Provably Secure Machine Learning

Jacob Steinhardt

ARO Adversarial Machine Learning Workshop
September 14, 2017

Why Prove Things?

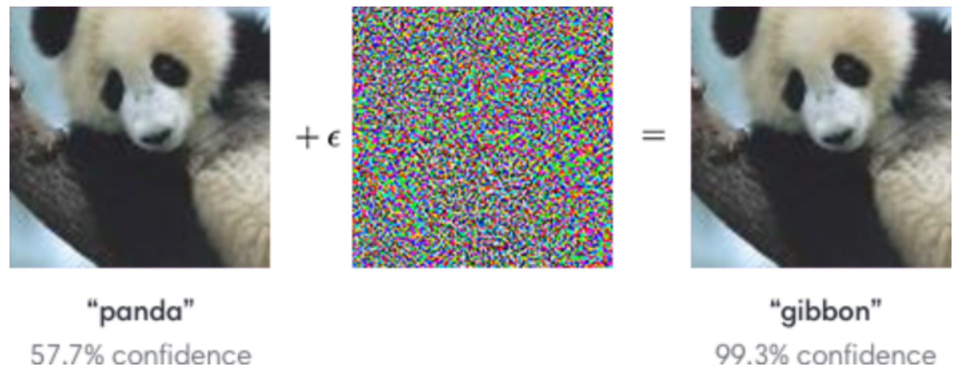
Attackers often have more motivation/resources than defenders

Heuristic defenses: arms race between attack and defense

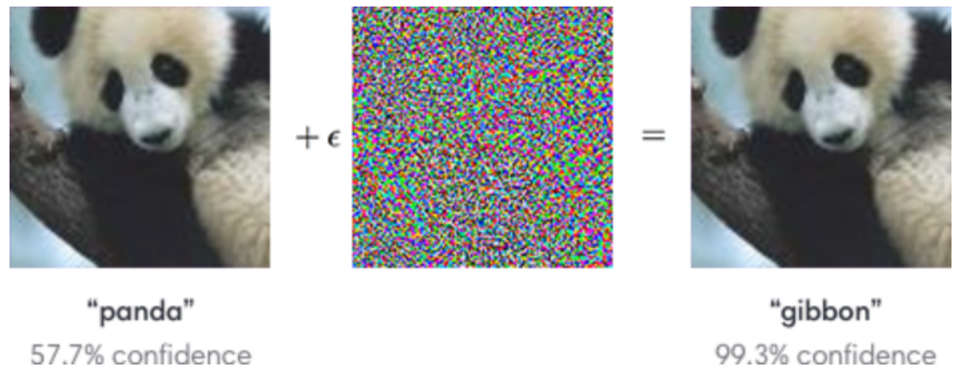
Proofs break the arms race, provide absolute security

- for a given threat model...

Example: Adversarial Test Images



Example: Adversarial Test Images



[Szegedy et al., 2014]: first discovers adversarial examples

[Goodfellow, Shlens, Szegedy, 2015]: Fast Gradient Sign Method (FGSM) + adversarial training

[Papernot et al., 2015]: defensive distillation

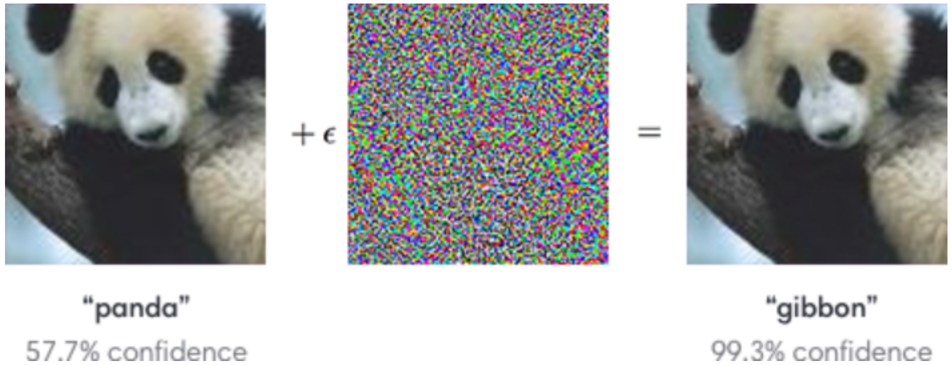
[Carlini and Wagner, 2016]: distillation is not secure

[Papernot et al., 2017]: FGSM + distillation only make attacks harder to find

[Carlini and Wagner, 2017]: all detection strategies fail

[Madry et al., 2017]: a secure network, finally??

Example: Adversarial Test Images



[Szegedy et al., 2014]: first discovers adversarial examples

[Goodfellow, Shlens, Szegedy, 2015]: Fast Gradient Sign Method (FGSM) + adversarial training

[Papernot et al., 2015]: defensive distillation

[Carlini and Wagner, 2016]: distillation is not secure

[Papernot et al., 2017]: FGSM + distillation only make attacks harder to find

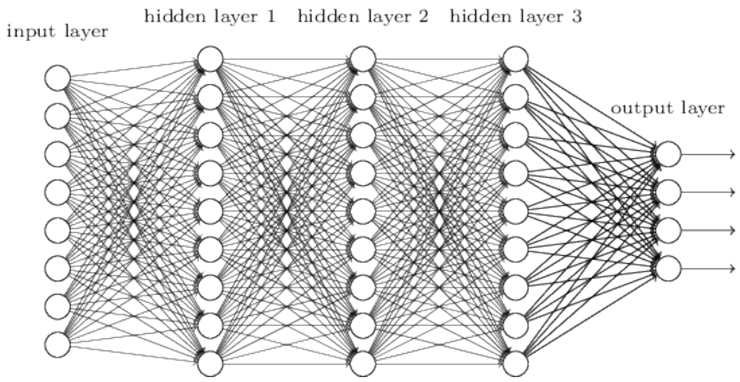
[Carlini and Wagner, 2017]: all detection strategies fail

[Madry et al., 2017]: a secure network, finally??

