Adversarial Machine Learning per reti neurali per la classificazione di malware

Francesco Baraldi – 256745@studenti.unimore.it – mat. 172703

Introduzione

Gli ultimi enormi sviluppi nell'uso delle reti neurali hanno aperto la possibilità di sfruttare queste tecnologie anche nel campo della sicurezza informatica, in particolare è possibile utilizzarle per la classificazione di applicazioni e software in benigni e malevoli (malware).
L'*adversarial machine learning* studia le tecniche per compromettere il corretto funzionamento degli algoritmi di deep learning, nello specifico algoritmi di classificazione; in particolare si cerca di costruire dei samples, detti *adversarial samples*, con delle perturbazioni impercettibili rispetto a un sample originale, ma che siano in grado di far sbagliare l'algoritmo. Queste tecniche vengono applicate solitamente per reti neurali deep nell'ambito della visione artificiale e nella classificazione di immagini, questo perché l'alta entropia di un'immagine può essere facilmente sfruttata per cambiarne la natura senza avere variazioni che si possano percepire a livello visivo.
L'obiettivo di questo progetto è invece quello di approfondire un approccio di adversarial machine learning applicato all'utilizzo di reti neurali per la classificazione di malware.

Classificazione di malware

Il primo passo è stato quello di allenare delle reti neurali per classificare i malware, per farlo sono state usate più versioni di multi-layer perceptron (MLP), alcune più semplici e altre più complesse, al fine di valutarne l'efficacia nella classificazione e soprattutto la facilità con cui queste reti possono essere ingannate. Le reti utilizzare sono le seguenti:

- DNN1: [8, 8, 8]
- DNN2: [20, 10, 8, 8]
- DNN3: [20, 20, 10, 10, 10]
- DNN4: [50, 50]
- DNN5: [100, 100]

Ogni rete un input layer, un output layer e gli hidden layers strutturati come descritto sopra e fornisce in output un singolo valore, 0 se il software è classificato come malware, 1 altrimenti.
Il dataset[1] utilizzato per l'addestramento contiene circa 15 mila istanze di software, ognuna delle quali ha 215 features binarie, ciascuna indica se il software utilizza oppure no determinate funzioni del sistema operativo o se svolge determinate azioni; infine, ogni istanza è etichettata come benigna o malevola.

[1] *Android Malware Dataset for Machine Learning, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning? select=drebin-215-dataset-5560malware-9476-benign.csv*
[2] *Deep Neural Network and Mixed Integer Linear Optimization – Matteo Fischetti, Jason Jo*

Ogni rete è stata addestrata sul dataset[1] per 50 epoche, usando come ottimizzatore lo SGD e i risultati del training sono mostrati nella tabella 1:

	DNN1	DNN2	DNN3	DNN4	DNN5
Accuracy sul training set	97.4 %	**97.5 %**	**97.5 %**	97.4 %	97.4 %
Accuracy sul test set	97.0 %	97.0 %	97.1 %	97.1 %	**97.2 %**

Table 1: Risultati del training

Come si può notare, le differenze nelle performance delle 5 reti testate sono trascurabili utilizzando 50 epoche, la differenza sarà invece più evidente nella facilità con cui queste reti vengono ingannate.

Creazione di adversarial samples

L'approccio utilizzato per costruire degli adversarial samples è quello proposto da Fischetti e Jo[2], infatti, come descrivono gli autori, è possibile modellare una rete neurale con un *Mixed Integer Linear Program* (MILP) e di conseguenza sfruttare il problema di ottimizzazione per diversi scopi, tra cui attaccare la rete con un approccio di adversarial machine learning. Questo è possibile adattando il problema di ottimizzazione inserendo vincoli e funzione obiettivo adatti allo scopo. È importante notare che una condizione per cui questo metodo funzioni è che la funzione di attivazione usata dalla rete deve essere la *Rectified Linear Unit* (ReLU).

