Program Analysis for System Security and Reliability

Dana Drachsler Cohen
Spring 2018
Last Time: Machine Learning Models

Models approximate a function $f^*$ (which is hard to define)

A model is an architecture and a set of parameters

\[ f_{w_1, \ldots, w_n, b}(x_1, \ldots, x_n) = \begin{cases} 1 & \sum w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases} \]

A loss function measures how well a model approximates the function

\[ \text{MSE} = \Sigma_{i \in D_T} (f^*(i) - f(i))^2 \]

Parameters are tuned to have lowest loss via gradient descent
Deep Models

A deep model is a directed graph of **neurons** organized in **layers**

A neuron is a simple model followed by an **activation function** e.g., ReLU:

\[ ReLU(a) = \max(0, a) \]
Feed Forward Neural Network (FF NN)

Neurons are connected to all neurons in the next layer

To compute output, feed forward the input through the network
Parameters are tuned via backpropagation
Convolutional Networks

Neural networks consisting of:

- convolutional layers
  - neurons are connected to local regions in the inputs and share weights
- pooling layers
  - neurons compute a downsample of the input to reduce dimensionality
- fully connected layers
  - identical to FF NN, used to perform the classification
Backpropagation of the Loss Function

(Back)propagates the common parts of the derivatives

$$\sum_{l_1^0 \in D_{Tr}} \frac{\partial \text{MSE}}{\partial l_1^2} \cdot \frac{\partial l_1^2}{\partial p_1^2} \cdot \frac{\partial p_1^2}{\partial l_1^1} \cdot \frac{\partial l_1^1}{\partial p_1^1} \cdot \frac{\partial p_1^1}{\partial w_{1,1}^1}$$

Input:
- $l_1^0$

Intermediate Layers:
- $p_1^1 = w_{1,1}^1 l_1^0 + b_1^1$
- $l_1^1 = \text{ReLU}(p_1^1)$
- $p_2^1 = w_{2,1}^1 l_1^0 + b_2^1$
- $l_2^1 = \text{ReLU}(p_2^1)$

Output:
- $f(l_1^0)$

Output Layer:
- $p_1^2 = w_{1,1}^2 \cdot l_1^1 + w_{1,2}^2 \cdot l_1^2 + b_1^2$
- $l_1^2 = \text{ReLU}(p_1^2)$

$$\frac{\partial \text{MSE}}{\partial w_{1,1}^2} = \sum_{l_1^0 \in D_{Tr}} \frac{\partial \text{MSE}}{\partial l_1^2} \cdot \frac{\partial l_1^2}{\partial p_1^2} \cdot \frac{\partial p_1^2}{\partial p_1^1} \cdot \frac{\partial p_1^1}{\partial w_{1,1}^1}$$
Adversarial Examples

• Given:
  ▪ neural network $f: X \rightarrow C$
  ▪ input $x \in X$
  ▪ target label $t \in C$, such that $f(x) \neq t$

• Compute $\eta$ such that $f(x + \eta) = t$
  By minimizing the loss function with respect to the input
Targeted Fast Gradient Sign Method (T-FGSM)

1. **Compute perturbation:**
   \[
   \eta = \epsilon \cdot \text{sign}(\nabla_x \text{loss}_t(x)), \text{ where }
   \nabla_x \text{loss}_t = \left(\frac{\partial \text{loss}_t}{\partial x_1}, ..., \frac{\partial \text{loss}_t}{\partial x_n}\right) \quad \text{sign}(x) = \begin{cases} 
   -1, & \text{if } x < 0 \\
   0, & \text{if } x = 0 \\
   1, & \text{if } x > 0
   \end{cases}
   \]

2. **Perturb the input:**
   \[x' = x + \eta\]

3. **Check whether:**
   \[f(x') = t\]

Explaining and Harnessing Adversarial Examples, Goodfellow et al., 2015
Today

Two more attacks

Robustness
Adversarial Examples

• Given:
  ▪ neural network $f: X \rightarrow C$
  ▪ input $x \in X$
  ▪ target label $t \in C$, such that $f(x) \neq t$

• Compute $\eta$ such that $f(x + \eta) = t$
  By minimizing the loss function with respect to the input