1 proof = 3 years of research

Formal Verification is Hard

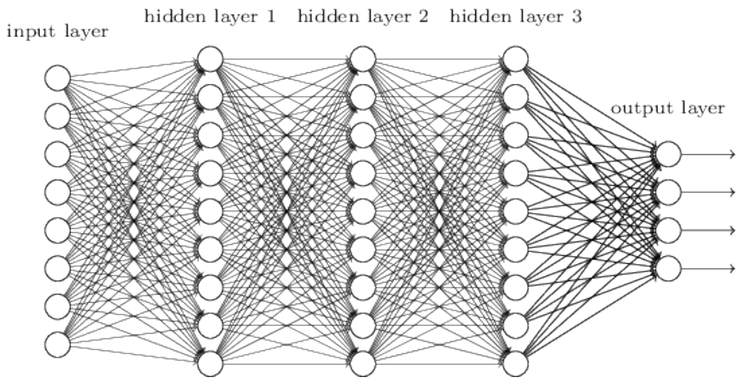
```
int get(int[] arr, int index){
  if(index > arr.length){
    throw new RuntimeException();
  }
  return arr[index];
}
```



- **Traditional software:** designed to be secure
- **ML systems:** learned organically from data, no explicit design

Formal Verification is Hard

```
int get(int[] arr, int index){
  if(index > arr.length){
    throw new RuntimeException();
  }
  return arr[index];
}
```

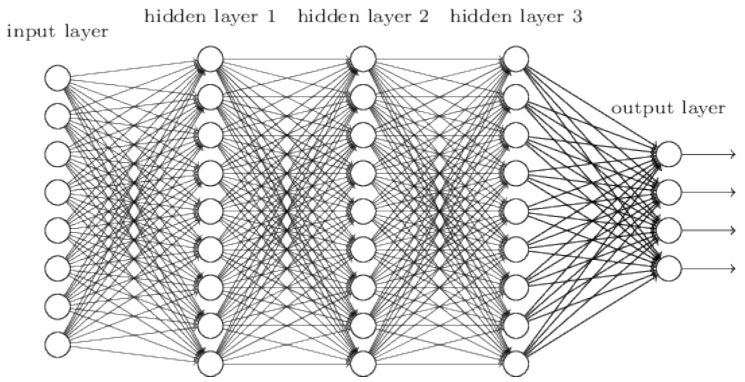


- **Traditional software:** designed to be secure
- **ML systems:** learned organically from data, no explicit design

Hard to analyze, limited levers

Formal Verification is Hard

```
int get(int[] arr, int index){
  if(index > arr.length){
    throw new RuntimeException();
  }
  return arr[index];
}
```



- **Traditional software:** designed to be secure
- **ML systems:** learned organically from data, no explicit design

Hard to analyze, limited levers

Other challenges:

- adversary has access to sensitive parts of system
- unclear what spec should be (car doesn't crash?)

What To Prove?

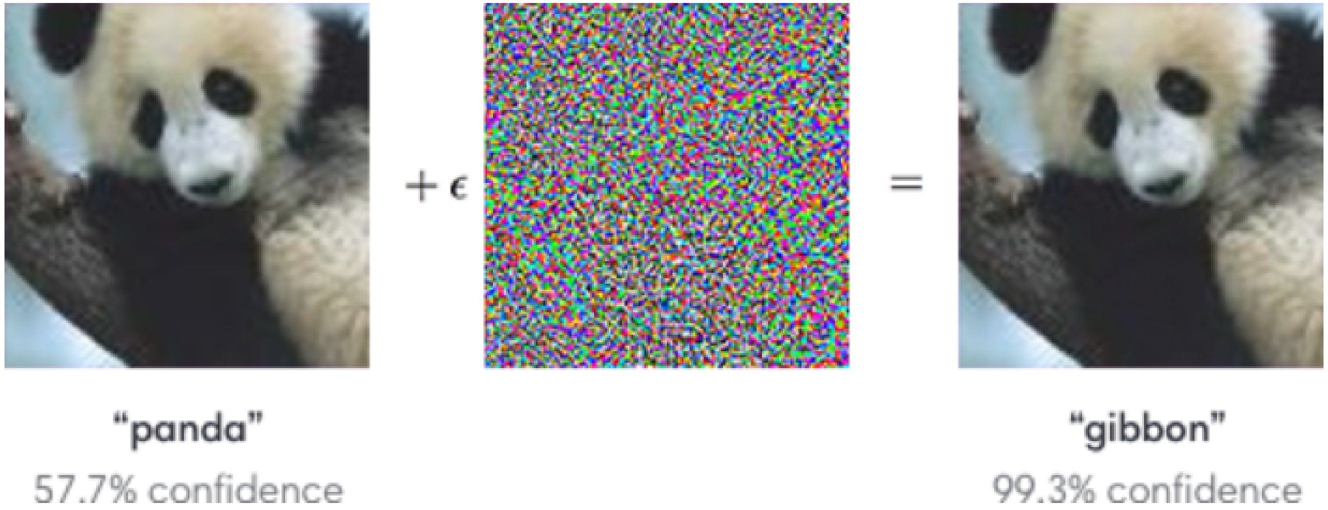
- Security against test-time attacks
- Security against training-time attacks
- Lack of implementation bugs

What To Prove?

- **Security against test-time attacks**
- Security against training-time attacks
- Lack of implementation bugs

Test-time Attacks

Adversarial examples:



Can we prove no adversarial examples exist?

Formal Goal

Goal

Given a classifier $f : \mathbb{R}^d \to \{1, \ldots, k\}$, and an input x, show that there is no x' with $f(x) \neq f(x')$ and $\|x - x'\| \leq \epsilon$.

- Norm: ℓ^∞-norm: $\|x\| = \max_{j=1}^d |x_j|$
- Classifier: f is a neural network

[Katz, Barrett, Dill, Julian, Kochenderfer 2017]

Approach 1: Reluplex

Assume f is a ReLU network: layers $x^{(1)}, \ldots, x^{(L)}$, with

$$x_i^{(l+1)} = \max(a_i^{(l)} \cdot x^{(l)}, 0)$$

Want to bound maximum change in output $x^{(L)}$.

Can write as an **integer-linear program (ILP)**:

$$y = \max(x, 0) \iff$$
$$x \leq y \leq x + b \cdot M,$$
$$0 \leq y \leq (1 - b) \cdot M,$$
$$b \in \{0, 1\}$$

Check robustness on 300-node networks

- time ranges from 1s to 4h (median 3m-4m)

[Raghunathan, S., Liang]

Approach 2: Relax and Dualize

Still assume f is ReLU

Can write as a **non-convex quadratic program** instead.

[Raghunathan, S., Liang]

Approach 2: Relax and Dualize

Still assume f is ReLU

Can write as a **non-convex quadratic program** instead.

