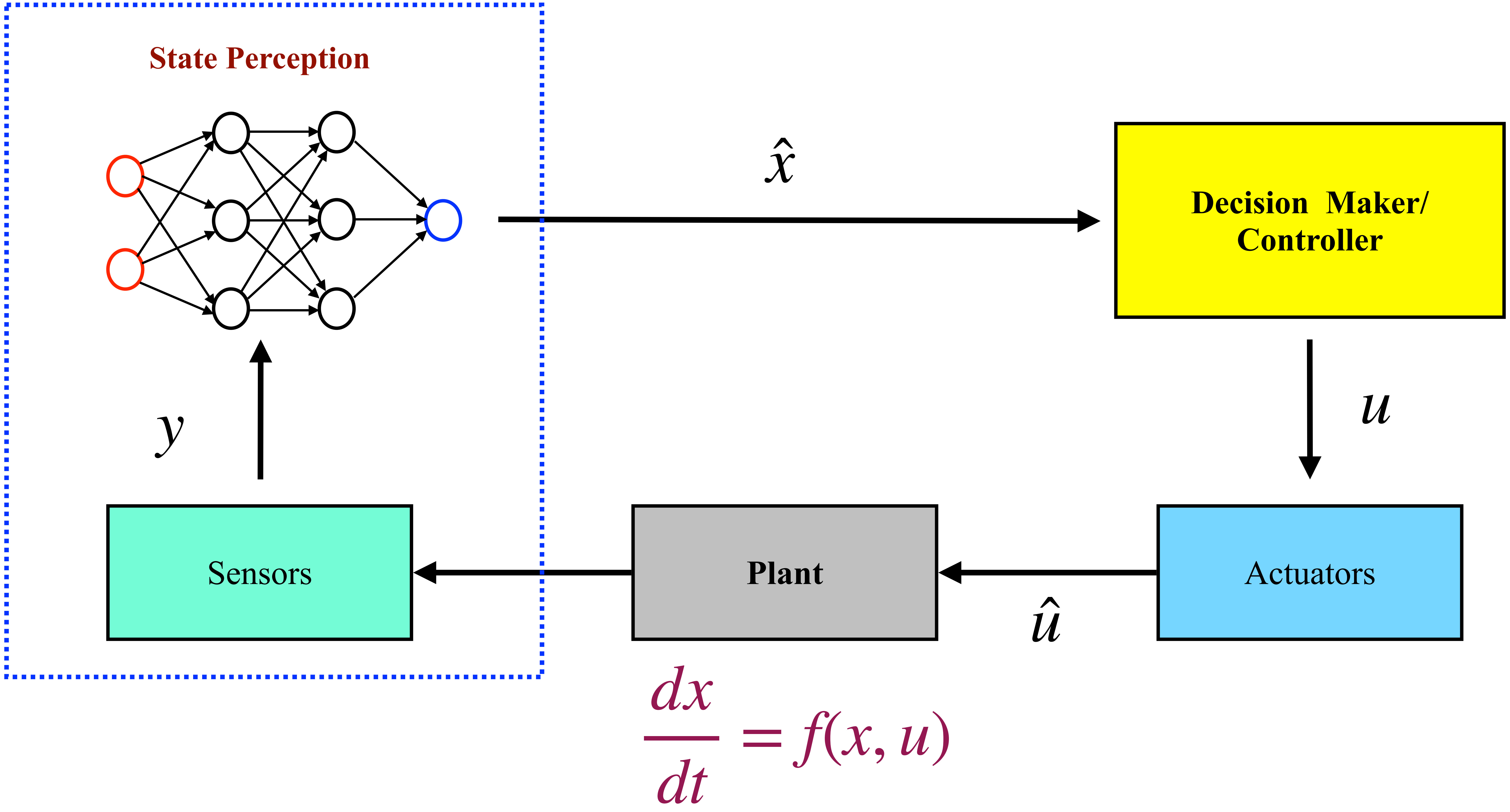
Functional Overapproximation for Neural-Network Controlled Systems

Xin Chen

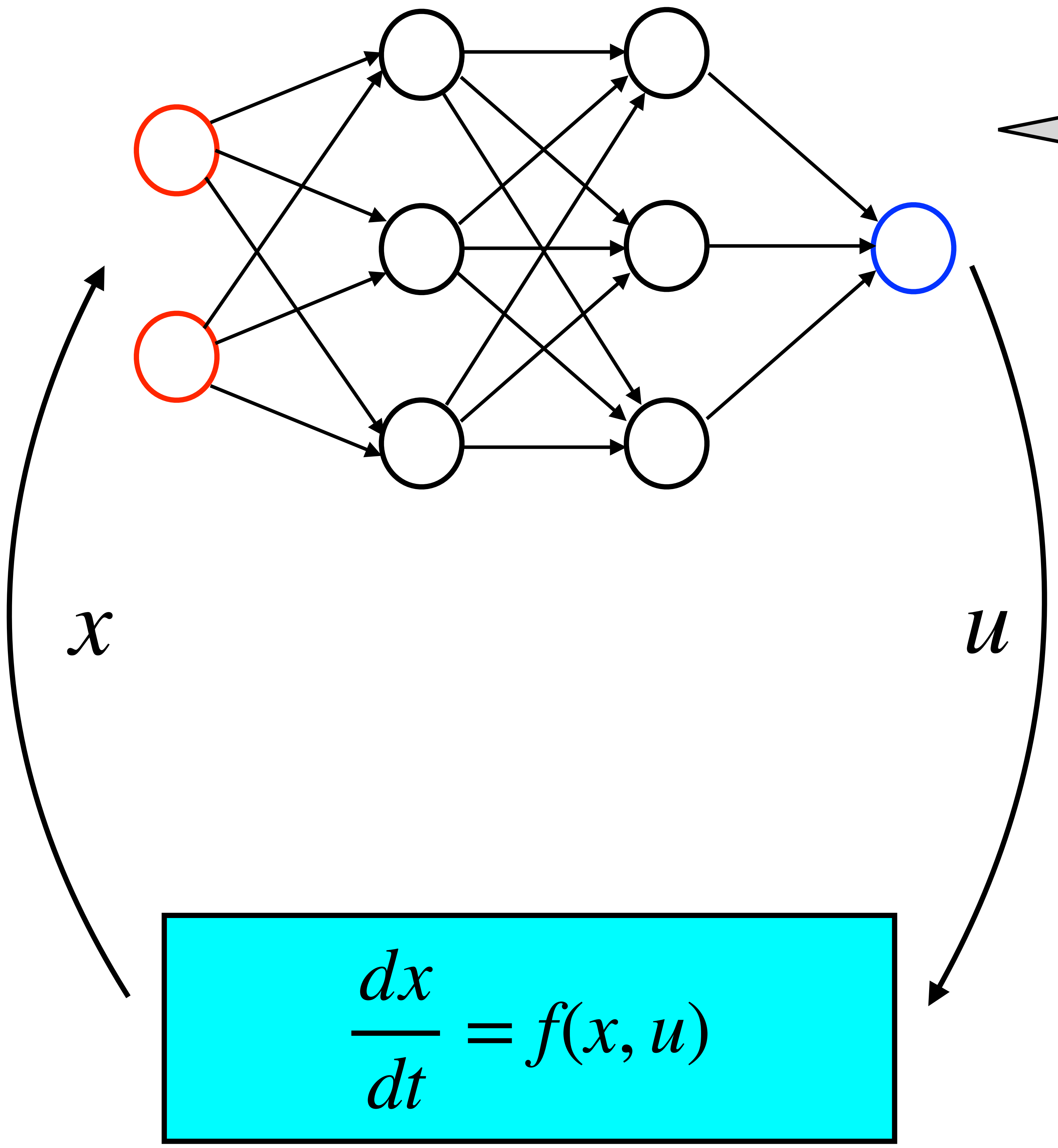
Assistant Professor, University of Dayton, USA.

The 18th International Summer School on Trustworthy Software. 2022.

Neural-Network Controlled System (NNCS)



Simplified NNCS



- The controller is a **feedforward** neural network.
- The neural network takes the system states as the input at the time $t = i\delta$ for $i = 0, 1, \ldots.$
- The neural network produces the value for the control input u which will be used in the current step, i.e., $t \in [i\delta, (i+1)\delta]$.
- The response time of the neural network controller is ignored.

Reachability Analysis of NNCS

- NNCS are still **state feedback systems**, and we may still use the main reachability analysis framework for NNCS.
- The plant of an NNCS is still defined by an **ODE**.
- The feedback law is not defined by a simple expression but a **neural network**.
- Assume that all of the existing techniques can still be well used on the system components except the neural network.
- We need at least a new approach to compute the reachable set **under a neural network mapping**.

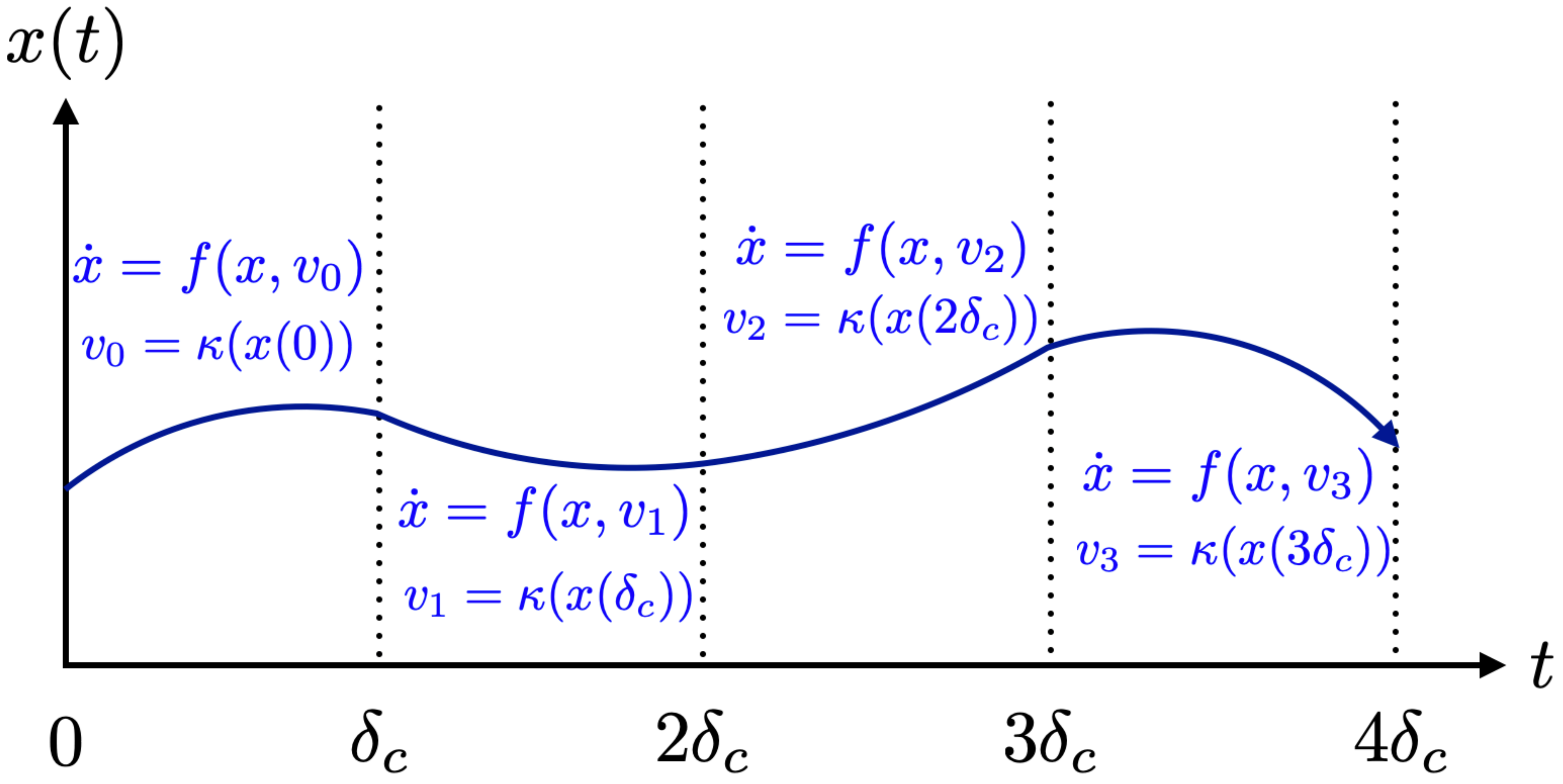
Formal Definition of an Execution

- We assume that the plant is defined by the ODE $\dot{x} = f(x, u)$.

- The input-output mapping of the neural network controller is denoted by κ.

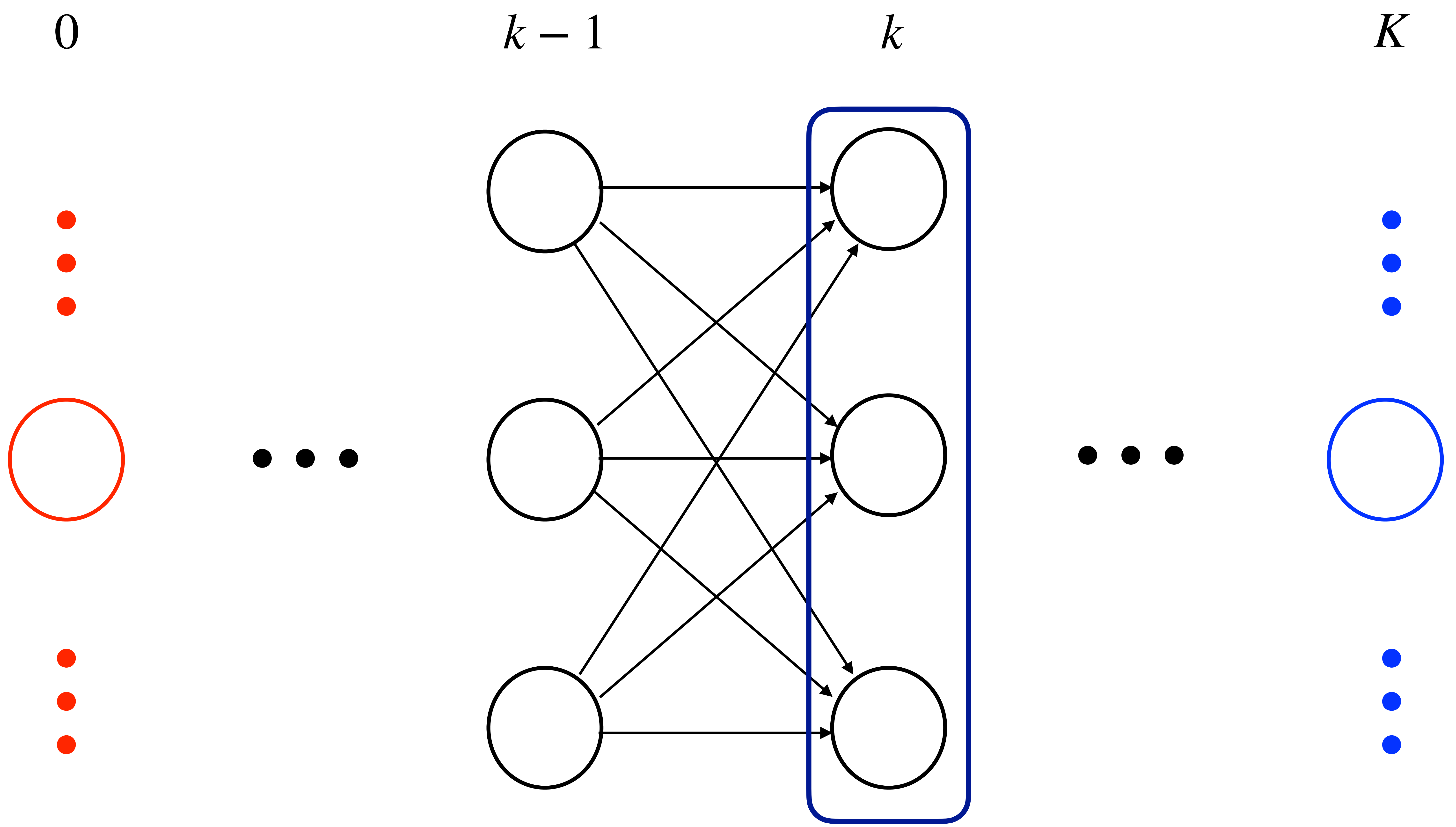
- In the i^{th} control step, the control input u is updated as $v_i = \kappa(x(i\delta_c))$. Then the system evolves according to the ODE $\dot{x} = f(x, v_i)$.

- Regardless of the possible disturbances from the environment, NNCS are **deterministic systems**.



Computing Neural Network Output Ranges

Input-Output Mapping of a Neural Network



Input: $\overrightarrow{a}_0$

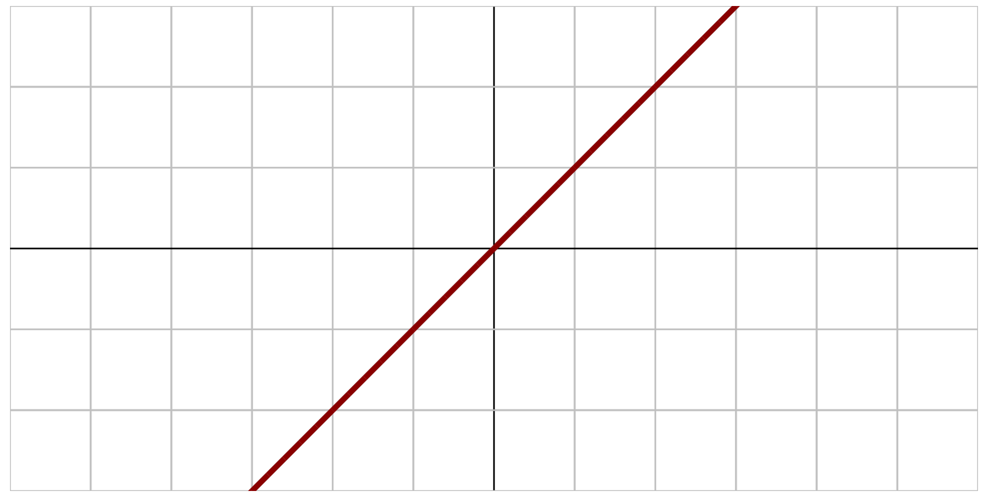
Input: $\overrightarrow{a}_K = \kappa(\overrightarrow{a}_0)$

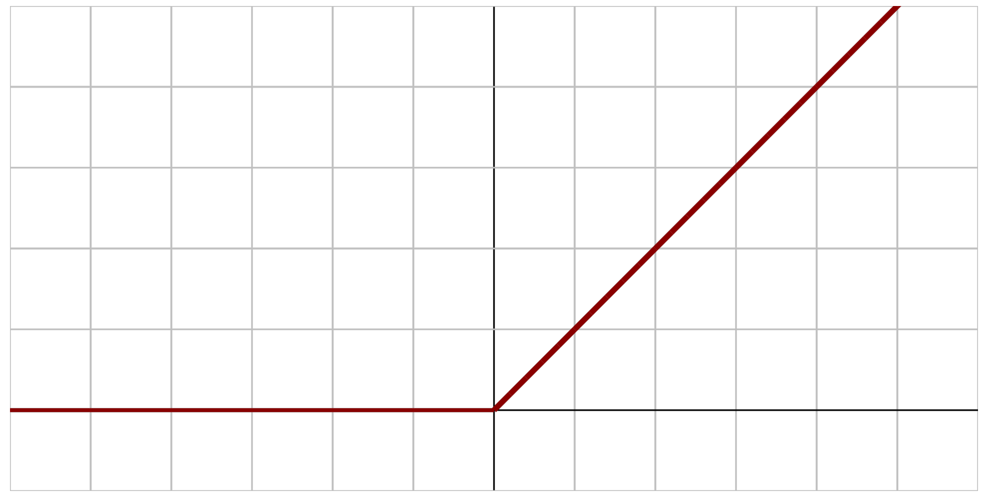
Output of the k-th layer: $\overrightarrow{a}_k = \sigma(W_k \overrightarrow{a}_{k-1} + \overrightarrow{b}_k)$

Activation Functions

Identity

$$\sigma(x) = x$$



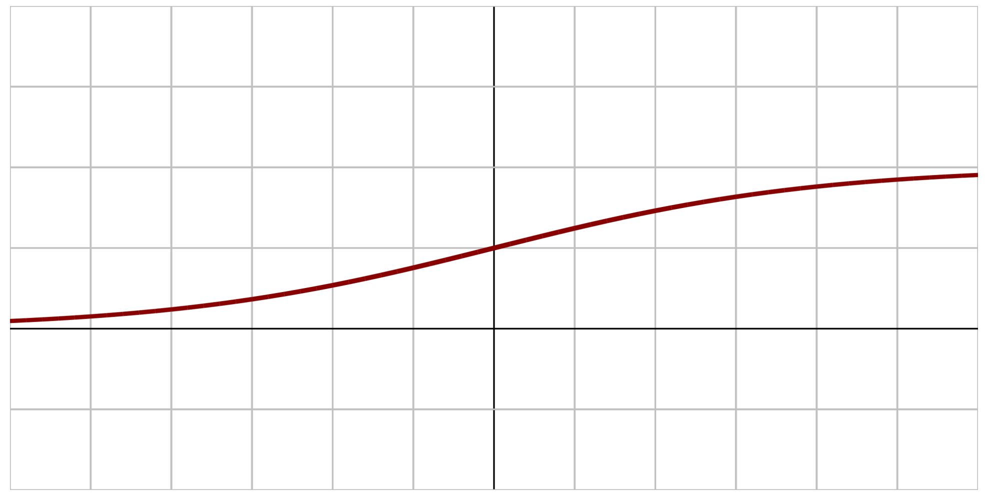


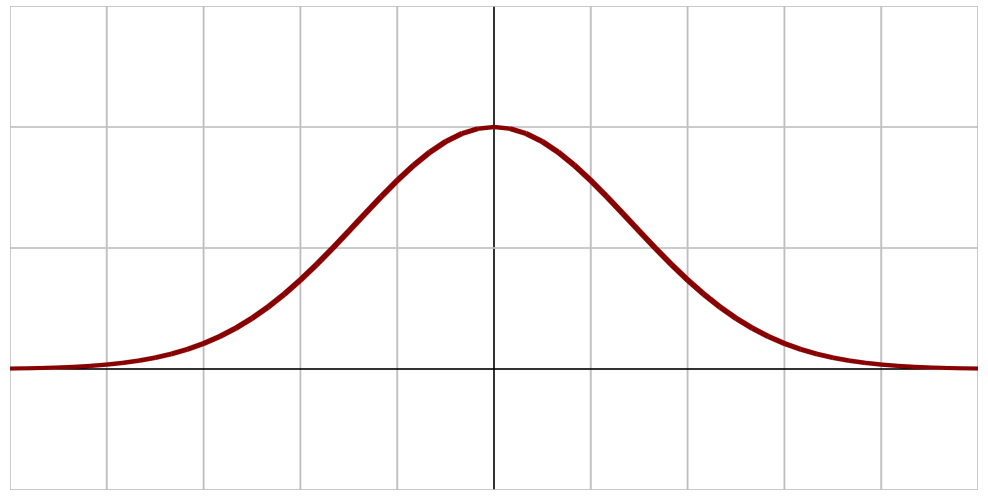
ReLU

$$\sigma(x) = \max\{0,x\}$$

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



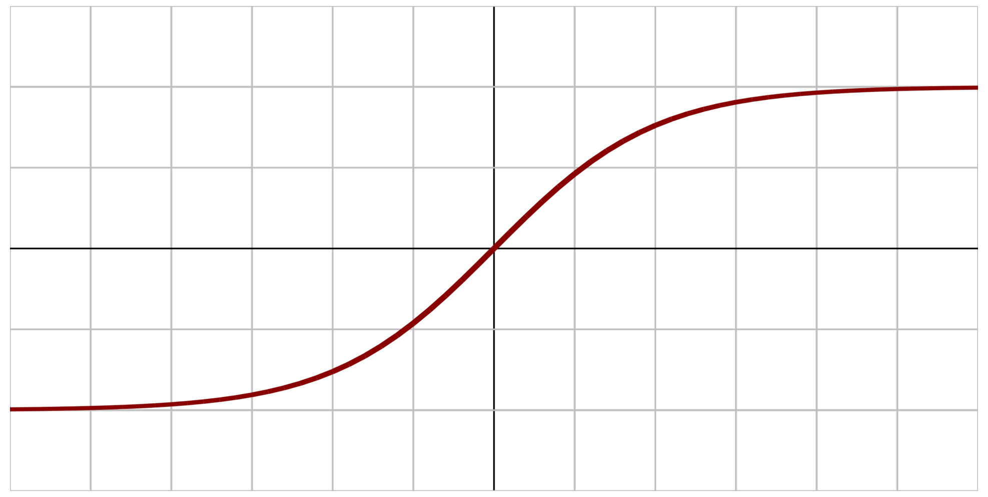


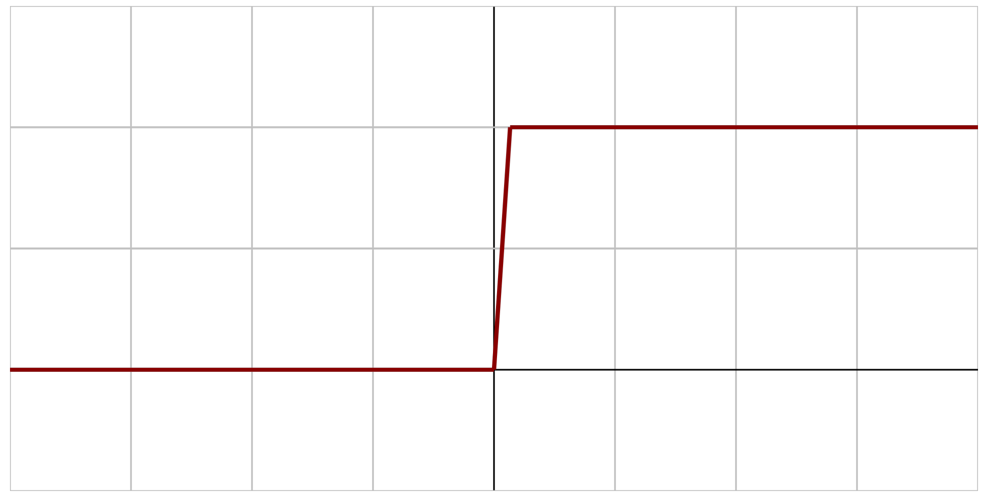
Gaussian

$$\sigma(x) = e^{-x^2}$$

Tanh

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

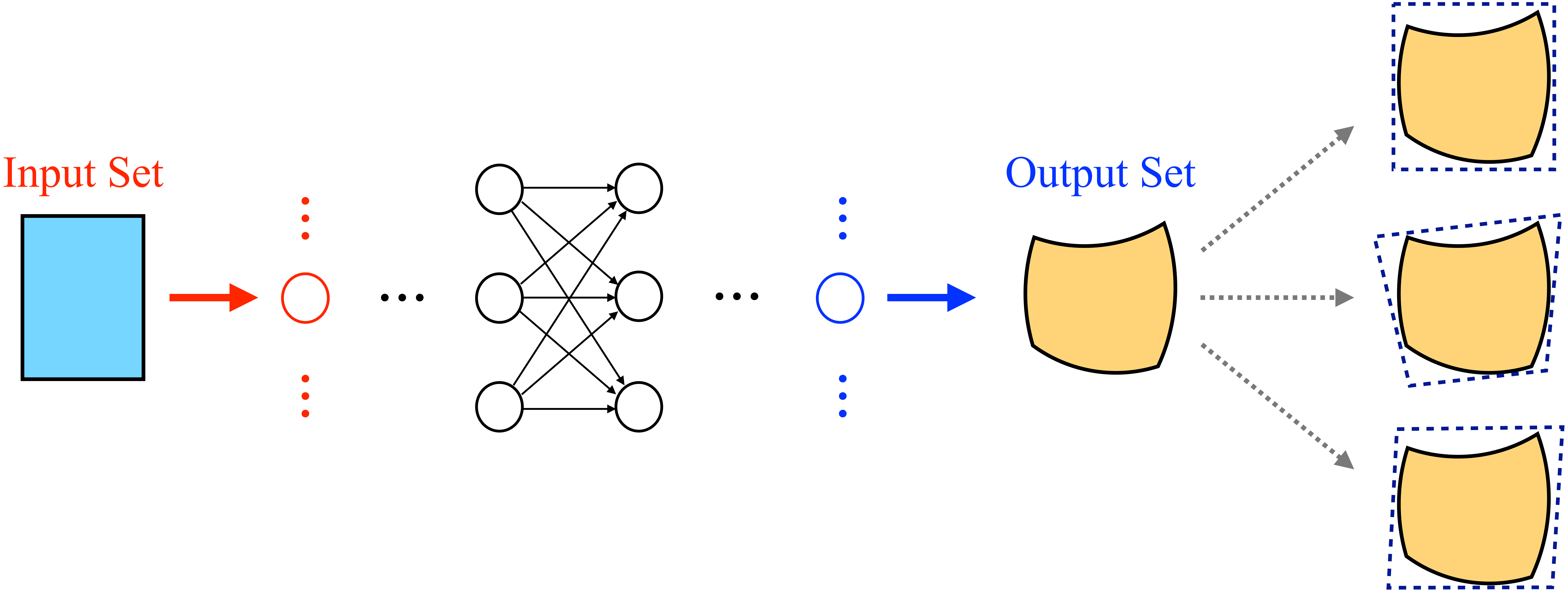




Threshold

$$\sigma(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

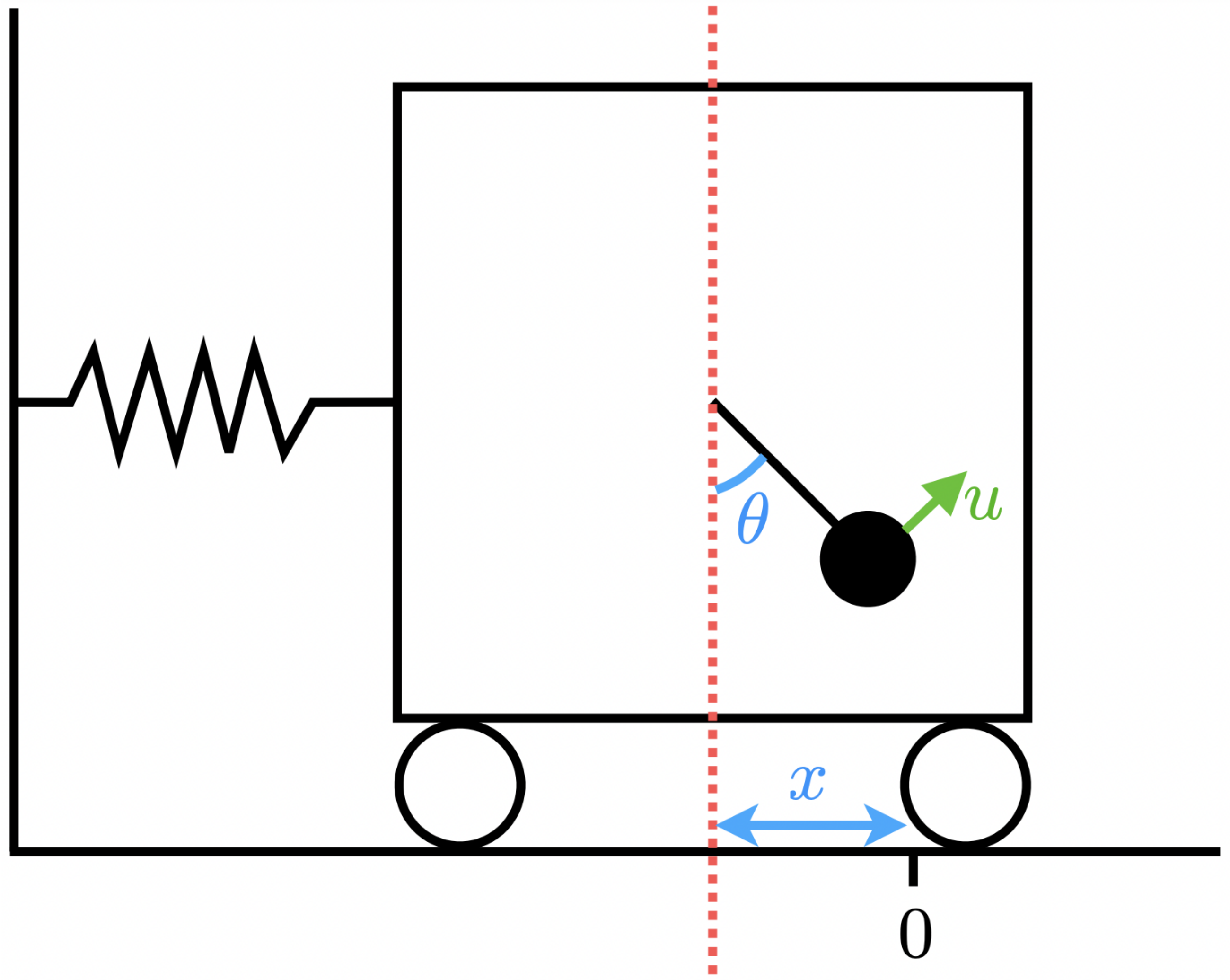
Existing Techniques for Computing NN Outputs



- **Constant bounds.** [Huang et al. 2017], [Katz et al. 2017], [Dutta et al. 2018].
- **Polynomial bounds.** [Zhang et al. 2018], [Wang et al. 2018], [Weng et al. 2018].
- **Geometric objects.** [Gehr et al., 2018], [Singh et al., 2019], [Tran et al. 2020].