Does the adversary need it?
Minimal Adversarial Examples

• Given:
  ▪ neural network $f: X \rightarrow C$
  ▪ input $x \in X$
  ▪ target label $t \in C$, such that $f(x) \neq t$

• Compute a minimal $\eta$ such that $f(x + \eta) = t$
  
  By computing the saliency map of $f$, indicating where $f$ changes most
High Level Idea through an Example

Assume we try to learn the logical AND function:

\[ f^*(x_1, x_2) = x_1 \land x_2 \]

Inputs are real numbers in \([0,1] \times [0,1]\) and thus:

\[ f^*(x_1, x_2) = \text{round}(x_1) \land \text{round}(x_2) \]

For example, \(f^*(0.7,0.3) = 0\) \(f^*(0.7,0.6) = 1\)
Illustration of a FF NN for $f^*$
The Gradient of the FF NN

The gradient of the FF NN function is its partial derivatives:

\[ \nabla f(x_1, x_2) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right) \]

What can be learned from \( \nabla f(x_1, x_2) \)?
The partial derivative $\frac{\partial f}{\partial x_2}$
The partial derivative \( \frac{\partial f}{\partial x_2} \)
Take Aways

For some inputs, small perturbations can significantly change the output

For other inputs, it is not the case

The gradient of $f$ guides towards adversarial examples
Computing the gradient in FF NN

\[
\frac{\partial p_1^1}{\partial l_0^0} = w_{1,1}^1, \quad \frac{\partial l_1^1}{\partial l_0^1} = \frac{\partial ReLU(p_1^1)}{\partial p_1^1} \cdot \frac{\partial p_1^1}{\partial l_0^1} = w_{1,1}^1 \cdot \frac{\partial l_1^1}{\partial l_0^1} + w_{1,2} \cdot \frac{\partial l_2^1}{\partial l_0^1} \\
\]

\[
l_1^1 = ReLU(p_1^1) \\
p_1^1 = w_{1,1}^1 l_1^0 + b_1^1 \\
\]

\[
\frac{\partial p_2^1}{\partial l_0^1} = w_{2,1}^1, \quad \frac{\partial l_2^1}{\partial l_0^1} = \frac{\partial ReLU(p_2^1)}{\partial p_2^1} \cdot \frac{\partial p_2^1}{\partial l_1^0} \\
\]

\[
l_2^1 = ReLU(p_2^1) \\
p_2^1 = w_{2,1}^1 l_1^0 + b_2^1 \\
\]

\[
f(l_0^0) \\
p_2^2 = w_{1,1}^2 \cdot l_1^1 + w_{1,2}^2 \cdot l_2^1 + b_2^2 \\
\]

\[
l_2^1 = ReLU(p_2^2) \\
p_2^2 = w_{2,1}^2 \cdot l_1^1 + w_{2,2}^2 \cdot l_2^1 + b_2^2 \\
\]

\[
f(l_0^0) = \frac{\partial ReLU(p_2^2)}{\partial p_2^2} \cdot \frac{\partial p_2^2}{\partial l_1^0} \\
\]

The Limitations of Deep Learning in Adversarial Settings, Papernot et al., 2015
Adversarial Examples via $f$’s Gradient

The goal of adversarial examples is:

given a network $f$, an input $x$, and a target $t$ (s.t., $f(x) \neq t$),
find $\eta$: such that $f(x + \eta) = t$

Since, $f = (f_1, ..., f_n)$, we extend the gradient to the Jacobian:

Goal: increase $\frac{\partial f_t}{\partial x_i}$, decrease $\frac{\partial f_j}{\partial x_i}$, $\forall j \neq t$
Adversarial Examples via $f$’s Gradient

The Jacobian matrix guides for every class, what part of the input can changed to find adversarial example:

$$\frac{\partial f_j}{\partial x_i} = 0, \text{ no change}$$
$$\frac{\partial f_j}{\partial x_i} > 0, \text{ increase } x_i$$
$$\frac{\partial f_j}{\partial x_i} < 0, \text{ decrease } x_i$$

For each $x_i$, we can increase or decrease
Which $x_i$ should we change?
Adversarial Examples via $f$’s Gradient

For a target $t$, increase those for which $\frac{\partial f_t}{\partial x_i}$ increases, while the combined classification of the other labels $f_j$ decreases.