Every quadratic program can be relaxed to a **semi-definite program**

[Raghunathan, S., Liang]

Approach 2: Relax and Dualize

Still assume f is ReLU

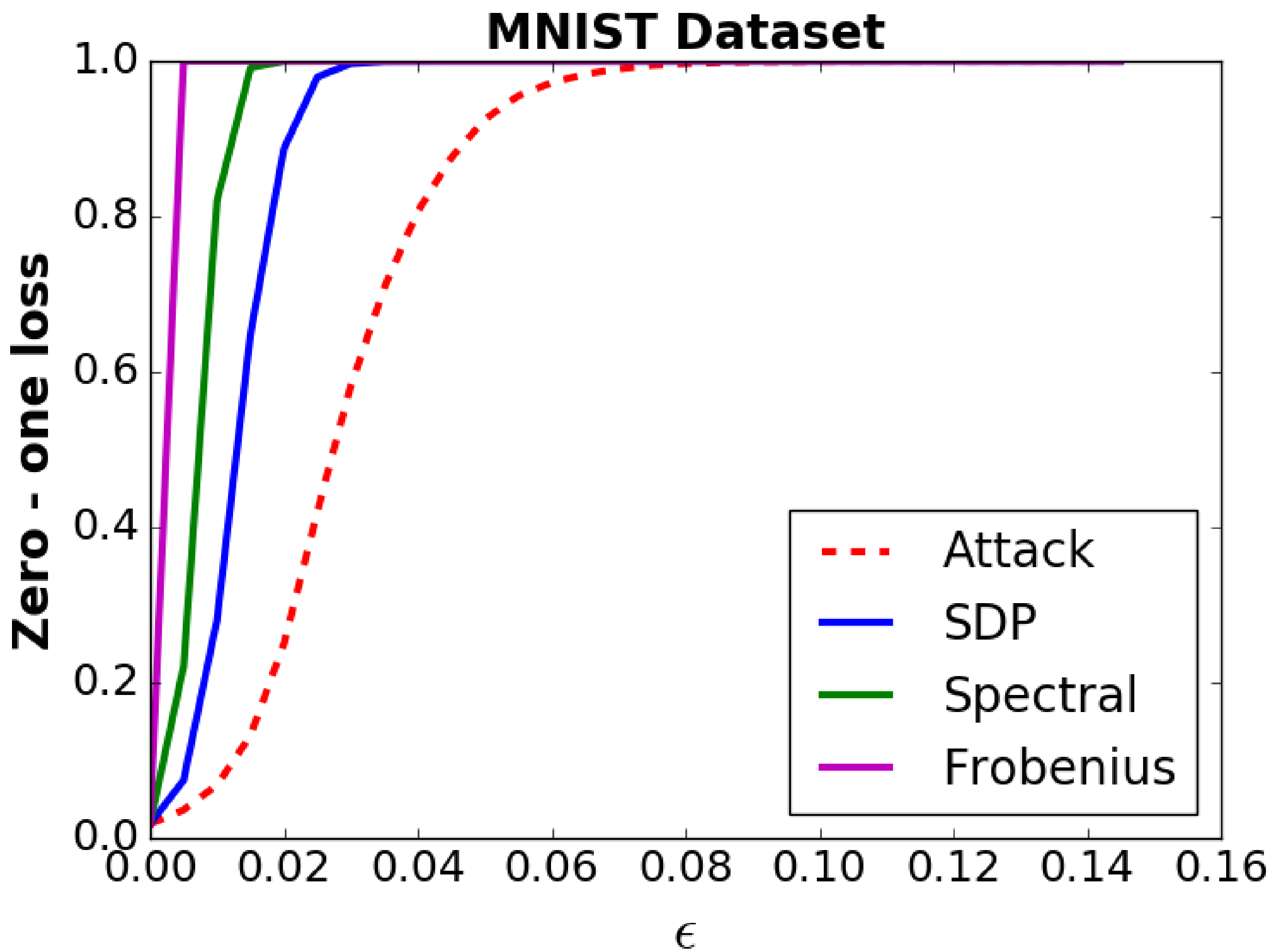
Can write as a **non-convex quadratic program** instead.

Every quadratic program can be relaxed to a **semi-definite program**

Advantages:

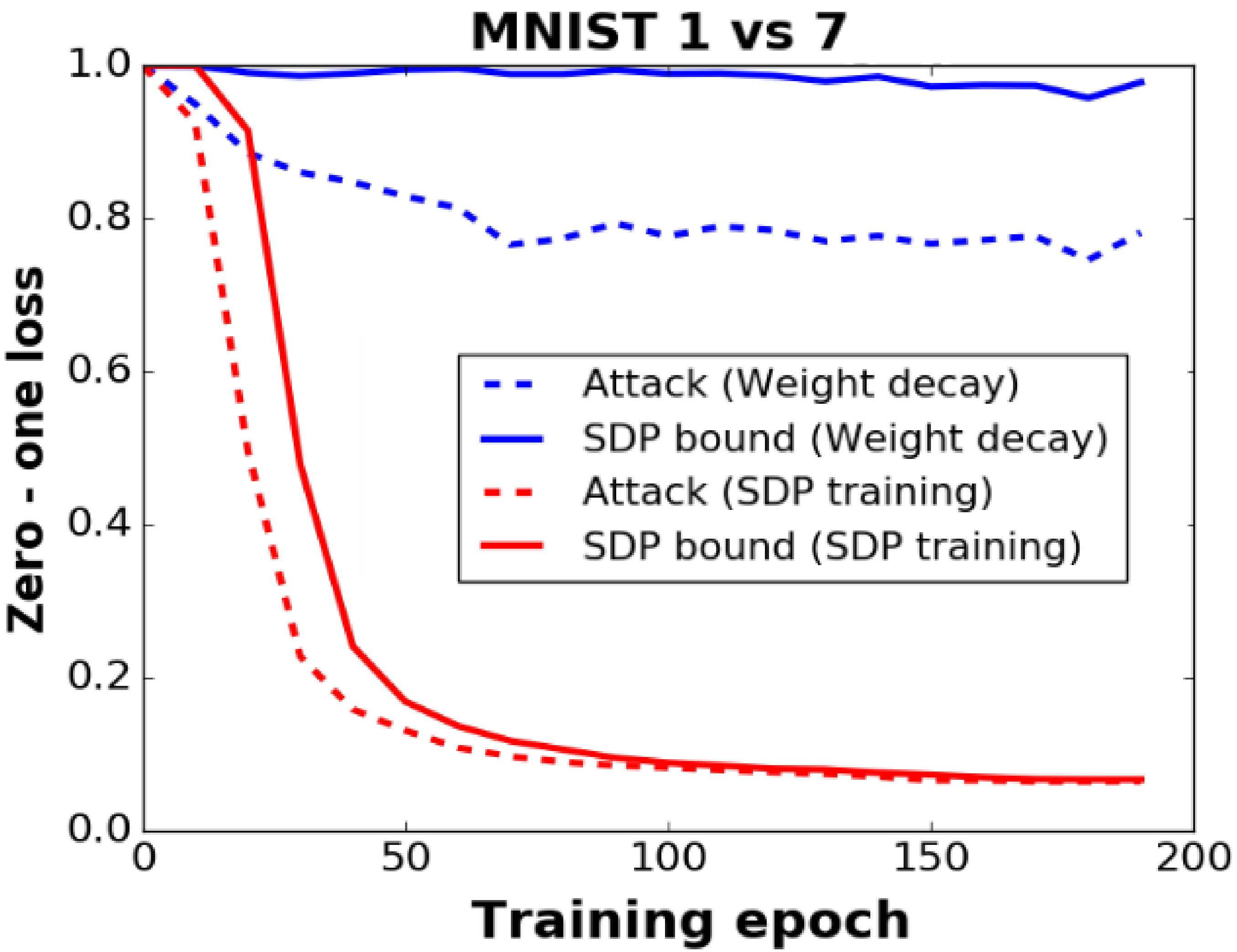
- always polynomial-time
- duality: get **differentiable** upper bounds
- can train against upper bound to generate robust networks