To compute NNCS reachable sets, is it sufficient to just add a method of computing neural network outputs?

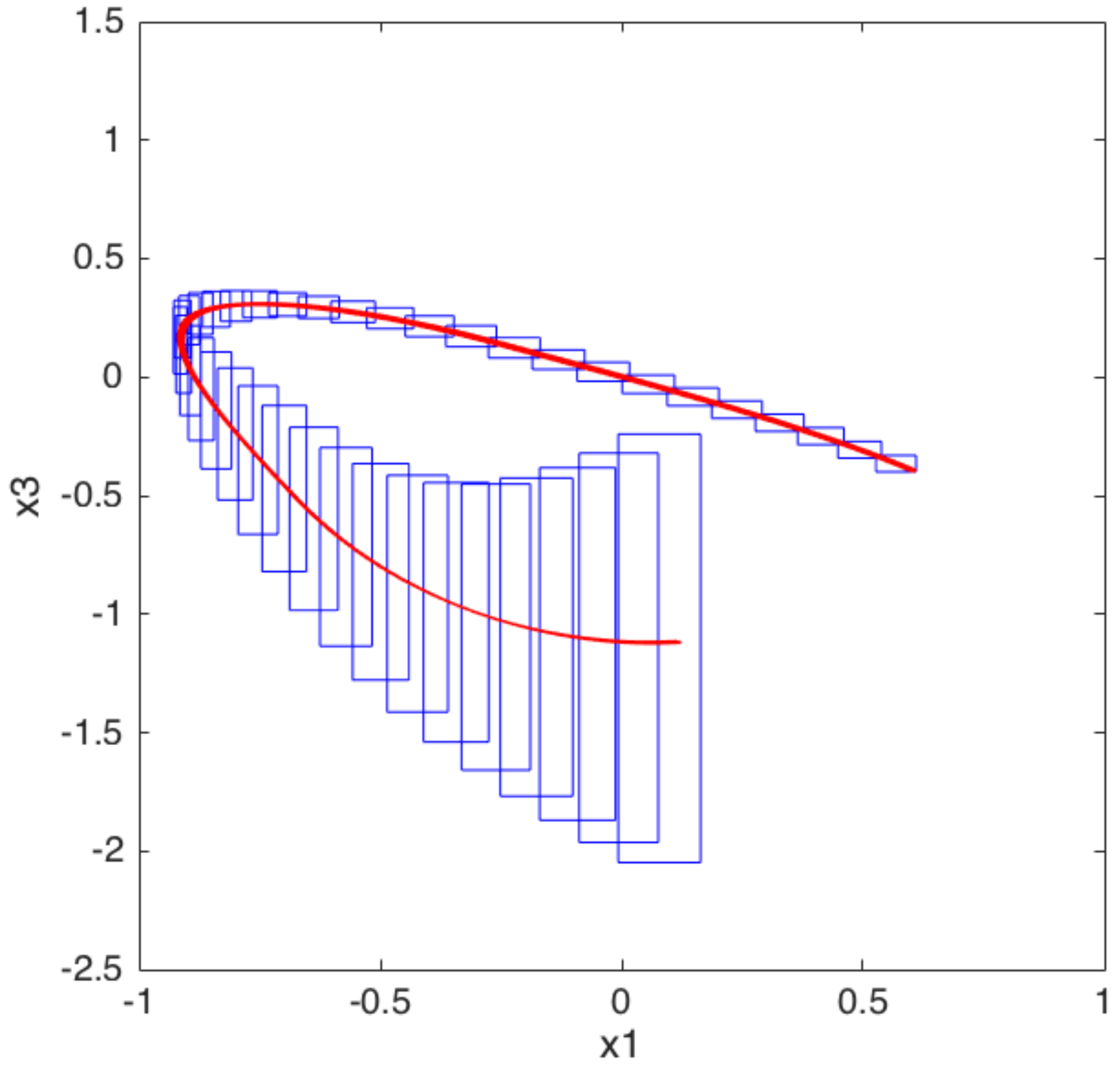
Attempt: Sherlock + Flow*



$$\begin{cases} \dot{x} & = & v \\ \dot{v} & = & -x + 0.1\sin(\theta) \\ \dot{\theta} & = & \omega \\ \dot{\omega} & = & u \end{cases}$$

$x(0) \in [0.6, 0.61], v(0) \in [-0.7, -0.69],$

$\theta(0) \in [-0.4, -0.39], \omega(0) \in [0.59, 0.6]$



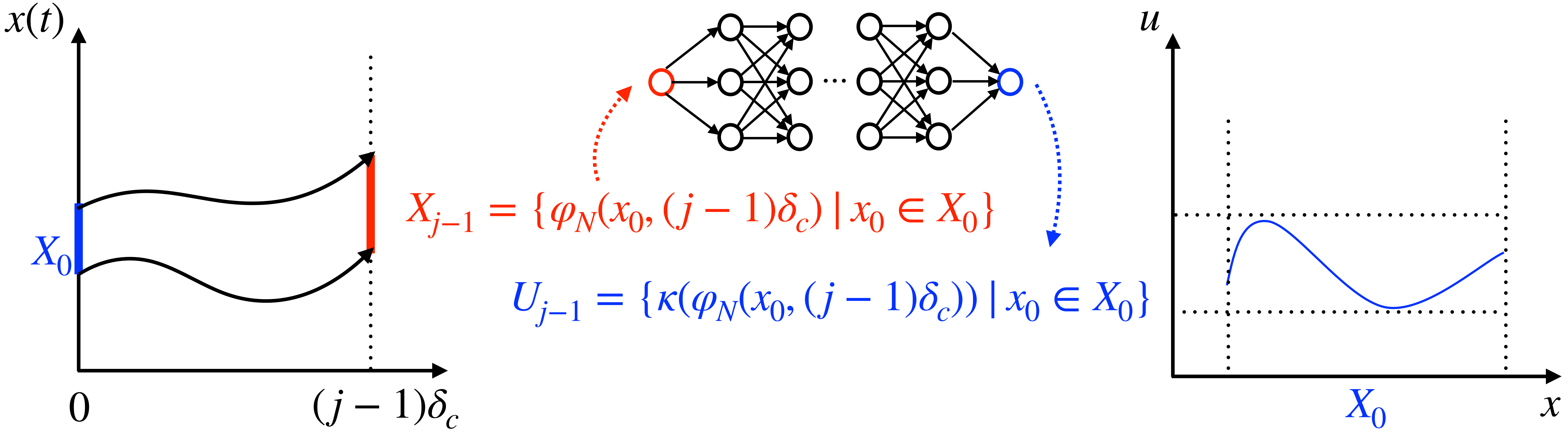
Neural network output (control input) ranges are computed by **Sherlock [Dutta et al. 2018]**. Reachable sets are computed by **Flow***.

Facts

- The ODEs are well handled by Flow*.
- Sherlock computes the **exact interval range** of the control input in each control step.
- In every control step, Sherlock computes the output range of the controller regarding to the current state set, and Flow* computes the reachable set under the updated ODE model.
- Overestimation still accumulates heavily and the reachable set computation for NNCS fails.

Where is the overestimation from?

Investigating NNCS Flowmap

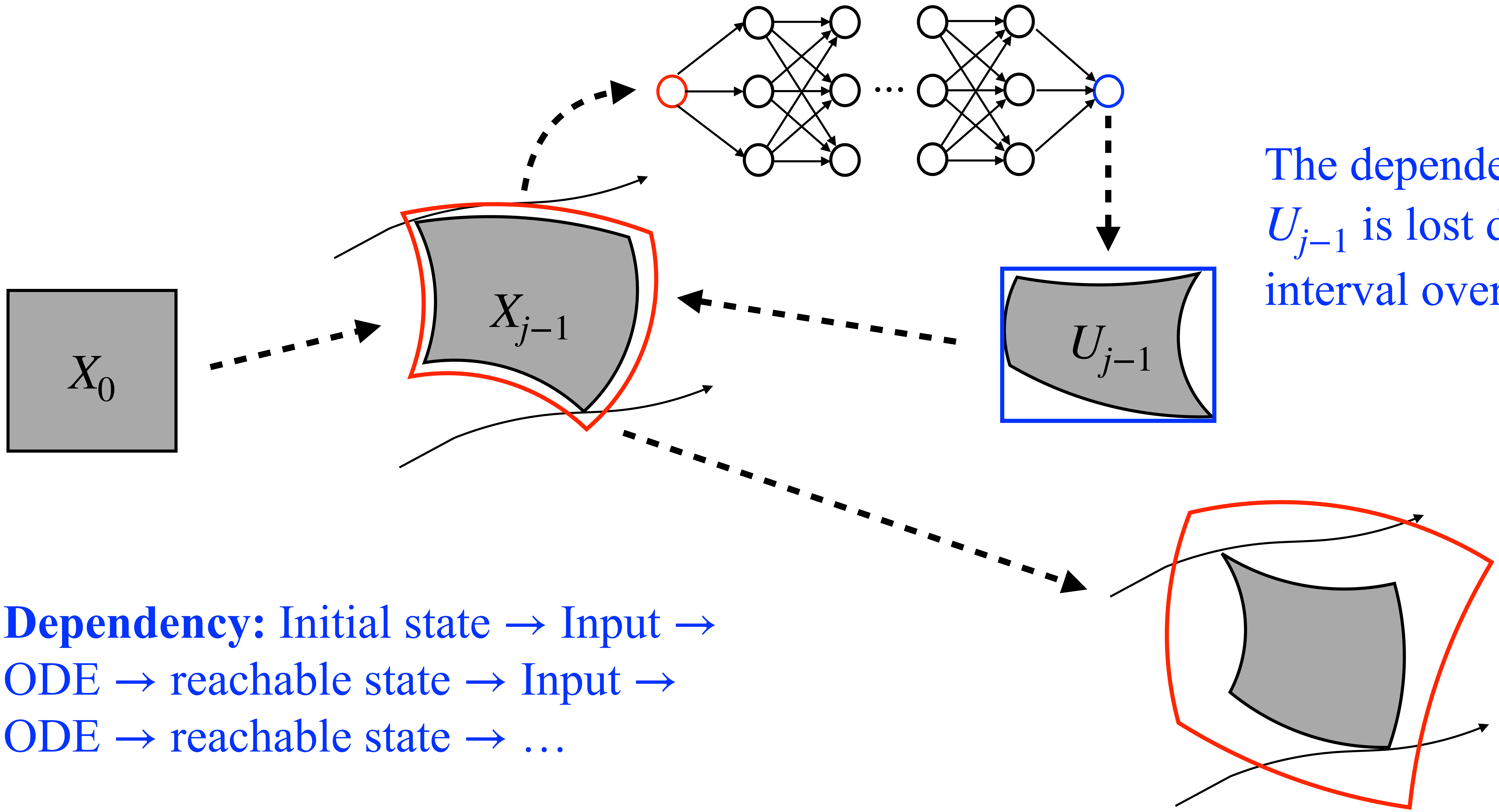


$X_{j-1} = \{\varphi_N(x_0, (j-1)\delta_c) \mid x_0 \in X_0\}$

$U_{j-1} = \{\kappa(\varphi_N(x_0, (j-1)\delta_c)) \mid x_0 \in X_0\}$

- We use φ_N to denote the flowmap function of an NNCS.
- Then the reachable set at $t = (j-1)\delta_c$ is $x_{j-1} = \varphi_N(x_0, (j-1)\delta_c, 0)$.
- The control input applied in the j^{th} step $t \in [(j-1)\delta_c, j\delta_c]$ is $v_{j-1} = \kappa(\varphi_N(x_0, (j-1)\delta_c, 0))$.
- Hence, v_j is uniquely determined by the initial state via the composite mapping $\kappa(\varphi_N(\,\cdot\,, (j-1)\delta_c, 0))$.

Main Issue in the Previous Solution



The dependency from X_0 to U_{j-1} is lost due to the interval overapproximation.

Dependency: Initial state → Input → ODE → reachable state → Input → ODE → reachable state → …

How can we track the dependency in a flowmap function?

Our Solutions

Polynomial Regression (ReLU):

[HSCC’19] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan.
Reachability Analysis for Neural Feedback Systems Using Regressive Polynomial Rule Inference.
In Hybrid Systems: Computation and Control (HSCC), pp. 157-168, 2019.

End-to-End Bernstein Approximation (Continuous):

[TECS’19] Chao Huang, Jiameng Fan, Wenchao Li, Xin Chen, and Qi Zhu.
ReachNN: Reachability Analysis of Neural-Network Controlled Systems.
In ACM Transactions on Embedded Computing Systems (TECS), volume 18, number 5s, pp. 106:1-106:22. ACM, 2019.

Layer-by-Layer Propagation using Polynomial Arithmetic (Continuous):

[ATVA’22] Chao Huang, Jiameng Fan, Xin Chen, Wenchao Li, and Qi Zhu.
POLAR: A Polynomial Arithmetic Framework for Verifying Neural-Network Controlled Systems
Available at arxiv.org/abs/2106.13867.

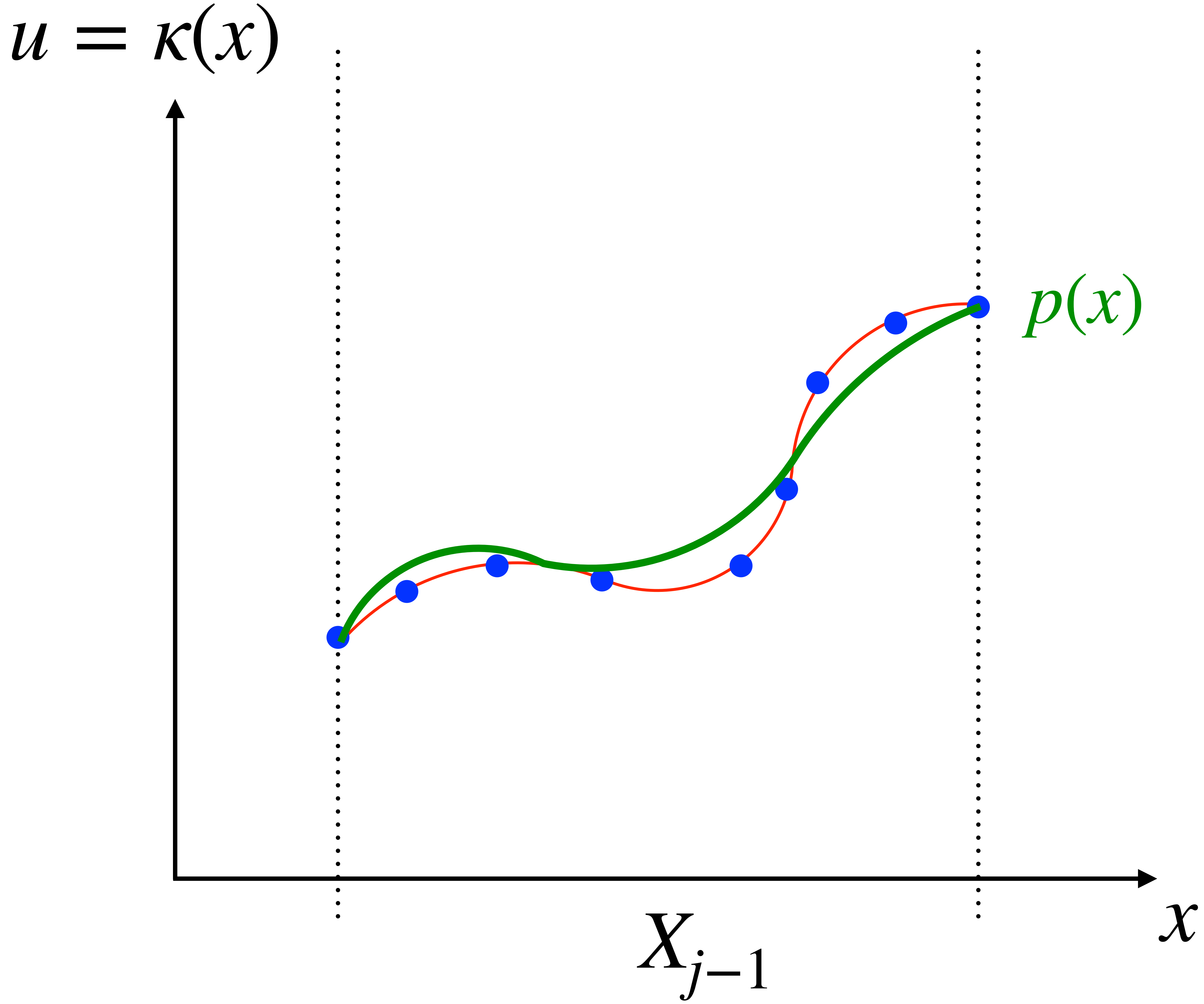
Polynomial Regression

Main idea:

- We compute a **polynomial regression** p based on a finite set of random samples in X_{j-1}.
- Then p can be viewed as a polynomial approximation of the neural network κ.
- Next, we compute an interval I which contains the difference $\varepsilon(x) = \kappa(x) - p(x)$ for all $x \in X_{j-1}$.
- Hence, $p + I$ is a functional overapproximation of κ.

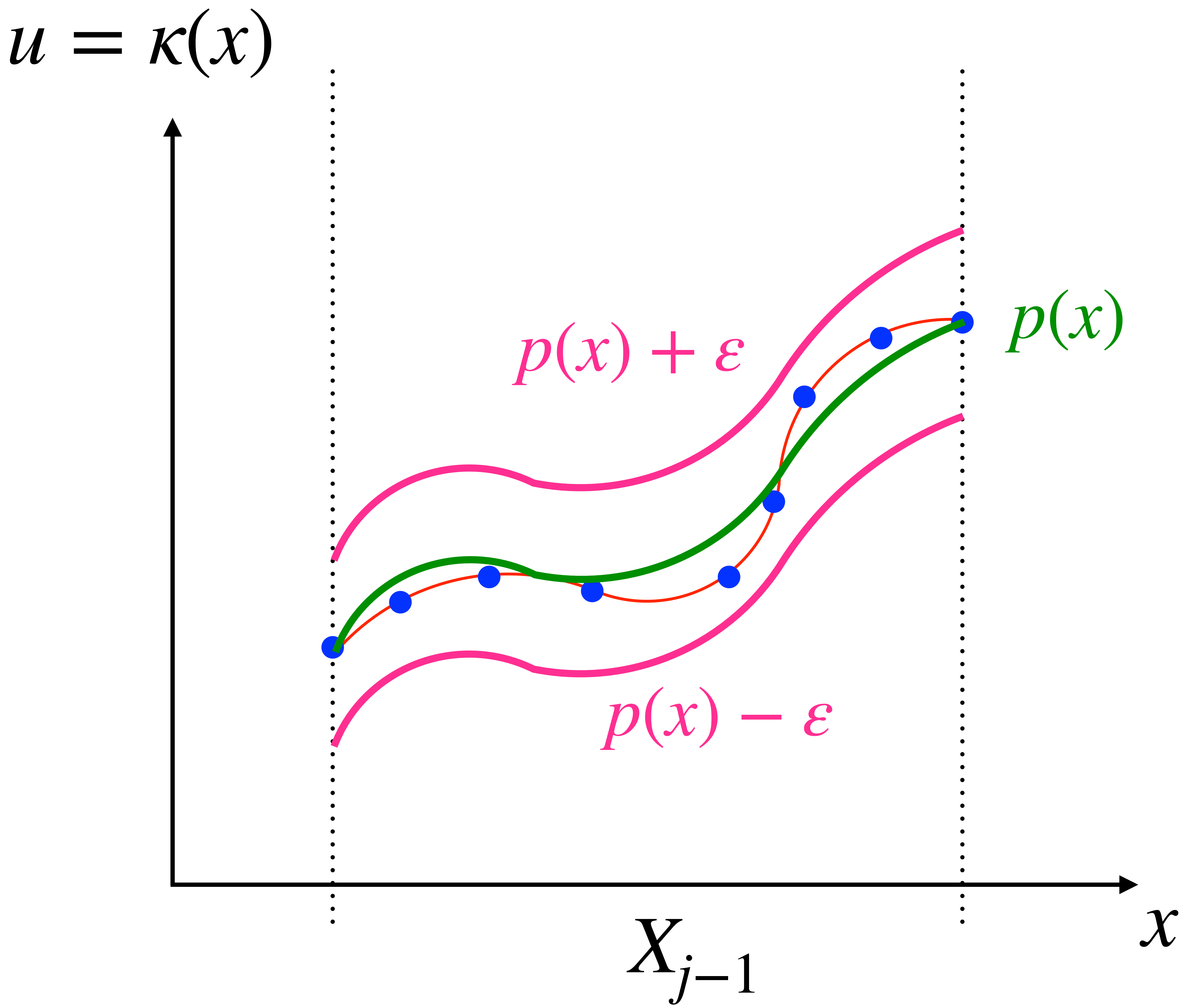
Step 1: Computing a Polynomial Regression

- Generate a finite set of **random** samples in the set X_{j-1}.
- Compute their images under the neural network mapping $\kappa(\,\cdot\,)$.
- Compute a **polynomial regression** p which is as close as possible to the samples.
- Then p will be used as an approximation of κ.



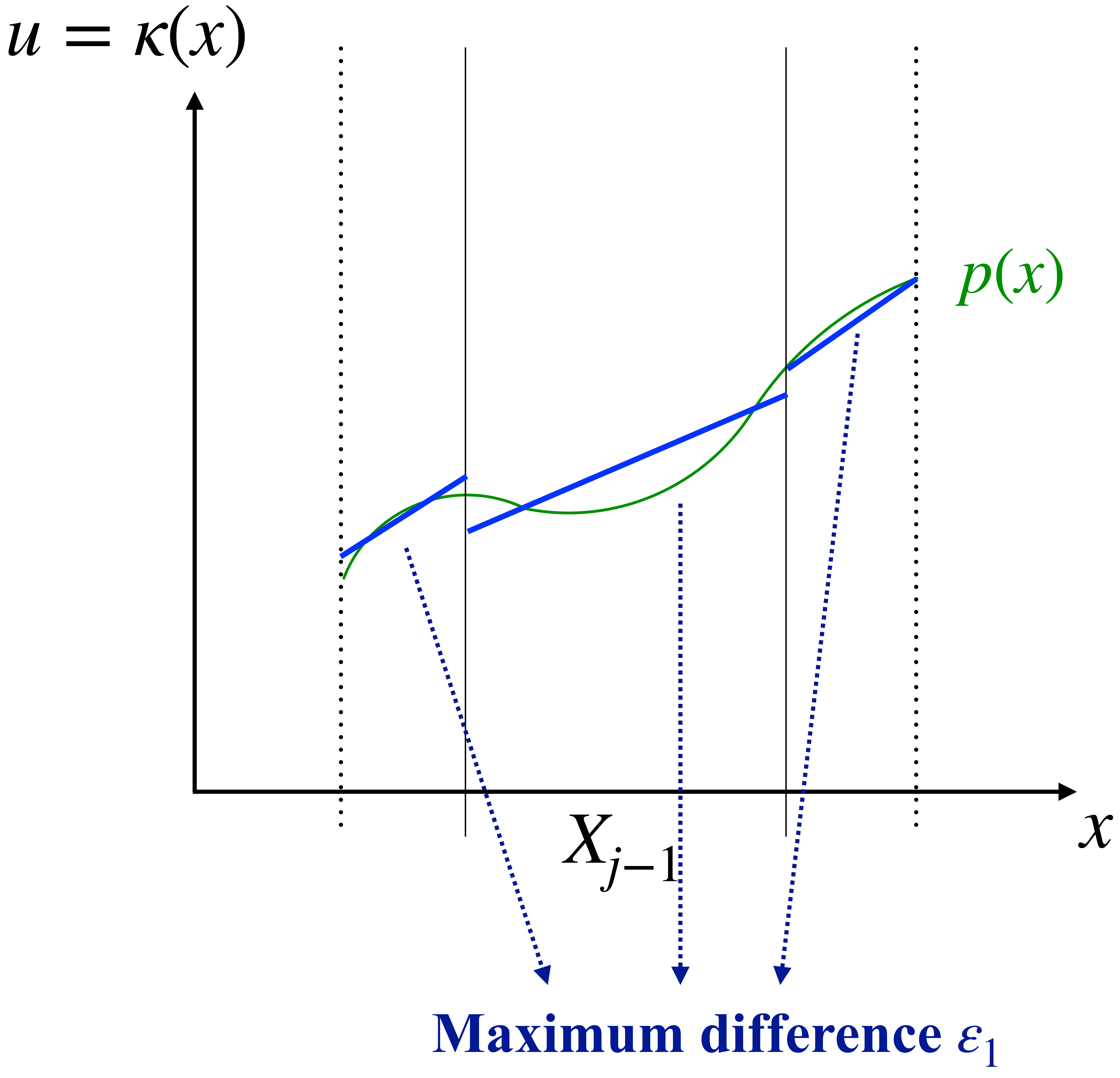
Step 2: Remainder Evaluation

- **Purpose:** We want to find a remainder interval $[-\varepsilon, \varepsilon]$ which is guaranteed to contain the approximation error of p, i.e., $\kappa(x) - p(x) \leq \varepsilon$ for all $x \in X_{j-1}$.
- **Information we know:** The range X_{j-1} of the samples, the polynomial approximation p.
- **Information we "do not know":** Although κ can be explicitly expressed, the form is often too complicated.



Step 2.1: Computing Piecewise Linear Polynomials

- We adaptively choose a subset of X_{j-1} and compute a linear polynomial over it.
- The error between a linear polynomial and the polynomial regression is controlled to be less than a threshold.
- Then, the error between the piecewise linear polynomials to the polynomial regression is defined as the maximum error ε_1.