Formally,

$$S(x_1, ..., x_n, t)[i] = \begin{cases} 0 & \text{if } \frac{\partial f_t}{\partial x_i} < 0 \text{ or } \Sigma_{j \neq t} \frac{\partial f_j}{\partial x_i} > 0 \\ 1 & \text{otherwise} \end{cases}$$

However, some $i$ affect more significantly
Saliency Maps

A matrix defining the intensity of inputs whose increase helps the most to accomplish the goal

\[ S(x_1, \ldots, x_n, t)[i] = \begin{cases} 
    0 & \text{if } \frac{\partial f_t}{\partial x_i} < 0 \text{ or } \sum_{j \neq t} \frac{\partial f_j}{\partial x_i} > 0 \\
    \left(\frac{\partial f_t}{\partial x_i}\right) \cdot \left|\sum_{j \neq t} \frac{\partial f_j}{\partial x_i}\right| & \text{otherwise}
\end{cases} \]
Adversarial Examples with Saliency Maps

Given a FF NN $f$, an input $x$, and a target $t$:

define $x' = x$

while $f(x') \neq t$ \& max iterations not exceeded

compute the saliency map $S(x'_1, ..., x'_n, t)$

let $i$ be the index maximizing $S(x'_1, ..., x'_n, t)$

$x'_i += \theta$
Remarks

Like previous attacks, this attack is not guaranteed to find an adversarial example

However, it enables to easily define:
  which inputs change
  by how much they change
Recap

A different attack to find adversarial examples using the Jacobian of $f$ instead of the gradient of $\text{loss}(f)$
White-Box Attacks

So far we focused on white-box attacks, which require to know:
- The model architecture and parameters (sometimes the loss)
- The training set

Safe-critical applications will not publish their network or dataset

Are neural networks safe from such attacks?
Black-Box Attacks

The attacker has a limited access to the model:

- can only query the model to get classifications
- cannot pose too many queries without being exposed
Main Idea

The attacker leverages prior knowledge:

- The dataset type (e.g., images, text)
- The common architecture (e.g., convolutional network)

The attacker *trains* another model used to find adversarial examples

- What architecture?
- With which input-output examples?
Training a New Model

Since the attacker knows the task at hand, she can pick a architecture that is commonly used

e.g., convolutional networks for classifying images

How to generate the input-output examples?

Idea: Assume the attacker can collect an initial set of inputs similar to the dataset, then synthesize more inputs and ask for the classification to get new input-output examples
Training Procedure

Define architecture for $\hat{f}$, the function approximating $f$

Let $D = \{(i_1, o_1), ..., (i_k, o_k)\}$ be the initial training set

Repeat $N$ times:
  Train $\hat{f}$ on $D$
  Generate new inputs and query $f$ for their output
  Extend $D$ with these input-output examples
Generating Inputs

The goal is to compute $\hat{f}$ which is close to $f$.

We can generate random inputs to extend $D$ with.

Instead, we generate inputs that improve the confidence in $\hat{f}$.

We pick inputs whose neighborhood shows a large variance of $\hat{f}$.
Generating Inputs based on Gradient

The gradient of (a single class) $\hat{f}$ points to where it changes most

Given a point $(i, o) \in D$, compute the gradient of $\hat{f}_o$:

$$\left( \frac{\partial f_o}{\partial i_1}, \ldots, \frac{\partial f_o}{\partial i_n} \right)$$

Based on the signs of the partial derivatives, perturb $i$ by $\lambda$:

$$i' = i + \lambda \cdot \text{sign} \left( \frac{\partial f_o}{\partial i_1}, \ldots, \frac{\partial f_o}{\partial i_n} \right)$$
Training Procedure

Define architecture for $\hat{f}$, the function approximating $f$

Let $D = \{(i_1, o_1), \ldots, (i_k, o_k)\}$ be the initial training set

Repeat $N$ times:

Train $\hat{f}$ on $D$

For every $(i, o) \in D$,

compute $i' = i + \lambda \cdot \text{sign} \left( \frac{\partial f_o}{\partial i_1}, \ldots, \frac{\partial f_o}{\partial i_n} \right)$, add $(i', f(i'))$ to $D$
In practice

