ECE/CS 584: Verification of Embedded and Cyber-physical Systems

Lecture 11: Neural Network Verification:
falsification, training verifiable NNs, and practical verifiers

Prof. Huan Zhang

huan@huan-zhang.com

Deadlines

Project proposal due 3/3

Discuss your class projects on Canvas, try to find a teammate with common interests.

- To be effective, *don't* just post your name and say you are looking for a teammate
- Be sure to introduce your technical strengths, research interests, and your thoughts about the project to find people with similar interests
- Do it as soon as possible! Discuss with your teammates to finalize project ideas

Deadlines

Project proposal due 3/3

Up to 4 pages

- Introduction of the problem or system under study and why it is important
- Give clear mathematical description of your problem
- Related work (What has been done before? do a thorough literature review!)
- Proposed methodology (what is your planned technique to solve this problem? What are the risks?)
- Timeline and targets (what goals do you aim to achieve?)

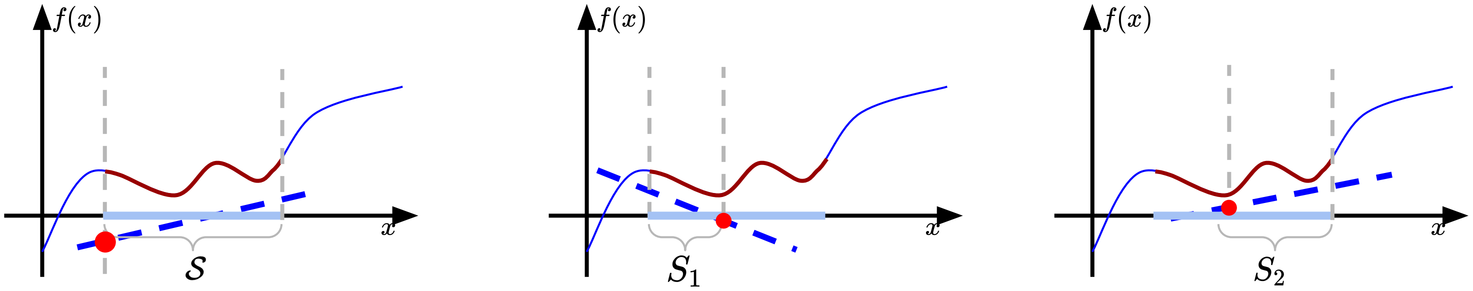
Deadlines

Homework 2 will be released on Wednesday (Feb 21), due 3/10

Include two programming assignments

Contact the TA, **Sanil Chawla** <schawla7@illinois.edu> for technical assistance

Review: bound propagation & branch and bound

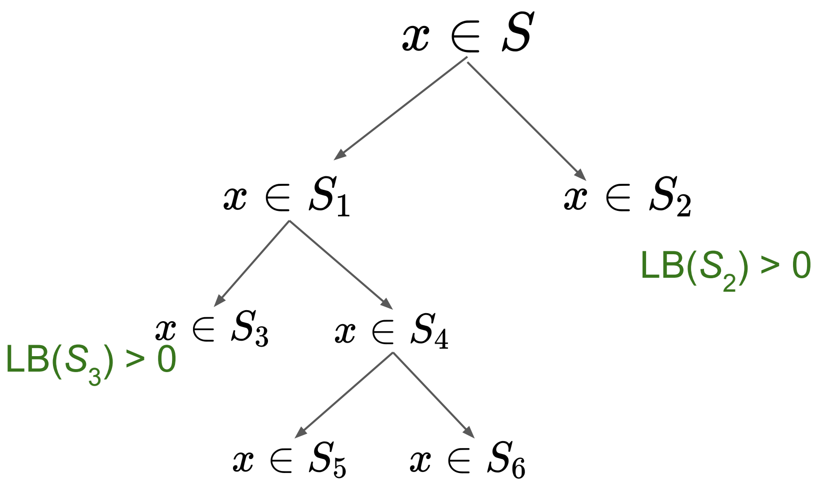


Goal: improve the loose lower bound

Review: Branch and bound

If LB(S_i) > 0, if can be removed from our problem since the property is verified on this subdomain S_i; branch and bound is needed for unverified subdomains only.