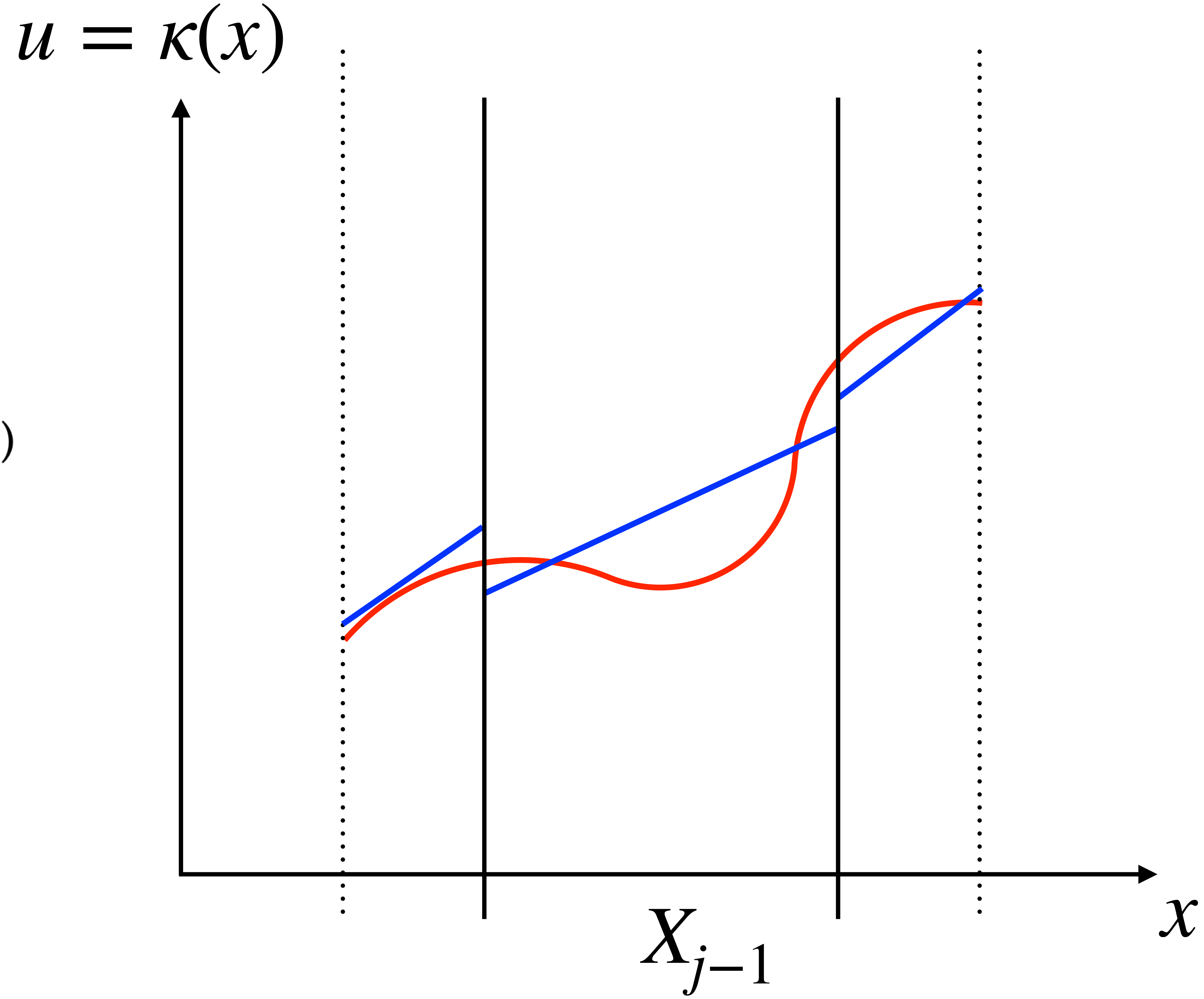


Step 2.2: Computing the Error between the PWL model and κ

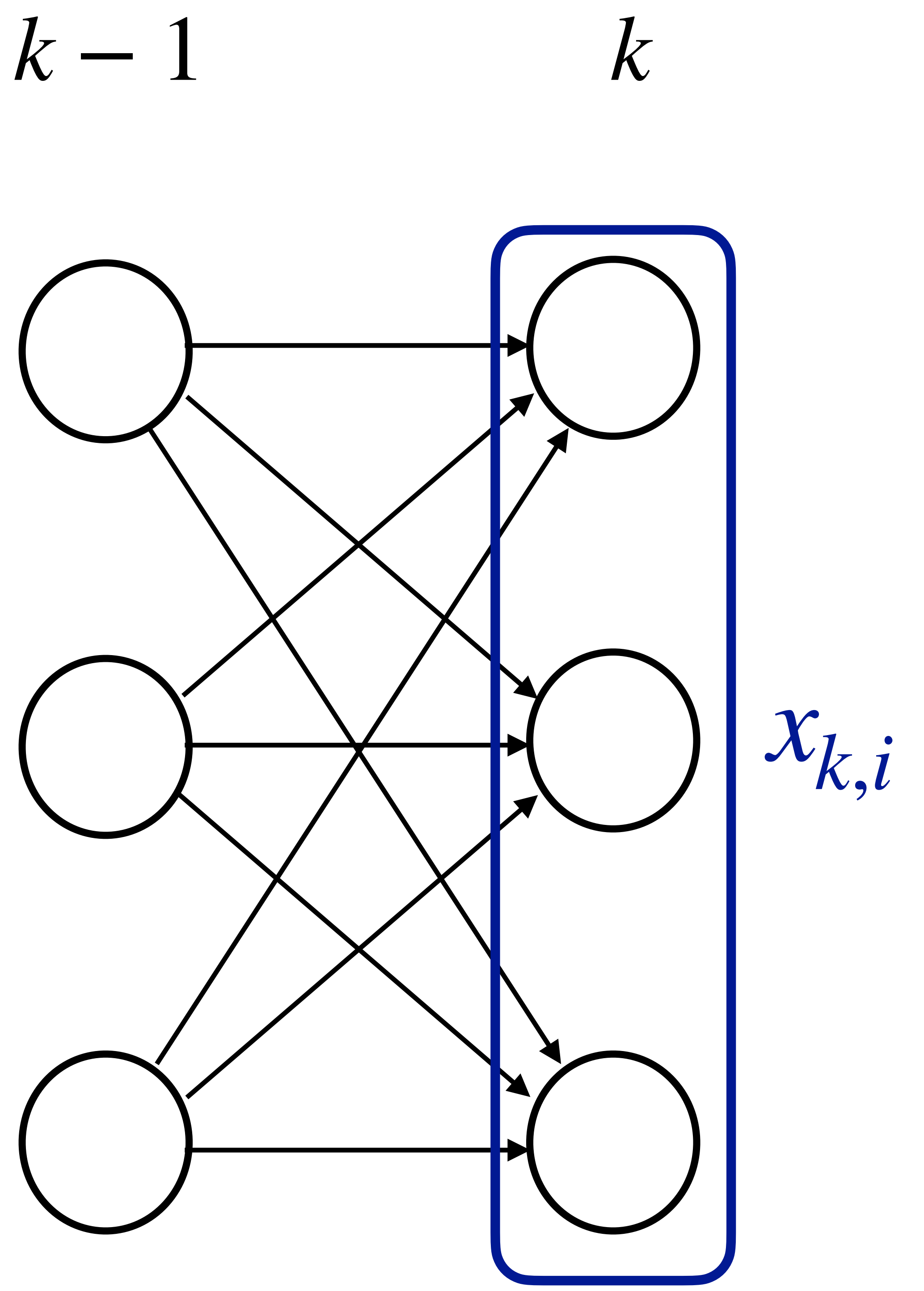
Combined MILP Model: Given the constraints $\Psi_N(\mathbf{x}, y, \mathbf{v})$ for a neural network N and constraints $\Psi_L(\mathbf{x}, z, \vec{l})$ for a PWL model L, the error interval is estimated by setting up a two MILPs as follows:

$$
\begin{array}{rll}
\max(\min) & z - y & \\
\text{s.t.} & \Psi_N(\mathbf{x}, y, \mathbf{v}) & \text{(*MILP encoding for NN*)} \\
& \Psi_L(\mathbf{x}, z, \vec{l}) & \text{(*MILP encoding for PWL*)} \\
& \mathbf{x} \in D,\ (\mathbf{v}, \vec{l}) \in \{0,1\}^{|\mathbf{v}|+|\vec{l}|} &
\end{array}
$$

- x is the vector of variables to denote the NN input.
- y is the vector of variables to denote the NN output.
- z is the vector of variables to denote the output of the PWL model.
- We want to find the maximum and minimum differences between z and y.



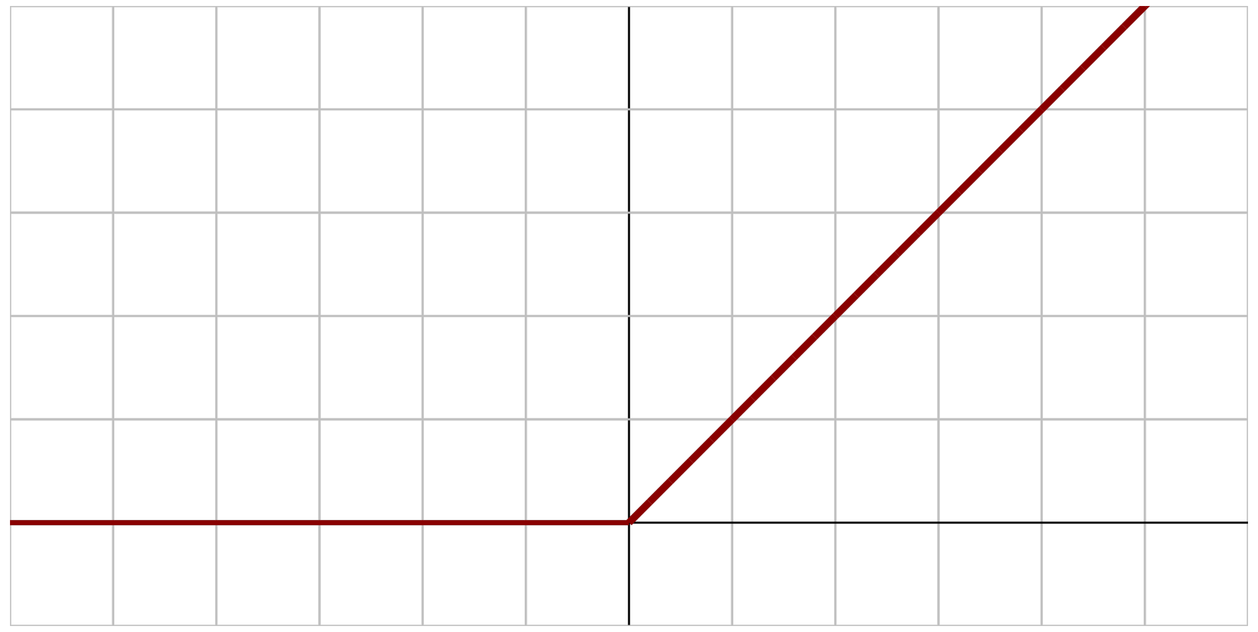
Background: MILP Encoding of ReLU NN



Output of the k-th layer: $x_{k,i} = \sigma(W_{i,k}\ x_{k-1,i} + b_{k,i})$

$x_{k,i} = \sigma(W_{i,k}\ x_{k-1,i} + b_{k,i})$, if $W_{i,k}\ x_{k-1,i} + b_{k,i} \geq 0$,
$x_{k,i} = 0$, otherwise.

Activation function:
$y = x$ if $x \geq 0$, $y = 0$ otherwise.

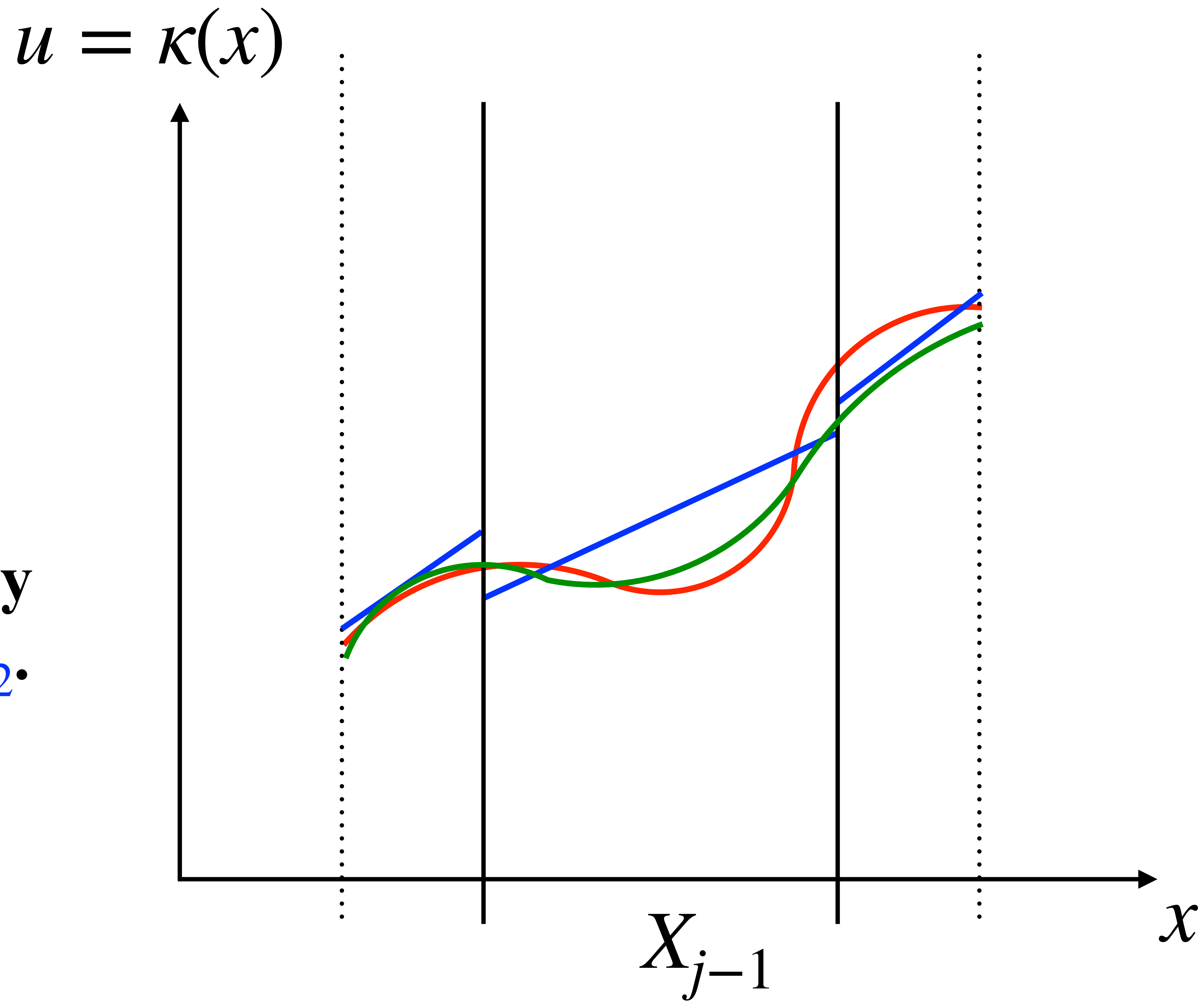


MILP Encoding:

$$y \geq x$$
$$y \leq x + Mv$$
$$y \geq 0$$
$$y \leq M(1-v)$$

$v \in \{0,1\}$, M is a very large number.

Error between κ and p

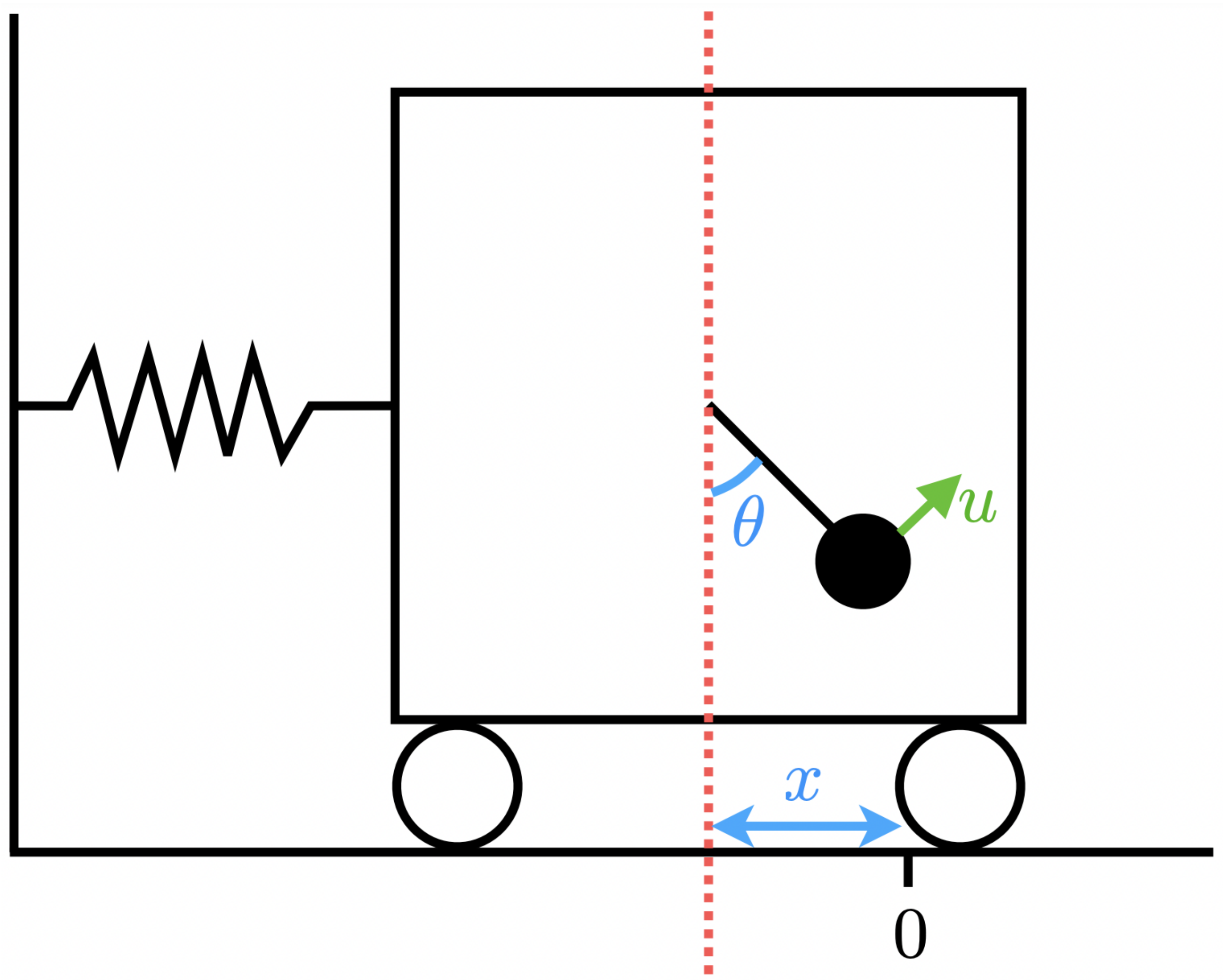


Difference between p and the PWL model: ε_1

Difference between the PWL model and κ: ε_2

Difference between p and κ can be conservatively computed by the triangle inequality: $\varepsilon = \varepsilon_1 + \varepsilon_2$.

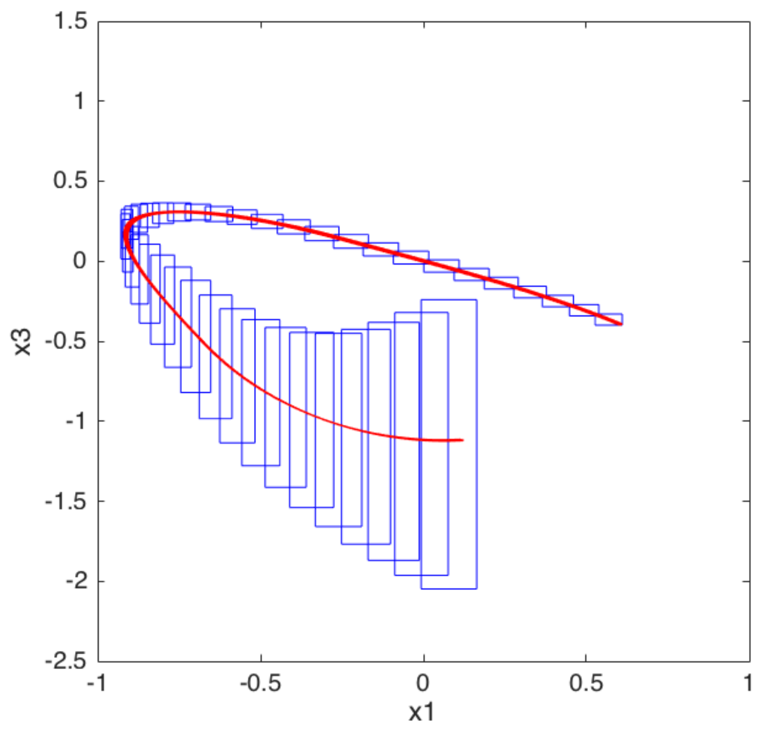
Revisit to the Case Study



$$\begin{cases} \dot{x} &= v \\ \dot{v} &= -x + 0.1\sin(\theta) \\ \dot{\theta} &= \omega \\ \dot{\omega} &= u \end{cases}$$

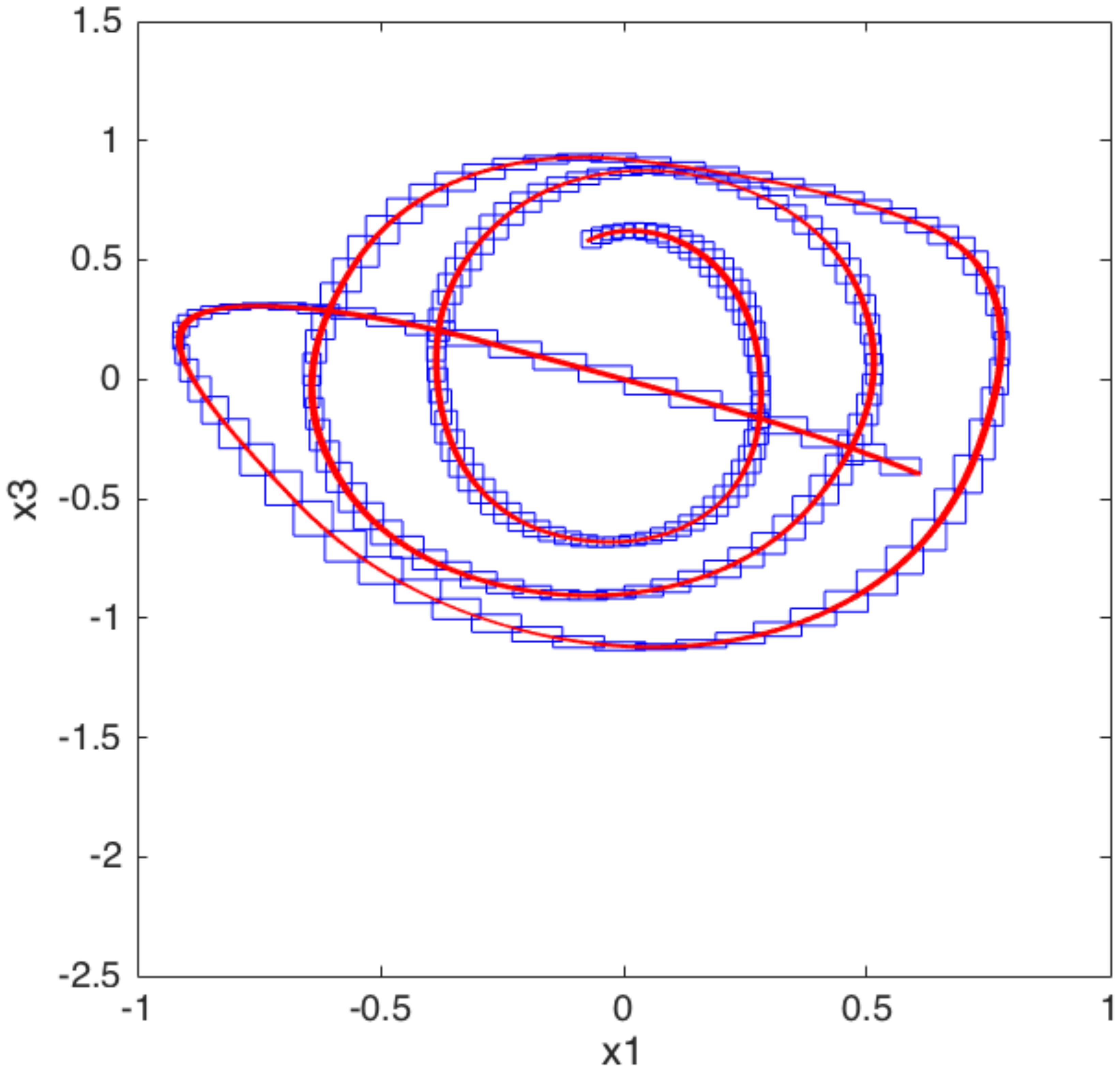
$$x(0) \in [0.6, 0.61], v(0) \in [-0.7, -0.69],$$

