UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE

Corso di Laurea Triennale in Informatica

Progettazione di un sistema di intelligenza artificiale per il riconoscimento e la classificazione di malware

Design of an artificial intelligence system for malware detection and classification

CANDIDATO:
Alessio Russo

MATRICOLA:
317228

RELATORE:
Prof.ssa Eleonora Iotti

CORRELATORE:
Prof. Roberto Alfieri

Anno Accademico 2023–2024

Dedico questa tesi ai miei genitori che, con fiducia e costanza, mi hanno sostenuto durante questo percorso. Grazie per aver creduto in me e per il vostro supporto.

Indice

1 Introduzione **1**

2 Virus Informatici **3**

2.1 Malware di vecchia e di nuova generazione 3
2.2 Tipologie di malware 4
2.2.1 Virus e worm 4
2.2.2 Trojan e rootkit 5
2.2.3 Spyware e keylogger 5
2.2.4 Downloader e backdoor 5
2.2.5 Ransomware 6
2.3 Riconoscimento di malware 6
2.3.1 Analisi teorica del problema 6
2.3.2 Tecniche di offuscamento 8
2.3.3 Analisi statica e dinamica 9
2.3.4 Analisi basata su firme 10
2.3.5 Analisi comportamentale 11
2.3.6 Analisi basata su machine learning 11

3 Reti Neurali Artificiali **13**

3.1 Reti neuronali e neurali 13
3.2 Neurone artificiale 14
3.3 Tipologie di addestramento 16
3.3.1 Addestramento supervisionato 16
3.3.2 Addestramento per rinforzo 17
3.3.3 Addestramento non supervisionato 18
3.4 Reti feed-forward 18
3.5 Single-Layer Perceptron 19
3.6 Multi-Layer Perceptron 22
3.6.1 Addestramento della rete 23
3.6.2 Discesa del gradiente 24
3.6.3 Algoritmo di backpropagation 25
3.6.4 Teorema di approssimazione universale 28

3.7 Reti neurali e classificatori 28

4 Progettazione 31

4.1 Dipendenze del sistema 31
4.2 Riconoscimento di malware 31
4.2.1 Caricamento e pre-elaborazione dei dati 32
4.2.2 Generazione del seed 33
4.2.3 Suddivisione del dataset in training e test set 33
4.2.4 Standardizzazione delle features 33
4.2.5 Creazione dei data loader 34
4.2.6 Definizione del modello 35
4.2.7 Funzione di loss e ottimizzatore 36
4.2.8 Addestramento del modello 37
4.2.9 Analisi dell'addestramento 39
4.2.10 Valutazione del modello 40
4.3 Classificazione di malware 41
4.3.1 Famiglie di malware 41
4.3.2 Estrazione delle features 43
4.3.3 Operazioni preliminari 43
4.3.4 Definizione del modello 44
4.3.5 Funzione di loss e ottimizzatore 44
4.3.6 Addestramento del modello 45
4.3.7 Analisi dell'addestramento 46
4.3.8 Valutazione del modello 47

5 Risultati 49

5.1 Strumenti per l'analisi 49
5.1.1 Accuracy 50
5.1.2 Precision 50
5.1.3 Recall 50
5.1.4 Confusion Matrix 50
5.2 Riconoscimento di malware 50
5.3 Classificazione di malware 52
5.4 Analisi di DikeDataset 53

6 Conclusioni 57

6.1 Discussione dei risultati 57
6.2 Sviluppi futuri 58

Bibliografia 59

A Portable Executable 63

Elenco delle figure

1.1 Percentuale dei file analizzati da VirusTotal (5 Giu-12 Giu, 2024) 2

3.1 Rappresentazione semplificata di un neurone biologico [1] 13
3.2 Esempio di classificazione binaria [1] 16
3.3 Esempio di regressione [1] 17
3.4 Esempio di clustering [1] 18
3.5 Rete neurale feed-forward 19
3.6 Single-Layer Perceptron con quattro neuroni nello strato di input e uno nello strato di output 20
3.7 Output del neurone al variare della preattivazione [1] 21
3.8 Classi linearmente separabili [1] 21
3.9 Multi-Layer Perceptron con uno strato nascosto [2] 22

4.1 Quantità di file benigni (0) e malware (1) in CLaMP 32
4.2 Grafico che mostra l'andamento della funzione di perdita durante il processo di addestramento 39
4.3 Grafico che mostra la precisione del modello durante il processo di addestramento 39
4.4 Distribuzione dei malware in BIG 2015 43
4.5 Grafico che mostra l'andamento della funzione di perdita durante il processo di addestramento 47
4.6 Grafico che mostra la precisione del modello durante il processo di addestramento 47