Results



MNIST Dataset
Zero - one loss
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
ϵ
Attack
SDP
Spectral
Frobenius

Results



MNIST 1 vs 7
Zero - one loss
Training epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
Attack (Weight decay)
SDP bound (Weight decay)
Attack (SDP training)
SDP bound (SDP training)

What To Prove?

- Security against test-time attacks
- **Security against training-time attacks**
- Lack of implementation bugs

Training-time attacks

Attack system by manipulating training data: *data poisoning*

Traditional security: keep attacker away from important parts of system

Data poisoning: attacker has access to most important part of all

Training-time attacks

Attack system by manipulating training data: *data poisoning*

Traditional security: keep attacker away from important parts of system

Data poisoning: attacker has access to most important part of all

Huge issue in practice...

Training-time attacks

Attack system by manipulating training data: *data poisoning*

Traditional security: keep attacker away from important parts of system

Data poisoning: attacker has access to most important part of all

Huge issue in practice...

How can we keep adversary from subverting the model?

Formal Setting

Adversarial game:

- Start with clean dataset $\mathcal{D}_c = \{x_1, \dots, x_n\}$
- Adversary adds ϵn bad points $\mathcal{D}_p$
- Learner trains model on $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_p$, outputs model θ and incurs loss $L(\theta)$

Learner's goal: ensure $L(\theta)$ is low no matter what adversary does

- under a priori assumptions,
- or for a specific dataset $\mathcal{D}_c$.

Formal Setting

Adversarial game:

- Start with clean dataset $\mathcal{D}_c = \{x_1, \ldots, x_n\}$
- Adversary adds ϵn bad points $\mathcal{D}_p$
- Learner trains model on $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_p$, outputs model θ and incurs loss $L(\theta)$

Learner's goal: ensure $L(\theta)$ is low no matter what adversary does

- under a priori assumptions,
- or for a specific dataset $\mathcal{D}_c$.

In high dimensions, most algorithms fail!

Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

Learning from Untrusted Data

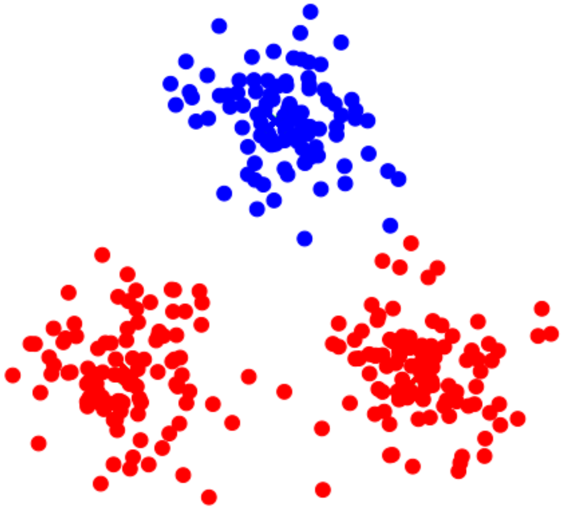
A priori assumption: covariance of data is bounded by σ.

Theorem: as long as we have a small number of "verified" points, can be robust to any fraction of adversaries (even e.g. 90%).

Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

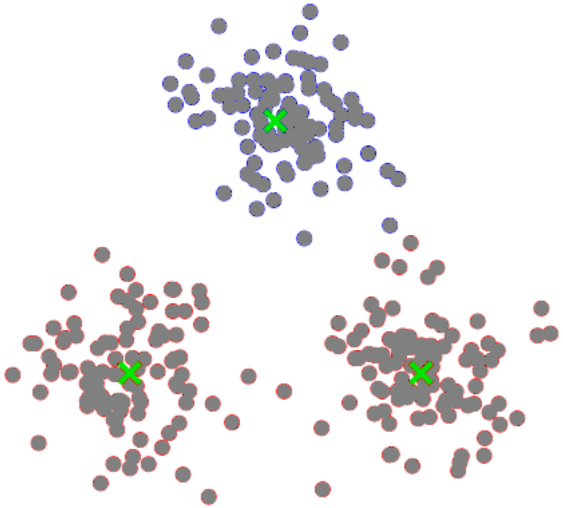
Theorem: as long as we have a small number of "verified" points, can be robust to any fraction of adversaries (even e.g. 90%).



Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

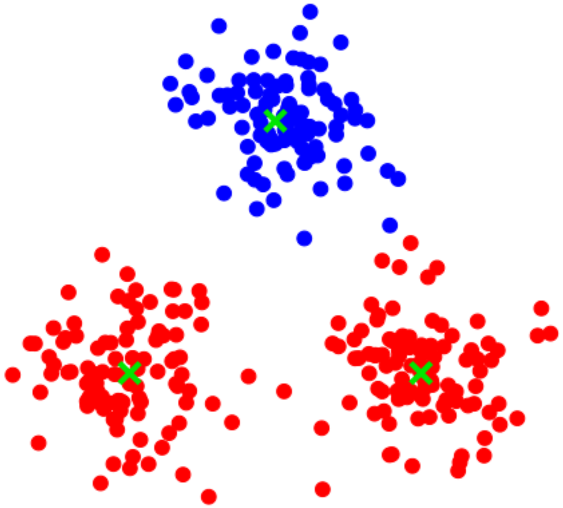
Theorem: as long as we have a small number of "verified" points, can be robust to any fraction of adversaries (even e.g. 90%).



Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

Theorem: as long as we have a small number of "verified" points, can be robust to any fraction of adversaries (even e.g. 90%).