$$y^* = \min_{x \in \mathcal{S}} f(x)$$



List of unverified subproblems

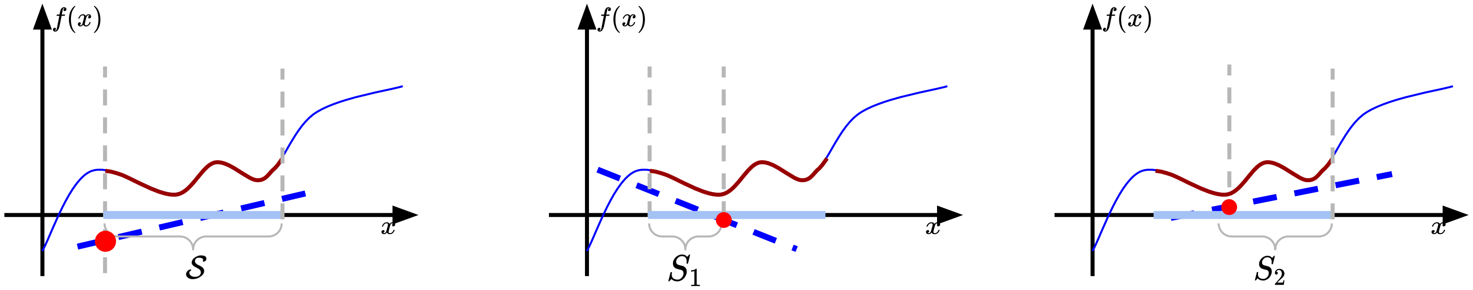
{S}

{S_1 , $\cancel{S_2}$}

{$\cancel{S_3}$, S_4}

{S_5 , S_6}

Review: Branch and bound on input



Split each into domain S, typically by

$S = \{x_1 \in [-1, 1], x_2 \in [-1, 1]\}$ =>

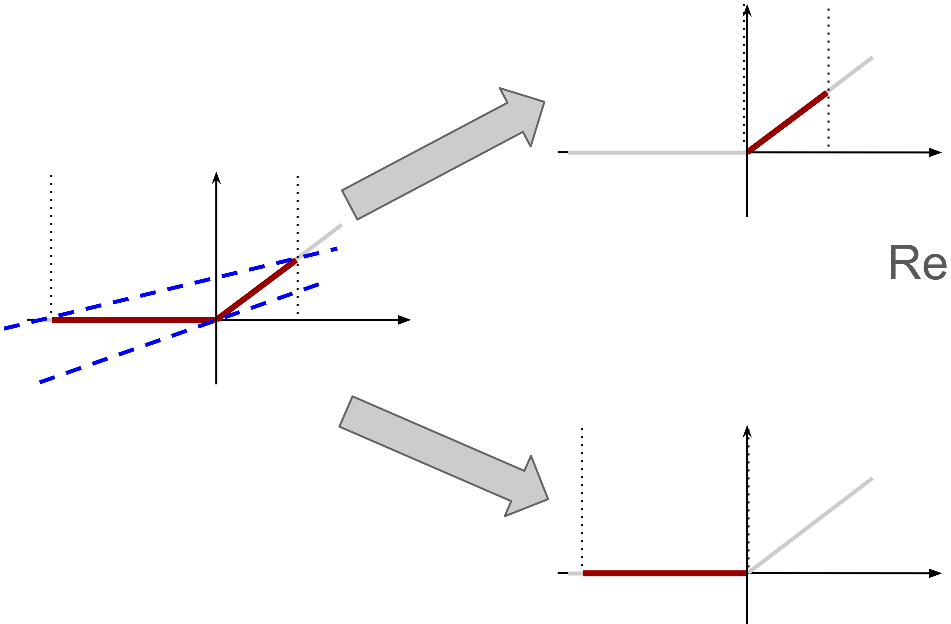
$S_1 = \{x_1 \in [-1, 0], x_2 \in [-1, 1]\}$, $S_2 = \{x_2 \in [0, 1], x_2 \in [-1, 1]\}$

Implementation is easy

Does not work well when input dimension is very high (e.g., image inputs)

Review: Branch and bound on ReLU

Implicitly split input domain S by considering a ReLU neuron in two cases: active and inactive.



ReLU becomes linear in both subproblems with split constraint (handled using **β-CROWN**)

What we have learned so far about NN verification

- Verification as optimization problems
- Mixed Integer programming formulation for verifying ReLU networks
- Linear programming formulation
- Interval bound propagation (IBP)
- Linear bound propagation algorithm (CROWN)
- Bound optimization to improve tightness (α-CROWN)
- Branch and bound to further improve tightness (β-CROWN)

What are missing to solve practical NN verification problems?

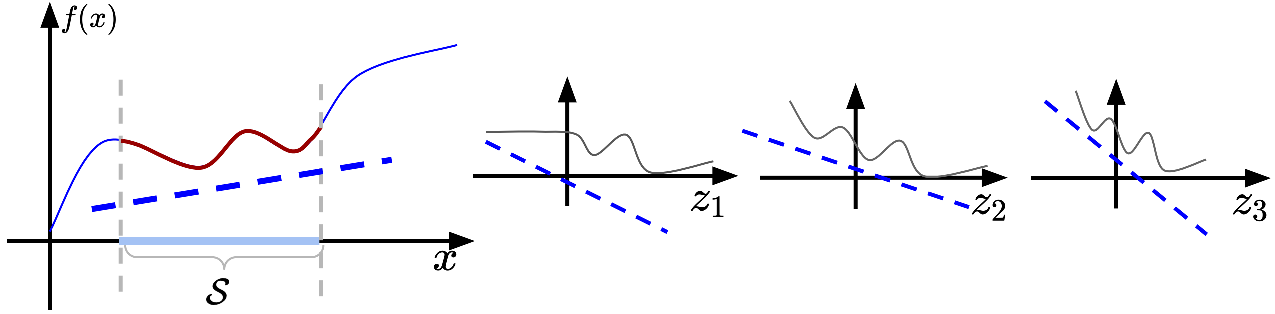
Several topics we will discuss today

Bound propagation on general computation graph

Falsification methods

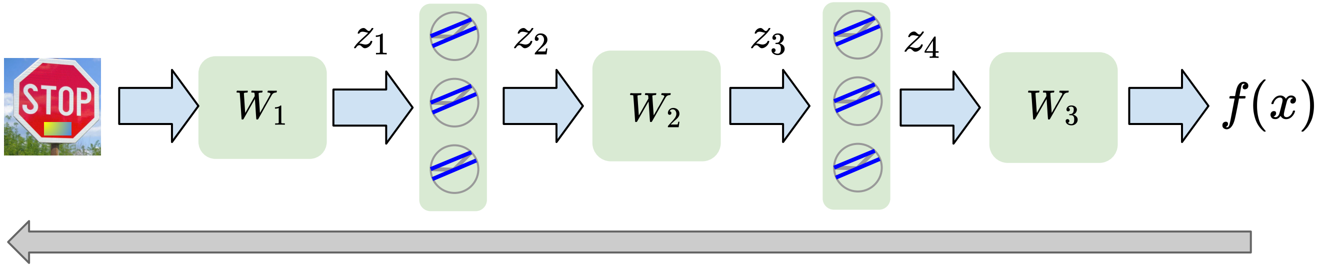
Adversarial training and verification-friendly networks

Bound propagation for feedforward neural networks



Initial (in)equality:

$$f(x) = W_3 z_4$$

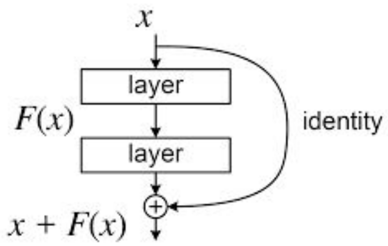


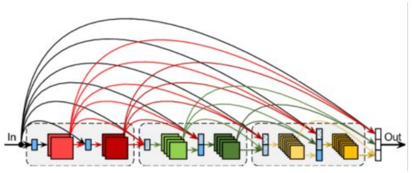
Always keep valid lower bounds

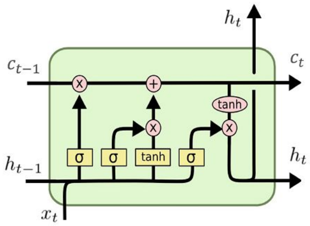
CROWN main theorem (simplified): $f(x) \geq a_{\text{CROWN}}^{\top} x + b_{\text{CROWN}} \quad \forall x \in \mathcal{S}$

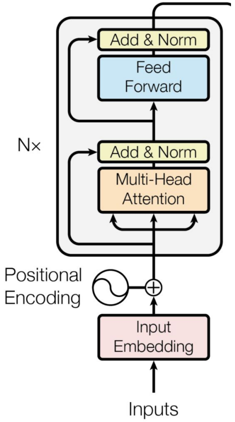
How about more complex networks?