5.1 Confusion matrix generata dal modello per il riconoscimento di malware 51
5.2 Confusion matrix generata dal modello per la classificazione di malware 52
5.3 Quantità di file benigni (0) e malware (1) in DikeDataset 53
5.4 Confusion matrix generata dal testing su DikeDataset 54
5.5 Distribuzione predetta per i malware di DikeDataset 55

Elenco degli algoritmi

1 Algoritmo di discesa del gradiente 25
2 Algoritmo di addestramento per un $(n+1, m+1, 1)$-MLP . . . 27
3 Caricamento e pre-elaborazione del dataset CLaMP 32
4 Generazione del seed 33
5 Suddivisione di CLaMP in training e test set 33
6 Standardizzazione delle features di CLaMP 33
7 Creazione dei tensori 34
8 Creazione dei Dataset e DataLoader 35
9 $(52, 32, 16, 1)$-MLP utilizzato per il riconoscimento di malware . 35
10 Definizione della rete, della loss function e dell'ottimizzatore . . 37
11 Addestramento del modello 38
12 Valutazione del modello 40
13 $(66, 32, 16, 9)$-MLP utilizzato per la classificazione di malware . . 44
14 Definizione della rete, della loss function e dell'ottimizzatore . . 45
15 Addestramento del modello 46
16 Valutazione del modello 48

Capitolo 1

Introduzione

Nell'era digitale moderna, la sicurezza informatica rappresenta una delle principali sfide a livello globale. Con l'aumento esponenziale delle minacce cibernetiche, la protezione dei sistemi informativi e dei dati sensibili è diventata una priorità assoluta per le organizzazioni pubbliche e private. Tra le varie minacce, i software malevoli (anche detti **malware**, abbreviazione del termine inglese *malicious software*) rappresentano una preoccupazione significativa per la loro capacità di diffondersi e danneggiare i sistemi informatici. In generale, qualsiasi software che esegue intenzionalmente azioni dannose sui dispositivi delle vittime, come computer o smartphone, è considerato malware.

Si stima che circa 1 milione di nuovi malware vengano creati ogni giorno [3] e che i danni causati dal crimine informatico raggiungeranno i 10,5 mila miliardi di dollari entro il 2025 [4]. Se fosse misurato come un Paese, il crimine informatico sarebbe la terza più grande economia del mondo, dopo Stati Uniti e Cina, superando la ricchezza di intere nazioni.

Cybersecurity Ventures, un noto gruppo di ricerca specializzato nello studio della cybereconomia globale, prevede che i **ransomware**, un particolare tipo di malware, capace di auto-installarsi sui computer delle vittime, crittografandone i file e richiedendo un riscatto per la loro restituzione, costeranno alle vittime circa 265 miliardi di dollari all'anno entro il 2031, con un nuovo attacco (ad utenti o aziende) ogni 2 secondi [4].

Per proteggere utenti e dispositivi, è fondamentale sviluppare sistemi capaci di determinare se un dato programma ha intenti dannosi. Il presente lavoro di tesi si concentra dunque sulla progettazione di un sistema basato su reti neurali per il riconoscimento e la classificazione di malware nei file Portable Executable.

La scelta di concentrare l'analisi sul formato **Portable Executable** (spesso abbreviato in PE) è motivata da due fattori. Innanzitutto, il formato PE è quello utilizzato dai file eseguibili, dai file oggetto e dalle librerie del sistema operativo Microsoft Windows, che già nel 2020 aveva raggiunto un miliardo di dispositivi attivi [5]. In secondo luogo, come mostrato in Figura 1.1, circa il 43% dei file inviati a VirusTotal (un noto antivirus online che permette l'analisi di file e URL di proprietà di Google Inc.) sono file PE [6].

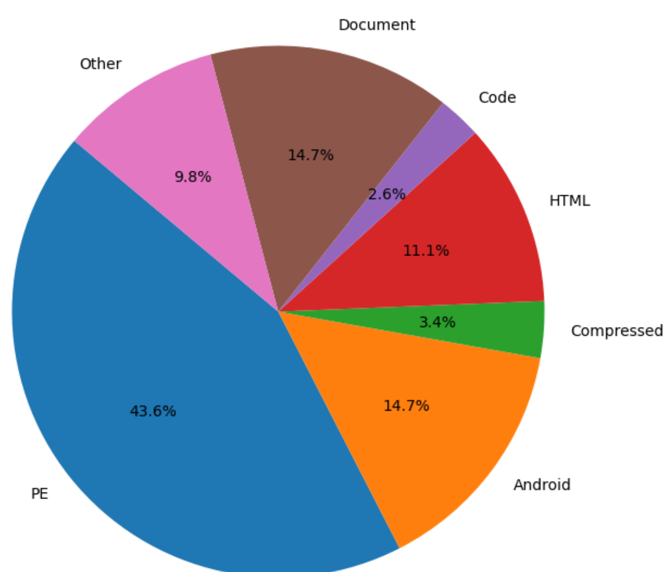

Figura 1.1: Percentuale dei file analizzati da VirusTotal (5 Giu-12 Giu, 2024)