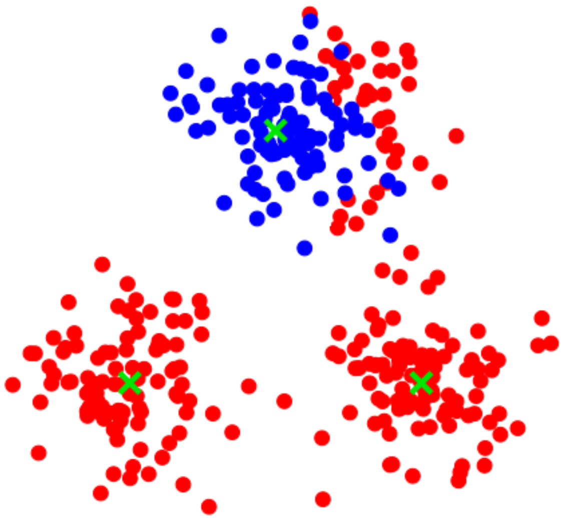


[Charikar, S., Valiant 2017]

Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

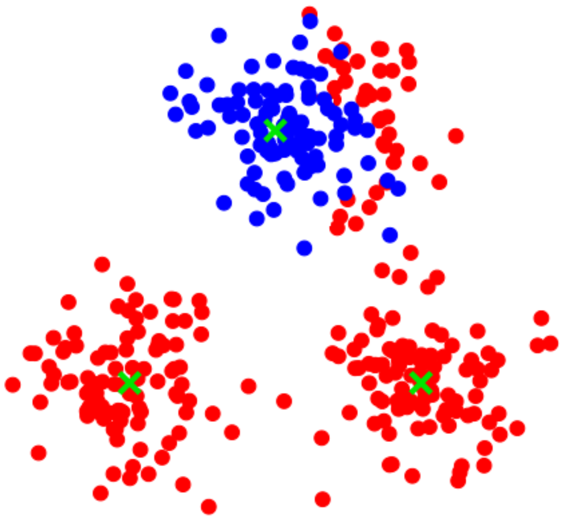
Theorem: as long as we have a small number of "verified" points, can be robust to any fraction of adversaries (even e.g. 90%).



Learning from Untrusted Data

A priori assumption: covariance of data is bounded by σ.

Theorem: as long as we have a small number of “verified” points, can be robust to any fraction of adversaries (even e.g. 90%).



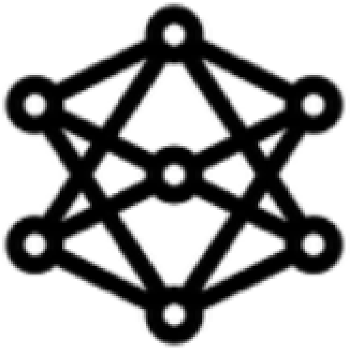
Growing literature: 15+ papers since 2016 [DKKLMS16/17, LRV16, SVC16, DKS16/17, CSV17, SCV17, L17, DBS17, KKP17, S17, MV17]

What about certifying a specific algorithm on a specific data set?

Certified Defenses for Data Poisoning

Clean
training data
$\mathcal{D}_c$

$$\hat{\theta} \stackrel{\text{def}}{=} \arg\min_\theta L(\theta; \mathcal{D}_c)$$



Model

Certified Defenses for Data Poisoning

Clean training data

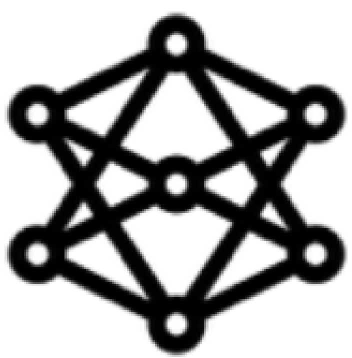
$\mathcal{D}_c$

Poisoned training data

$\mathcal{D}_p$

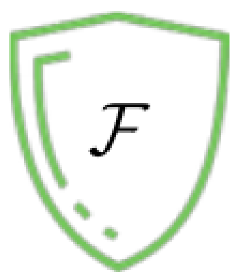
$|\mathcal{D}_p| = \epsilon |\mathcal{D}_c|$

$$\hat{\theta}_{\mathcal{D}_p} \stackrel{\text{def}}{=} \arg\min_\theta L(\theta; \mathcal{D}_c \cup \mathcal{D}_p)$$



Model

Certified Defenses for Data Poisoning



Outlier removal

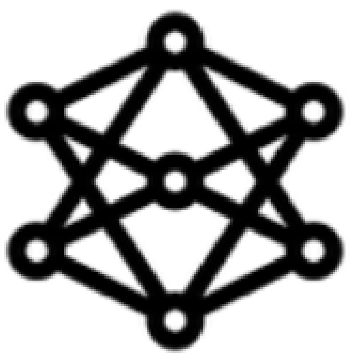
Defender discards outliers outside some feasible set $\mathcal{F}$