$$\theta(0) \in [-0.4, -0.39], \omega(0) \in [0.59, 0.6]$$

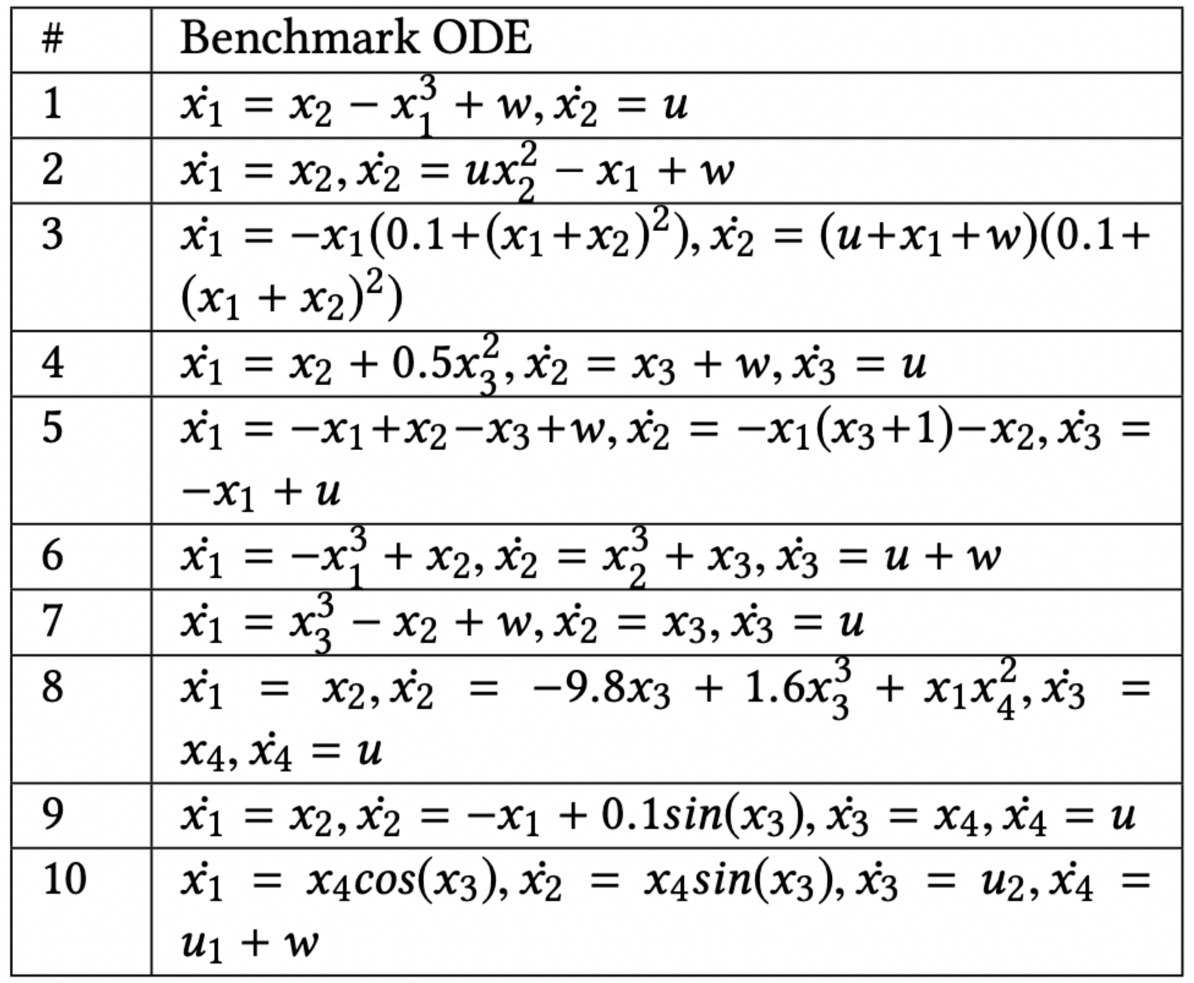


Sherlock + Flow*

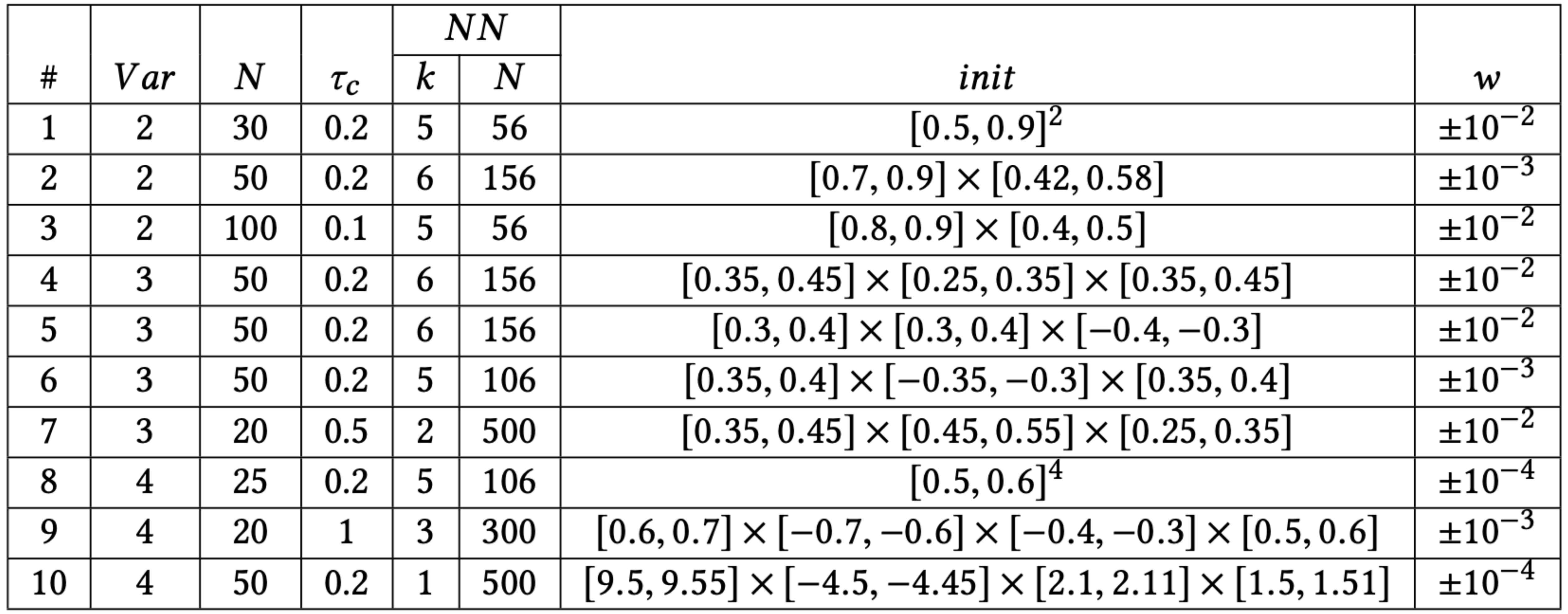
Using polynomial regression



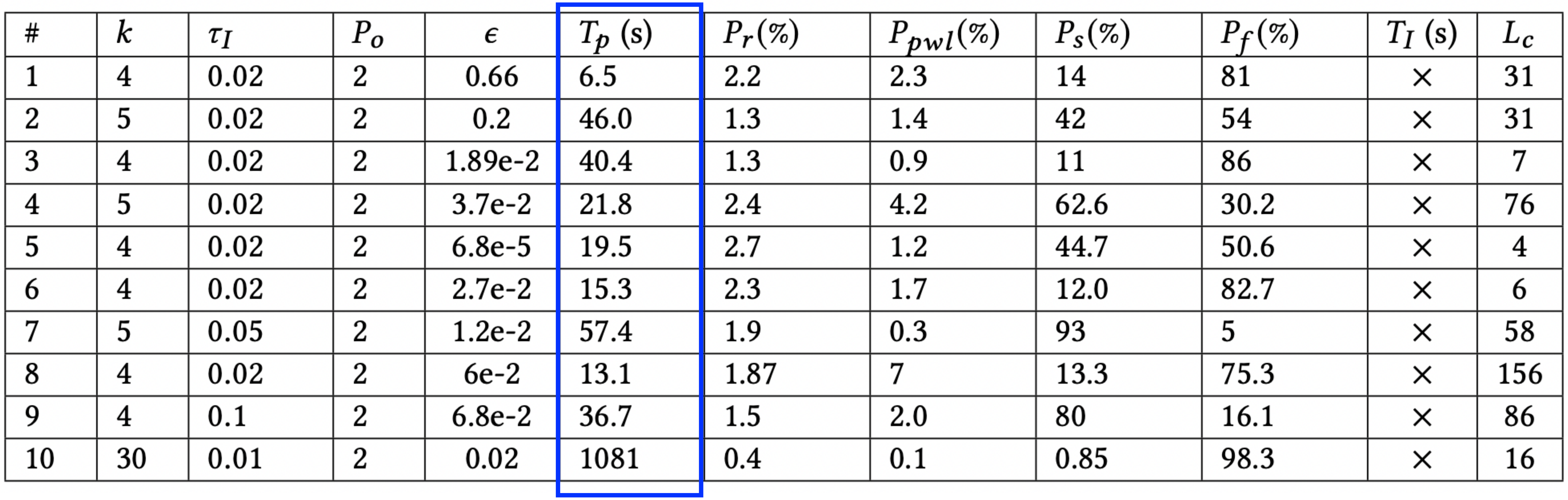
More Experiments



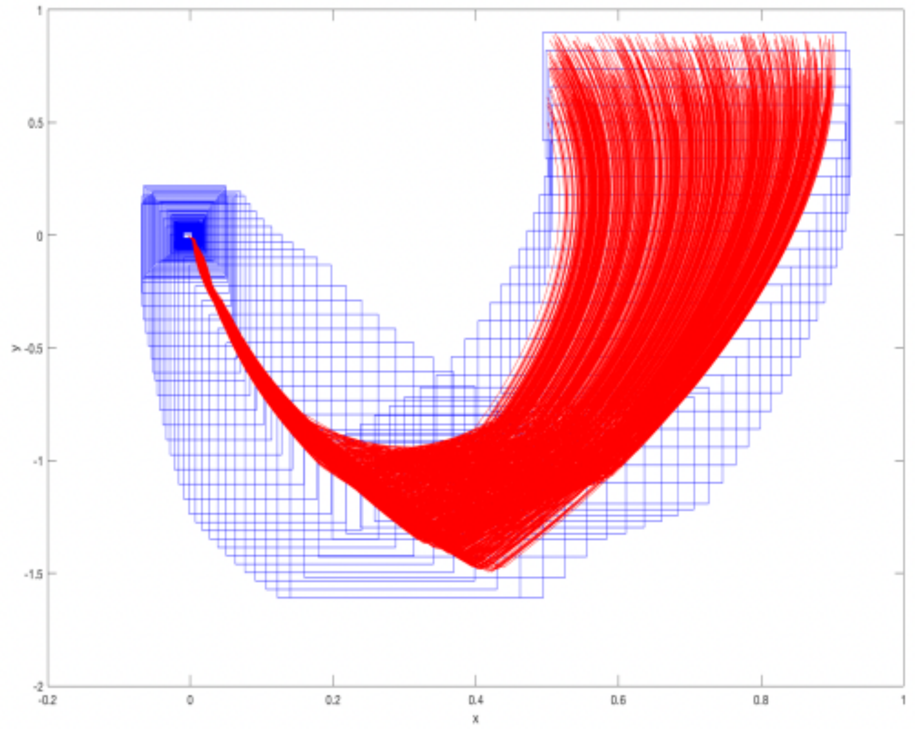
#	Benchmark ODE
1	$\dot{x_1} = x_2 - x_1^3 + w, \dot{x_2} = u$
2	$\dot{x_1} = x_2, \dot{x_2} = ux_2^2 - x_1 + w$
3	$\dot{x_1} = -x_1(0.1+(x_1+x_2)^2), \dot{x_2} = (u+x_1+w)(0.1+(x_1+x_2)^2)$
4	$\dot{x_1} = x_2 + 0.5x_3^2, \dot{x_2} = x_3 + w, \dot{x_3} = u$
5	$\dot{x_1} = -x_1+x_2-x_3+w, \dot{x_2} = -x_1(x_3+1)-x_2, \dot{x_3} = -x_1 + u$
6	$\dot{x_1} = -x_1^3 + x_2, \dot{x_2} = x_2^3 + x_3, \dot{x_3} = u + w$
7	$\dot{x_1} = x_3^3 - x_2 + w, \dot{x_2} = x_3, \dot{x_3} = u$
8	$\dot{x_1} = x_2, \dot{x_2} = -9.8x_3 + 1.6x_3^3 + x_1x_4^2, \dot{x_3} = x_4, \dot{x_4} = u$
9	$\dot{x_1} = x_2, \dot{x_2} = -x_1 + 0.1sin(x_3), \dot{x_3} = x_4, \dot{x_4} = u$
10	$\dot{x_1} = x_4cos(x_3), \dot{x_2} = x_4sin(x_3), \dot{x_3} = u_2, \dot{x_4} = u_1 + w$

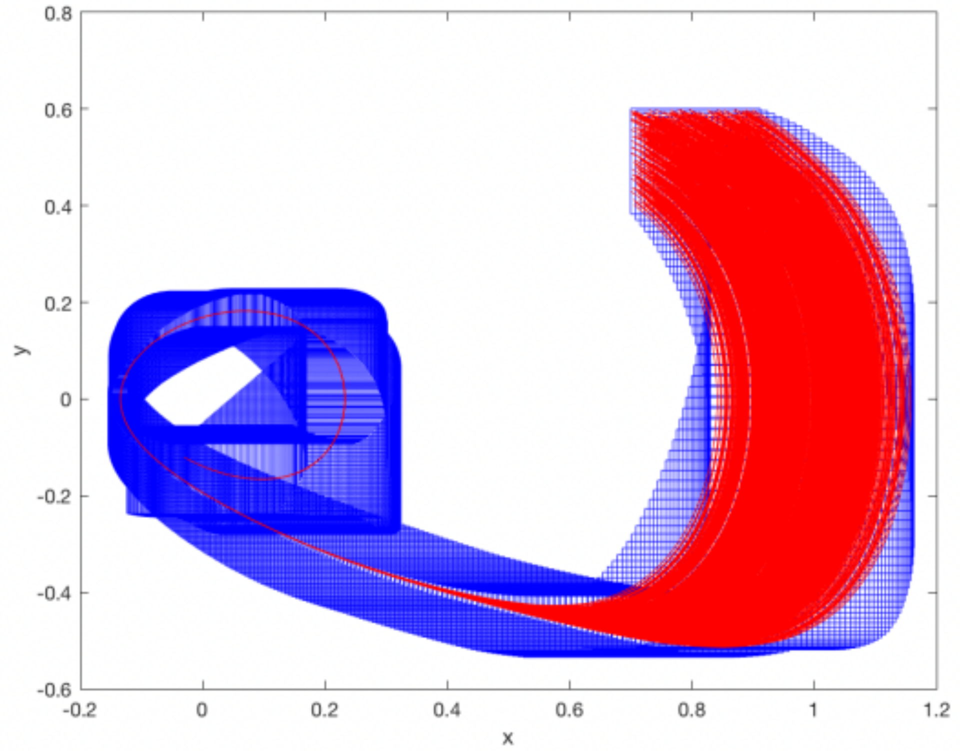


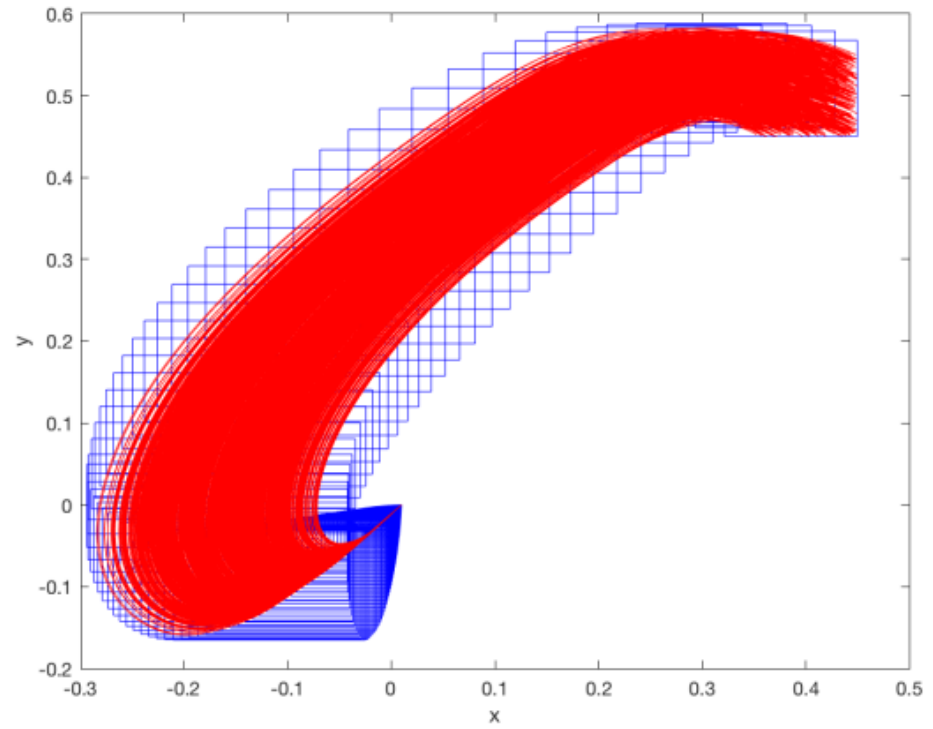
				NN			
#	Var	N	τ_c	k	N	$init$	w
1	2	30	0.2	5	56	$[0.5, 0.9]^2$	$\pm 10^{-2}$
2	2	50	0.2	6	156	$[0.7, 0.9] \times [0.42, 0.58]$	$\pm 10^{-3}$
3	2	100	0.1	5	56	$[0.8, 0.9] \times [0.4, 0.5]$	$\pm 10^{-2}$
4	3	50	0.2	6	156	$[0.35, 0.45] \times [0.25, 0.35] \times [0.35, 0.45]$	$\pm 10^{-2}$
5	3	50	0.2	6	156	$[0.3, 0.4] \times [0.3, 0.4] \times [-0.4, -0.3]$	$\pm 10^{-2}$
6	3	50	0.2	5	106	$[0.35, 0.4] \times [-0.35, -0.3] \times [0.35, 0.4]$	$\pm 10^{-3}$
7	3	20	0.5	2	500	$[0.35, 0.45] \times [0.45, 0.55] \times [0.25, 0.35]$	$\pm 10^{-2}$
8	4	25	0.2	5	106	$[0.5, 0.6]^4$	$\pm 10^{-4}$
9	4	20	1	3	300	$[0.6, 0.7] \times [-0.7, -0.6] \times [-0.4, -0.3] \times [0.5, 0.6]$	$\pm 10^{-3}$
10	4	50	0.2	1	500	$[9.5, 9.55] \times [-4.5, -4.45] \times [2.1, 2.11] \times [1.5, 1.51]$	$\pm 10^{-4}$

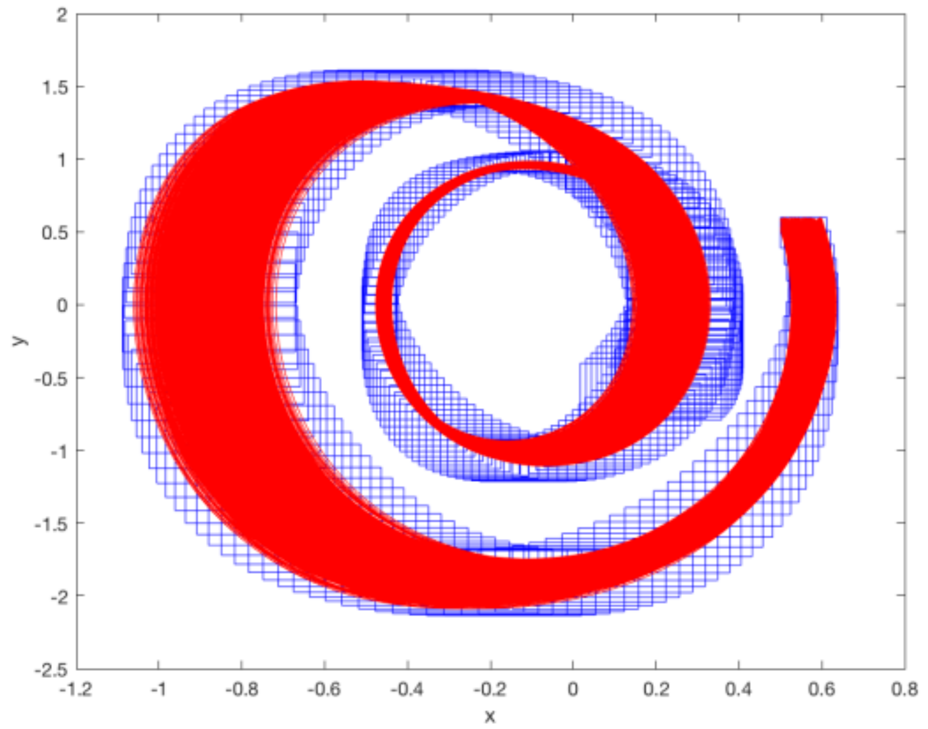


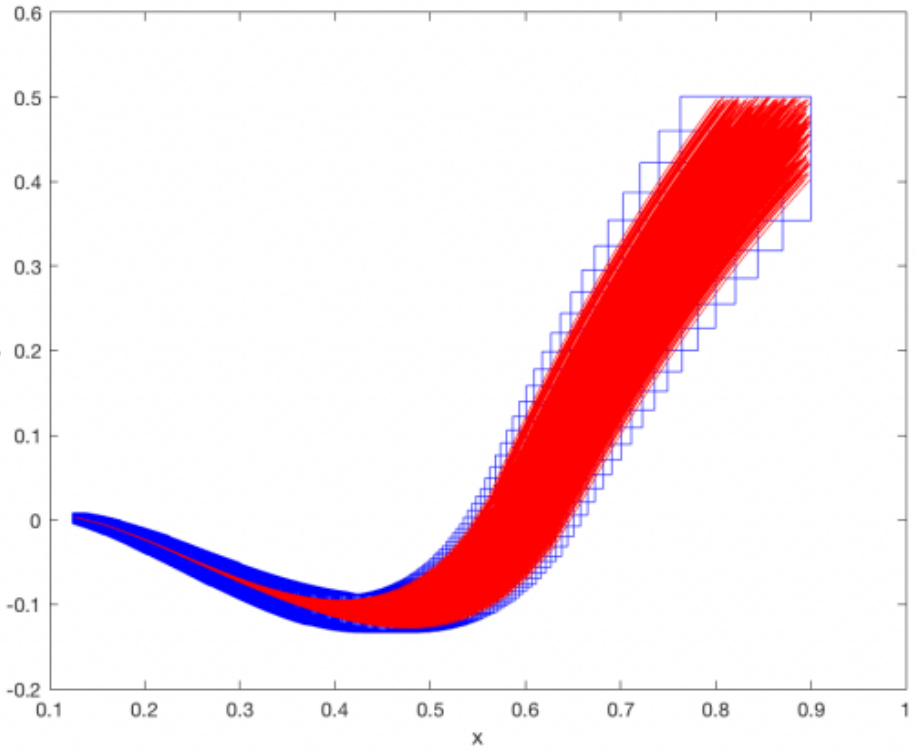
#	k	τ_I	P_o	ϵ	T_p (s)	$P_r(\%)$	$P_{pwl}(\%)$	$P_s(\%)$	$P_f(\%)$	T_I (s)	L_c
1	4	0.02	2	0.66	6.5	2.2	2.3	14	81	×	31
2	5	0.02	2	0.2	46.0	1.3	1.4	42	54	×	31
3	4	0.02	2	1.89e-2	40.4	1.3	0.9	11	86	×	7
4	5	0.02	2	3.7e-2	21.8	2.4	4.2	62.6	30.2	×	76
5	4	0.02	2	6.8e-5	19.5	2.7	1.2	44.7	50.6	×	4
6	4	0.02	2	2.7e-2	15.3	2.3	1.7	12.0	82.7	×	6
7	5	0.05	2	1.2e-2	57.4	1.9	0.3	93	5	×	58
8	4	0.02	2	6e-2	13.1	1.87	7	13.3	75.3	×	156
9	4	0.1	2	6.8e-2	36.7	1.5	2.0	80	16.1	×	86
10	30	0.01	2	0.02	1081	0.4	0.1	0.85	98.3	×	16

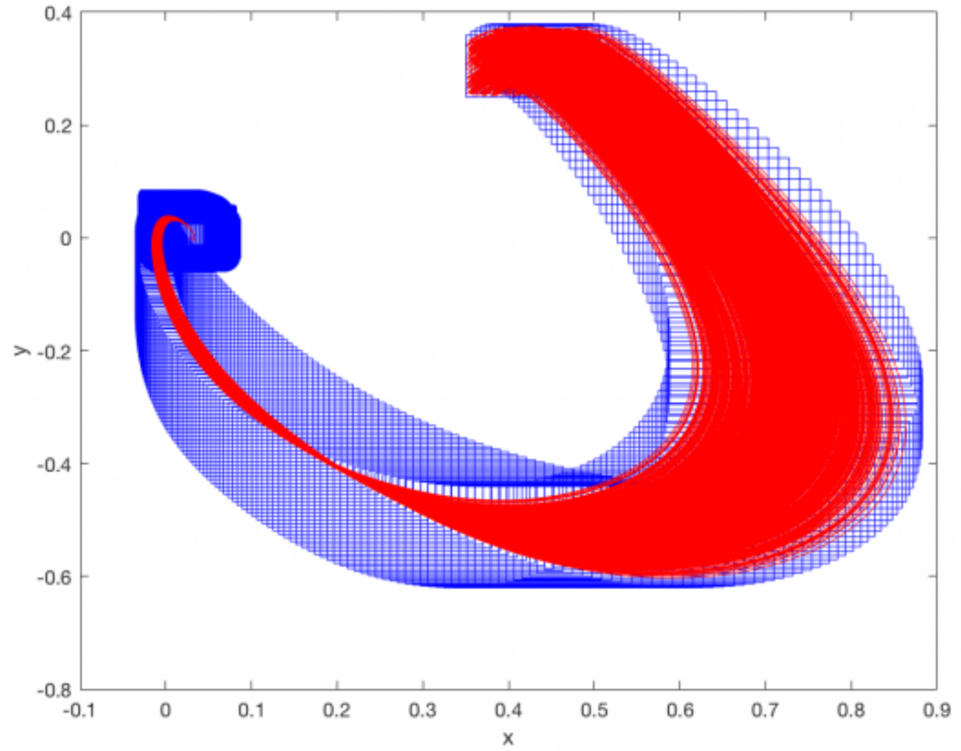

1
0.5
0
-0.5
-1
-1.5
-2
y
-0.2
0
0.2
0.4
0.6
0.8
1
x

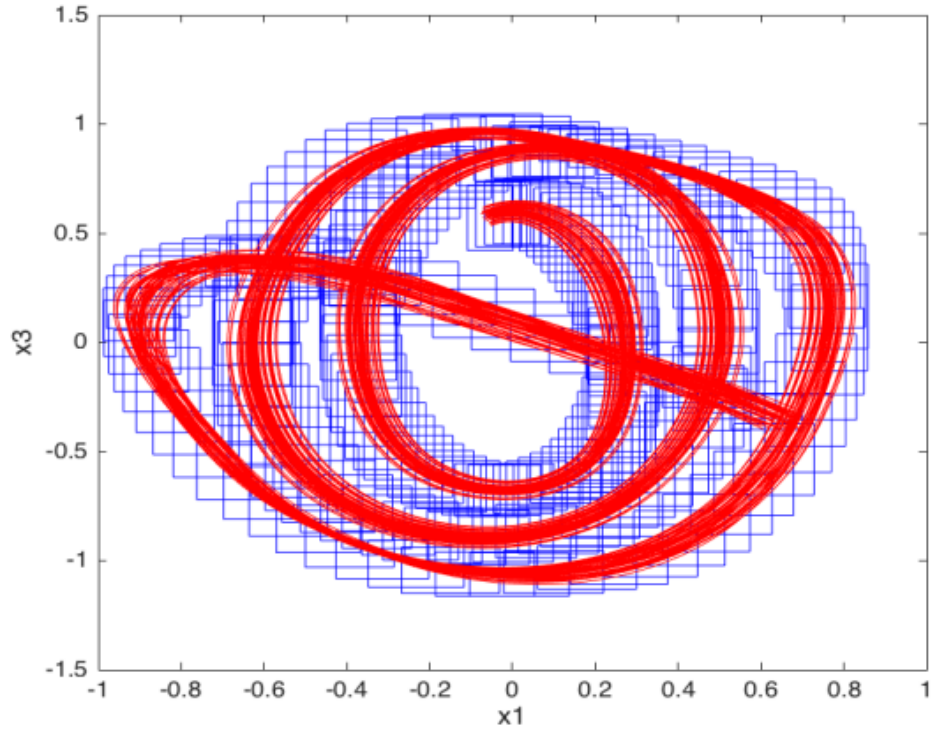

0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
y
-0.2
0
0.2
0.4
0.6
0.8
1
1.2
x

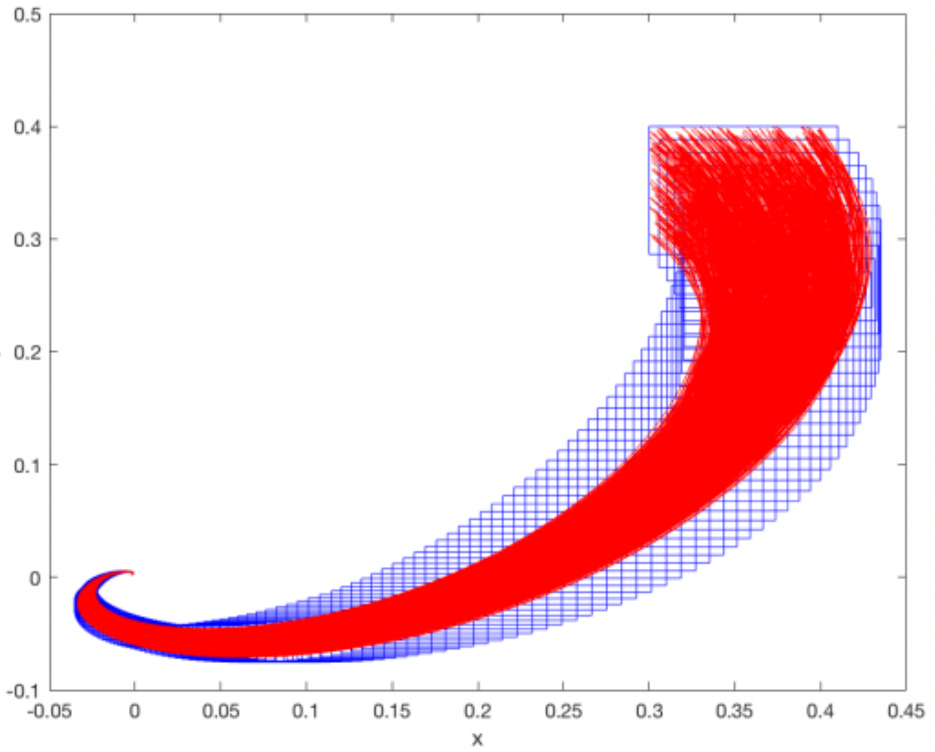

0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
y
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
0.4
0.5
x

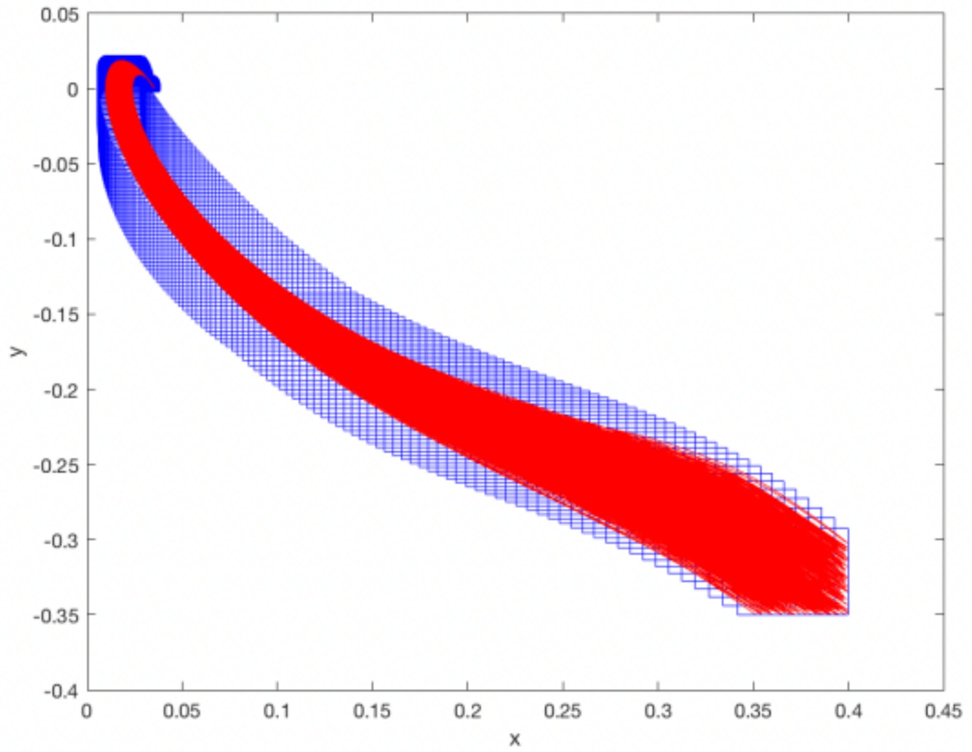

2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
y
-1.2
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
x


0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
y
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
x


0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
y
-0.1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
x


1.5
1
0.5
0
-0.5
-1
-1.5
x3
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
x1


0.5
0.4
0.3
0.2
0.1
0
-0.1
y
-0.05
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
x


0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
-0.35
-0.4
y
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
x

Pros and Cons

- Pros:
  - **Low overestimation accumulation:** It is a functional overapproximation method for the NNCS flowmaps.
  - **Good average efficiency:** Only need to solve an MILP problem.
  - **Allows to consider uncertainties:** Noises in sensing and actuating.

- Cons:
  - The activation functions have to be ReLU.
  - Very low efficiency in the worst case: Too many pieces in the PWL model, MILP problems are sometimes hard to solve.

End-to-End Bernstein Approximation

Main idea:

- We compute a **(Multivariate) Bernstein polynomial** p_B for the neural network κ over the domain X_{j-1}.

- Next, we compute an interval I which contains the difference $\varepsilon(x) = \kappa(x) - p_B(x)$ for all$x \in X_{j-1}$.