Il resto del documento è perciò organizzato come segue: nel Capitolo 2 verrà fornita una panoramica sui virus informatici, evidenziando le principali sfide nella loro rilevazione. Nel Capitolo 3 verranno presentati i concetti fondamentali dell'intelligenza artificiale, con un approfondimento sul funzionamento delle reti neurali multistrato. Il Capitolo 4 descriverà la metodologia adottata per la progettazione del sistema, inclusi i dettagli relativi alla raccolta dei dati, alla selezione delle caratteristiche e all'architettura dei modelli. Il Capitolo 5 illustrerà i risultati degli esperimenti condotti. Infine, nel Capitolo 6 saranno discussi i risultati ottenuti e le possibili direzioni future del progetto.

Capitolo 2

Virus Informatici

Come già anticipato, un malware è qualsiasi software progettato per interrompere l'attività di un sistema informatico, raccogliere informazioni sensibili o accedere a risorse private. Ciò che contraddistingue un malware è quindi la sua **intenzione** malevola. Il termine **badware** si riferisce invece sia ai programmi intenzionalmente dannosi, sia ai software che causano danni in modo non intenzionale [7].

2.1 Malware di vecchia e di nuova generazione

Una prima e importante distinzione da fare è quella tra i malware tradizionali e quelli di nuova generazione. Inizialmente, i software malevoli venivano eseguiti in modalità utente, erano costituiti da un singolo processo e non utilizzavano tecniche complesse per nascondersi. Questo tipo di software, generalmente progettato per svolgere compiti semplici, viene spesso definito **malware tradizionale** [3]. Al giorno d'oggi, però, i malware vengono sviluppati per essere eseguiti in modalità kernel.

La principale differenza tra i programmi eseguiti in modalità utente e quelli in modalità kernel risiede nel livello di accesso e controllo che ciascuno di essi ha sul sistema operativo e sull'hardware del computer.

I programmi eseguiti in modalità utente operano in un ambiente sicuro, con accesso limitato alle risorse. Questa modalità di funzionamento è progettata per proteggere l'integrità del sistema ed impedire che applicazioni malfunzionanti o malevole possano danneggiare il sistema operativo o influenzare altri programmi. In modalità utente, i programmi non possono accedere direttamente all'hardware né alle aree protette della memoria, ma devono passare attraverso il sistema operativo, utilizzando API e servizi di sistema.

Al contrario, i programmi eseguiti in modalità kernel hanno pieno accesso a tutte le risorse del sistema, inclusi l'hardware e la memoria protetta. La modalità kernel è il livello più privilegiato del sistema operativo, in cui vengono eseguite operazioni fondamentali come la gestione della memoria, il controllo dei processi e la comunicazione con i dispositivi hardware. I programmi eseguiti in modalità kernel, come i driver di dispositivo e alcune parti del sistema operativo, possono eseguire qualsiasi istruzione e accedere a qualsiasi area di memoria.

Questa nuova classe di software, detta dei **malware di nuova generazione** [3], oltre ad essere potenzialmente più distruttiva, spesso attiva simultaneamente diversi processi ed è più difficile da rilevare. Grazie all'uso di tecniche di offuscamento (vedi paragrafo 2.3.2 *Tecniche di offuscamento*), infatti, riesce a eludere facilmente i software di protezione, come firewall e antivirus, diventando persistente nel sistema.

2.2 Tipologie di malware

Esistono diversi tipi di malware, classificabili in base al modo in cui interagiscono con il dispositivo della vittima. Tuttavia, occorre sottolineare che con l'avvento dei malware di nuova generazione, la classificazione sta diventanto sempre più complessa, poiché questi spesso presentano caratteristiche riconducibili a più di una tipologia di malware.

2.2.1 Virus e worm

I **virus** [7] sono programmi progettati per diffondersi senza che l'utente ne sia a conoscenza. In particolare, i virus sono in grado di auto-eseguirsi (e.g. inserendo il proprio codice nel percorso di esecuzione di un altro programma) e auto-replicarsi (e.g. sostituendo ad altri file eseguibili una copia di se stesso). Alcuni virus sono programmati per danneggiare il computer distruggendo programmi, cancellando file o riformattando il disco rigido. Altri, invece, non sono progettati per danneggiare direttamente il sistema, ma semplicemente per replicarsi e farsi notare attraverso la presentazione di messaggi testuali, video e audio. Anche questi virus, apparentemente innocui, riescono però a danneggiare l'utente: occupando grandi quantità di memoria, possono infatti provocare crash di sistema o perdite di dati.