Il modello matematico che descrive la rete è il seguente:
Si considera una rete con $K + 1$ layer numerati da 0 a K, il layer 0 è di input mentre il layer K è di output. Ogni layer $k \in \{0, \dots, K\}$ ha n_k neuroni.
Variabili:

- $x^k \in \mathbb{R}^{n_k}$ è l'output del layer k, x_j^k è l'output del j-esimo neurone del layer k, con $k \in \{1, \dots, K\}$
- $s_j^k \in \mathbb{R}$ è la variabile slack per ogni neurone di ogni layer, con $k \in \{1, \dots, K\}$
- $z_j^k \in \{0,1\}$ è la variabile di attivazione per ogni neurone di ogni layer, con $k \in \{1, \dots, K\}$
- $error$ è la variabile che indica il numero di features cambiate rispetto all'input originale
- $output$ è la variabile di output della rete

Funzione obiettivo: $\min error$

[1] *Android Malware Dataset for Machine Learning, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning? select=drebin-215-dataset-5560malware-9476-benign.csv*
[2] *Deep Neural Network and Mixed Integer Linear Optimization – Matteo Fischetti, Jason Jo*

Vincoli:

$\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - s_j^k,\ k \in \{1, \dots K-1\}, j \in \{1, \dots, n_k\}$ (1)

$\sum_{i=1}^{n_0} (x_i^0 - input_i)^2 \leq error$ (2)

$\sum_{i=1}^{n_{K-1}} w_{i0}^{K-1} x_i^{K-1} + b_0^{K-1} = output$ (3)

$error \leq MAX\_ERROR$ (4)

$output \geq 0.55$ (5)

$z_j^k = 1 \rightarrow x_j^k \leq 0,\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (6)

$z_j^k = 0 \rightarrow s_j^k \leq 0,\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (7)

$x_j^k, s_j^k \geq 0,\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (8)

$z_j^k \in \{0,1\},\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (9)

$lb_j^k \leq x_j^k \leq ub_j^k,\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (10)

$\overline{lb_j^k} \leq s_j^k \leq \overline{ub_j^k},\ k \in \{1, \dots, K\}, j \in \{1, \dots, n_k\}$ (11)

Questo modello matematico rappresenta un generico multi-layer perceptron e genera un sample che sia il più simile possibile a un sample dato in input, infatti, minimizzando l'errore si modificano meno features possibili, soddisfacendo però il vincolo che l'output sia maggiore di 0.55, cioè che la rete classifichi il sample come benigno. In questo modo è possibile fornire al modello un'istanza che originariamente viene classificata dalla rete come malware e ottenerne una versione leggermente modificata ma che venga classificata dalla rete come benigna.
In particolare, con il vincolo (1) si rappresenta il funzionamento interno della rete neurale, il vincolo (4) impone che il numero delle features cambiate sia sotto una certa soglia, il vincolo (3) impone che l'uscita dell'output layer sia uguale alla variabile $output$, e il vincolo (5) impone che questa variabile sia maggiore di 0.55 e quindi che l'istanza sia classificata dalla rete come benigna. Infine i vincoli (6) e (7) sono gli *indicator constraints* e modellano la non linearità data dalla funzione di attivazione ReLU.

Un problema di ottimizzazione di questo tipo, anche usando i migliori solver moderni potrebbe presentare dei tempi di risoluzione molto alti fino a diventare inaccettabili a seconda della complessità della rete neurale considerata, per questo motivo diventano cruciali gli upper bound che vengono imposti alle variabili continue che rappresentano le uscite dei vari neuroni e le relative variabili slack. Al modello è stato quindi applicato il processo presentato da Fischetti e Jo[2] per il calcolo di upper bound migliori, in quanto quelli calcolati automaticamente dai solver non garantiscono un miglioramento delle prestazioni. L'algoritmo considerato consiste in un procedimento iterativo in cui per ogni neurone $UNIT(j,k)$, si eliminano tutti i layer successivi e tutti i neuroni del layer corrente escluso il neurone considerato, dopodiché si risolve il problema massimizzando le variabili x_j^k e s_j^k. I risultati ottenuti possono essere usati come upper bound per le rispettive variabili nel problema originale. È importante notare che questo procedimento è indipendente dall'input

[1] *Android Malware Dataset for Machine Learning, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning? select=drebin-215-dataset-5560malware-9476-benign.csv*
[2] *Deep Neural Network and Mixed Integer Linear Optimization – Matteo Fischetti, Jason Jo*

e quindi si può eseguire “una tantum” per ogni rete; inoltre anche imponendo dei limiti ai tempi di esecuzione, e quindi accettando soluzioni approssimate, si ottengono comunque risultati accettabili.