- Then $p + I$ is a functional overapproximation of κ.

- The main idea is very similar to the previous method, but we do not need an intermediate approximation for the error bound evaluation.

Background: Univariate Bernstein Polynomial

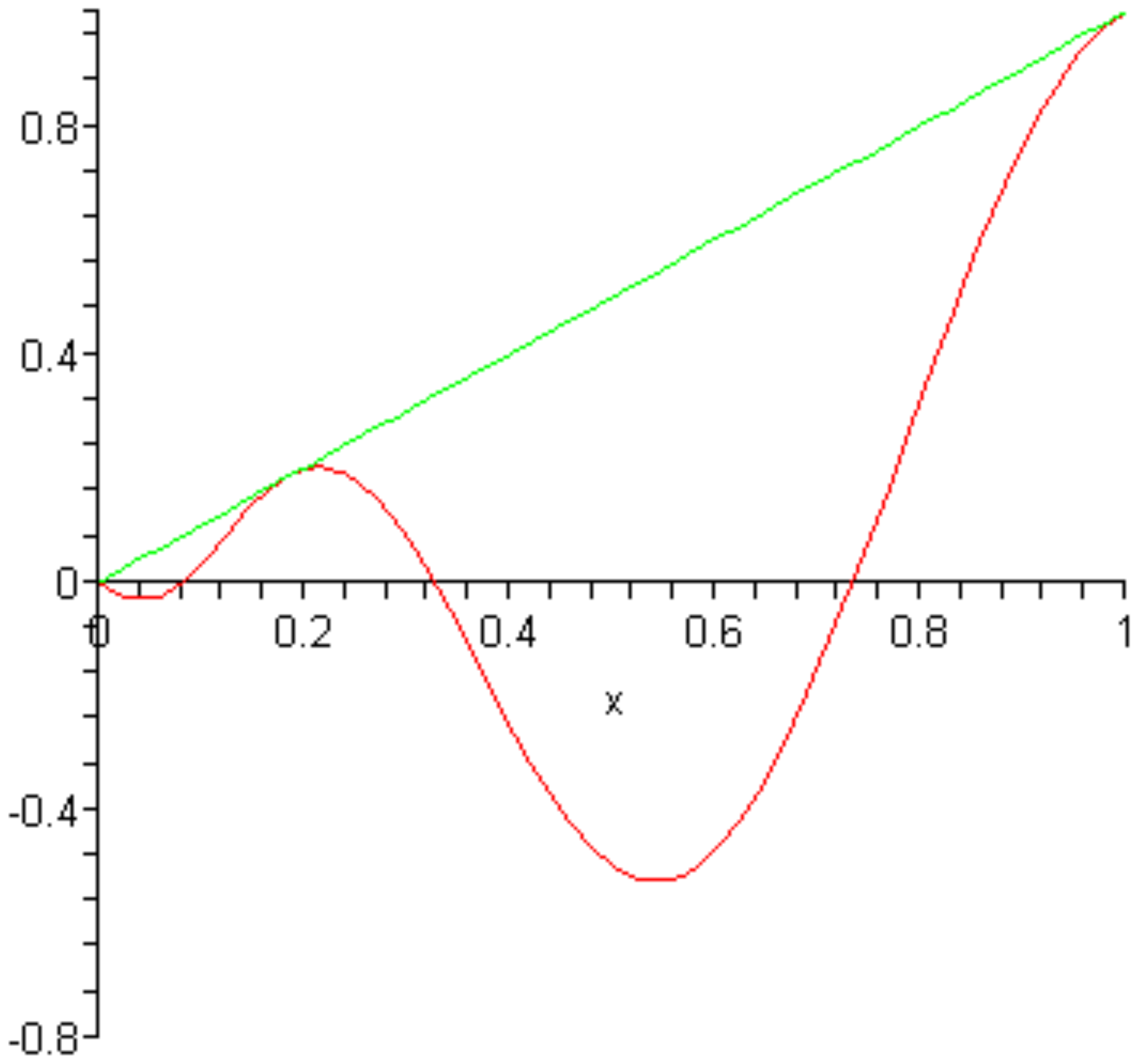
Given a **continuous** function $f(x)$ over the domain $x \in [0,1]$, its degree n Bernstein interpolation can be computed as

$$p(x) = a_0 b_{0,n}(x) + a_1 b_{1,n}(x) + \cdots + a_n b_{n,n}(x)$$

such that $a_v = f(v/n)$. The monomials $b_{i,n}(x)$ are called Bernstein basis.

Advantages:

- The **best** polynomial approximation.
- The error $|f(x) - p(x)|$ **converges to 0** when $n \to \infty$.
- The size of $p(x)$ is **linear** in n.



[Wikipedia: Bernstein polynomial]

Examples of Bernstein Basis

Order 1: $b_{0,0}(x) = 1,$

Order 2: $b_{0,1}(x) = 1 - x,$ $b_{1,1}(x) = x$

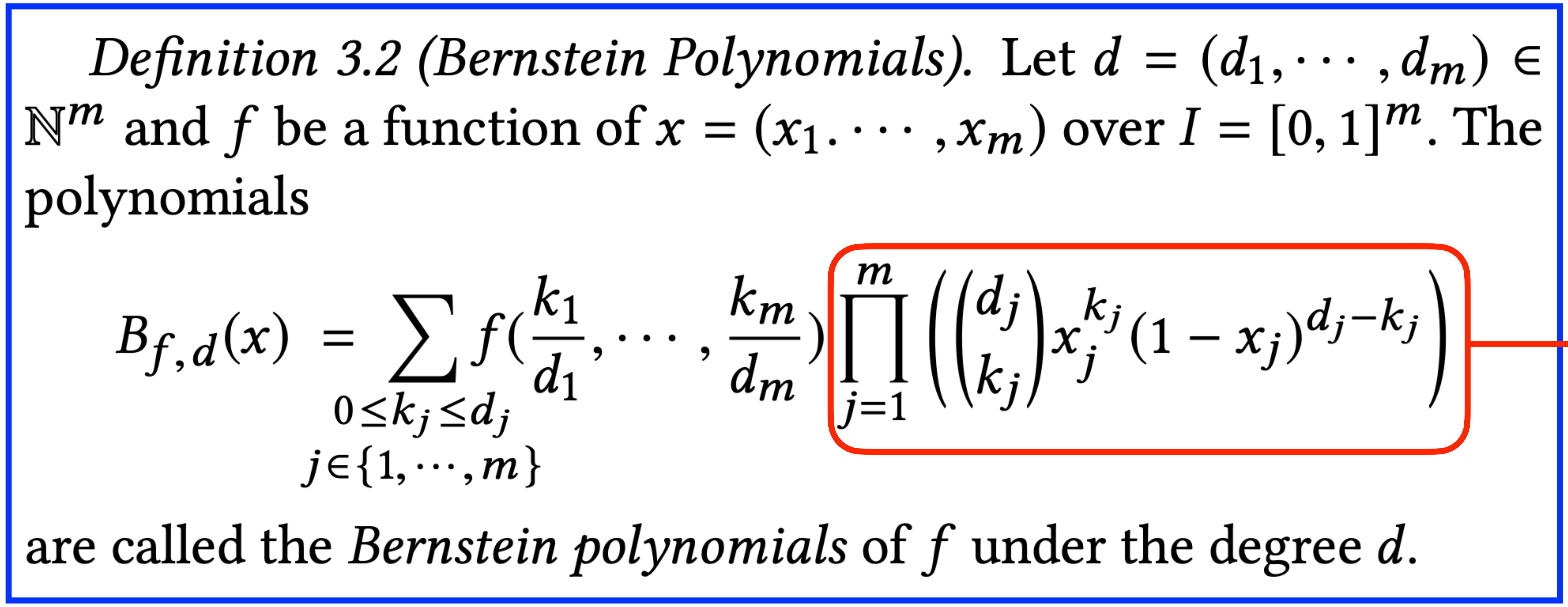
Order 3: $b_{0,2}(x) = (1 - x)^2,$ $b_{1,2}(x) = 2x(1 - x),$ $b_{2,2}(x) = x^2$

Order 4: $b_{0,3}(x) = (1 - x)^3,$ $b_{1,3}(x) = 3x(1 - x)^2,$ $b_{2,3}(x) = 3x^2(1 - x),$ $b_{3,3}(x) = x^3$

[Wikipedia: Bernstein polynomial]

Bernstein polynomials of different orders use different bases.

Background: Multivariate Bernstein Polynomial



Definition 3.2 (Bernstein Polynomials). Let $d = (d_1, \cdots, d_m) \in \mathbb{N}^m$ and f be a function of $x = (x_1. \cdots, x_m)$ over $I = [0, 1]^m$. The polynomials

$$B_{f,d}(x) = \sum_{\substack{0 \le k_j \le d_j \\ j \in \{1, \cdots, m\}}} f\left(\frac{k_1}{d_1}, \cdots, \frac{k_m}{d_m}\right) \boxed{\prod_{j=1}^{m} \left(\binom{d_j}{k_j} x_j^{k_j} (1 - x_j)^{d_j - k_j} \right)}$$

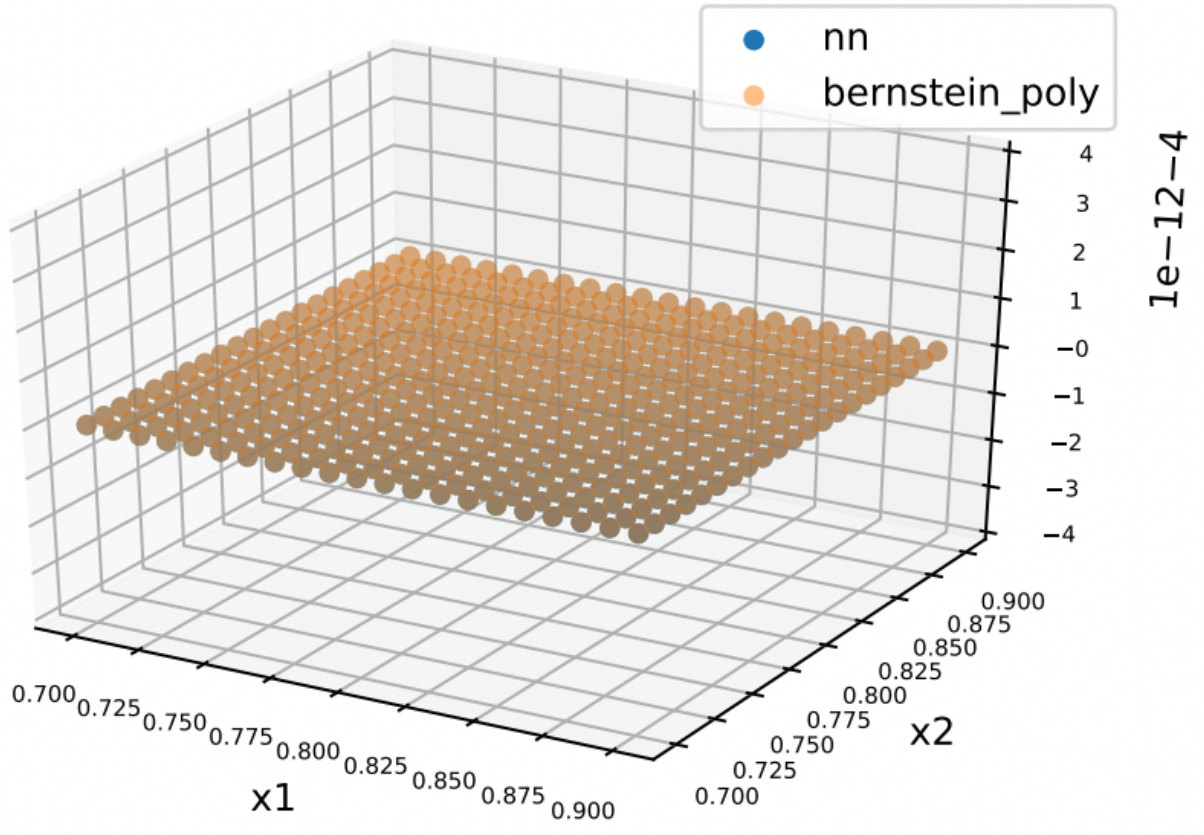
are called the *Bernstein polynomials* of f under the degree d.

multivariate Bernstein basis

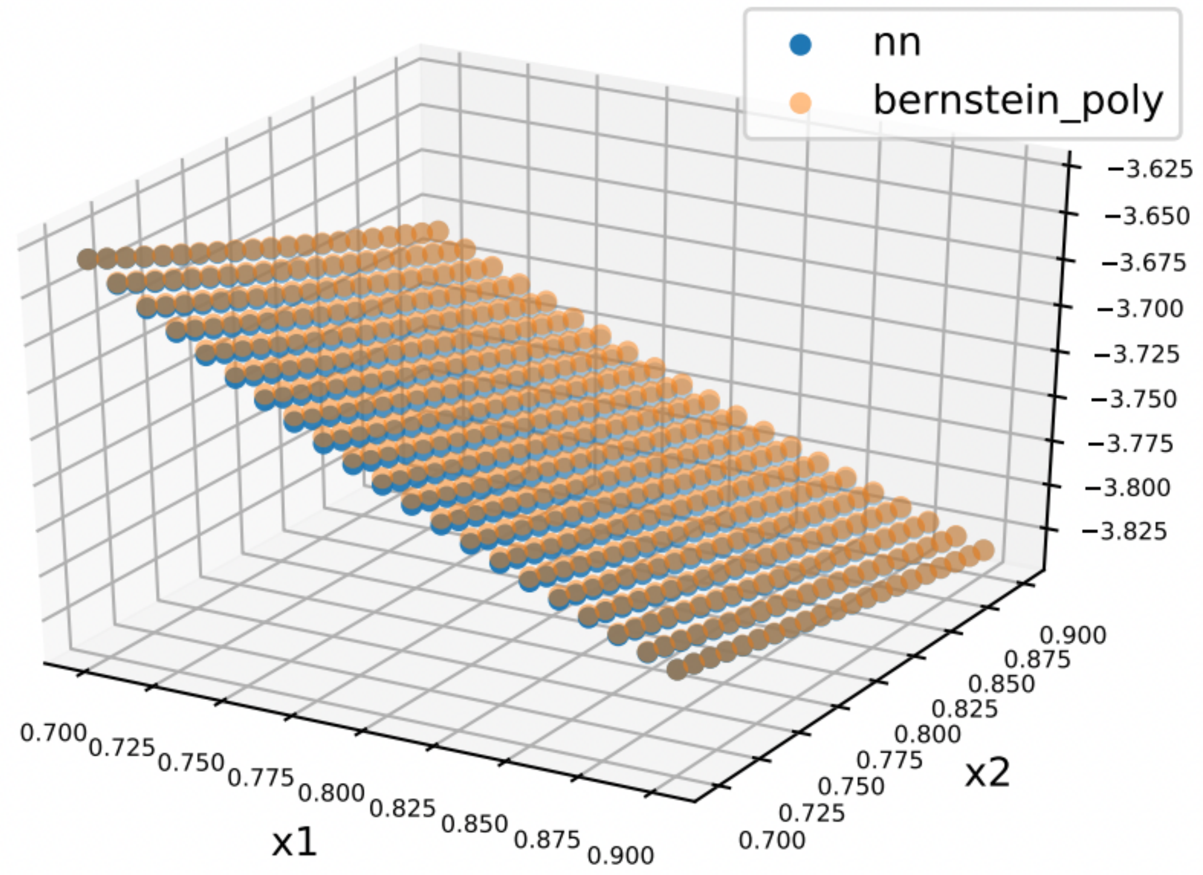
- The domain $[0,1]^m$ can be shifted to any m-dimensional box.
- The degree of the polynomial is $d_1 + d_2 + \cdots + d_m$.
- There are totally $\prod_{j=1}^{m} d_j$ coefficients need to be computed.

Approximating a Neural Network

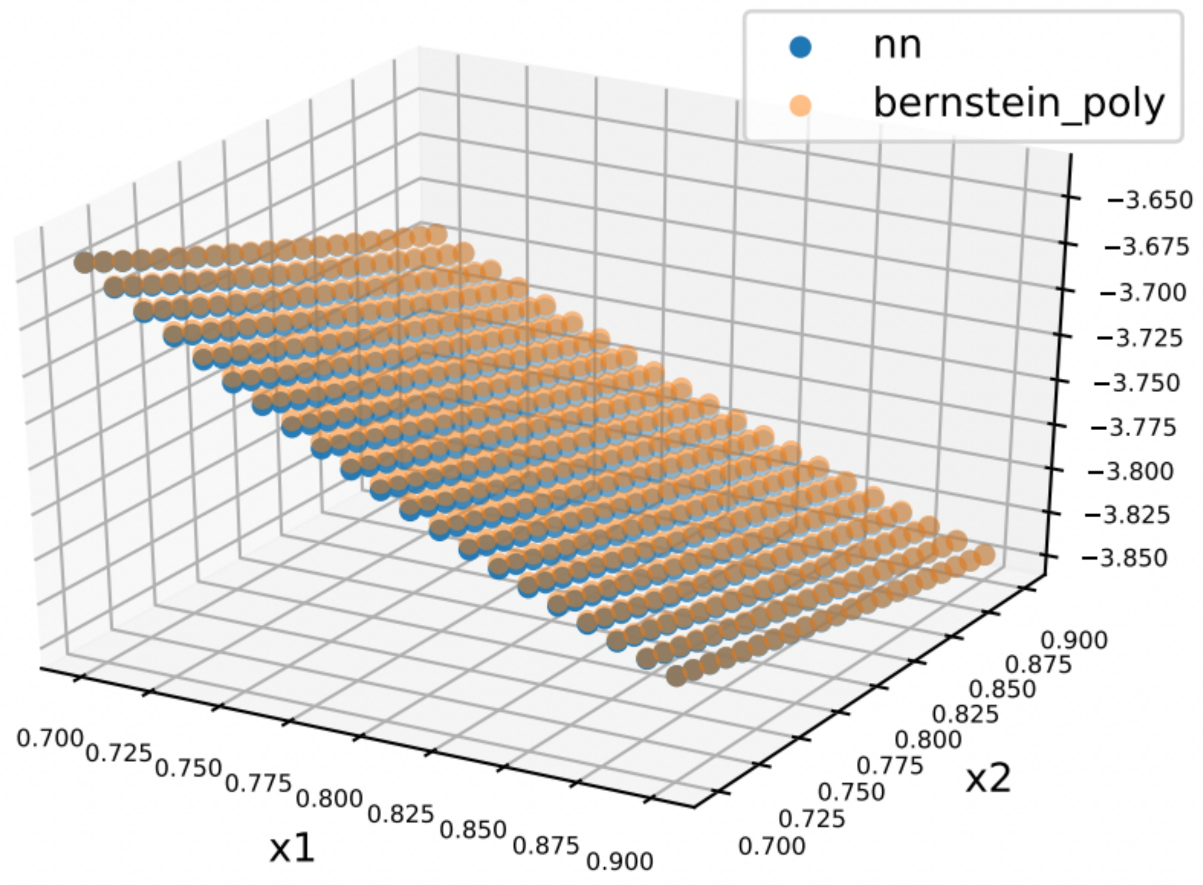
- A Bernstein polynomial can be directly computed for κ over a given input range.
- Since Bernstein approximation only requires the original function to be continuous, we may handle most of the activation functions.
- We often do not need high-degree Bernstein polynomials.



(a) Approximation for ReLU neural network

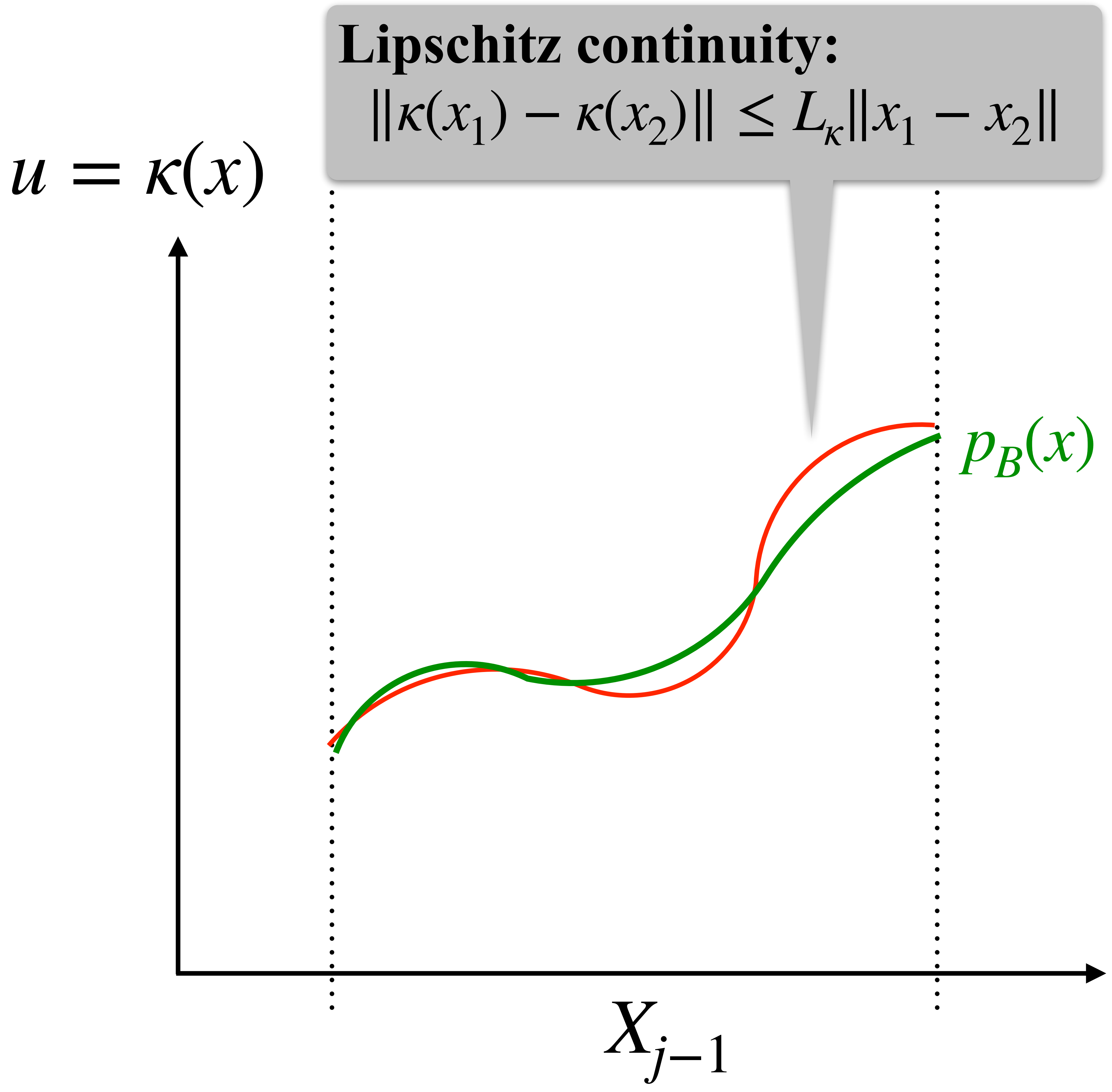


(b) Approximation for sigmoid neural network



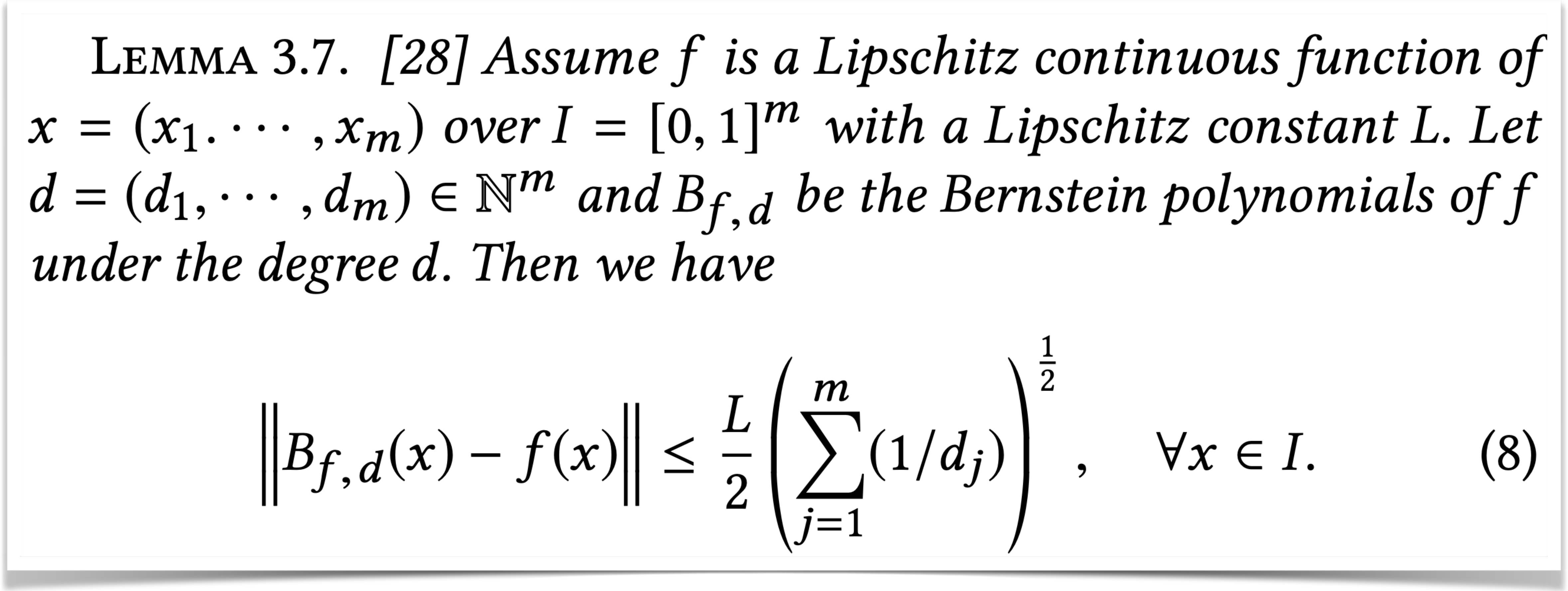
(c) Approximation for tanh neural network

Error Bound Evaluation using Lipschitz Constant



Lipschitz continuity:
$\|\kappa(x_1) - \kappa(x_2)\| \le L_\kappa \|x_1 - x_2\|$

The error bound is proportional to the Lipschitz constant which is often conservatively estimated.

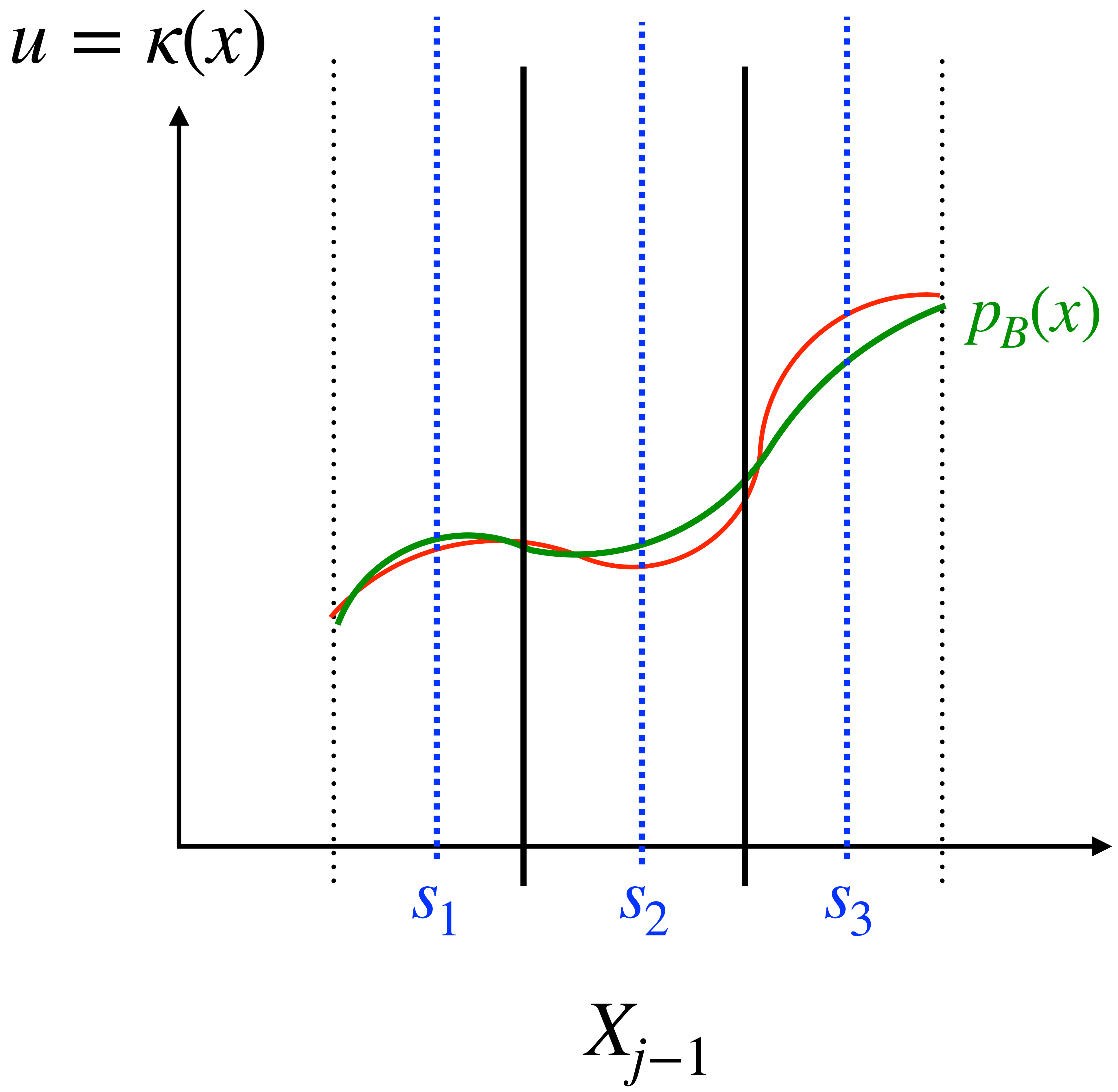


Lemma 3.7. *[28] Assume f is a Lipschitz continuous function of $x = (x_1. \cdots, x_m)$ over $I = [0,1]^m$ with a Lipschitz constant L. Let $d = (d_1, \cdots, d_m) \in \mathbb{N}^m$ and $B_{f,d}$ be the Bernstein polynomials of f under the degree d. Then we have*

$$\left\| B_{f,d}(x) - f(x) \right\| \le \frac{L}{2} \left(\sum_{j=1}^{m} (1/d_j) \right)^{\frac{1}{2}}, \quad \forall x \in I. \tag{8}$$

[28] George G Lorentz. 2013. *Bernstein polynomials.* American Mathematical Soc.