Most modern neural networks have more than the “linear” feedforward structure



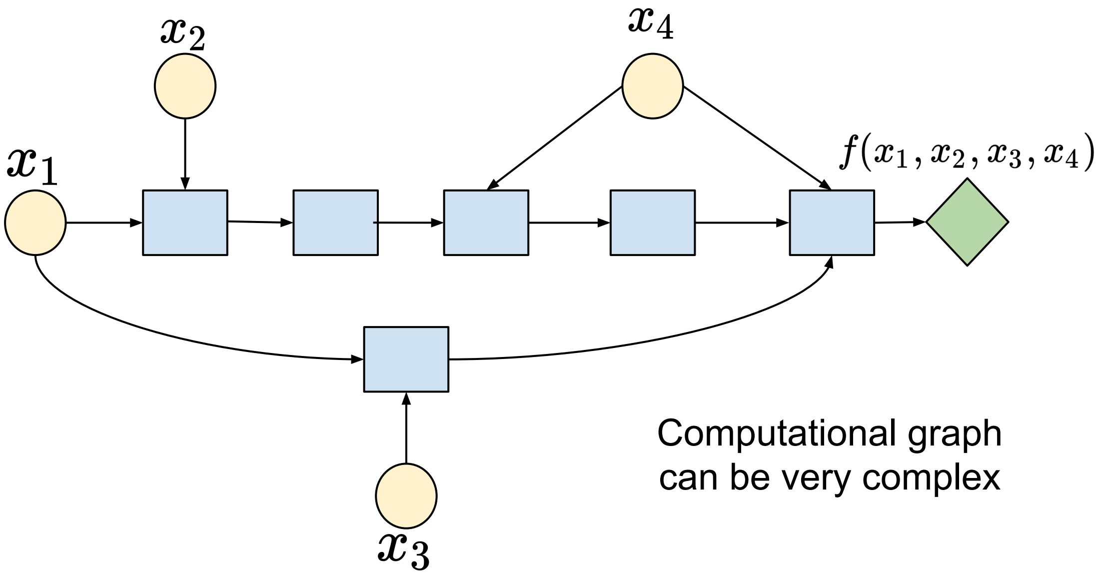






Verification on general computation graphs

The idea of bound propagation can be generalized to general computation graphs, as a graph algorithm

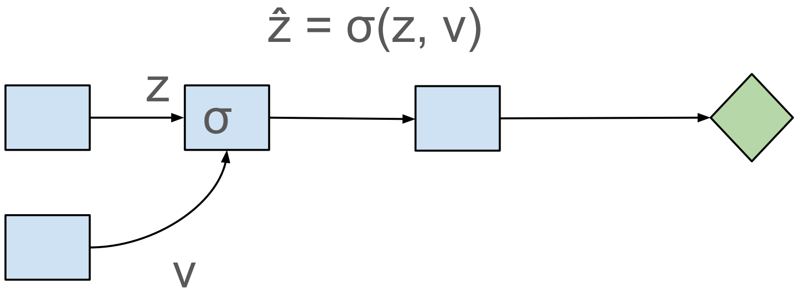


Computational graph can be very complex

Requirement:
Each computation  can be **bounded using linear hyperplanes** w.r.t. its inputs

Bound propagation on computation graphs

One compute node can have multiple inputs



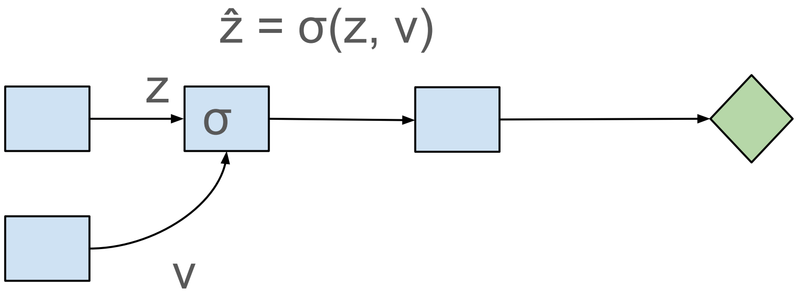
For example:

$\hat{z} = z + v$

$\hat{z} = z \times v$

Bound propagation on computation graphs

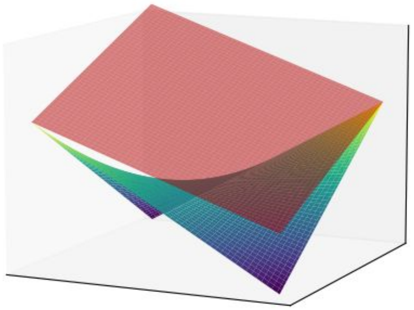
First step: bound σ using linear bounds of z and v



$$\underline{a}^\top z + \underline{b}^\top v + \underline{c} \leq \sigma(z, v) \leq \overline{a}^\top z + \overline{b}^\top v + \overline{c}$$

For ẑ = z + v, the function is already linear in z and v

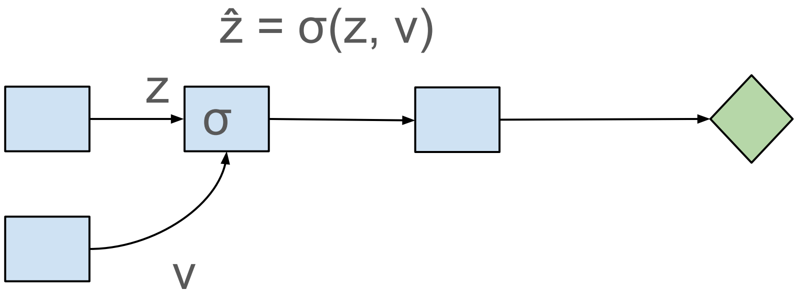
For ẑ = z × v, need intermediate layer bounds for z and v



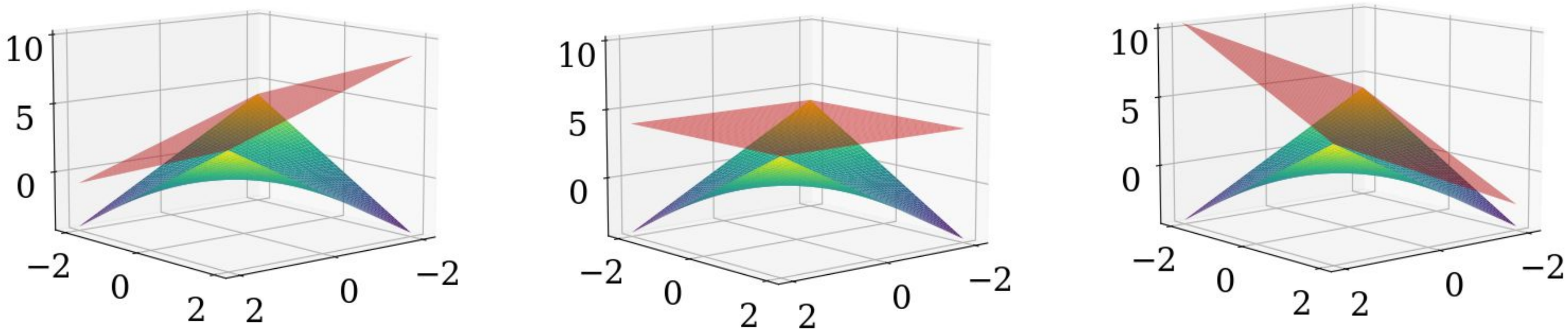
Bound propagation on computation graphs

For $\hat{z} = z \times v$, need intermediate layer bounds for z and v

$$\underline{a}^\top z + \underline{b}^\top v + \underline{c} \leq \sigma(z, v) \leq \overline{a}^\top z + \overline{b}^\top v + \overline{c}$$



These bounds can be optimized as well!