Certified Defenses for Data Poisoning

Clean
training data
$\mathcal{D}_c$

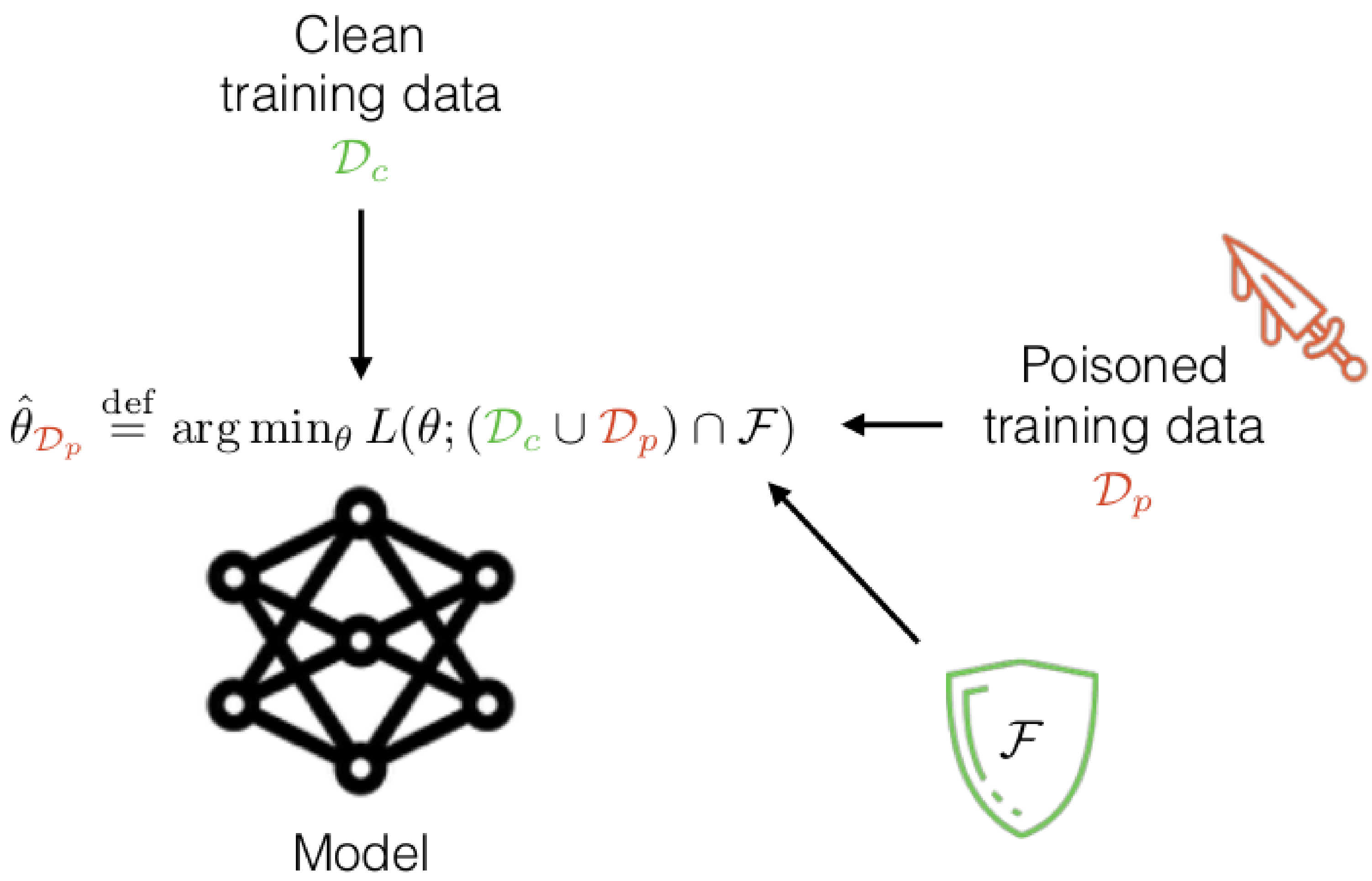
Poisoned
training data
$\mathcal{D}_p$

$$\hat{\theta}_{\mathcal{D}_p} \stackrel{\text{def}}{=} \arg\min_\theta L(\theta; \mathcal{D}_c \cup \mathcal{D}_p)$$



Model

Certified Defenses for Data Poisoning

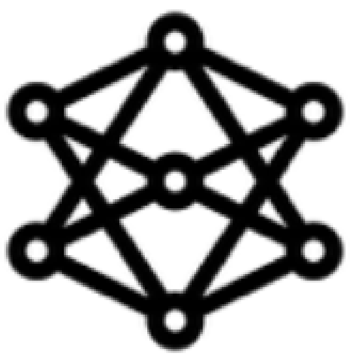


Certified Defenses for Data Poisoning

Clean
training data
$\mathcal{D}_c$

$$\hat{\theta}_{\mathcal{D}_p} \stackrel{\text{def}}{=} \arg\min_\theta L(\theta; (\mathcal{D}_c \cup \mathcal{D}_p) \cap \mathcal{F})$$

Poisoned
training data
$\mathcal{D}_p$



Model

What's the **worst case** test loss over all possible attacks?

Impact on training loss

Worst-case impact is solution to **bi-level optimization problem**:

$$\text{maximize}_{\hat{\theta},\mathcal{D}_p}\, L(\hat{\theta}) \;\; \text{subject to} \;\; \hat{\theta} = \text{argmin}_{\theta} \textstyle\sum_{x \in \mathcal{D}_c \cup \mathcal{D}_p} \ell(\theta; x),$$
$$\mathcal{D}_p \subseteq \mathcal{F}$$

Impact on training loss

Worst-case impact is solution to **bi-level optimization problem**:

$$\text{maximize}_{\hat{\theta}, \mathcal{D}_p} \, L(\hat{\theta}) \;\; \text{subject to} \;\; \hat{\theta} = \operatorname{argmin}_{\theta} \sum_{x \in \mathcal{D}_c \cup \mathcal{D}_p} \ell(\theta; x),$$
$$\mathcal{D}_p \subseteq \mathcal{F}$$

(Very) NP-hard in general

Impact on training loss

Worst-case impact is solution to **bi-level optimization problem**:

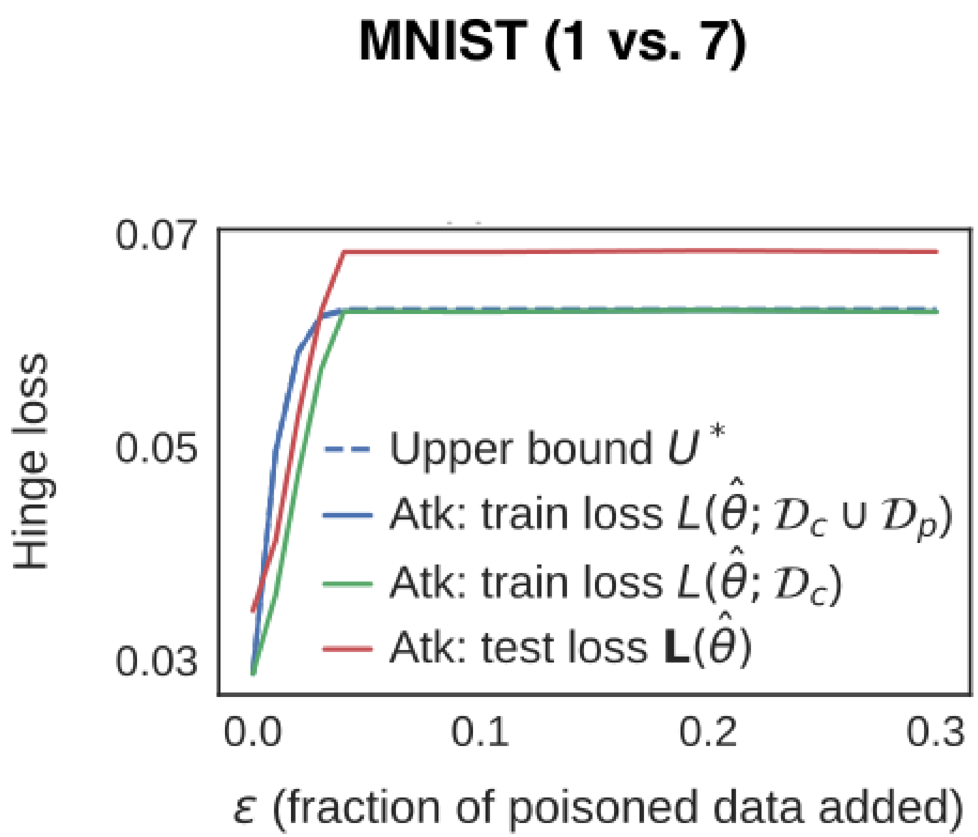
$$\text{maximize}_{\hat{\theta}, \mathcal{D}_p} L(\hat{\theta}) \;\text{ subject to }\; \hat{\theta} = \text{argmin}_\theta \textstyle\sum_{x \in \mathcal{D}_c \cup \mathcal{D}_p} \ell(\theta; x),$$
$$\mathcal{D}_p \subseteq \mathcal{F}$$