Error Bound Evaluation using Samples



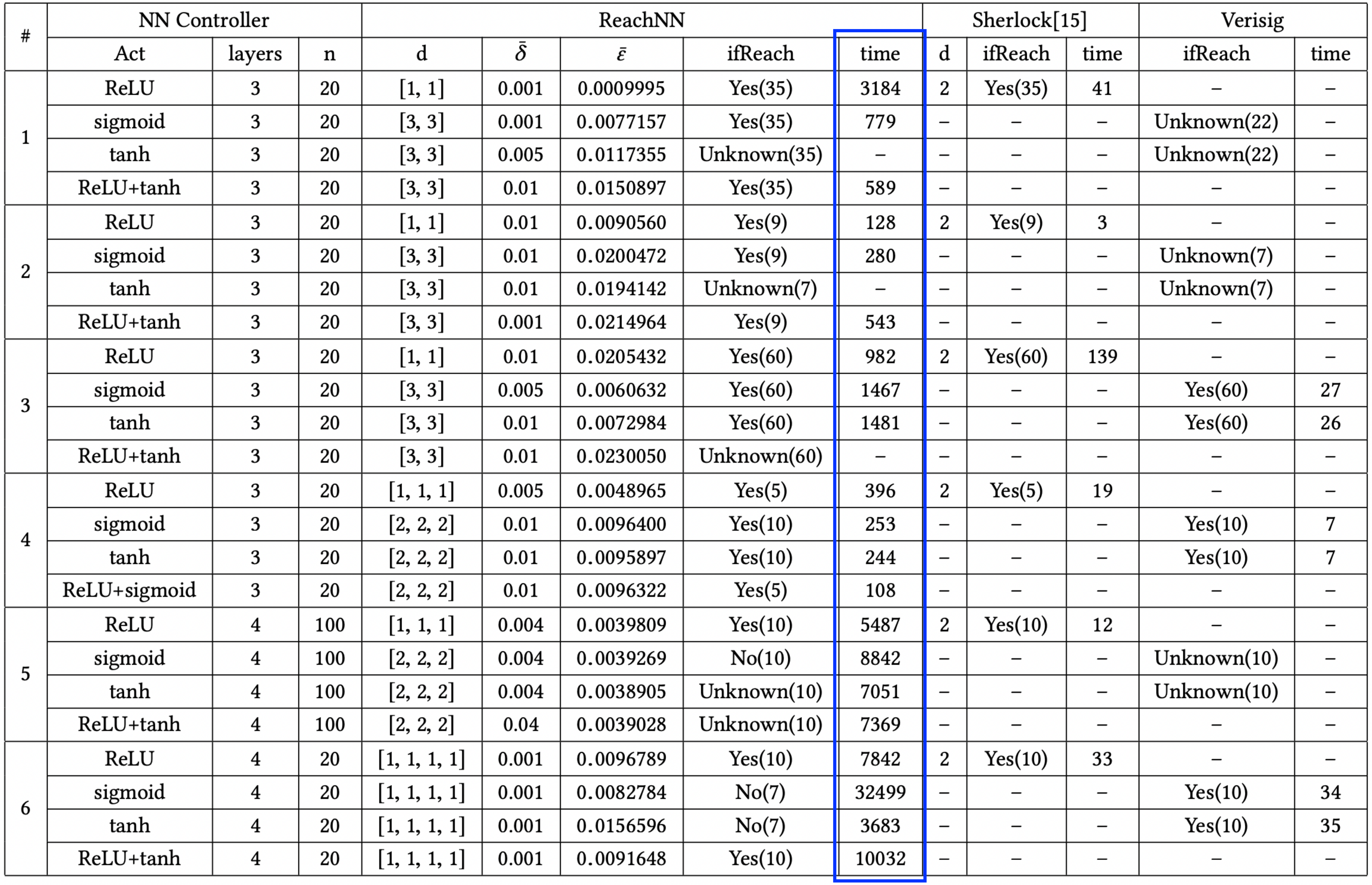
$$\varepsilon = \frac{L}{2}\sqrt{\sum_{j=1}^{m}\left(\frac{u_j - l_j}{n_j}\right)^2} \; + \; \max_{s} \|p_B(s) - \kappa(s)\|$$

n_j: number of subdivisions in the j-th dimension.

s: a sample.

$$X_{j-1} \subseteq [l_1, u_1] \times \cdots \times [l_m, u_m]$$

Experiments



#	NN Controller			ReachNN					Sherlock[15]			Verisig	
	Act	layers	n	d	$\bar{\delta}$	$\bar{\varepsilon}$	ifReach	time	d	ifReach	time	ifReach	time
1	ReLU	3	20	[1, 1]	0.001	0.0009995	Yes(35)	3184	2	Yes(35)	41	–	–
	sigmoid	3	20	[3, 3]	0.001	0.0077157	Yes(35)	779	–	–	–	Unknown(22)	–
	tanh	3	20	[3, 3]	0.005	0.0117355	Unknown(35)	–	–	–	–	Unknown(22)	–
	ReLU+tanh	3	20	[3, 3]	0.01	0.0150897	Yes(35)	589	–	–	–	–	–
2	ReLU	3	20	[1, 1]	0.01	0.0090560	Yes(9)	128	2	Yes(9)	3	–	–
	sigmoid	3	20	[3, 3]	0.01	0.0200472	Yes(9)	280	–	–	–	Unknown(7)	–
	tanh	3	20	[3, 3]	0.01	0.0194142	Unknown(7)	–	–	–	–	Unknown(7)	–
	ReLU+tanh	3	20	[3, 3]	0.001	0.0214964	Yes(9)	543	–	–	–	–	–
3	ReLU	3	20	[1, 1]	0.01	0.0205432	Yes(60)	982	2	Yes(60)	139	–	–
	sigmoid	3	20	[3, 3]	0.005	0.0060632	Yes(60)	1467	–	–	–	Yes(60)	27
	tanh	3	20	[3, 3]	0.01	0.0072984	Yes(60)	1481	–	–	–	Yes(60)	26
	ReLU+tanh	3	20	[3, 3]	0.01	0.0230050	Unknown(60)	–	–	–	–	–	–
4	ReLU	3	20	[1, 1, 1]	0.005	0.0048965	Yes(5)	396	2	Yes(5)	19	–	–
	sigmoid	3	20	[2, 2, 2]	0.01	0.0096400	Yes(10)	253	–	–	–	Yes(10)	7
	tanh	3	20	[2, 2, 2]	0.01	0.0095897	Yes(10)	244	–	–	–	Yes(10)	7
	ReLU+sigmoid	3	20	[2, 2, 2]	0.01	0.0096322	Yes(5)	108	–	–	–	–	–
5	ReLU	4	100	[1, 1, 1]	0.004	0.0039809	Yes(10)	5487	2	Yes(10)	12	–	–
	sigmoid	4	100	[2, 2, 2]	0.004	0.0039269	No(10)	8842	–	–	–	Unknown(10)	–
	tanh	4	100	[2, 2, 2]	0.004	0.0038905	Unknown(10)	7051	–	–	–	Unknown(10)	–
	ReLU+tanh	4	100	[2, 2, 2]	0.04	0.0039028	Unknown(10)	7369	–	–	–	–	–
6	ReLU	4	20	[1, 1, 1, 1]	0.001	0.0096789	Yes(10)	7842	2	Yes(10)	33	–	–
	sigmoid	4	20	[1, 1, 1, 1]	0.001	0.0082784	No(7)	32499	–	–	–	Yes(10)	34
	tanh	4	20	[1, 1, 1, 1]	0.001	0.0156596	No(7)	3683	–	–	–	Yes(10)	35
	ReLU+tanh	4	20	[1, 1, 1, 1]	0.001	0.0091648	Yes(10)	10032	–	–	–	–	–

Pros and Cons (Comparing to the first method)

- **Pros:**
  - **Generality:** Works on all continuous activation functions.
  - **Stable time cost:** Does not require to solve optimization problems.
  - The polynomial computation does not require to go though the neural network.

- **Cons:**
  - **Very low efficiency in the worst case:** The computation cost of a Bernstein polynomial is exponential in the number of variables. The number of samples in the remainder estimation is also exponential in the number of variables.

Layer-by-Layer Propagation using Polynomial Arithmetic

Main idea:

- The functional overapproximation $p + I$ for κ over X_{j-1} is computed in a layer-by-layer propagation manner using an extension of Taylor model arithmetic.

- We use univariate Bernstein polynomial to approximate a non-differentiable activation function, and use both univariate Taylor and Bernstein polynomials to approximate differentiable activation functions.

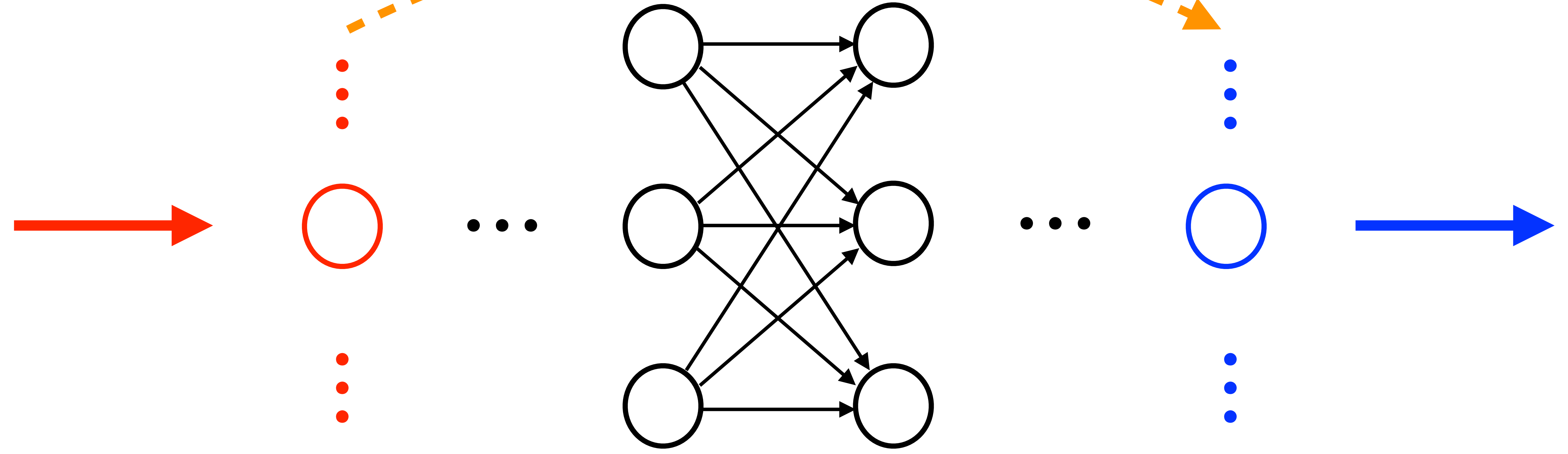
- Taylor model remainders can also be represented symbolically under linear mappings to reduce the error accumulation.

Main Difference from the Previous Methods

Sherlock, ReachNN

$\forall x \in X_{j-1} \,.\, \kappa(x) \in p(x) + I$

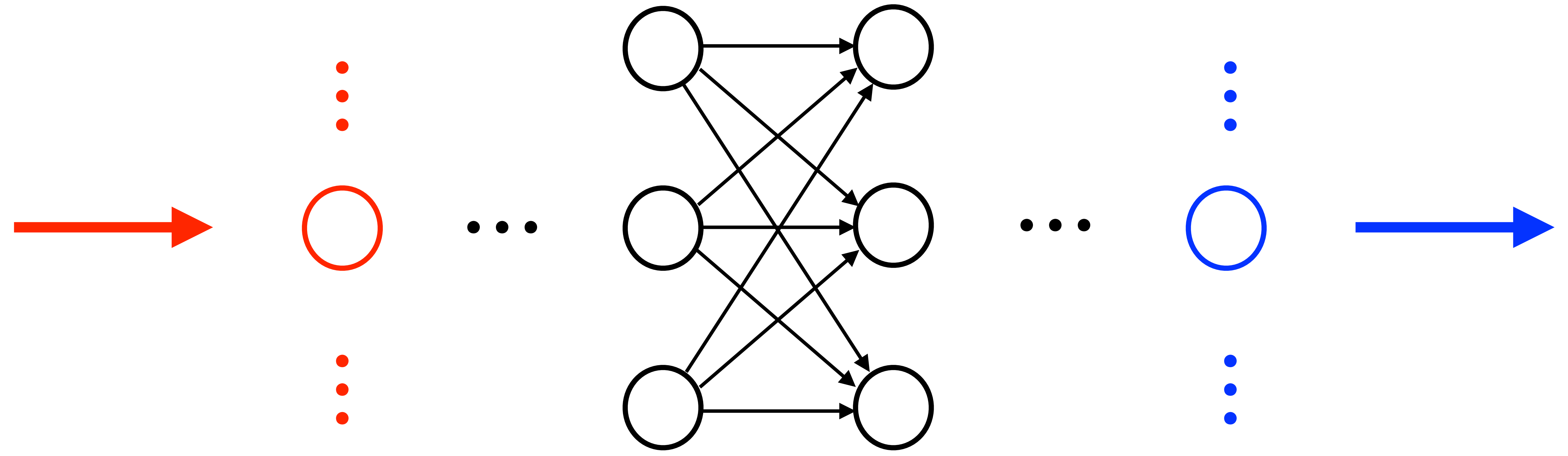
$X_{j-1} = q(x_0) + J$



$U_{j-1} = p(q(x_0) + J) + I$

POLAR

$X_{j-1} = q(x_0) + J$



Computed by layer-by-layer propagation

$U_{j-1} = p_r(x_0) + I_r$

Advantages

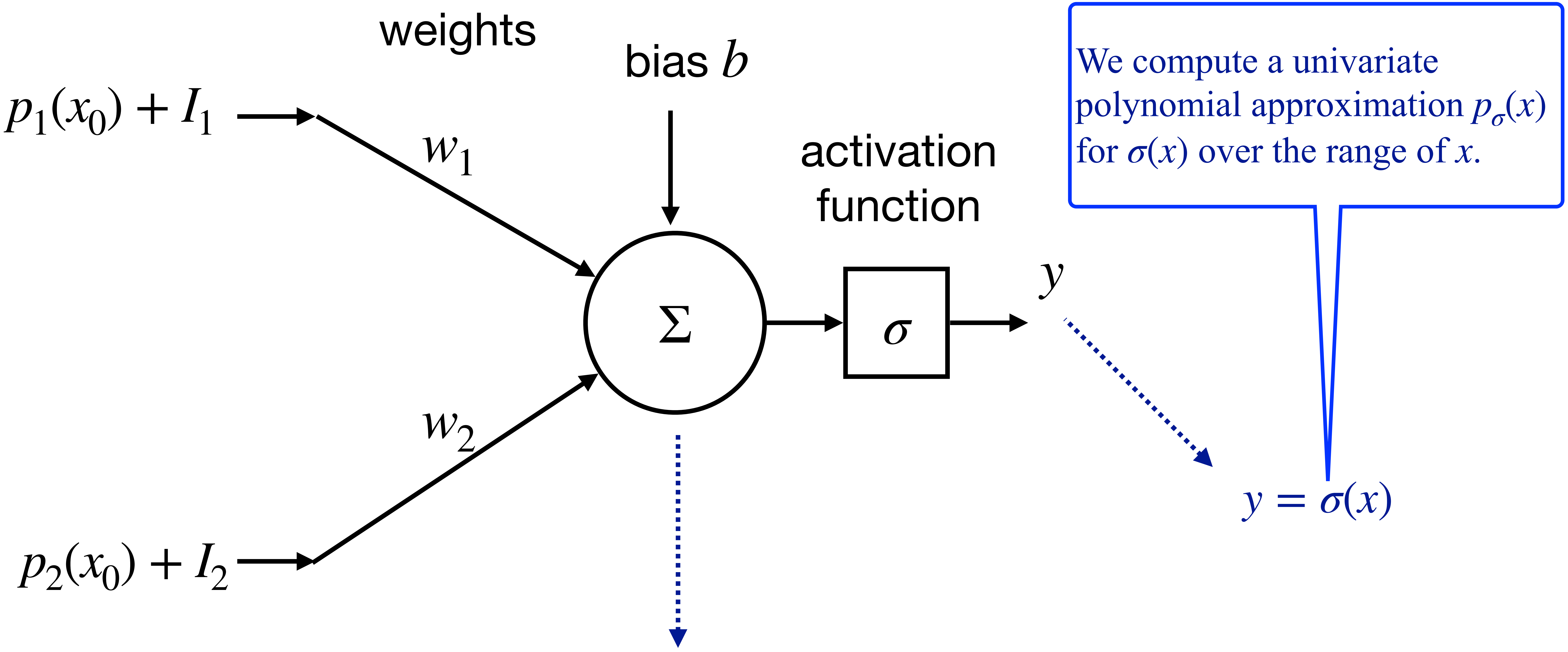
- There is **no** intermediate end-to-end overapproximation for the neural network. Notice that the size of this overapproximation may be **exponential** in the number of the neural network inputs.

- All of the Taylor models computed in the layer-by-layer propagation are only over the variable(s) x_0, and their sizes are **independent** from the size of the neural network.

- For each neuron, we only need to compute one Taylor model.

- All of the Taylor and Bernstein polynomials are univariate, since the activation functions are univariate.

Main Algorithm



weights

bias b

$p_1(x_0) + I_1$

w_1

$p_2(x_0) + I_2$

w_2

Σ

activation function

σ

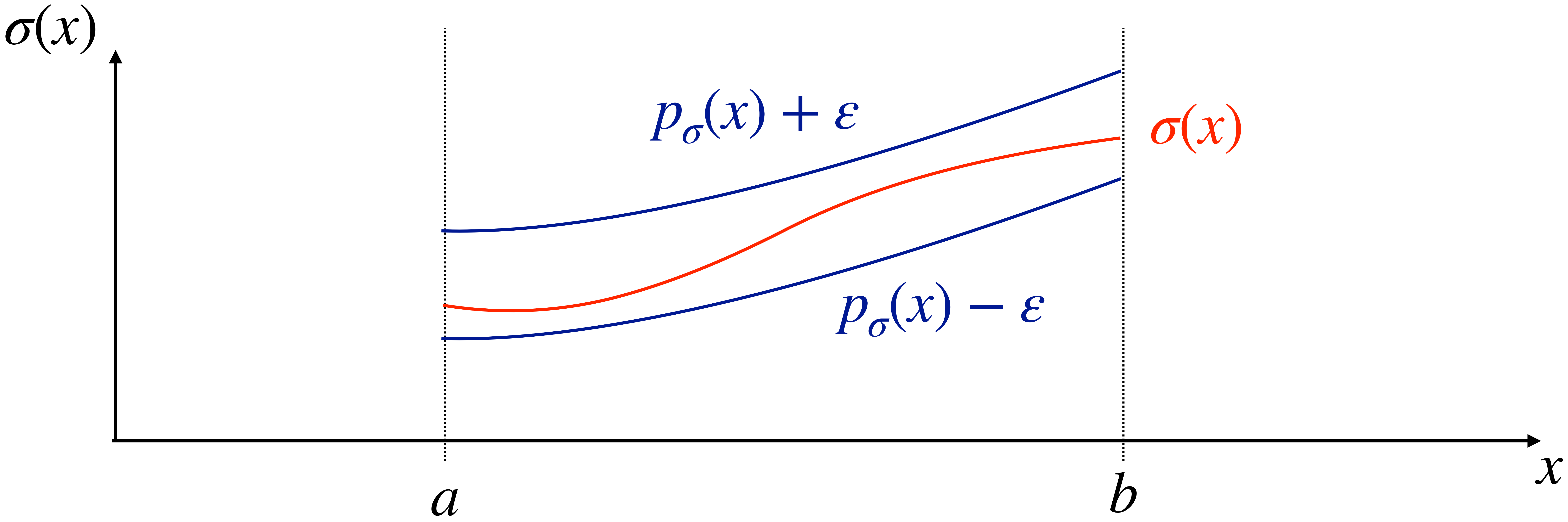
y

We compute a univariate polynomial approximation $p_\sigma(x)$ for $\sigma(x)$ over the range of x.

$y = \sigma(x)$

$$x = w_1(p_1(x_0) + I_1) + w_2(p_2(x_0) + I_2) + b$$

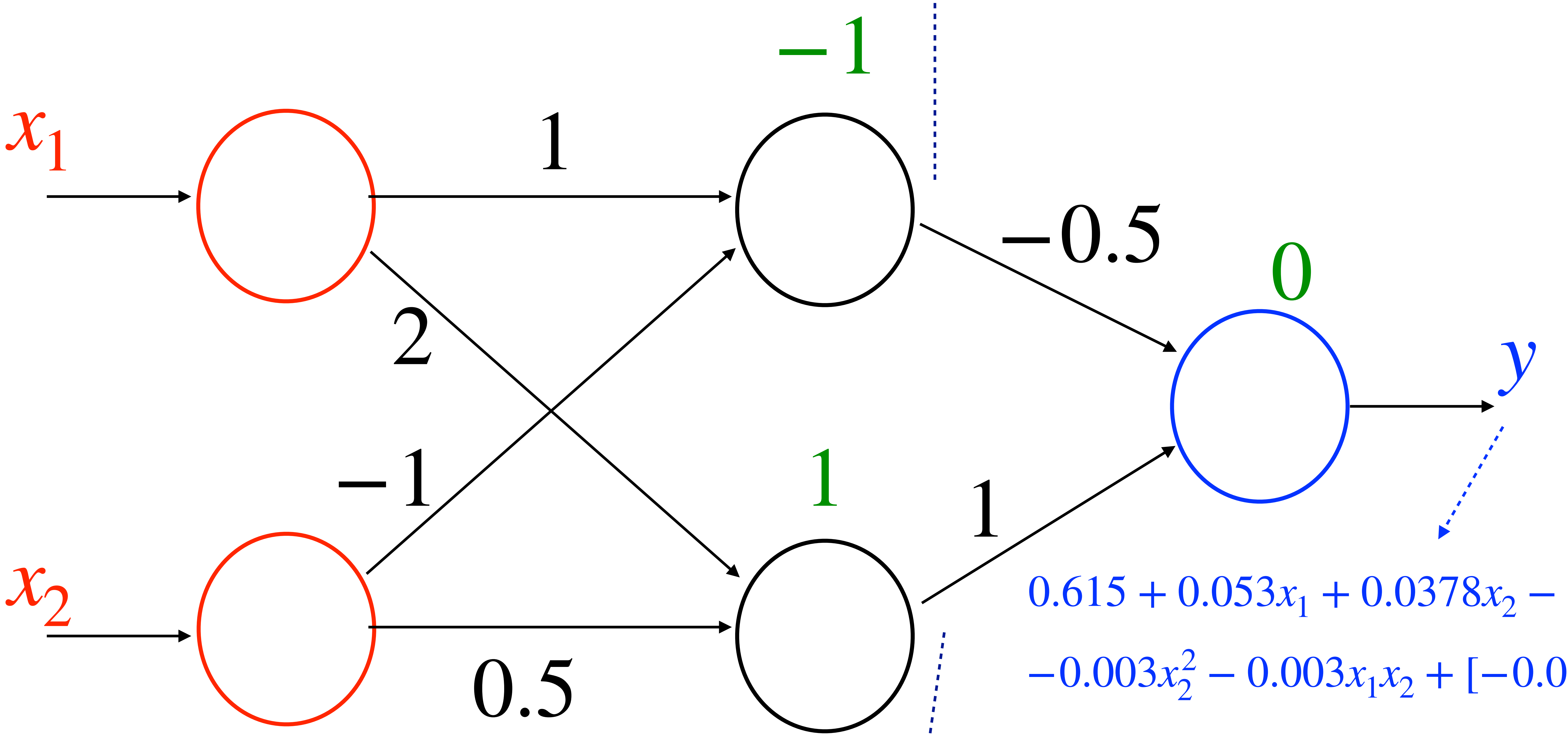
Remainder Evaluation



$$\varepsilon = \max_{i=1,\cdots,m}\left(\left|\boldsymbol{B_\sigma}\left(\frac{b-a}{m}(i-\frac{1}{2})+a\right)-\boldsymbol{\sigma}\left(\frac{b-a}{m}(i-\frac{1}{2})+a\right)\right|+\boldsymbol{L}\cdot\frac{b-a}{m}\right)$$

Example

$$0.329 + 0.171x_1 - 0.171x_2 + 0.015x_1^2 + 0.0152x_2^2 - 0.03x_1x_2 + [-0.066,0.066]$$



$$0.615 + 0.053x_1 + 0.0378x_2 - 0.014x_1^2$$
$$-0.003x_2^2 - 0.003x_1x_2 + [-0.038,0.038]$$

$$0.654 + 0.315x_1 + 0.079x_2 - 0.049x_1^2 - 0.003x_2^2 - 0.025x_1x_2 + [-0.088,0.088]$$

Taylor vs. Bernstein

- A Bernstein approximation is often better than the Taylor approximation of the same order.

- The remainder evaluation for a Bernstein polynomial is also more accurate than that for a Taylor polynomial of the same order.

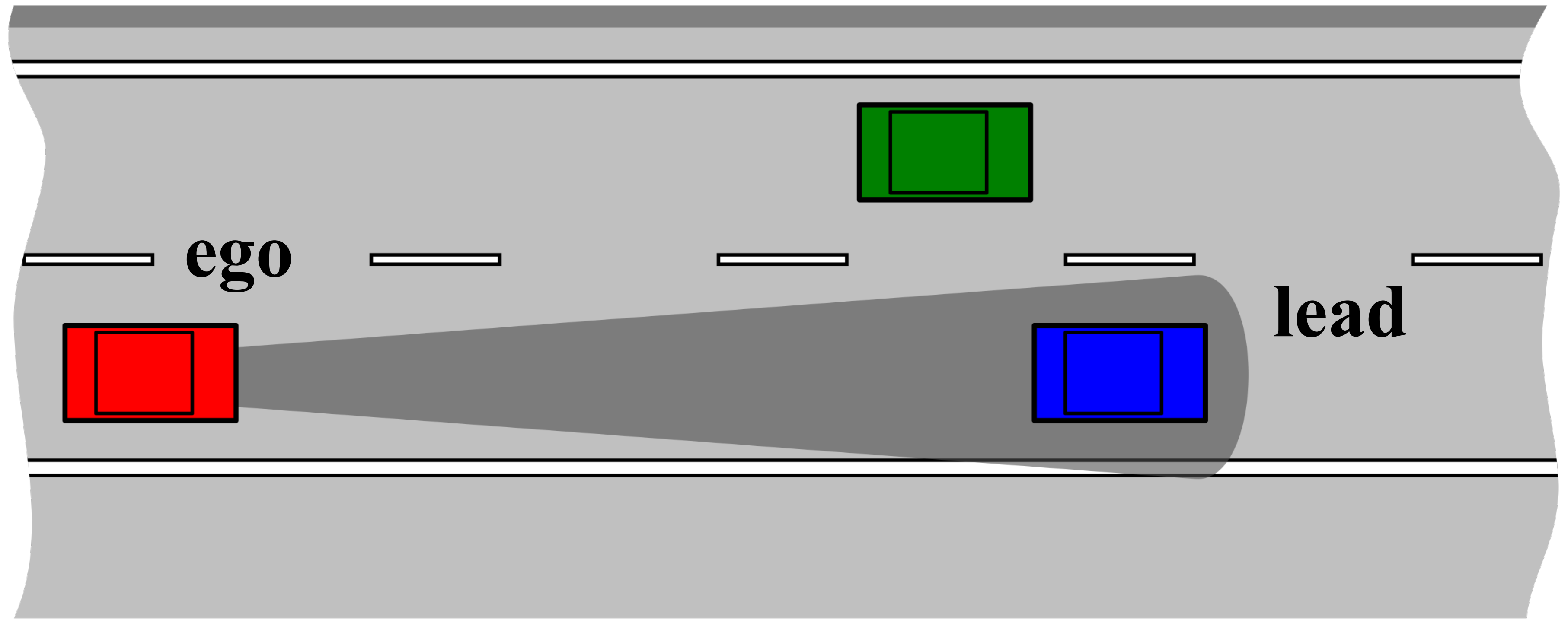
- However, the Taylor model composition used for each neuron is simplified by truncating the higher-order terms, and it could make a Bernstein approximation worse than a Taylor approximation.