I **worm** [7] sono invece progettati per diffondersi attraverso Internet, replicandosi sui dispositivi collegati alla rete. I worm si diffondono principalmente

tramite la posta elettronica, e spesso si nascondono all'interno di altri file, come documenti Word o Excel, cercando indirizzi e-mail memorizzati nel computer ospite e inviando copie di sé stessi come allegati agli indirizzi raccolti.

Spesso i worm fungono da veicolo per l'installazione automatica su un gran numero di macchine di altri malware, come backdoor o keylogger. Questa tecnica prende il nome di **dropping**.

2.2.2 Trojan e rootkit

I **trojan horse** [7] (o semplicemente *trojan*) si presentano come software legittimi per ingannare gli utenti e indurli a installarli. Una volta eseguiti, permettono ai malintenzionati di ottenere accesso non autorizzato ai sistemi. In genere, i trojan non sono in grado di auto-replicarsi.

I **rootkit** [7] sono software progettati per nascondere l'esistenza di determinati processi ai metodi di rilevamento tradizionali. L'obiettivo dei rootkit è ottenere privilegi di amministratore sul sistema infetto, consentendo di coprire la presenza di altri tipi di malware.

2.2.3 Spyware e keylogger

Gli **spyware** [7] sono malware utilizzati in attacchi malevoli come furto di identità, phishing e ingegneria sociale, minacce progettate per rubare denaro agli utenti, alle aziende e alle banche. Gli spyware tengono traccia delle attività svolte sul computer, come i siti web visitati o i programmi utilizzati.

Talvolta, i dati raccolti dagli spyware vengono utilizzati da un particolare tipo di malware, detto **adware**, che mostra agli utenti pubblicità indesiderate.

I **keylogger** [7] sono invece un tipo particolare di spyware, il cui obiettivo è registrare i tasti premuti sulla tastiera. Questi programmi vengono generalmente utilizzati per raccogliere dati sensibili, come password e informazioni bancarie.

2.2.4 Downloader e backdoor

I **downloader** sono un tipo di malware progettato per scaricare e installare altri malware su un sistema infetto. Di solito, un downloader viene distribuito attraverso email di phishing, siti web compromessi o allegati dannosi. Una volta eseguito, il downloader si connette a un server remoto e scarica altri tipi di malware, aumentando così il livello di compromissione del sistema. I

downloader sono spesso utilizzati nelle prime fasi di un attacco informatico, per stabilire una presenza iniziale e preparare il terreno per ulteriori azioni malevole.

L'attività dei downloader è spesso legata a quella delle **backdoor**, spesso installate da questo tipo di malware, utilizzate per ottenere l'accesso a un sistema informatico, a una rete o a un'applicazione bypassando le normali procedure di autenticazione. La presenza di una backdoor rappresenta una grave vulnerabilità per la sicurezza di un sistema.

2.2.5 Ransomware

Già introdotti nel Capitolo 1, i **ransomware** sono un tipo di malware che crittografa i file presenti su un computer o su una rete, rendendoli inaccessibili per l'utente. Dopo aver completato la crittografia, viene spesso richiesto un riscatto (in inglese *ransom*) in cambio della chiave di decrittazione necessaria a ripristinare l'accesso ai dati. Spesso, il pagamento del riscatto viene richiesto in criptovaluta per garantire l'anonimato degli estorsori. Gli attacchi ransomware possono avere gravi conseguenze per individui, aziende e istituzioni, causando perdite finanziarie e interruzioni operative significative.

2.3 Riconoscimento di malware

Questo paragrafo esamina il problema del rilevamento di malware, un aspetto cruciale per la sicurezza informatica.

2.3.1 Analisi teorica del problema

Poiché il primo malware registrato apparteneva alla famiglia dei virus, spesso i primi approcci teorici al problema si concentrano sulla rilevazione dei virus. In particolare, è dimostrato che la rilevazione dei virus è *NP-completa* [3].

Problemi NP-completi

I problemi NP-completi sono una classe particolare di problemi. Per comprendere cosa siano, è utile introdurre alcuni concetti alla base della teoria della complessità:

- **Classe P.** Questa classe include tutti i problemi decisionali (ovvero i problemi con risposta affermativa o negativa) che possono essere risolti in tempo polinomiale da un algoritmo deterministico. In altre parole,

esiste un algoritmo che può risolvere questi problemi in un tempo che è una funzione polinomiale della dimensione dell'input.

- **Classe NP.** Questa classe include tutti i problemi decisionali che possono essere risolti in tempo polinomiale da un algoritmo non deterministico. Le soluzioni dei problemi di classe NP sono generalmente difficili da determinare, mentre possono essere verificate in tempo polinomiale da un algoritmo deterministico. In altre parole, data una possibile soluzione, possiamo verificarne la correttezza rapidamente (in tempo polinomiale).