(Very) NP-hard in general

Key insight: approximate test loss by train loss, can then upper bound via a saddle point problem (tractable)

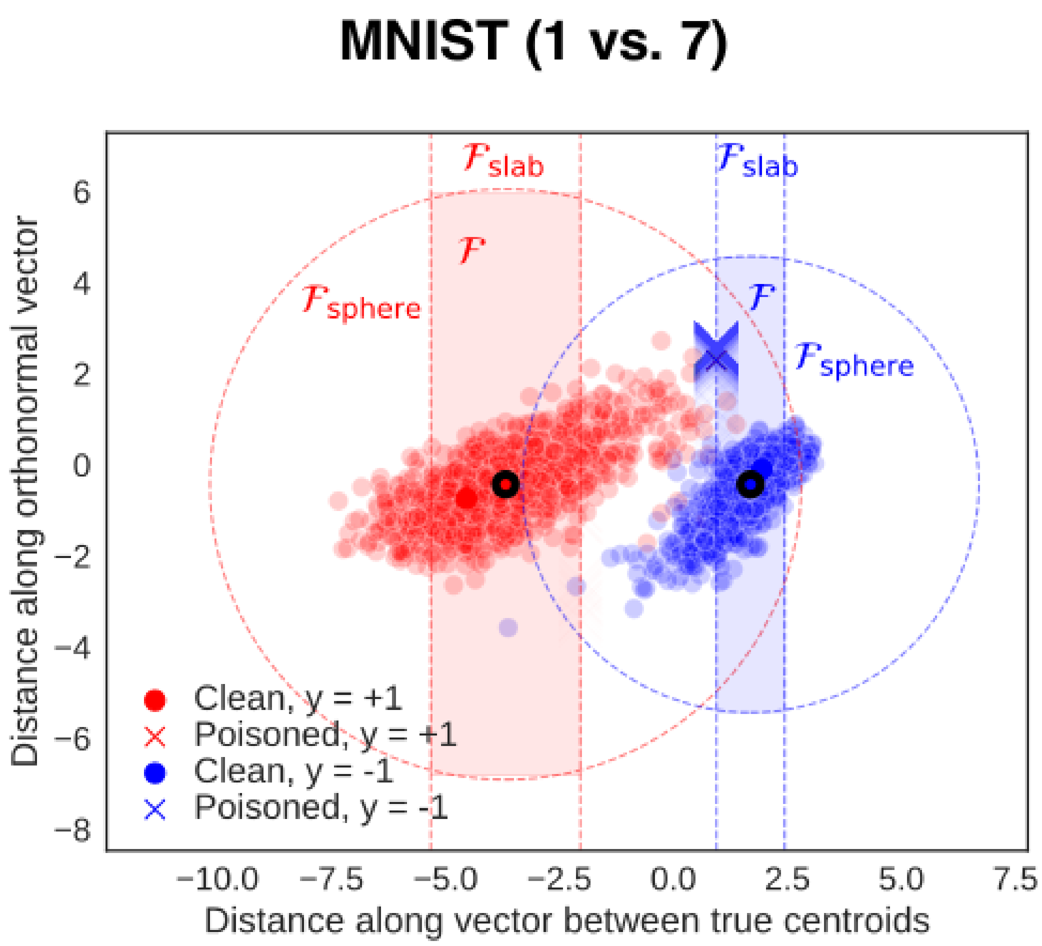
- **automatically generates** a nearly optimal attack