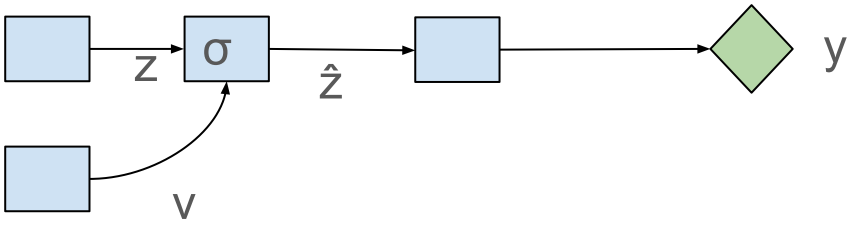
Bound propagation on computation graphs

$$\underline{a}^\top z + \underline{b}^\top v + \underline{c} \leq \sigma(z, v) \leq \overline{a}^\top z + \overline{b}^\top v + \overline{c}$$

Choose lower or upper based on the sign of $a_{\hat{z}}$

$$y \leq a_z^\top z + a_v^\top v + b'$$

$$y \leq a_{\hat{z}}^\top \hat{z} + b$$



Bound propagation on computation graphs

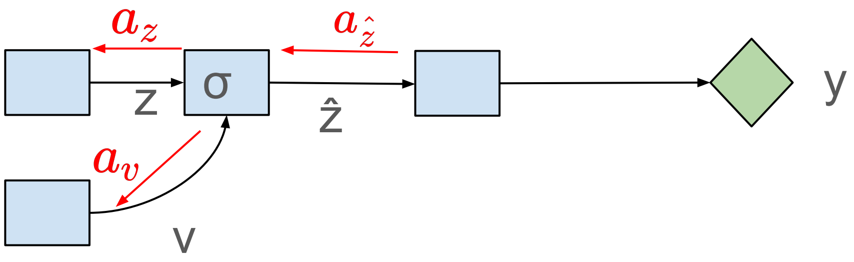
$$\underline{a}^\top z + \underline{b}^\top v + \underline{c} \le \sigma(z, v) \le \overline{a}^\top z + \overline{b}^\top v + \overline{c}$$