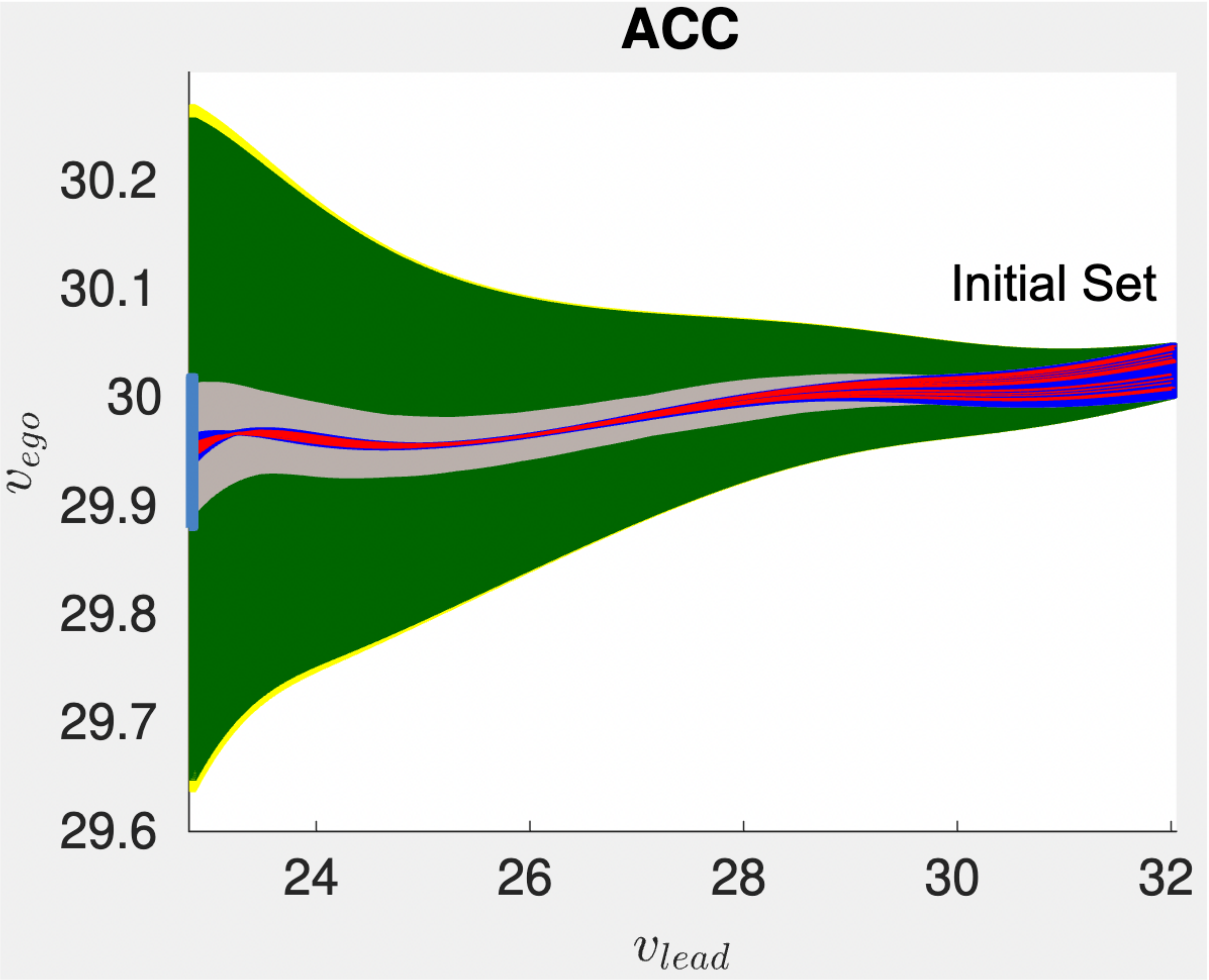
- Hence, for a differentiable activation function, we compute both Taylor and Bernstein overapproximations, and choose the more accurate one after composing them with the input Taylor model.

Case Study 1: Adaptive Cruise Control

$$\dot{x}_{lead} = v_{lead}, \quad \dot{v}_{lead} = \gamma_{lead}, \quad \dot{\gamma}_{lead} = -2\gamma_{lead} + 2a_{lead} - uv_{lead}^2,$$

$$\dot{x}_{ego} = v_{ego}, \quad \dot{v}_{ego} = \gamma_{ego}, \quad \dot{\gamma}_{ego} = -2\gamma_{ego} + 2a_{ego} - uv_{ego}^2,$$





NN controller size: 20 × 20 × 20 (tanh)

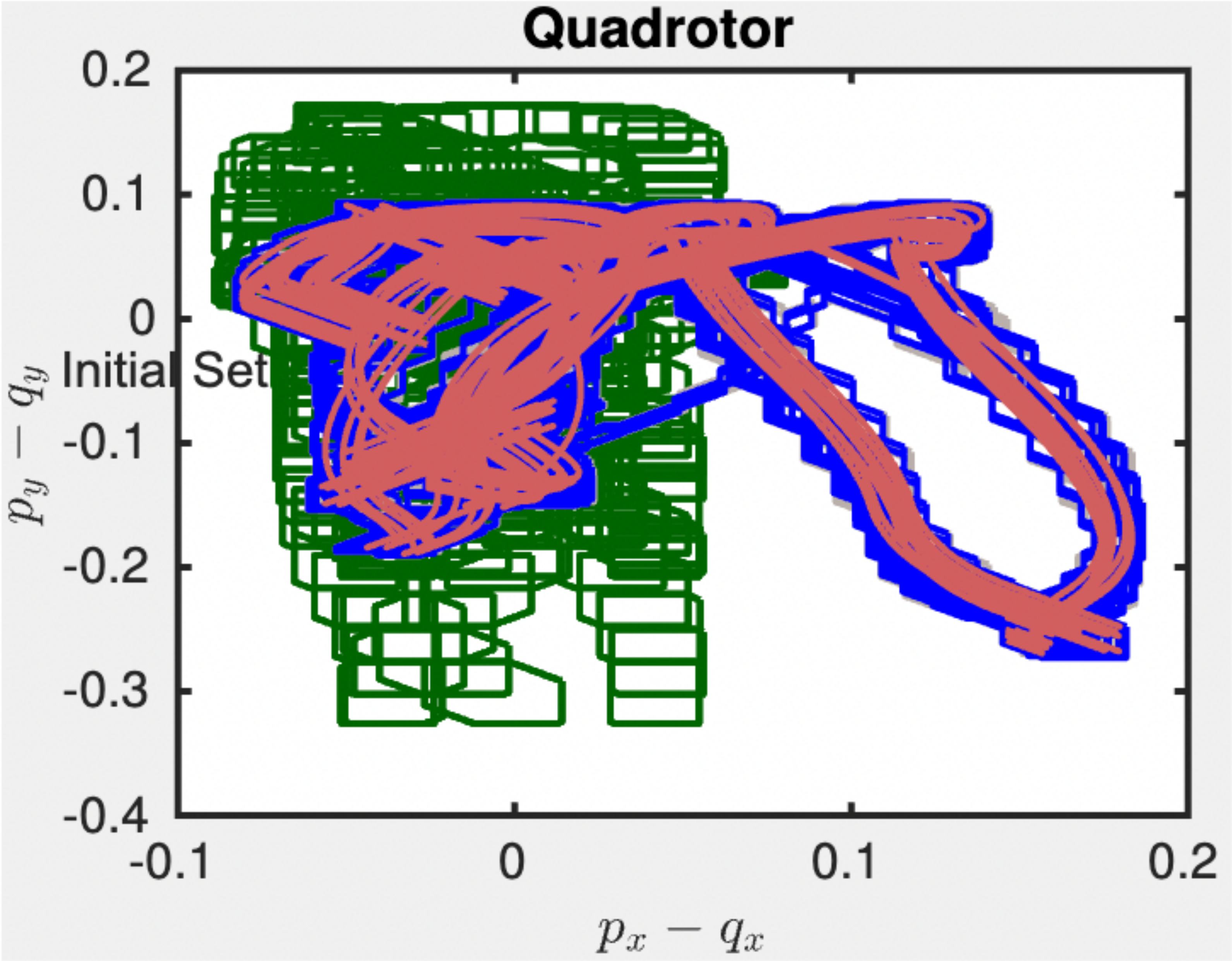
α-β-CROWN + Flow*:	-
NNV:	-
Verisig 2.0:	3344s
POLAR:	343s

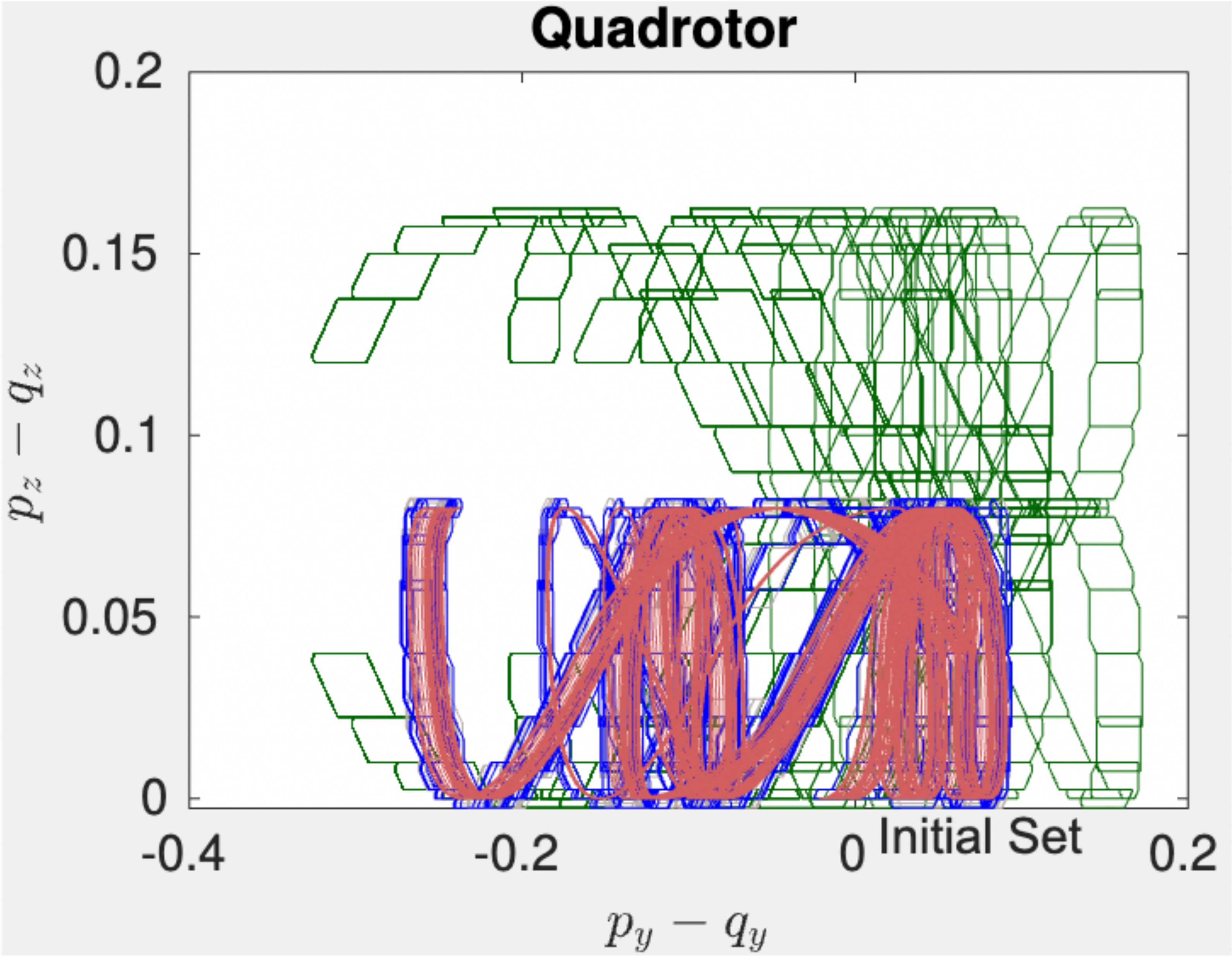
Case Study 2: Quadrotor (QMPC)

Quadrotor: $\dot{p}_x = v_x, \quad \dot{p}_y = v_y, \quad \dot{p}_z = v_z, \quad \dot{v}_x = gtan\theta, \quad \dot{v}_y = -gtan\phi, \quad \dot{v}_z = \tau - g,$

Planner: $\dot{q}_x = b_x, \quad \dot{q}_y = b_y, \quad \dot{q}_z = b_z, \quad \dot{b}_x = 0, \quad \dot{b}_y = 0, \quad \dot{b}_z = 0.$

Safety: $-0.32 \leq p_x - q_x, p_y - q_y, p_z - q_z \leq 0.32.$





NN controller size:
20 × 20 (tanh)

The neural network controller has a post-processing module which maps the neural network output to a control input according to a look-up table.

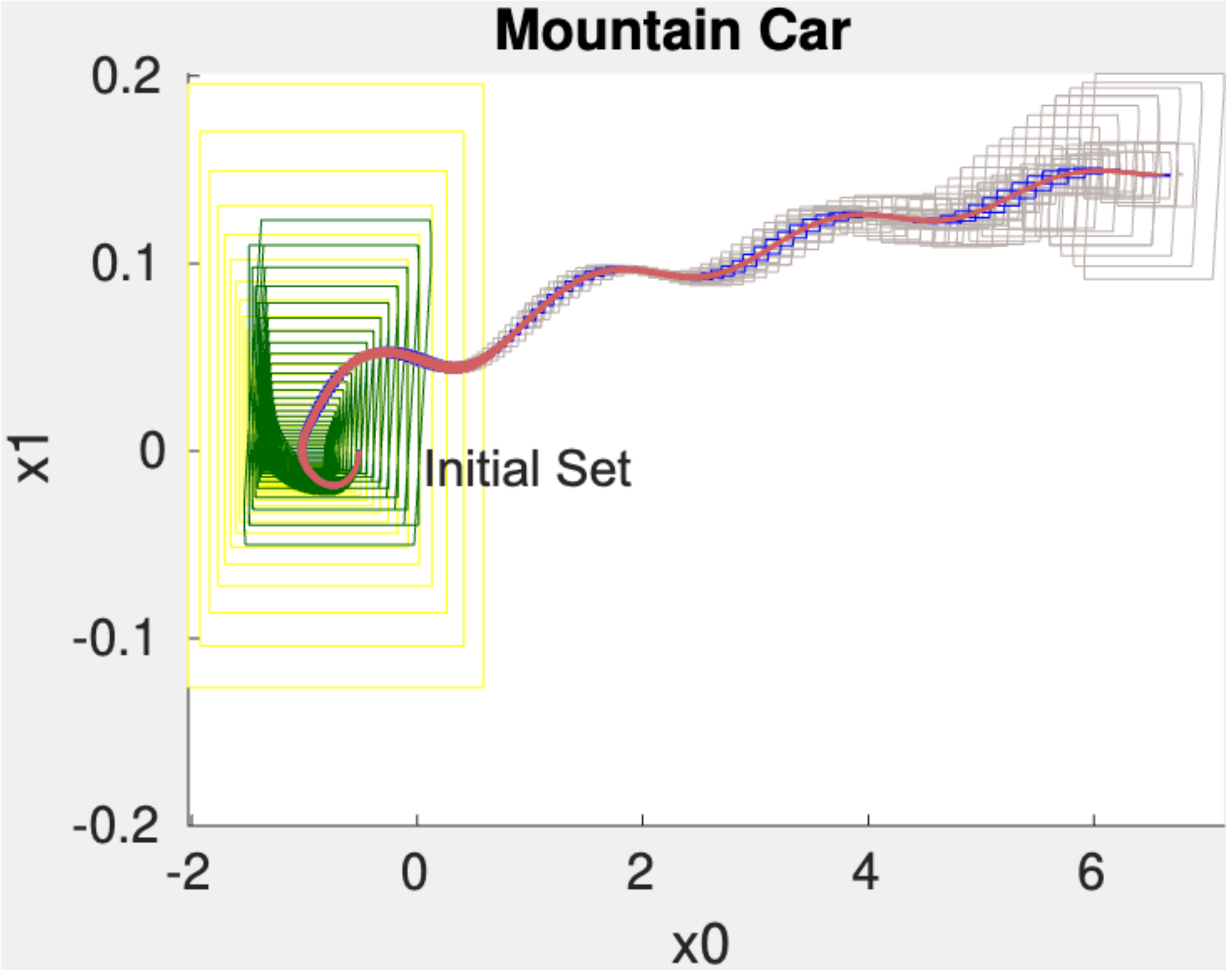
α-β-CROWN + Flow*:	-
Verisig 2.0:	652s
POLAR:	61s

R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, I. Lee. *Verifying the safety of autonomous systems with neural network controllers.* ACM Transactions on Embedded Computing Systems (TECS) 20 (1) (2020) 1– 26.

Case Study 3: Mountain Car

Discrete-time dynamics:

$$x_0[t+1] = x_0[t] + x_1[t],$$
$$x_1[t+1] = x_1[t] + 0.0015 \cdot u[t] - 0.0025 \cdot \cos(3 \cdot x_0[t]).$$

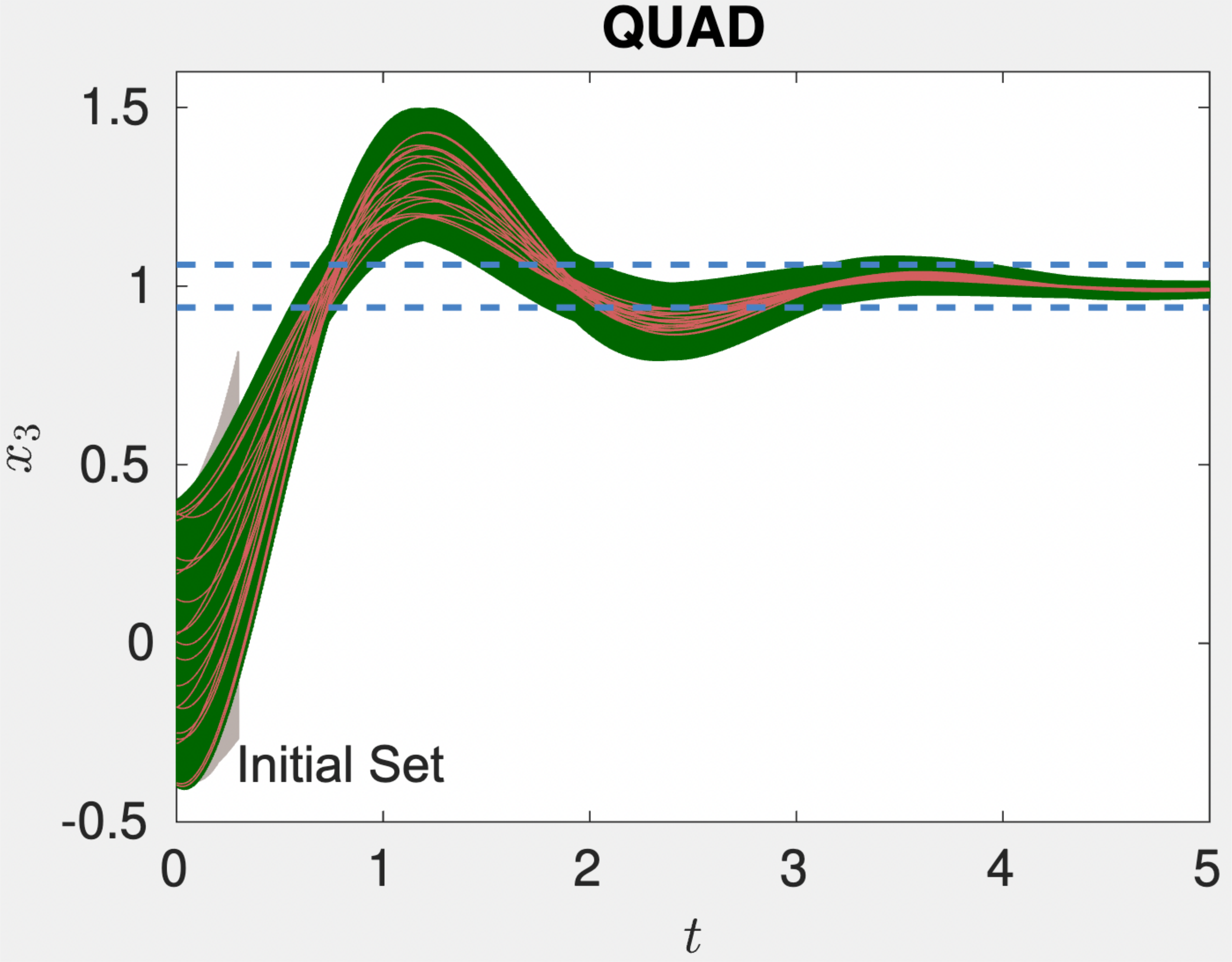


NN controller size: 16 × 16 (sigmoid+tanh)

α-β-CROWN + Flow*:	-
NNV:	-
Verisig 2.0:	237s
POLAR:	32s

Case Study 4: Quadrotor (QUAD)

$$\begin{cases}
\dot{x}_1 = \cos(x_8)\cos(x_9)x_4 + (\sin(x_7)\sin(x_8)\cos(x_9) - \cos(x_7)\sin(x_9))\,x_5 \\
\qquad + (\cos(x_7)\sin(x_8)\cos(x_9) + \sin(x_7)\sin(x_9))\,x_6 \\
\dot{x}_2 = \cos(x_8)\sin(x_9)x_4 + (\sin(x_7)\sin(x_8)\sin(x_9) + \cos(x_7)\cos(x_9))\,x_5 \\
\qquad + (\cos(x_7)\sin(x_8)\sin(x_9) - \sin(x_7)\cos(x_9))\,x_6 \\
\dot{x}_3 = \sin(x_8)x_4 - \sin(x_7)\cos(x_8)x_5 - \cos(x_7)\cos(x_8)x_6 \\
\dot{x}_4 = x_{12}x_5 - x_{11}x_6 - g\sin(x_8) \\
\dot{x}_5 = x_{10}x_6 - x_{12}x_4 + g\cos(x_8)\sin(x_7) \\
\dot{x}_6 = x_{11}x_4 - x_{10}x_5 + g\cos(x_8)\cos(x_7) - g - u_1/m \\
\dot{x}_7 = x_{10} + \sin(x_7)\tan(x_8)x_{11} + \cos(x_7)\tan(x_8)x_{12} \\
\dot{x}_8 = \cos(x_7)x_{11} - \sin(x_7)x_{12} \\
\dot{x}_9 = \dfrac{\sin(x_7)}{\cos(x_8)}x_{11} - \sin(x_7)x_{12} \\
\dot{x}_{10} = \dfrac{J_y - J_z}{J_x}x_{11}x_{12} + \dfrac{1}{J_x}u_2 \\
\dot{x}_{11} = \dfrac{J_z - J_x}{J_y}x_{10}x_{12} + \dfrac{1}{J_y}u_3 \\
\dot{x}_{12} = \dfrac{J_x - J_y}{J_z}x_{10}x_{11} + \dfrac{1}{J_z}\tau_\psi
\end{cases}$$

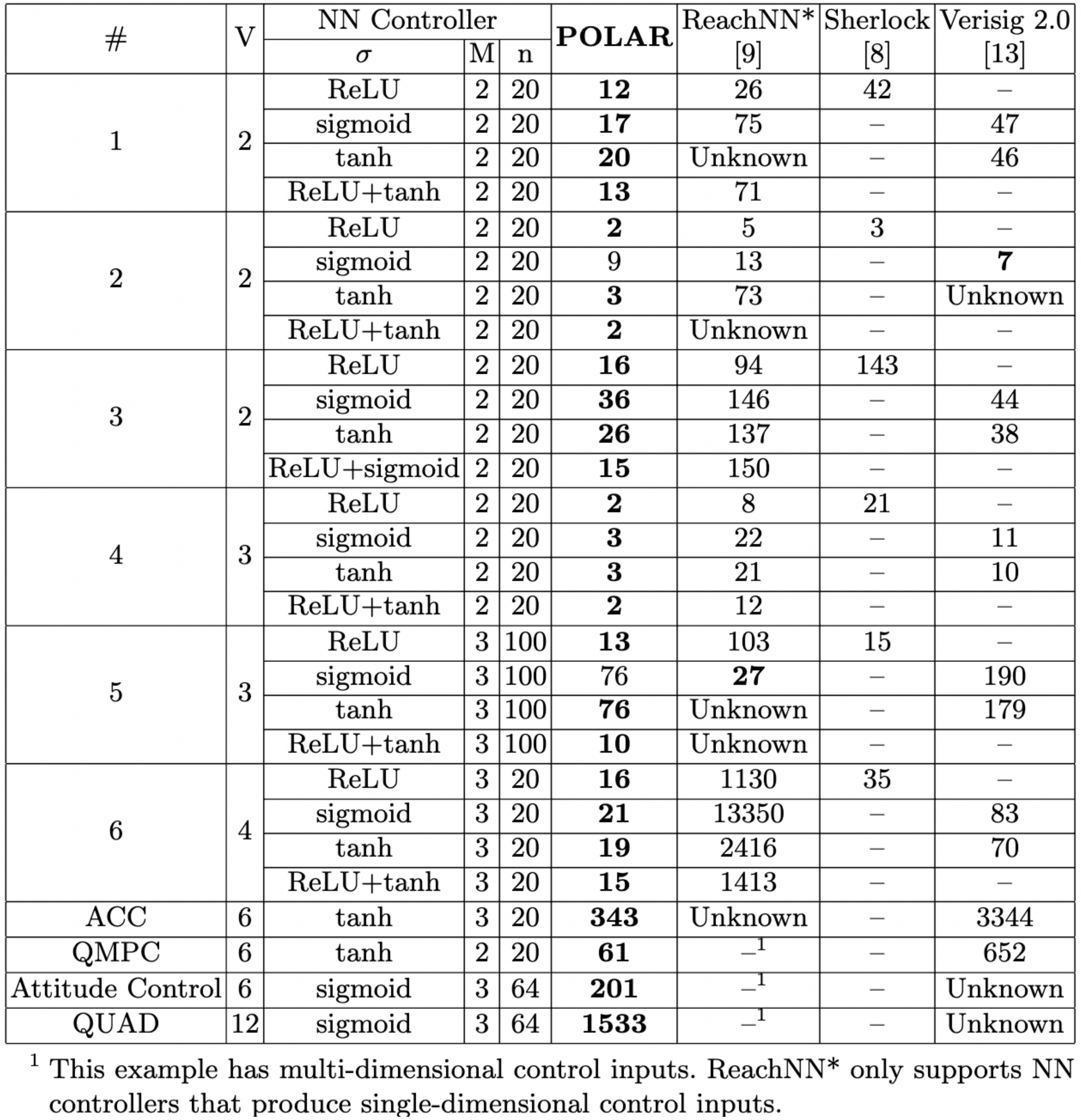


Verisig 2.0: -

POLAR: 1533s

NN controller size: $64 \times 64 \times 64$ (sigmoid)

More Experiments



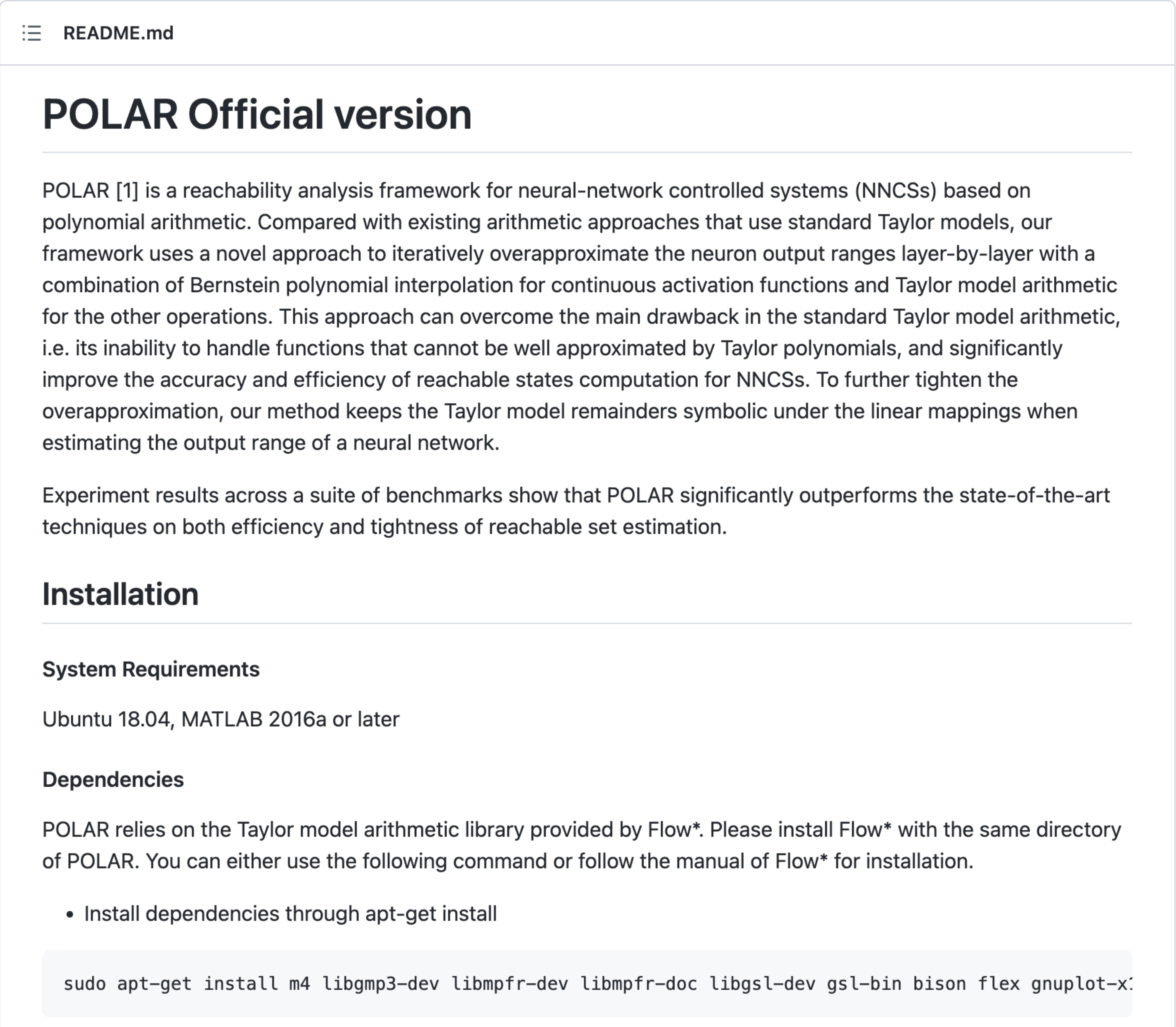
#	V	NN Controller			POLAR	ReachNN*	Sherlock	Verisig 2.0
		σ	M	n		[9]	[8]	[13]
1	2	ReLU	2	20	**12**	26	42	–
		sigmoid	2	20	**17**	75	–	47
		tanh	2	20	**20**	Unknown	–	46
		ReLU+tanh	2	20	**13**	71	–	–
2	2	ReLU	2	20	**2**	5	3	–
		sigmoid	2	20	9	13	–	**7**
		tanh	2	20	**3**	73	–	Unknown
		ReLU+tanh	2	20	**2**	Unknown	–	–
3	2	ReLU	2	20	**16**	94	143	–
		sigmoid	2	20	**36**	146	–	44
		tanh	2	20	**26**	137	–	38
		ReLU+sigmoid	2	20	**15**	150	–	–
4	3	ReLU	2	20	**2**	8	21	–
		sigmoid	2	20	**3**	22	–	11
		tanh	2	20	**3**	21	–	10
		ReLU+tanh	2	20	**2**	12	–	–
5	3	ReLU	3	100	**13**	103	15	–
		sigmoid	3	100	76	**27**	–	190
		tanh	3	100	**76**	Unknown	–	179
		ReLU+tanh	3	100	**10**	Unknown	–	–
6	4	ReLU	3	20	**16**	1130	35	–
		sigmoid	3	20	**21**	13350	–	83
		tanh	3	20	**19**	2416	–	70
		ReLU+tanh	3	20	**15**	1413	–	–
ACC	6	tanh	3	20	**343**	Unknown	–	3344
QMPC	6	tanh	2	20	**61**	–[1]	–	652
Attitude Control	6	sigmoid	3	64	**201**	–[1]	–	Unknown
QUAD	12	sigmoid	3	64	**1533**	–[1]	–	Unknown

[1] This example has multi-dimensional control inputs. ReachNN* only supports NN controllers that produce single-dimensional control inputs.