Results

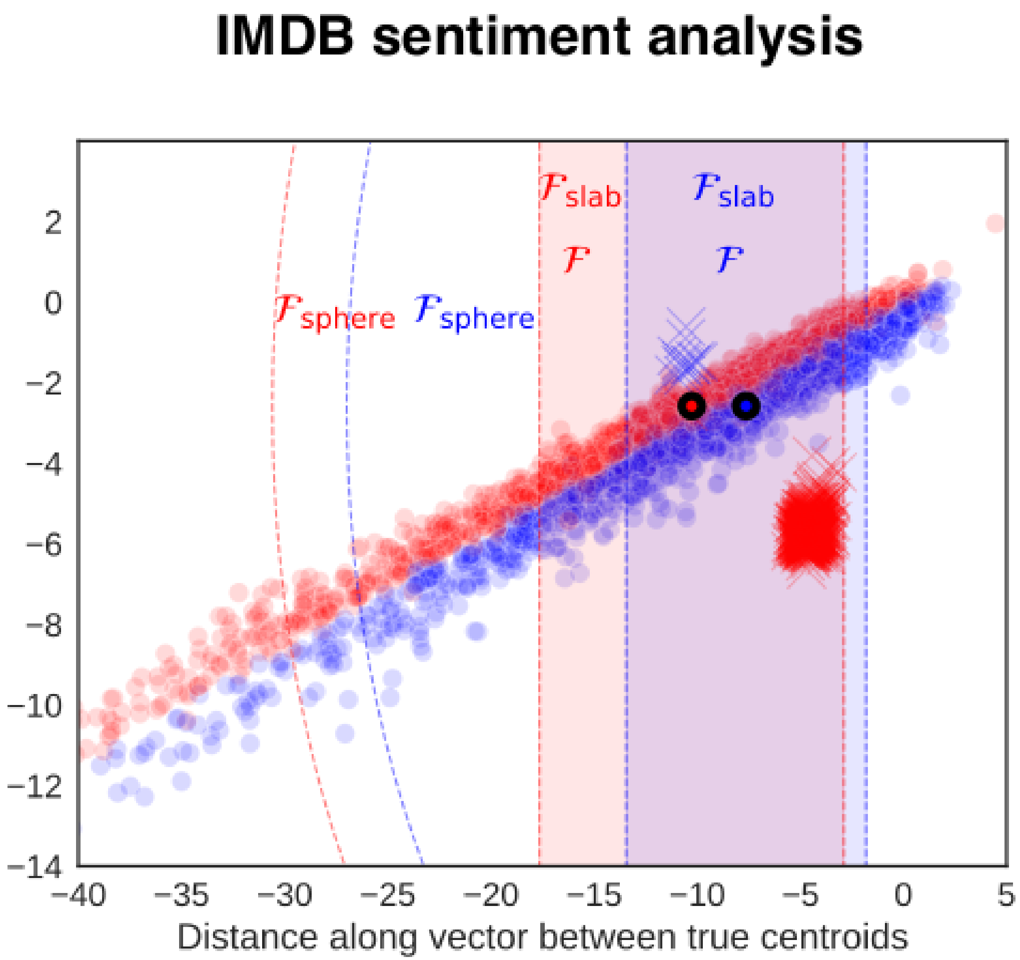


MNIST (1 vs. 7)
0.07
0.05
0.03
Hinge loss
Upper bound U^*
Atk: train loss $L(\hat{\theta}; \mathcal{D}_c \cup \mathcal{D}_p)$
Atk: train loss $L(\hat{\theta}; \mathcal{D}_c)$
Atk: test loss $\mathbf{L}(\hat{\theta})$
0.0
0.1
0.2
0.3
ε (fraction of poisoned data added)

Results


MNIST (1 vs. 7)
$\mathcal{F}_{\text{slab}}$
$\mathcal{F}_{\text{slab}}$
$\mathcal{F}$
$\mathcal{F}_{\text{sphere}}$
$\mathcal{F}$
$\mathcal{F}_{\text{sphere}}$
Clean, y = +1
Poisoned, y = +1
Clean, y = -1
Poisoned, y = -1
Distance along orthonormal vector
Distance along vector between true centroids
6
4
2
0
−2
−4
−6
−8
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

Results



11% --> 23% error with 3% poisoned data

What To Prove?

- Security against test-time attacks

- Security against training-time attacks

- **Lack of implementation bugs**

Theano / **Theano**

Watch 512

Code | Issues 591 | Pull requests 114 | Boards | Reports | Projects 0 | Wiki

Insidious random data/memory corruption bug causing incorrect computation and training divergence #4770

Closed **xuancong84** opened this issue on Jul 20, 2016 · 17 comments

xuancong84 commented on Jul 20, 2016 • edited

It seems that sometimes by chance, Theano's (for all versions including bleeding-edge) internal memory state can get corrupted silently, with all subsequent training/testing operations produces erroneous results without throwing any exceptions/warnings. The error will accumulate until some point when the training starts to always diverge. The problem can be solved by aborting the current process, reloading the last-known good model and resuming training.

Theano / Theano

Watch 512

Code | Issues 591 | Pull requests 114 | Boards | Reports | Projects 0 | Wiki

Insidious random data/memory corruption bug causing incorrect computation and training divergence #4770

Closed **xuancong84** opened this issue on Jul 20, 2016 · 17 comments

xuancong84 commented on Jul 20, 2016 • edited

It seems that sometimes by chance, Theano's (for all versions including bleeding-edge) internal memory state can get corrupted silently, with all subsequent training/testing operations produces erroneous results without throwing any exceptions/warnings. The error will accumulate until some point when the training starts to always diverge. The problem can be solved by aborting the current process, reloading the last-known good model and resuming training.

lamblin closed this 24 days ago

xuancong84 commented 24 days ago

@lamblin any idea why it diverges?
Actually, running on CPU gives more reproducible results. You should run it on GPU. Anyway, Theano has some serious bugs, I no longer use it.

[Selsam and Liang 2017]

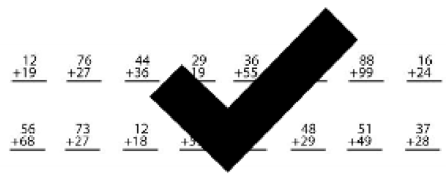
Developing Bug-Free ML Systems

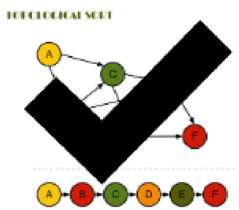
```
-- Formal specification
def gsplus_spec (f : ℝ → ℝ) : Prop :=
 ∀ x, f x = ∇ splus x

-- Incorrect implementation
def gsplus (x : ℝ) : ℝ := 1 / (1 + exp x)

-- Proof
theorem gsplus_correct : gsplus_spec gsplus :=
 -- first take a few actions to simplify the goal,
 -- leaving the unprovable goal:
 -- x : ℝ ⊢ 1 / (1 + exp x) = (exp x) / (1 + exp x)

-- Revised implementation
def gsplus (x : ℝ) : ℝ := (exp x) / (1 + exp x)

-- Revised proof
theorem gsplus_correct : gsplus_spec gsplus :=
 -- now the proof goes through successfully

-- Execute with floating point numbers
vm_eval gsplus π -- answer: 0.958576
```

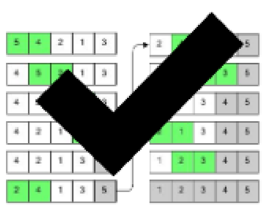
Provable Generalization via Recursion

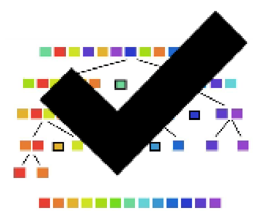
Verification of Perfect Generalization

We successfully verified a learned recursive program for each task via the *oracle matching* procedure.


Grade-School Addition


Topological Sort


Bubble Sort


Quicksort

Summary

Formal verification can be used in **many contexts:**

- test-time attacks
- training-time attacks
- implementation bugs
- checking generalization

High-level ideas:

- cast as **optimization problem**: rich set of tools
- **train/optimize** against certificate
- **re-design** system to be amenable to proof

Are we verifying the right thing?

"Real" goal not easy to state:

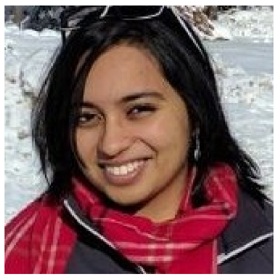
- ℓ^∞-perturbations are arbitrary
- low test error $\implies$ specific inputs could still be bad
- what does security even mean for non-convex models?

How do we specify our real end goals?

- "my car won't crash"
- "my newsfeed won't disseminate propaganda"
- "my trading algorithm won't lose $$$"

Acknowledgments

Collaborators:





Funding:

the Hertz FOUNDATION freedom to innovate

NSF

future of life INSTITUTE

NIPS Workshop on Secure ML: Please submit your work!