Choose lower or upper based on the sign of $a_{\hat{z}}$

$$y \le a_z^\top z + a_v^\top v + b'$$

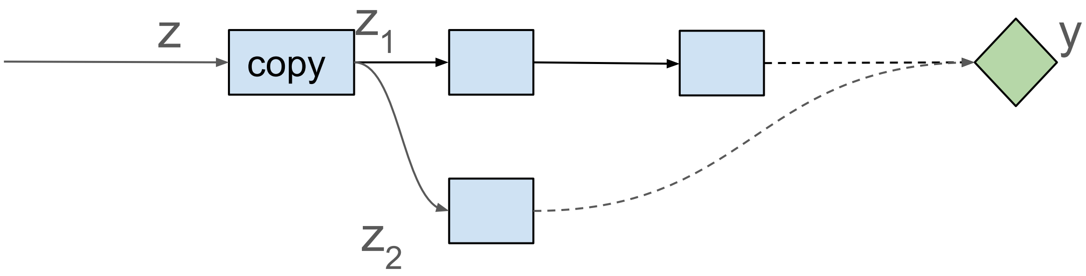
$$y \le a_{\hat{z}}^\top \hat{z} + b$$

We essentially propagate those coefficients on the graph



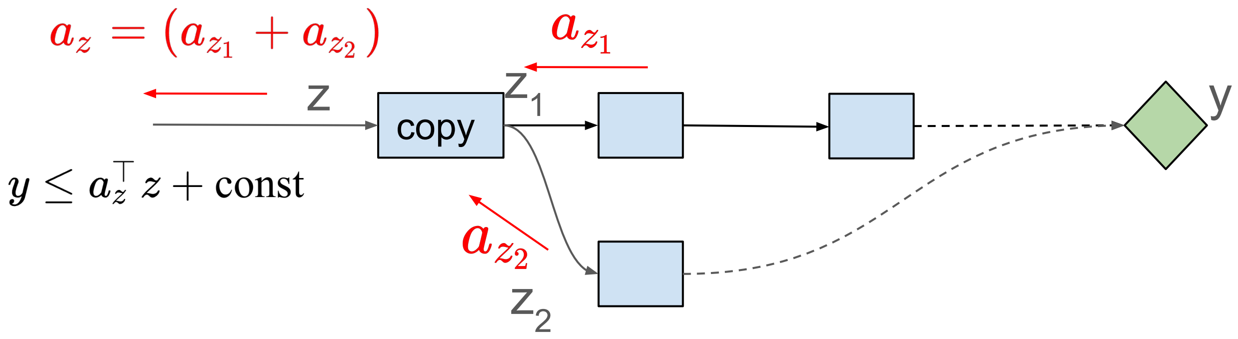
Bound propagation on computation graphs

One compute node's output can be used by multiple nodes



Bound propagation on computation graphs

One compute node's output can be used by multiple nodes

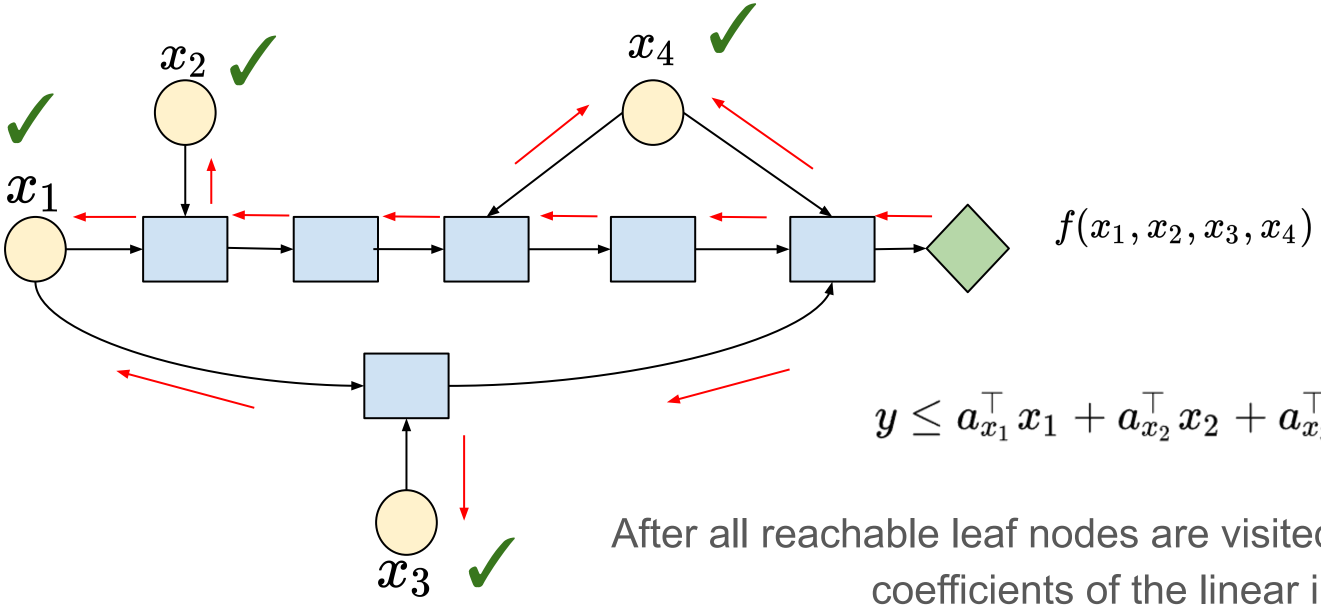


$$y \leq a_{z_1}^\top z_1 + a_{z_2}^\top z_2 + \text{const}$$

Must wait until both coefficients become available

Bound propagation on computation graphs

When to stop? Reaching a leaf node.

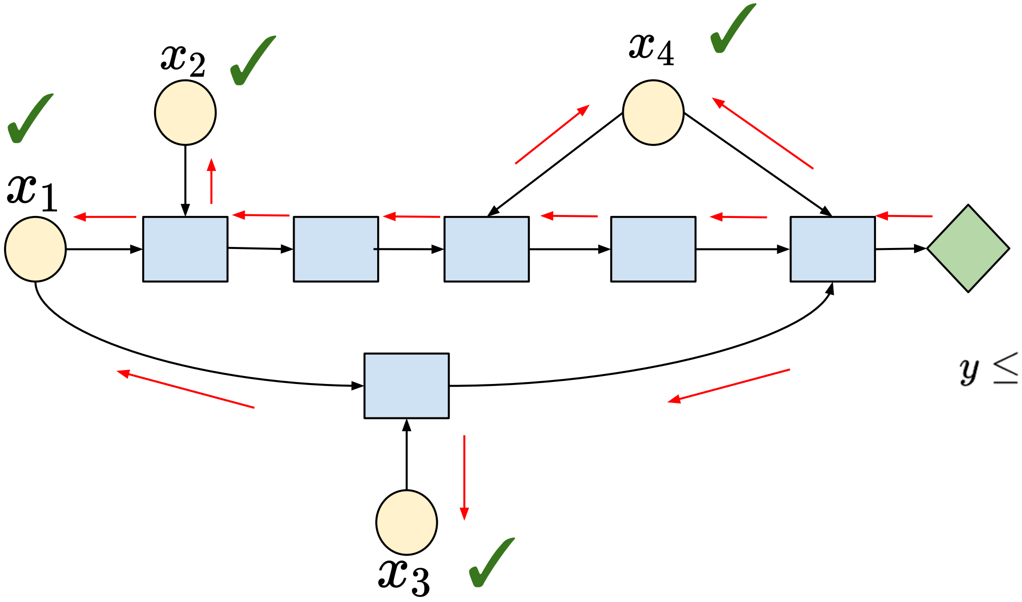


$$y \leq a_{x_1}^\top x_1 + a_{x_2}^\top x_2 + a_{x_3}^\top x_3 + a_{x_3}^\top x_3 + \text{const}$$

After all reachable leaf nodes are visited, we obtained all linear coefficients of the linear inequality

Verification Beyond Neural Networks

Actually, CROWN can work on general computation graphs, not limited to neural networks!
Using CROWN for some novel applications that requires bounding is a great project idea!



$f(x_1, x_2, x_3, x_4)$

$$y \le a_{x_1}^\top x_1 + a_{x_2}^\top x_2 + a_{x_3}^\top x_3 + a_{x_3}^\top x_3 + \text{const}$$

Falsification methods

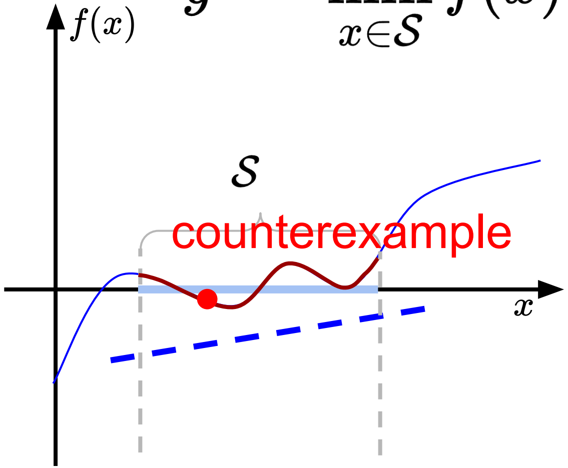
Lower bounds are good for verification, but are not very helpful for finding a counterexample (feasible solution) so far. Back to our original problem:

$$\exists\ x \in S \ \wedge\ y \leq 0 \ \wedge\ y = f(x)$$

In some case, we want to find some x such that **y ≤ 0**

- Use these counterexamples to fix model bugs
  Similar to failed test cases in software engineering
- For large models, lower bound can be loose.
  Verification is challenging and falsification has more hope
- Also called "adversarial attacks" in ML literature
  Counterexample == adversarial example