- **NP-completezza.** Un problema è NP-completo se 1) appartiene ad NP e 2) ogni problema in NP può essere ridotto ad esso in tempo polinomiale. Questo significa che se trovassimo un algoritmo in tempo polinomiale per risolvere un problema NP-completo, potremmo usarlo per risolvere qualsiasi altro problema in NP in tempo polinomiale.

Risolvere i problemi NP-completi è una sfida centrale nella teoria della complessità computazionale. Poiché non si conosce ancora un algoritmo in tempo polinomiale che possa risolvere qualsiasi problema NP-completo (e non è noto se tale algoritmo esista), le strategie per affrontare questi problemi variano in base alle esigenze pratiche e alla natura specifica del problema. Di seguito sono riportate le principali strategie utilizzate per la risoluzione dei problemi NP-completi:

- **Algoritmi esatti.** Tecniche come il backtracking o la programmazione dinamica permettono di individuare soluzioni ottime, ma in generale hanno complessità esponenziale nel peggiore dei casi.

- **Algoritmi di approssimazione.** Questi algoritmi non trovano necessariamente la soluzione ottimale, ma garantiscono una soluzione vicina all'ottimale in tempo polinomiale (e.g. gli algoritmi greedy prendono decisioni locali ottimali nella speranza di trovare una soluzione globale vicina all'ottimale).

- **Algoritmi euristici.** Questi algoritmi non garantiscono né ottimalità né una soluzione vicina all'ottimale, ma possono trovare buone soluzioni in tempi ragionevoli per casi reali.

- **Algoritmi randomizzati.** Questi algoritmi utilizzano la randomizzazione per esplorare lo spazio delle soluzioni e possono offrire buone soluzioni in tempi ragionevoli (e.g. algoritmi Monte Carlo, che forniscono soluzioni corrette con alta probabilità, e algoritmi Las Vegas, sempre corretti ma con tempo di esecuzione casuale).

Problema del rilevamento

Tornando al problema del rilevamento di malware, come affermato da Ö. Aslan e R. Samet [3], e dimostrato da F. Cohen [8], la rilevazione dei virus informatici è **indecidibile** poiché il processo stesso di rilevamento contiene una contraddizione.

Se il problema del riconoscimento di virus viene trattato come un problema decisionale, D (il decisore) dovrà stabilire se P è un virus o meno. Secondo Cohen, non è possibile determinare se P è un virus perché, se P fosse un virus, verrebbe identificato come tale da D, e quindi non sarebbe in grado di apportare modifiche ad altri programmi, non comportandosi più come un virus. Se invece il decisore D non riconoscesse P come un virus, P interagirebbe con altri programmi per diffondersi e infettarli.

Questo processo decisionale comporta quindi una contraddizione, rendendo impossibile identificare P come un virus.

Secondo M. Chess e R. White, inoltre, non esiste un programma che rilevi tutti i virus senza falsi positivi (FP), poiché i virus sono polimorfici (vedi paragrafo successivo) e possono esistere in diverse forme [9].

2.3.2 Tecniche di offuscamento

Nella pratica, sebbene siano state sviluppate diverse tecniche per identificare e neutralizzare questo tipo di minacce, nessuna di queste si è rivelata capace di contrastare tutte le tipologie di attacco esistenti. Questo perché i malware di nuova generazione spesso utilizzano diverse tecniche di offuscamento che complicano il processo di rilevamento. Le più comuni **tecniche di offusamento** [3] sono:

- **Crittografia.** I malware talvolta utilizzano la crittografia per nascondere i blocchi di codice malevolo (il cosidetto **payload**) all'interno del codice complessivo.

- **Oligomorfismo.** Spesso vengono utilizzate tecniche di crittografia asimmetrica per crittografare e decrittografare la sezione malevola del codice, complicando ulteriormente il rilevamento del malware.

- **Polimorfismo.** In maniera analoga al metodo oligomorfico, viene utilizzata la crittografia asimmetrica per codificare parte del codice. In questo caso, però, la parte malevola del codice viene crittografata a strati.

- **Metamorfismo.** Anziché utilizzare la crittografia, viene adoperata una tecnica di offuscamento del codice dinamica, in cui l'opcode (operation code, codice operativo), una parte delle istruzioni in linguaggio macchina che specifica l'operazione che deve essere eseguita, cambia a ogni iterazione quando il processo malevolo viene eseguito. È molto difficile rilevare questo tipo di malware perché ogni nuova copia ha una firma completamente diversa.