This approach has been shown to work in some cases

While success is not guaranteed, it still demonstrates that neural networks are not safe against adversarial examples even if used as black-box

Can we prove when a network is safe for some inputs?
Robustness of Neural Networks

Given a model $f$ and an input $x$, we say $f$ is robust for $x$ in a neighborhood $N_x$ if:

$$\text{for all } x' \in N_x: f(x) = f(x')$$

Common $N_x$:

$$N_x^\varepsilon = \left\{ x' \mid \|x' - x\|_p < \varepsilon \right\}$$

for $p = 0, 1, 2, \infty$
Checking Robustness of Neural Network

Testing:

Check a strict subset of points in $N_x$

If we find $x' \in N_x$ such that $f(x') \neq f(x)$, $f$ is not robust for $x$

Otherwise, cannot guarantee robustness
Checking Robustness of Neural Network

Verification:

Analyze all points in $N_x$

$f$ is robust for $x$ if and only if for all $x' \in N_x$, $f(x') = f(x)$

How?
Verification

In general, verifying that a function (e.g., neural network) verifies a property (e.g., robustness) is undecidable.

However, under certain assumptions, it may be decidable.

Today, we show one such procedure that analyzes robustness of feed forward neural networks.

Next week, we will see an approach for convolutional networks.
High-level Idea

We focus on robustness of $x$ for $p = \infty$

$$N_x^{\epsilon} = \left\{ x' \mid ||x' - x||_{\infty} < \epsilon \right\}$$

That is,

$$N_x^{\epsilon} = \left\{ x' \mid |x_0 - x'_0| < \epsilon \land \cdots \land |x_n - x'_n| < \epsilon \right\}$$

Goal: check if for all $x' \in N_x^{\epsilon}$, we have $f(x) = f(x')$

That is, check $|x_0 - x'_0| < \epsilon \land \cdots \land |x_n - x'_n| < \epsilon \land f(x) = f(x')$

We define a solver for determining whether this is satisfiable
Background: Theory and Interpretation

A theory is a signature $\Sigma$ and interpretation $I$ consisting of a domain and interpretation for the symbols in $\Sigma$

Example: $\Sigma = \{=, +, 0, 1, 2, 3, ...\}$ where $I$ interprets $\Sigma$ over the natural numbers: $=, +$ have the standard meaning and $0, 1, 2, 3, ...$ are interpreted to their corresponding natural numbers

Formulas over the theory: $\exists i. i + 1 = i \quad \exists i \forall j. i + j = j$
Satisfiability Modulo Theory (SMT)

An SMT problem is the decision problem of determining whether a logical formula in a certain theory is satisfiable.

For example, for $\Sigma = \langle =, +, 0, 1, 2, 3, ... \rangle$ where $I$ interprets $\Sigma$ over the natural numbers:

$\exists i \forall j. i + j = j$ is satisfiable

$\exists i. i + 1 = i$ is not satisfiable
Simplex

A solver for the SMT problem for the linear arithmetic theory

\[ \Sigma = \left( +, -, ;, \leq, \geq, =, 0, \frac{1}{1}, \frac{1}{2}, \ldots, \frac{2}{1}, \frac{2}{2}, \ldots \right) \]

Determines the satisfiability of a conjunction of formulas of the form

\[ \Sigma_{x_i \in X} c_i x_i \bowtie d_i \quad \bowtie \in \{\leq, \geq, =\} \]

\( X \) is a set of variables, \( c_i, d_i \) are constants
High-level Idea

Simplex is an iterative algorithm

At each iteration, Simplex has an assignment for all variables

Some of the variables satisfy their bounds, some violate

Each iteration fixes a violation by changing the assignment of a violating variable to its upper/lower bound

This may lead to another violation, which we try to fix later
Set up

For every formula $\Sigma_{x_i \in X} c_i x_i \bowtie d_i$, we create a new variable $b_i$:

$$\Sigma_{x_i \in X} c_i x_i = b_i$$

And add $d_i$ is a bound of $b_i$ (lower/upper/both)

Example: for $2x_1 + 3x_2 \geq 1$, we create $2x_1 + 3x_2 = b_1$ and $b_1 \geq 1$