$$y^* = \min_{x \in \mathcal{S}} f(x)$$

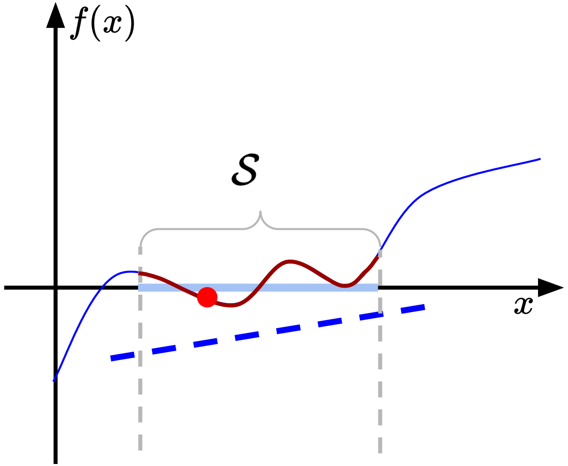


Falsification methods

To find a counterexample, we minimize the objective function using any method

- Randomly sample some x ∈ S and check f(x)?
- **Gradient-based method**

$$y^* = \min_{x \in \mathcal{S}} f(x)$$

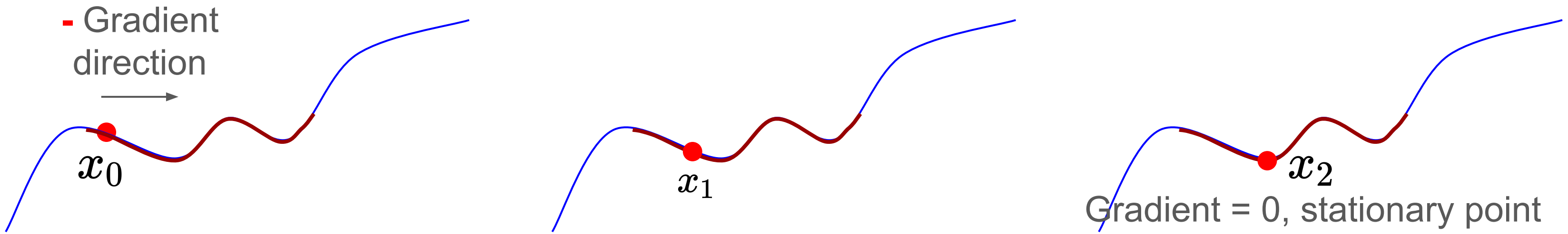


Projected gradient descent

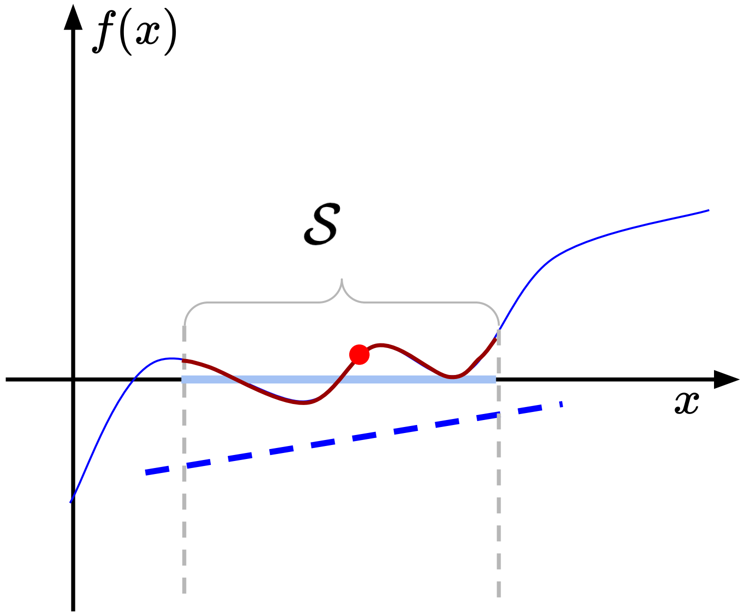
$$x_{t+1} \leftarrow \mathrm{Proj}_{\mathcal{S}}(x_t - \eta \nabla f(x_t))$$

Follow the “downhill” to decrease f(x)

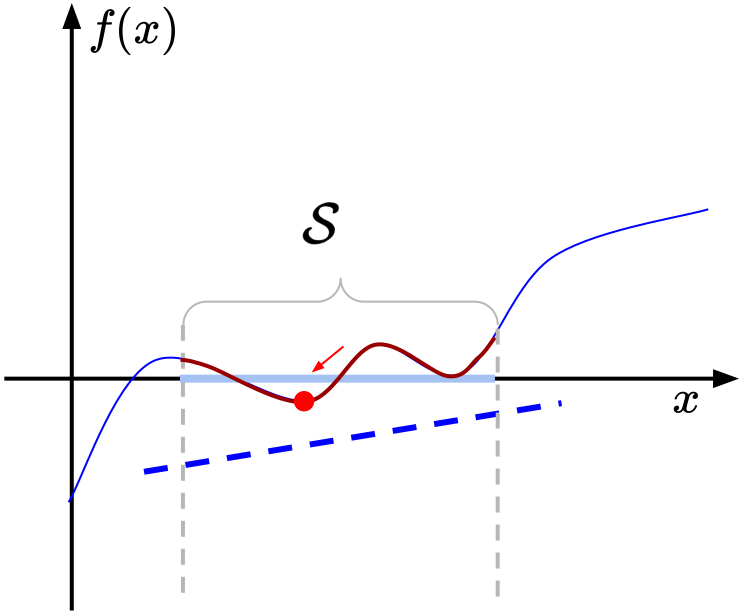
In the meanwhile, do not go outside of the set S



Projected gradient descent

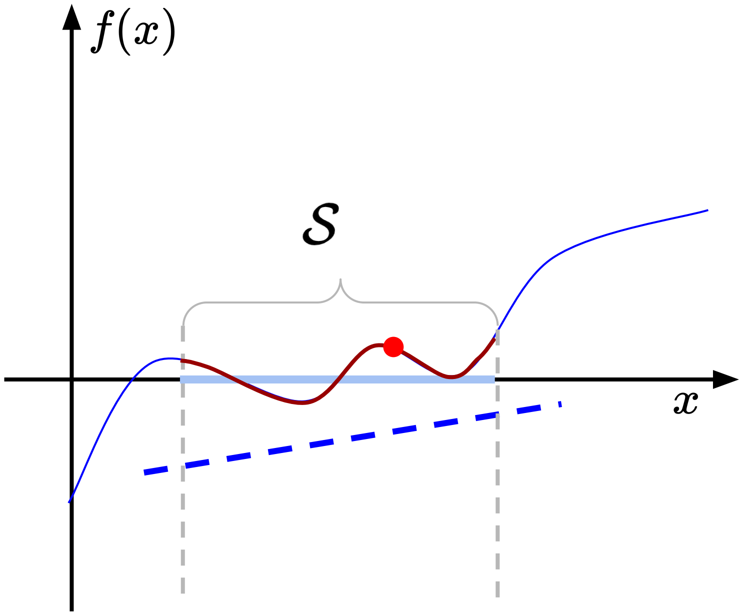


After a few iterations

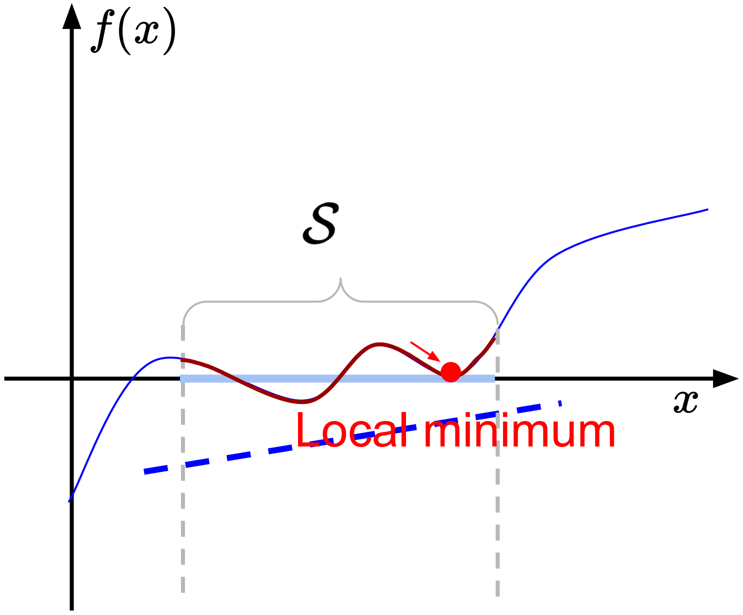


f(x) < 0, counterexample found!