- **Stealth.** La tecnica stealth consente al malware di monitorare, grazie ad una parte di esso che rimane costantemente in memoria, le chiamate dei programmi ad alcune funzioni del sistema operativo. In questo modo, ad esempio, un malware può rilevare il tentativo di un programma di analizzare un file infetto, e di ripulire preventivamente quel file dal suo codice, per poi reinfettarlo nuovamente. In questo caso, però, il rovescio della medaglia è che i software antivirus possono facilmente rilevare la parte di codice virale costantemente presente in memoria.

- **Packaging.** Il packaging è una tecnica di offuscamento che consiste nel comprimere il malware per impedirne il rilevamento.

2.3.3 Analisi statica e dinamica

Il processo di analisi dei malware consiste nell'identificare se un dato programma sia intenzionalmente malevolo, comprenderne il funzionamento, determinare quali dispositivi o software bersaglia e quali dati tenta di sottrarre o danneggiare. Esistono principalmente due approcci per questa analisi: statico e dinamico.

L'**analisi statica** prevede di esaminare il malware senza eseguirlo. Tuttavia, le tecniche di offuscamento possono rendere questa analisi costosa o inaffidabile, motivo per cui si è sviluppata l'analisi dinamica.

L'**analisi dinamica**, invece, consiste nello studio del comportamento di un programma dannoso durante la sua esecuzione in un ambiente controllato (e.g. una macchina virtuale, un simulatore, un emulatore o una sandbox). Questo approccio è più efficace rispetto all'analisi statica poiché rivela il comportamento reale del malware. Tuttavia, l'analisi dinamica è dispendiosa in termini di tempo e risorse, creando problemi di scalabilità. Inoltre, l'ambiente virtuale in cui questo viene eseguito differisce da quello reale, e il suo comportamento potrebbe essere diverso da quello reale. In alcuni casi, infatti, il malware potrebbe attivare la sezione malevola del suo codice solo sotto determinate con-

dizioni (e.g. una connessione internet attiva) che potrebbero non verificarsi in un ambiente virtuale.

2.3.4 Analisi basata su firme

Una delle tecniche tradizionali per l'*analisi statica* dei malware è quella basata su firme [3]. La **firma** di un programma è un codice, che dipende dalla sua struttura e che permette di identificarlo in modo univoco. Questa tecnica consiste nel confrontare le firme dei file sospetti con un database di firme di malware conosciuti. Particolarmente efficace nel rilevare malware già noti, presenta però una significativa limitazione: non è in grado di identificare nuove minacce o varianti sconosciute fino a quando le firme non vengono aggiornate.

L'approccio basato su firma è ampiamente utilizzato negli antivirus commerciali.

Generazione della firma

Il processo di generazione della firma inizia con l'estrazione di alcune caratteristiche fondamentali del malware. Queste informazioni vengono poi utilizzate per generare la firma, che viene memorizzata in un database dedicato. Quando un programma campione deve essere classificato come malware o benigno, la firma del campione viene estratta nello stesso modo e confrontata con quelle presenti nel database. Esistono diverse tecniche per creare una firma, tra cui:

- **Scansione delle stringhe.** Confronta la sequenza di byte nel file analizzato con quelle precedentemente salvate nel database. Le firme dei byte sono state ampiamente utilizzate dagli scanner antivirus per molti anni e spesso vengono usate per rilevare malware appartenenti alla stessa famiglia ma con firme diverse.

- **Scansione di testa e coda.** Invece di analizzare l'intero file, vengono presi solo i punti iniziali e finali del file per creare determinate firme. Questo metodo è molto conveniente per rilevare virus che si attaccano all'inizio e alla fine dei file.

- **Scansione del punto di ingresso**: Il punto di ingresso di un file indica dove inizia l'esecuzione quando il file viene avviato. Il malware di solito modifica il punto di ingresso di un programma, in modo che il codice dannoso venga eseguito prima del codice legittimo. Pertanto, alcuni malware possono essere rilevati estraendo la firma dalle sequenze nei punti di ingresso del programma.

- **Controllo dell'integrità (Firme Hash)**: Questo metodo genera un checksum crittografico, come MD5 o SHA-256, per ciascun file in un sistema a intervalli regolari. È utilizzato per identificare possibili modifiche causate da malware.

2.3.5 Analisi comportamentale

Una forma di *analisi dinamica* spesso utilizzata è l'analisi comportamentale [3], che osserva il comportamento dei programmi in esecuzione. L'approccio di rilevamento dei malware basato sul comportamento osserva i comportamenti dei programmi tramite strumenti di monitoraggio per determinare se il programma è malware o benigno. Sebbene il codice del programma possa essere modificato, il comportamento del programma rimarrà simile; pertanto, la maggior parte dei nuovi malware può essere rilevata con questo metodo. Tuttavia, alcuni malware non funzionano correttamente in ambienti protetti (macchine virtuali, sandbox), il che può portare a una classificazione errata dei campioni di malware come benigni.