We begin by assigning all variables to 0: $x_1 \rightarrow 0, x_2 \rightarrow 0, b_1 \rightarrow 0$
A matrix $T$ consisting of the coefficients in formulas of the form

$$\sum_{x_i \in X} c_i x_i$$

for $2x_1 + 3x_2 \geq 1$:

$$\begin{pmatrix} x_1 & x_2 \\ (2 & 3) \end{pmatrix}$$

Initially, each row represents one of the new variables, each column represents one of the original variables in $X$

$$\begin{pmatrix} x_1 & x_2 \\ b_1 & (2 & 3) \end{pmatrix}$$

$b_1 \geq 1$

$x_1 \to 0, x_2 \to 0, b_1 \to 0$
The tableau is changed in every iteration by replacing one of the column variables (called non-basic) with a row variable (basic)

- The formulas are updated accordingly
- The assignment to the variables is changed

An invariant of the execution is that the non-basic variables satisfy their bounds

- Initially, \( x_i \) are the non-basic and have no bounds
The Pivot Operation

While there is a basic variable violating its bound, pivot it with a non-basic variable

For example,

\[
\begin{array}{cc}
x_1 & x_2 \\
b_1 & (2 & 3) \\
\end{array}
\]

\[
b_1 \geq 1
\]

\[
x_1 \rightarrow 0, x_2 \rightarrow 0, b_1 \rightarrow 0
\]

We have \( b_1 \geq 1 \) but \( b_1 \rightarrow 0 \) and thus violates its bound

Pivot with \( x_1 \)
The Pivot Operation

For example, \[ \begin{align*} x_1 & \quad x_2 & \quad b_1 \geq 1 \\ b_1 & \quad (2 & \quad 3) & \quad x_1 \to 0, x_2 \to 0, b_1 \to 0 \end{align*} \]

\( b_1 = 0 < 1 \) and thus violates its bound, pivot with \( x_1 \)

Pivoting is done by changing the equation \( b_1 = 2x_1 + 3x_2 \)

\[ x_1 = \frac{b_1}{2} - \frac{3}{2}x_2 \]

resulting in

\[ \begin{align*} b_1 & \quad x_2 \\ x_1 & \quad (0.5 & \quad 1.5) \end{align*} \]

\[ b_1 \geq 1 \]

\[ x_1 \to 0, x_2 \to 0, b_1 \to 0 \]
The Pivot Operation

The assignment is then updated

The (old) non-basic variable is changed based on the basic variable’s bound (that caused the violation):

\[
\begin{align*}
x_1 & \rightarrow \frac{d_1 - b_1}{T[b_1, x_1]} = \frac{1 - 0}{2} = \frac{1}{2} \\
b_1 & \geq 1
\end{align*}
\]

The (old) basic variable is changed to match its bound \(d_1\):

\[
b_1 \rightarrow 1
\]

The new basic variable \((x_1)\) may violate its bounds now
Termination of Simplex

When all basic variables satisfy their bounds or when the pivot operation fails, completes

For $2x_1 + 3x_2 \geq 1$: $x_1 \rightarrow 0.5, x_2 \rightarrow 0, b_1 \rightarrow 1$

If all variables satisfy their bounds, the original conjunction of formulas is satisfiable

Otherwise, the conjunction is not satisfiable
A feed forward neural network computes a linear arithmetic functions followed by the ReLU

Define the initial variables to be the $p$-s and the $l$-s
Add constraints on $l_1^0$ (to capture $N_\mathbf{x}^\varepsilon$), on $l_1^2$ ($l_1^2 = f(x)$), and all the other equalities (e.g., $p_1^1 = w_{1,1}^1 l_1^0 + b_1^1$)
For the ReLU, add for each $l$ (except for $l_0$) a “dynamic” lower bound:

$$\max(0, l)$$

In each iteration, the value of the lower bound is determined by the assignment to $l$: it is either 0 or $l$
Recap

Reluplex is a solver for determining the satisfiability of conjunctions over $\geq, \leq, =, ReLU$

Can be used to prove robustness of inputs in feed forward neural networks
Summary

A white- and black-box attacks to find adversarial examples

Definition of robustness to express when a neighborhood of a given input is safe from adversarial examples

A verification procedure to check robustness of feed forward neural network with ReLU

Next time: AI²: Safety and Robustness Certification of Neural Networks with Abstract Interpretation