- All of the tools compute functional overapproximations using Taylor models.
- POLAR does not show its best performance, since we need to use the same hyperparameters in all of the tools to present a fair comparison.
- When using a time step which is same as the control stepsize, POLAR only costs 0.5 seconds to complete test #1.

Source Code of POLAR

GitHub Repository



README.md

POLAR Official version

POLAR [1] is a reachability analysis framework for neural-network controlled systems (NNCSs) based on polynomial arithmetic. Compared with existing arithmetic approaches that use standard Taylor models, our framework uses a novel approach to iteratively overapproximate the neuron output ranges layer-by-layer with a combination of Bernstein polynomial interpolation for continuous activation functions and Taylor model arithmetic for the other operations. This approach can overcome the main drawback in the standard Taylor model arithmetic, i.e. its inability to handle functions that cannot be well approximated by Taylor polynomials, and significantly improve the accuracy and efficiency of reachable states computation for NNCSs. To further tighten the overapproximation, our method keeps the Taylor model remainders symbolic under the linear mappings when estimating the output range of a neural network.

Experiment results across a suite of benchmarks show that POLAR significantly outperforms the state-of-the-art techniques on both efficiency and tightness of reachable set estimation.

Installation

System Requirements

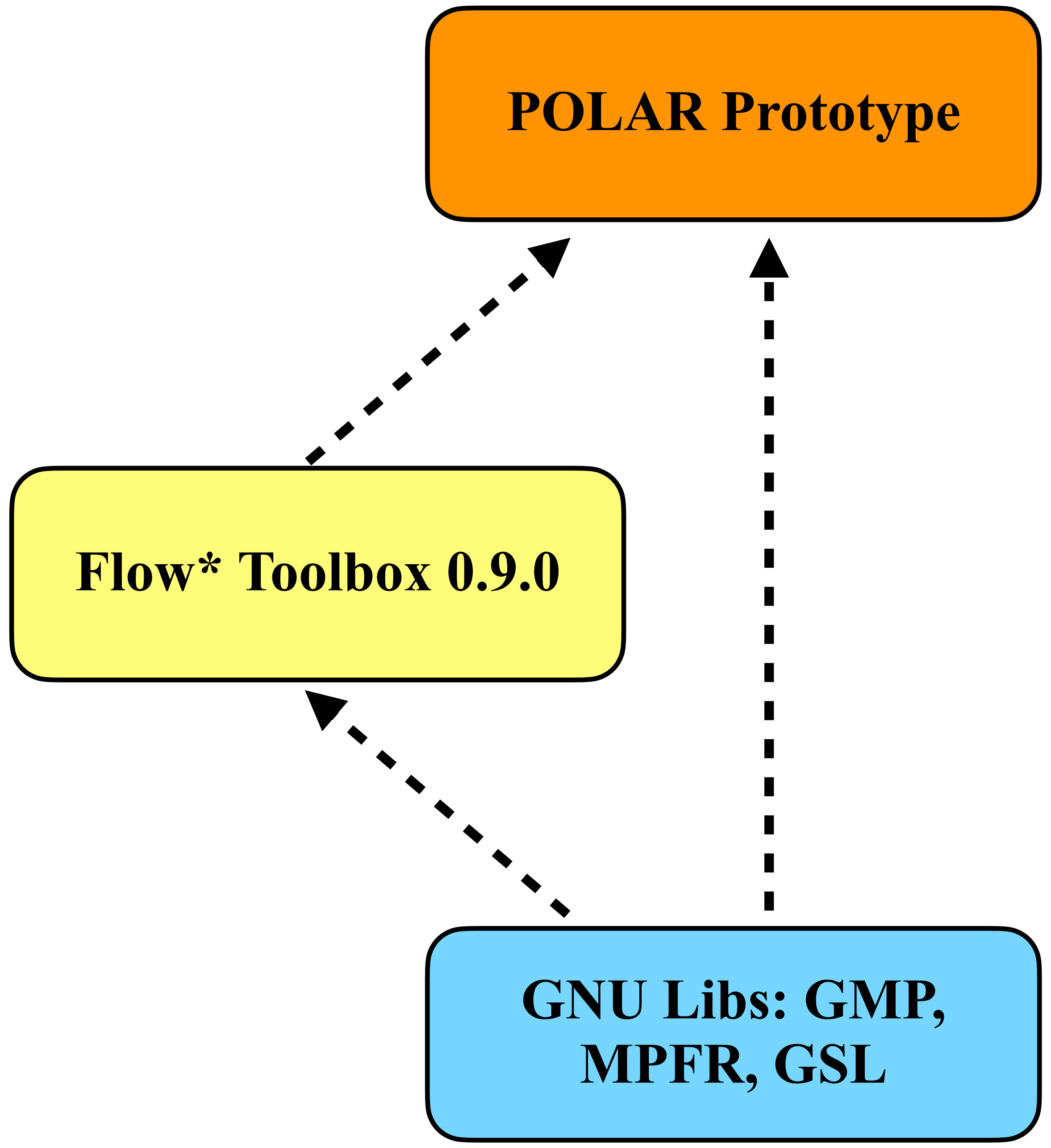
Ubuntu 18.04, MATLAB 2016a or later

Dependencies

POLAR relies on the Taylor model arithmetic library provided by Flow*. Please install Flow* with the same directory of POLAR. You can either use the following command or follow the manual of Flow* for installation.

- Install dependencies through apt-get install

```
sudo apt-get install m4 libgmp3-dev libmpfr-dev libmpfr-doc libgsl-dev gsl-bin bison flex gnuplot-x
```

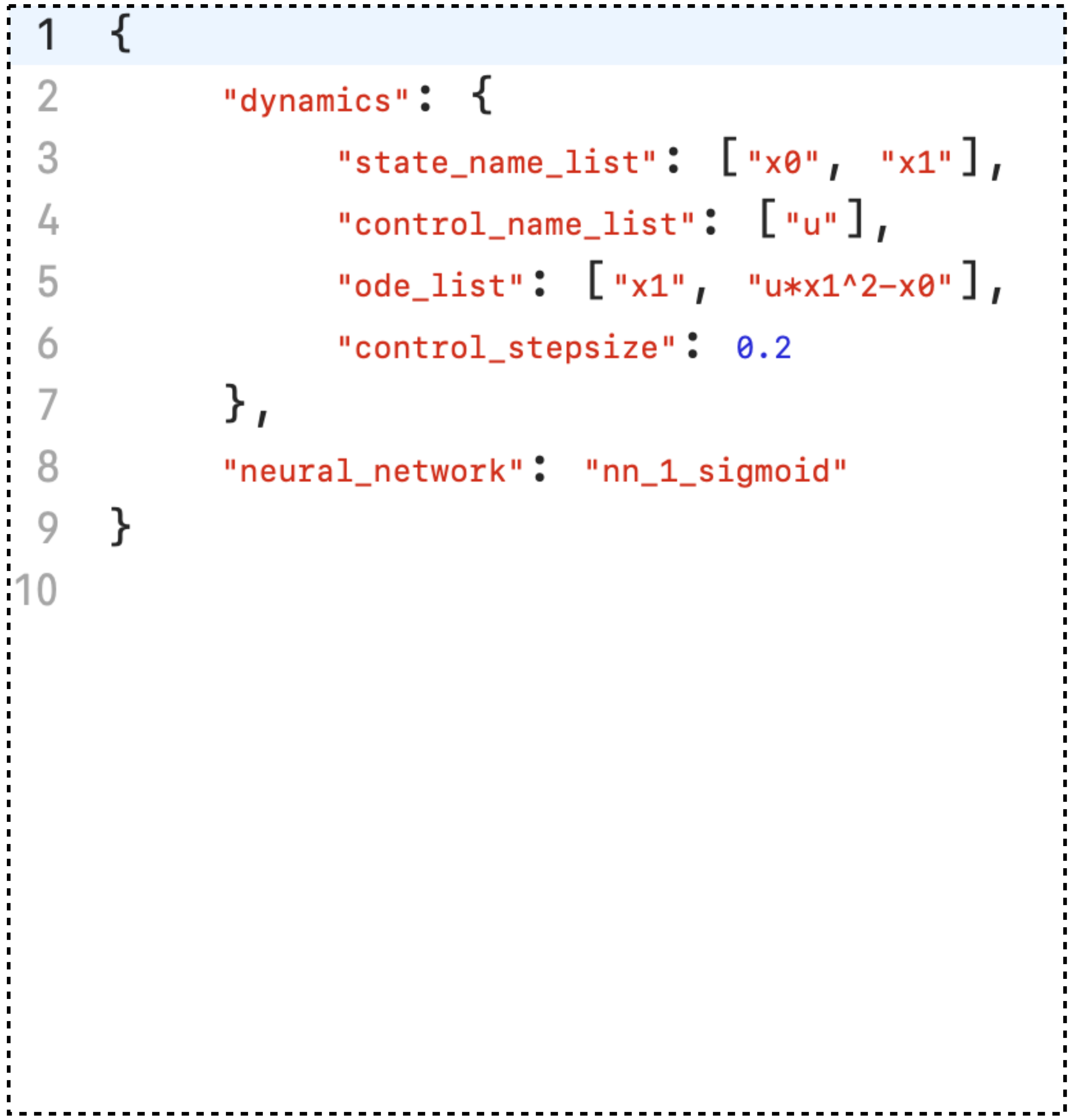


- The polynomial arithmetic framework is implemented in C++.
- Taylor models are handled by the Flow* toolbox version.
- Neural networks provided in popular formats (e.g., h5 file) are loaded by a Python program.
- Extendable API in C++ for more general CPS verification.

https://github.com/ChaoHuang2018/POLAR_Tool

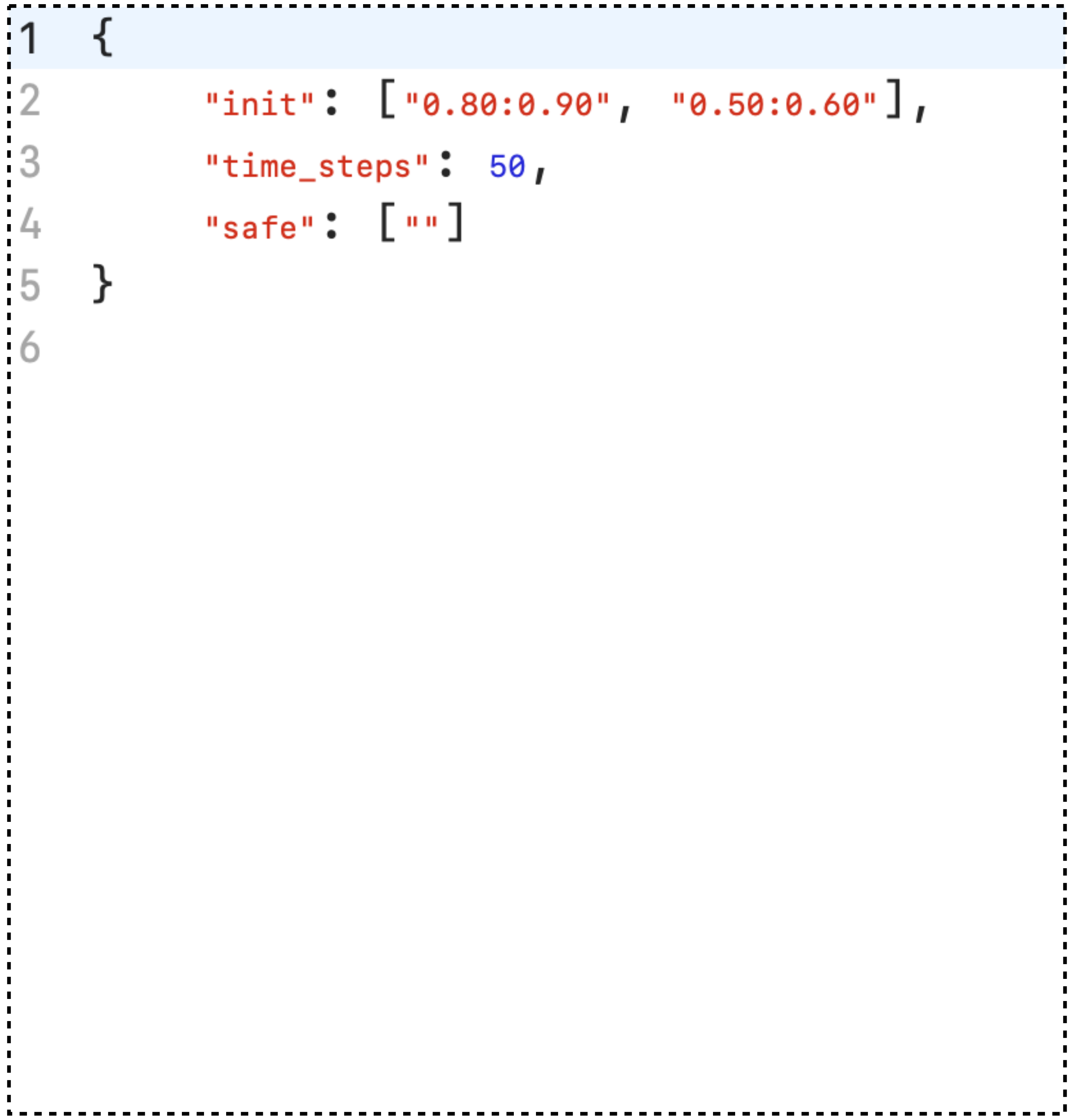
Simple Example: Benchmark 1

System



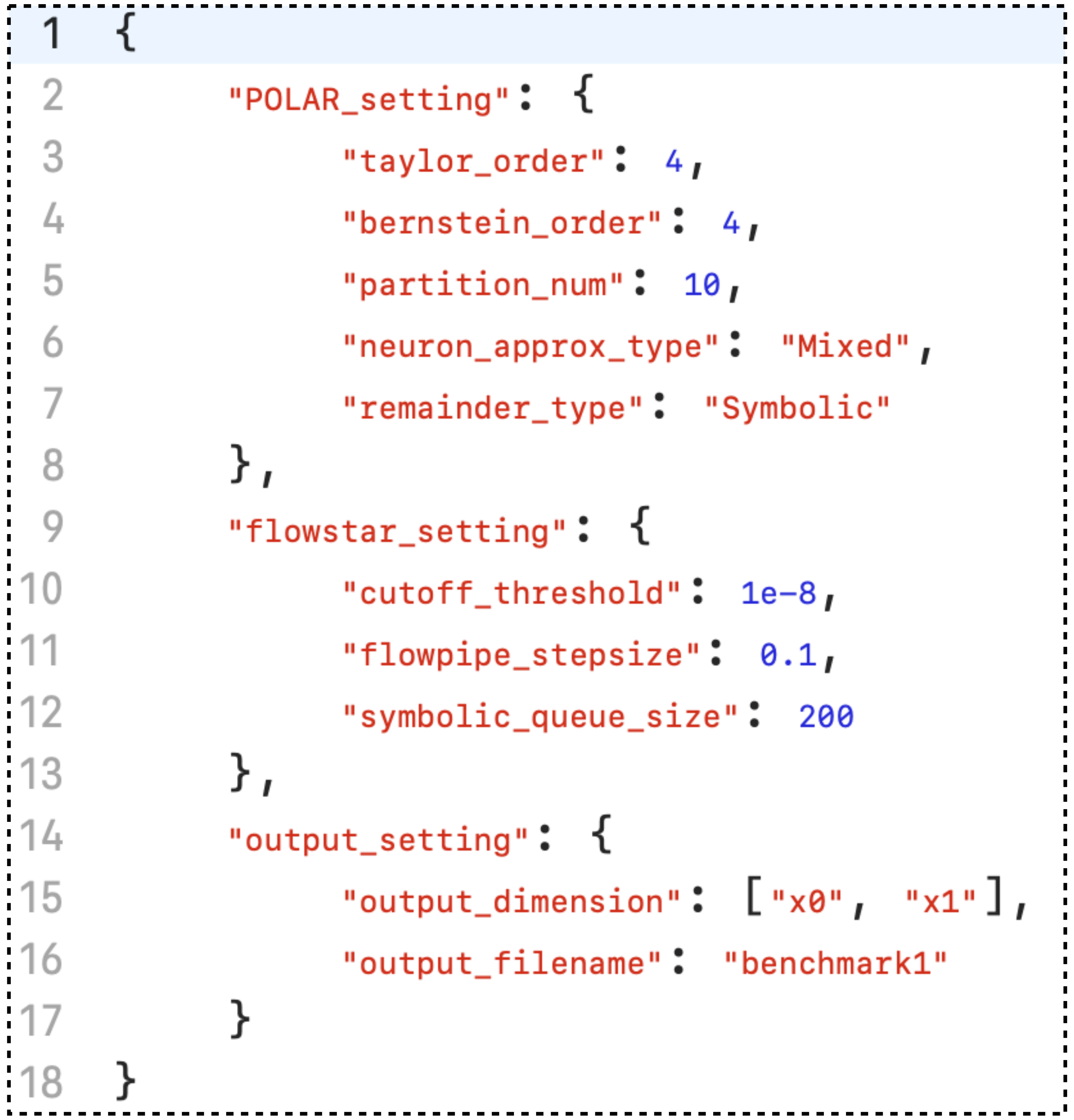
```
{
    "dynamics": {
        "state_name_list": ["x0", "x1"],
        "control_name_list": ["u"],
        "ode_list": ["x1", "u*x1^2-x0"],
        "control_stepsize": 0.2
    },
    "neural_network": "nn_1_sigmoid"
}
```

Reachability specification



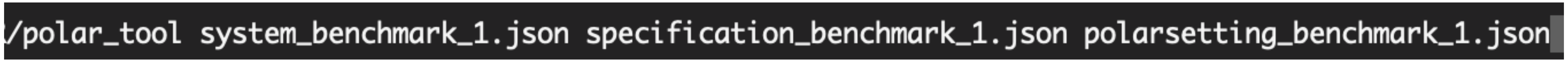
```
{
    "init": ["0.80:0.90", "0.50:0.60"],
    "time_steps": 50,
    "safe": [""]
}
```

Tool setting

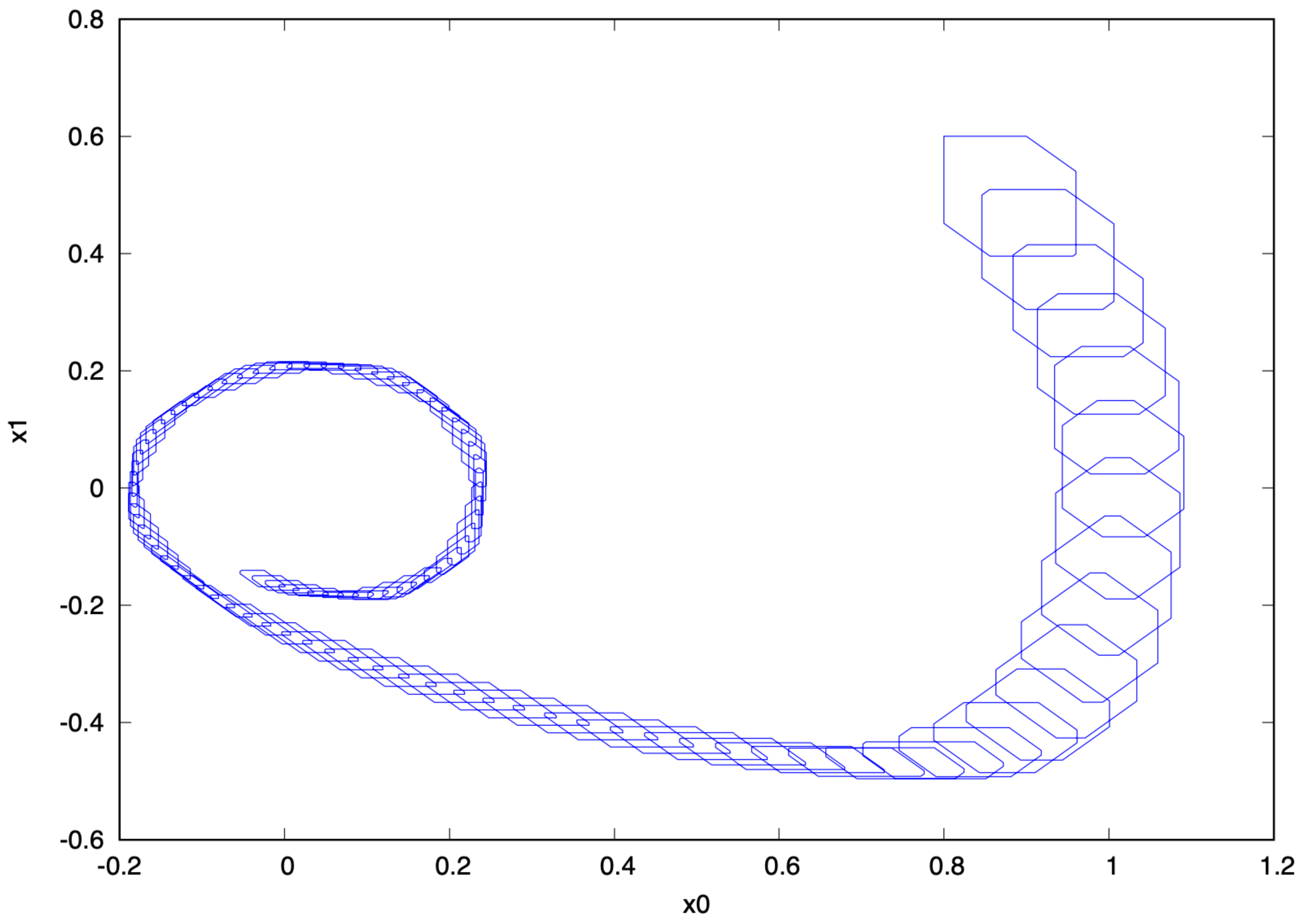


```
{
    "POLAR_setting": {
        "taylor_order": 4,
        "bernstein_order": 4,
        "partition_num": 10,
        "neuron_approx_type": "Mixed",
        "remainder_type": "Symbolic"
    },
    "flowstar_setting": {
        "cutoff_threshold": 1e-8,
        "flowpipe_stepsize": 0.1,
        "symbolic_queue_size": 200
    },
    "output_setting": {
        "output_dimension": ["x0", "x1"],
        "output_filename": "benchmark1"
    }
}
```

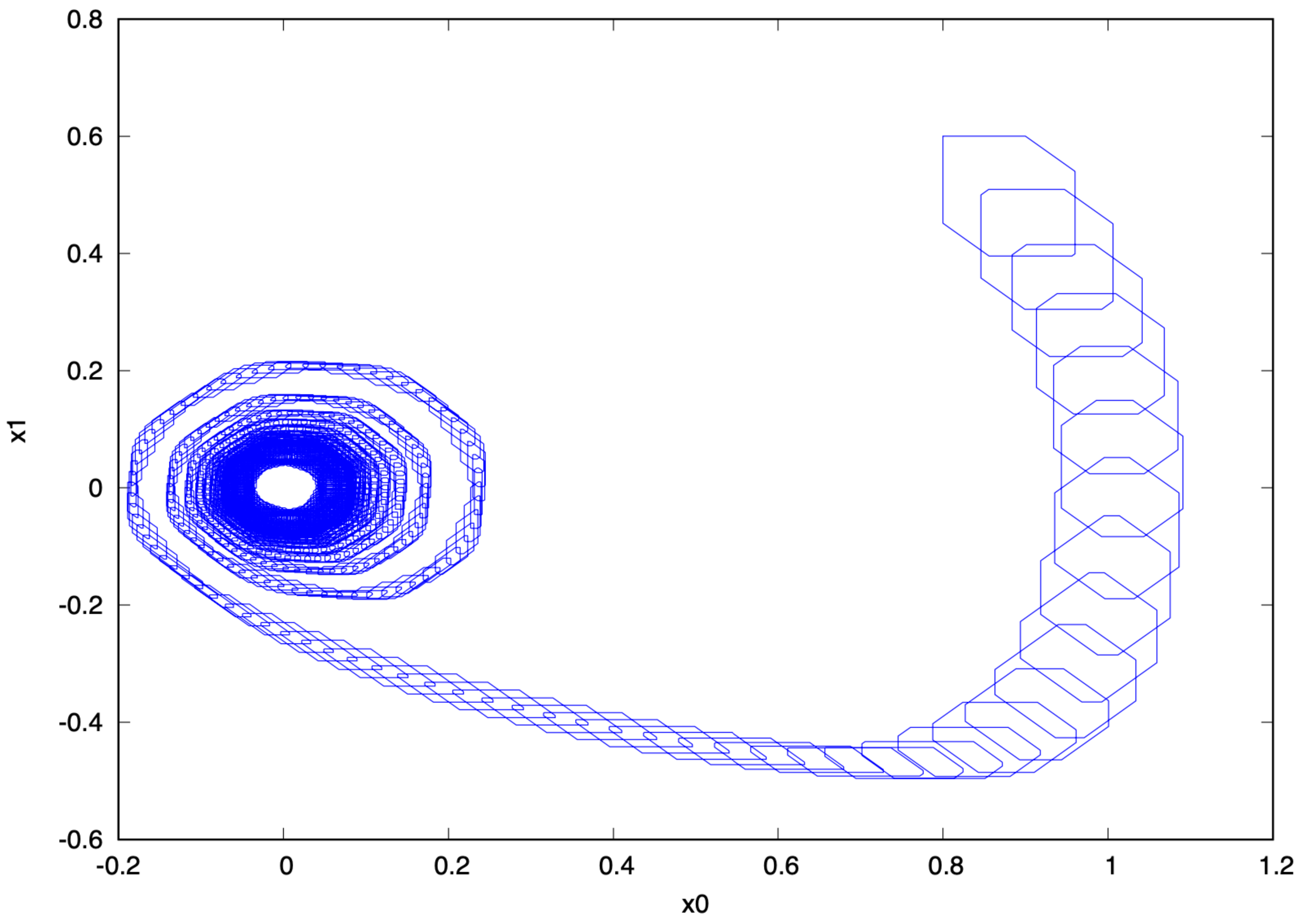
Simple Example



```
/polar_tool system_benchmark_1.json specification_benchmark_1.json polarsetting_benchmark_1.json
```



50 control steps, 2 seconds (M1 Mac)



500 control steps, 22 seconds (M1 Mac)

Conclusion

- We presented a series of approaches for computing functional overapproximations for NNCS flowmaps.
- All of the methods use Taylor model arithmetic but not limited to Taylor approximations.
- The **POLAR** framework has a time complexity which is **linear** in the size of neural networks.
- All of the methods satisfy the following property.

Theorem.
If $p(x_0, \tau) + I$ is the i-th TM computed in the j-th control step, then the actual reachable state at a time $t \in (j-1)\delta_c + (i-1)\delta + [0,\delta]$ from any $x_0 \in X_0$ at $x_0 \in X_0$ is contained in the box $p(x_0, t - (j-1)\delta_c - (i-1)\delta) + I$.

Further Observations

- It is hard to measure the **error size** in a pure range overapproximation. However, the remainder size of a functional overapproximation directly tells the approximation quality.

- It is often not hard to compute an accurate polynomial approximation, however evaluating a **guaranteed error** for it is not easy in general.

- Although all of our methods only consider deterministic behavior, they can be immediately extended to handle **sensing and actuation noises**. Also, the QMPC benchmark uses a look-up table to find control inputs.

- The methods may be extended to handle **all continuous operations** in learning-enabled CPS.

Future Directions

- Relational abstraction for machine learning components, forward and backward analysis.
- Falsification using functional overapproximations.
- Vulnerability or robustness checking for safe learning-enabled CPS.
- Tool improvement: using GPUs to perform numerical computation.

Thank You

for attending the course!