2.3.6 Analisi basata su machine learning

Le tecniche di machine learning e intelligenza artificiale stanno diventando sempre più popolari nel rilevamento di malware. Queste tecniche addestrano modelli su grandi quantità di dati per identificare modelli e anomalie che indicano la presenza di malware. I vantaggi di queste tecniche includono la capacità di rilevare minacce sconosciute e la loro adattabilità a nuove forme di attacchi. Tuttavia, l'implementazione di soluzioni di machine learning può essere complessa e richiedere risorse significative per la raccolta e l'analisi dei dati.

Si noti che l'obiettivo di questo progetto di tesi, come verrà approfondito nei capitoli successivi, è proprio quello di sviluppare una tecnica di analisi, basata sul machine learning, che combina la capacità delle analisi dinamiche di rilevare malware sconosciuti con la praticità delle analisi statiche, che non richiedono la creazione di ambienti protetti per l'esecuzione dei malware, e che possono quindi essere applicate direttamente sui dispositivi degli utenti.

Capitolo 3

Reti Neurali Artificiali

In questo capitolo verranno esplorati alcuni concetti fondamentali del machine learning, partendo dalle definizioni basilari e proseguendo con l'analisi delle principali tecniche e algoritmi utilizzati nel settore. Successivamente, si approfondirà l'argomento delle reti neurali multistrato, uno strumento chiave che costituirà la base del sistema presentato nel seguito.

3.1 Reti neuronali e neurali

Dal punto di vista biologico, il cervello è un organo formato da particolari cellule, dette **neuroni**. Poiché i neuroni sono connessi tra loro, è possibile descrivere il cervello come una rete di neuroni, detta **rete neuronale**. In particolare, la rete neuronale che forma un cervello umano è caratterizzata da un enorme numero di neuroni (nell'ordine di 10^{11}), quasi tutti uguali tra loro (è possibile individuare circa 20 tipi di neuroni diversi), tra loro collegati mediante un numero relativamente basso di connessioni (in media ogni neurone è collegato a mille altri neuroni).

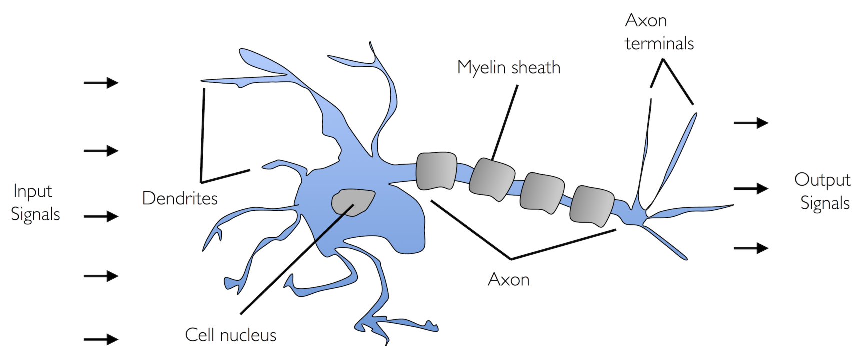

Figura 3.1: Rappresentazione semplificata di un neurone biologico [1]

La Figura 3.1 mostra una schematizzazione delle parti principali di un neurone.

Le connessioni tra neuroni sono fondamentali, poiché consentono di trasportare informazioni attraverso la rete, e ogni neurone riceve e invia informazioni da e per pochi altri neuroni, con un tempo di reazione dell'ordine di 1 ms (quindi, abbastanza lentamente se paragonato alle prestazioni di un calcolatore).

Il trasporto di informazioni, e quindi la comunicazione tra i neuroni, avviene attraverso l'interazione tra i dendriti e le sinapsi di diversi neuroni. In altre parole, ogni neurone possiede una struttura relativamente semplice capace di elaborare le informazioni trasportate dagli ioni tramite dendriti e sinapsi.

Una **rete neurale** (o *rete neurale artificiale*, Artificial Neural Network, ANN) è uno strumento software capace di simulare una rete di neuroni operante secondo gli stessi principi delle reti neuronali del cervello umano. Tuttavia, una rete neurale è generalmente molto più semplice rispetto a una rete neuronale biologica.

Quindi, sebbene si possa affermare che una rete neurale simula il comportamento del cervello umano, queste reti sono progettate per scopi specifici e solo in minima parte mirano a riprodurre i fenomeni neurofisiologici.

3.2 Neurone artificiale

In particolare, una rete neurale simula il comportamento di una rete neuronale replicando il funzionamento dei singoli neuroni. Un **neurone artificiale** (noto anche come *unità di McCulloch-Pitts*, o semplicemente *neurone*) è un modello matematico e computazionale che tenta di emulare alcune delle funzioni fondamentali di un neurone biologico.