Implementazione

Le reti neurali sono state implementate usando Pytorch nel file “nets.py”, nel file “dataset.py” invece è presente la classe Pytorch per rappresentare il dataset che servirà per istanziare successivamente i dataloader. Mentre nel file “MILP.py” sono presenti le funzioni per i problemi di ottimizzazione, implementati usando il solver Gurobi, uno per calcolare gli upper bound, uno per il problema originale di creare un adversarial sample partendo da un’istanza di input. Infine nel file “main.py”, presente anche in versione jupyter notebook “main.ipynb”, viene eseguito tutto il codice necessario, prima per allenare le reti, in seguito per calcolare i bound ottimi e infine per creare delle istanze avversarie, testando prima il modello senza upper bound, e poi gli stessi test vengono eseguiti applicando al modello i bound ottimi calcolati in precedenza. Per ogni rete, dopo essere stata allenata la prima volta, sono stati salvati i parametri su file così da non doverle riallenare ogni volta, e questi sono presenti nella cartella “params/”. Gli upper bound di ogni rete sono stati anch’essi salvati nella cartella “bounds/”, in questo modo a ogni esecuzione è possibile leggerli da file evitando di rieseguire il problema di ottimizzazione. Infine, il modello per la creazione di adversarial sample è stato testato per 50 input diversi e i risultati sono stati salvati su file in forma testuale nella cartella “outputs/”.

Risultati

I test sono stati eseguiti ponendo dei limiti di tempo all’esecuzione dei modelli, in particolare di 10 secondi per il calcolo degli upper bound, e di 2 minuti per il calcolo degli adversarial sample. In tabella 2 sono riportati i risultati del modello senza upper bound, mentre in tabella 3 quelli del modello con upper bound. Dai risultati si deduce che il modello base, quindi senza upper bound, non ha problemi per le reti più semplici ma incontra difficoltà con quelle più complesse, all’aumentare del numero di layer e del numero di neuroni per ogni layer. Il modello ottimizzato invece risolve tutte le istanze entro il limite di tempo imposto per le prime quattro reti, e mostra un lieve calo di performance solamente per l’ultima e più complessa rete, ma anche in questo caso ha delle prestazioni ottime. Inoltre, a prescindere dalla rete, il modello a cui sono stati applicati gli upper bound garantisce tempi di esecuzione molto inferiori.

[1] *Android Malware Dataset for Machine Learning, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning? select=drebin-215-dataset-5560malware-9476-benign.csv*
[2] *Deep Neural Network and Mixed Integer Linear Optimization – Matteo Fischetti, Jason Jo*

	Opt. Solved (%)	Avg. Time (s)	Avg. Gap (%)
DNN1	100.0	0.171	0.0
DNN2	100.0	1.931	0.0
DNN3	70.0	59.522	9.4
DNN4	92.0	24.904	1.6
DNN5	52.0	79.643	22.033

Table 2: Modello base

	Opt. Solved (%)	Avg. Time (s)	Avg. Gap (%)
DNN1	100.0	0.043	0.0
DNN2	100.0	0.480	0.0
DNN3	100.0	1.873	0.0
DNN4	100.0	3.019	0.0
DNN5	92.0	25.188	2.067

Table 3: Modello ottimizzato

Conclusioni

Il procedimento studiato risulta molto flessibile per modellare una rete neurale perché permette di approcciare diversi problemi semplicemente cambiando vincoli e funzione obiettivo in base al caso specifico. Un aspetto da tenere sempre presente però è quello della complessità e quindi dei tempi di esecuzione che possono aumentare molto all'aumentare della complessità dell'architettura della rete neurale. A questo proposito è fondamentale, come dimostrato dagli esperimenti, considerare gli upper bound per le variabili continue, in quanto l'utilizzo di upper bound corretti migliora di molto le prestazioni del modello.

[1] *Android Malware Dataset for Machine Learning, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning? select=drebin-215-dataset-5560malware-9476-benign.csv*

[2] *Deep Neural Network and Mixed Integer Linear Optimization – Matteo Fischetti, Jason Jo*