\mathcal{S}^*$ should it ever exit. This is acceptable in most cases, as it is only mandatory for the system to stay inside the user-specified allowable set $\mathcal{A}$, which contains $\mathcal{S}^*$. Generally, the system will exit $\mathcal{S}^*$ without exiting $\mathcal{A}$.

Testing hyperparameters: (1) For the metric "% of non-saturating states on $\partial\mathcal{S}^*$", we used 10K boundary samples. The critic used to compute the worst saturation used a batch of size 10K and took 50 gradient steps, during which its objective converged. (2) For the metric "% of simulated rollouts that are FI", we used 5K rollouts. To approximate the volume of the safe set, we calculated the percentage of samples in $\mathcal{D}$ falling within $\mathcal{S}^*$, for 1 million samples.

Details for k_{lqr}: We construct an LQR controller to stabilize the full 16D system to the origin in the typical way: we find the linearized system $\dot{x} = Ax + Bu$, let $Q = I_{16\times16}, R = I_{4\times4}$, and compute the linear feedback matrix K. For a nonlinear system such a quadcopter-pendulum, this stabilizing controller only has a small region of attraction about the origin. Thus, it may produce unsafe behavior when initialized further from the origin, so a CBF safeguard is useful.

Details on inverted pendulum volume comparison (from Fig. 1): For our toy inverted pendulum problem with a 2D state space and 1D input space, we are curious about how our learned, volume-regularized safe set compares to the largest possible safe set. The largest safe set is the set of all states from which a safe trajectory exists (that is, a trajectory keeping within allowable set $\mathcal{A}$). We can identify most of these states by checking, for every state x_{start} in the two-dimensional domain $\mathcal{D}$, if such a trajectory can be found. Specifically, we pose the following nonlinear program to the

do-mpc Python package [45]:

$$\min_{u(t)} \int_0^T \rho(x(t))\partial t \tag{29}$$

$$\text{s.t. } u(t) \in \mathcal{U} \ , \ \forall t \in [0, T] \tag{30}$$

$$x(0) = x_{start} \tag{31}$$

$$\dot{x}(t) = f(x) + g(x)u(t) \tag{32}$$

where recall that $\mathcal{A}$ is defined as $\{\rho\}_{\leq 0}$ and T is a sufficiently long time horizon. Besides MPC, another way to compute the largest safe set would be to use HJ reachability [21]. However, the problem of finding the largest safe set under input limits is NP-hard, so we can only compute this baseline for our toy inverted pendulum problem and not the higher-dimensional quadcopter-pendulum problem.

Testing robustness to model mismatch and stochastic dynamics:

Noise variance	% FI rollouts
1	99.42
2	99.11
5	93.47
10	85.84

Inertia off by a factor of...	% FI rollouts
0.75	99.66
1.00	99.62
1.25	99.20
1.5	97.51
2	89.18
5	53.33

Table 3: (Left) Rollout metrics computed for our learned CBF under stochastic dynamics (when the spread of the zero-mean, Gaussian noise is varied). (Right) Rollout metrics computed for our learned CBF under model mismatch (when the moments of inertia of the quadcopter are off by a factor).

We test whether our learned CBF still ensures safety when our assumption of a known, deterministic system is broken. First, we consider what happens if the model is unknown. Specifically, we consider the case where some model parameters (the quadcopter's moments of inertia) have been misidentified (are all off by a factor). We expect that if the inertia is greater than believed, our learned safe controller will probably intervene too late to save the higher-inertia system. In Table 4, we see this is true. Our safe controller becomes increasingly ineffective at preserving safety as the true inertia increases. On the other hand, when inertia is smaller than expected, the system will be easier to save than expected, which means the same level of safety is preserved (compare first and second rows of Table 4). Second, we consider what happens if the model is stochastic. We consider a system with additive white Gaussian noise: $\dot{x} = f(x) + g(x)u + w$, with $w \in \mathcal{N}(0, \sigma)$ (0-mean, σ-variance Gaussian). As the variance of the noise increases, the system departs further from its assumed dynamics, and our safe controller fails more and more to ensure safety. Note that we have run rollouts with $k_{nom,LQR}$ (a stabilizing LQR nominal controller).

Noise variance	% FI rollouts
1	99.42
2	99.11
5	93.47
10	85.84

Inertia off by a factor of...	% FI rollouts
0.75	99.66
1.00	99.62
1.25	99.20
1.5	97.51
2	89.18
5	53.33

Table 4: (Left) Rollout metrics computed for our learned CBF under stochastic dynamics (when the spread of the zero-mean, Gaussian noise is varied). (Right) Rollout metrics computed for our learned CBF under model mismatch (when the moments of inertia of the quadcopter are off by a factor).

Elaborating on limitations: We mentioned that we had to perform a change of variables on the states input to the neural CBF before it would train successfully. Specifically, we changed the pendulum's angular velocity to a linear velocity and the quadcopter's angular velocity to the linear velocity of its vertical body axis. However, we note that after we made this adjustment, the rest of the synthesis required no human intervention.