Projected gradient descent may fail

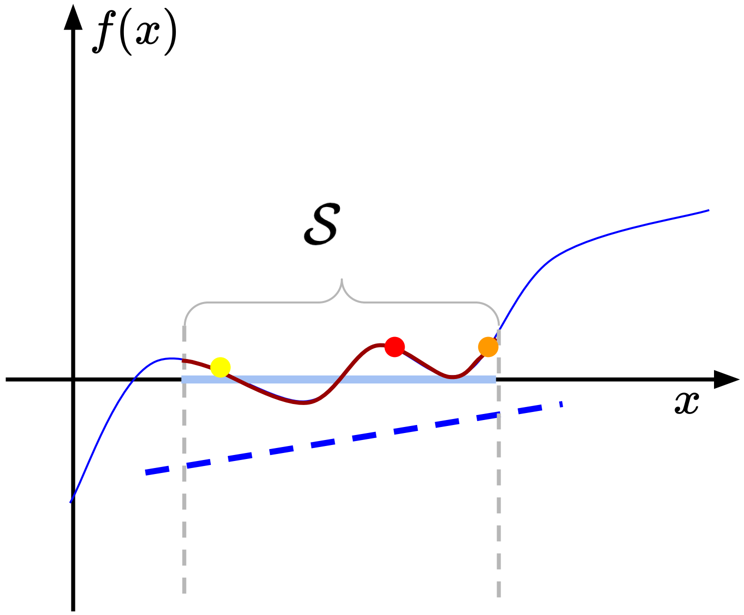


After a few iterations

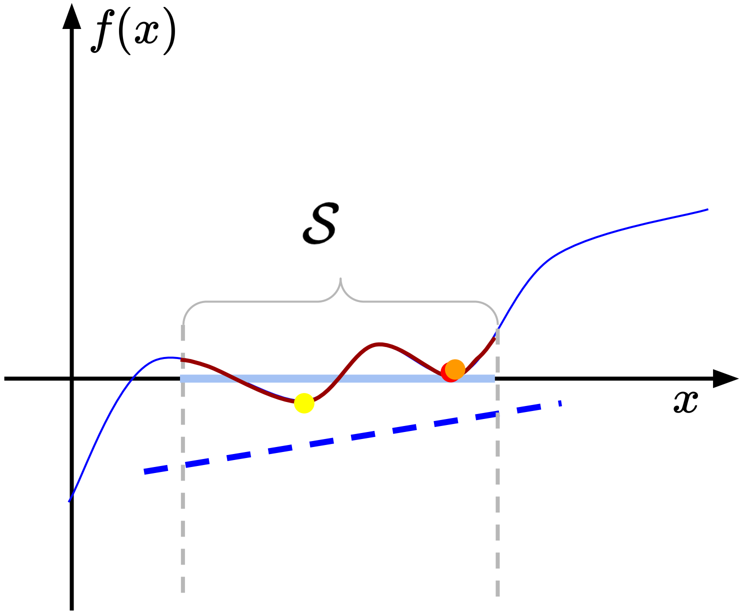


Get stuck! No further improvements possible

Projected gradient descent with random restarts



After a few iterations



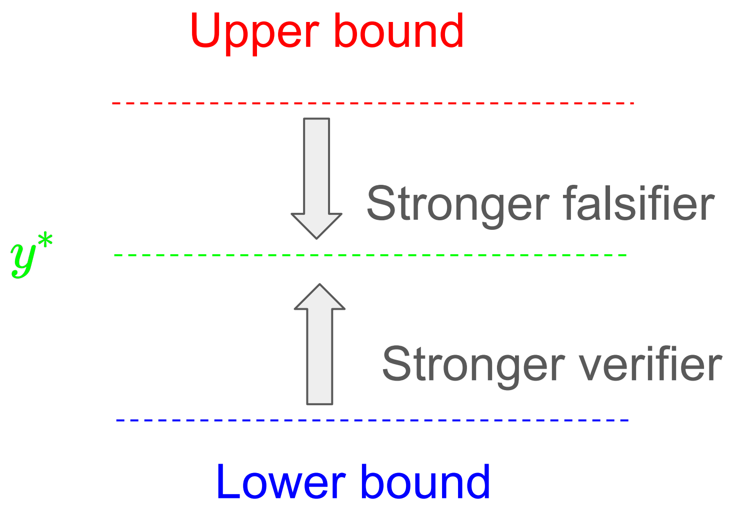
Try random starting points to increase chance

Falsification gives an upper bound of the verification problem

$$y^* = \min_{x \in \mathcal{S}} f(x)$$

Verification gives a lower bound of y*:
Lower bound ≥ 0 => verified

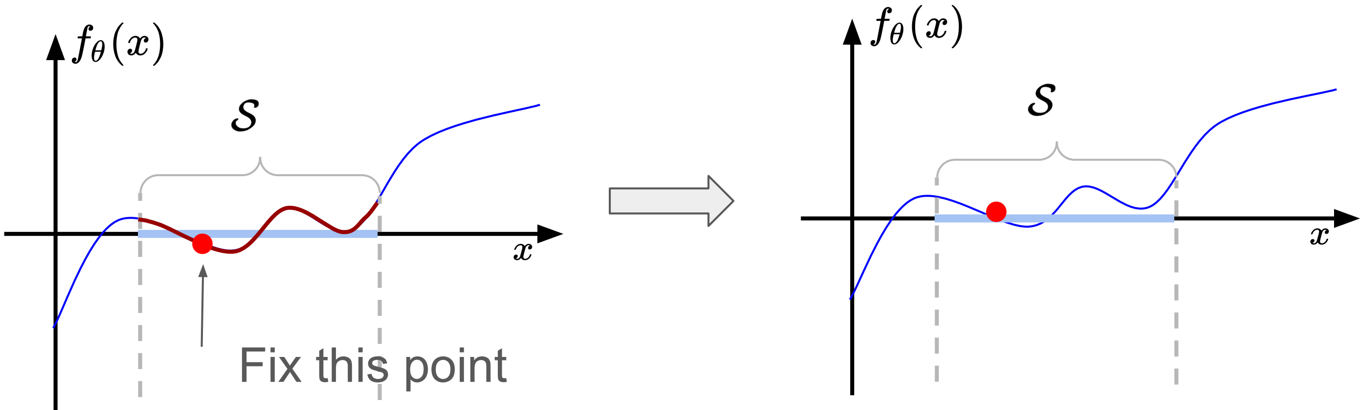
Falsification gives an upper bound of y*:
Upper bound ≤ 0 => falsified



Adversarial training

While not converged:

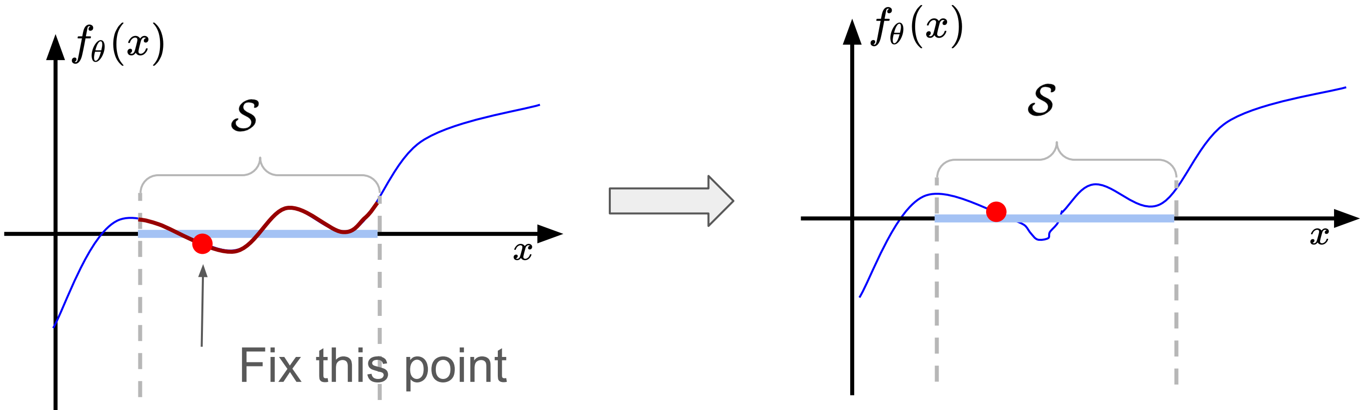
> $x \leftarrow$ sample training data
>
> $S \leftarrow$ a small neighborhood around x
>
> $x_{adv} \leftarrow$ counterexample: $x_{adv} \in S \wedge f_\theta(x_{adv}) \leq 0$
>
> Update model parameter θ using gradient ascent to maximize $f_\theta(x_{adv})$



Adversarial training

Pros: (relatively) efficient to find counterexamples, less impact on model performance

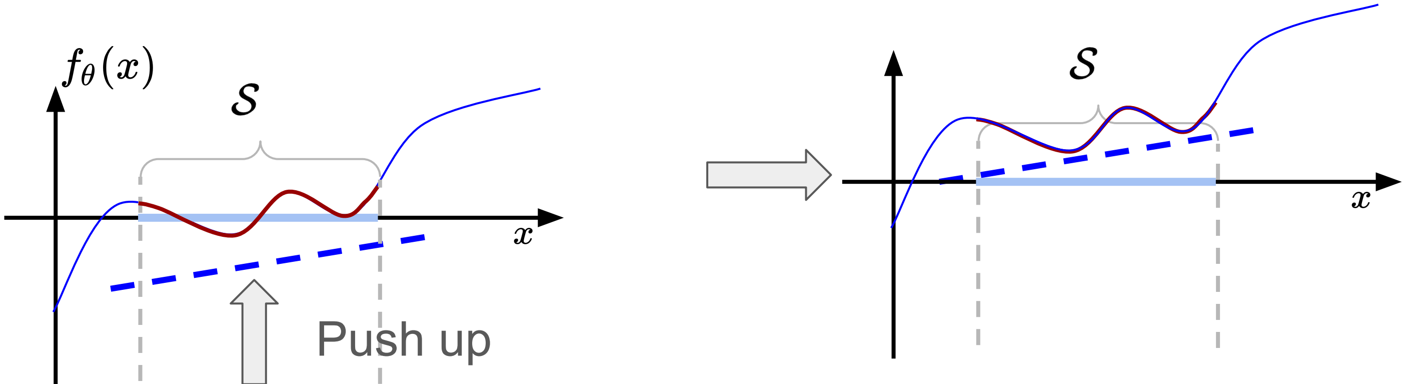
Cons: just fixing a finite number of counterexamples may not lead to a truly verified model; usually challenging for verifiers



Verification-guided training

While not converged:

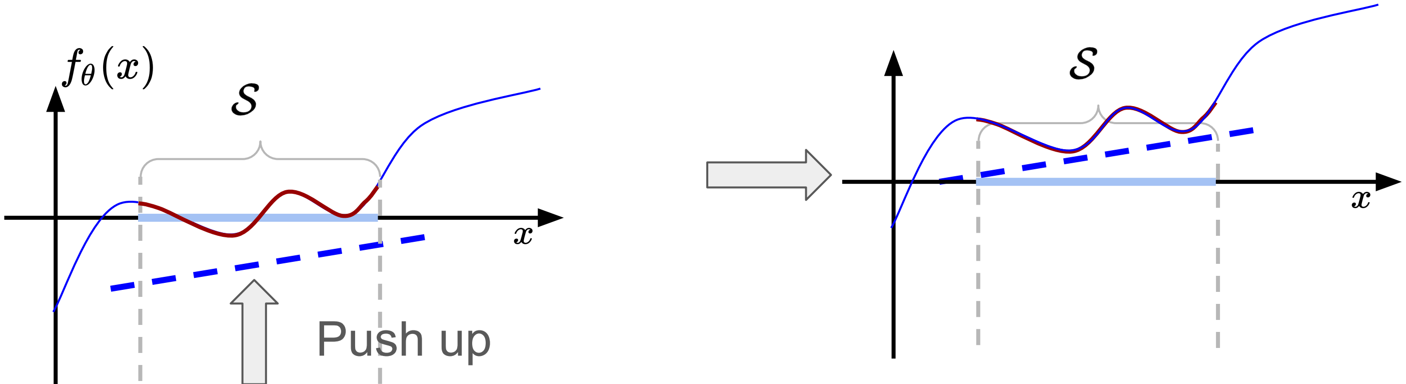
> $x \leftarrow$ sample training data
>
> $S \leftarrow$ a small neighborhood around x
>
> $LB_\theta(S) \leftarrow$ lower bound of $f_\theta(x)$ in S
>
> Update model parameter θ using gradient ascent to maximize $LB_\theta(S)$



Verification-guided training

Pros: Maximizing the lower bound guarantees f(x) > 0; models trained in this way are typically very easy to verify (verification friendly)

Cons: Calculating lower bounds is expensive; lower bounds can be too conservative, leading to poor model performance



Adversarial training vs Verification-guided training

CIFAR10: pixel-wise perturbation 8/255

Adversarial training: 90% clean accuracy, 70% accuracy under adversarial attack, ~0% verified accuracy

Verification-guided training: 55% clean accuracy, ~40% accuracy under adversarial attack, ~35% verified accuracy