Il neurone artificiale riceve uno o più input (analogamente ai dendriti nel neurone biologico), esegue una qualche forma di computazione su questi input, e poi produce un output (simile all'impulso nervoso in un neurone biologico). In particolare, un neurone artificiale implementa la seguente funzione reale di $n \in \mathbb{N}_+$ variabili reali:

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{3.1}$$

dove $\hat{y} \in \mathbb{R}$ è il valore prodotto dall'unità, talvolta indicato come $f(x)$, a fronte degli n ingressi reali $(x_i)_{i=1}^n$ utilizzando n coefficienti reali $(w_i)_{i=1}^n$ detti **pesi**, un coefficiente reale b detto **bias** e una funzione $\sigma : \mathbb{R} \to \mathbb{R}$ detta **funzione**

di attivazione. Normalmente, la funzione di attivazione di un neurone non è un'arbitraria funzione reale di variabile reale, ma viene scelta tra una serie di funzioni di uso comune, tra cui

Funzione identità

$$Id(x) = x \tag{3.2}$$

Funzione ReLU (Rectified Linear Unit)

$$R(x) = \max\{0, x\} \tag{3.3}$$

Funzione di Heaviside

$$H(x) = \begin{cases} 1 \text{ se } x \geq 0 \\ 0 \text{ se } x < 0 \end{cases} \tag{3.4}$$

Funzione logistica (o sigmoide)

$$S(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

Una rete neurale è un grafo orientato pesato in cui i nodi sono neuroni artificiali, mentre i pesi sugli archi rappresentano i pesi associati agli ingressi dei neuroni. Solitamente, gli autoanelli non sono permessi in una rete neurale per evitare i ritardi causati dai singoli neuroni.

Data una rete neurale con $n \in \mathbb{N}_+$ neuroni, per qualsiasi istante $t \in \mathbb{R}$, con $t \geq 0$, il vettore $s(t) \in \mathbb{R}^n$, che rappresenta le uscite correnti dei neuroni, è definito **stato della rete** all'istante t.

Poiché spesso tutti i neuroni di una rete utilizzano la stessa funzione di attivazione, è possibile affermare che una rete neurale è completamente descritta una volta noti i pesi e i bias associati ai suoi neuroni.

Inoltre, una volta identificati i pesi e i bias adeguati, è possibile utilizzare una rete neurale per implementare funzioni dello stato della rete. In questi casi, si ipotizza l'esistenza di neuroni che ricevono valori da elaborare dall'esterno e di neuroni i cui risultati vengono resi disponibili verso l'esterno. Per la costruzione di una rete neurale, si ricorre spesso ad algoritmi di apprendimento automatico (*machine learning*), che determinano i pesi e i bias della rete attraverso un processo automatico chiamato **addestramento**.

3.3 Tipologie di addestramento

Nel campo del machine learning esistono principalmente tre paradigmi, ciascuno associato a un particolare processo di addestramento: supervised learning, reinforcement learning e unsupervised learning. Ognuno di questi paradigmi ha applicazioni e obiettivi specifici:

3.3.1 Addestramento supervisionato

Nel **supervised learning** (o *addestramento supervisionato*), una rete viene addestrata utilizzando un set di dati, detto **training set**, che include sia gli input che i corrispondenti output desiderati. L'obiettivo è creare un modello in grado di effettuare previsioni accurate su nuovi dati non etichettati. Una specifica applicazione del supervised learning è la **classificazione**.

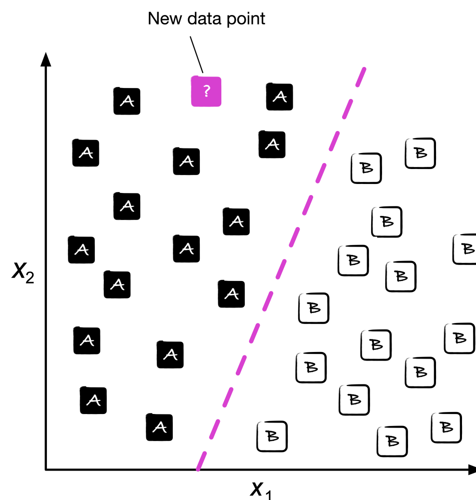

Figura 3.2: Esempio di classificazione binaria [1]

In contesti come quello in Figura 3.2, l'obiettivo è assegnare a ogni elemento di input un'etichetta specifica, scelta da un insieme di classi predefinite. È importante sottolineare che, nella classificazione, le etichette sono di natura discreta. Un esempio di classificazione è proprio l'identificazione di un software come malware o benigno.

È importante precisare che la classificazione non è limitata a problemi di classificazione binaria, in cui esistono solo due classi. Quando sono presenti più di due categorie, si parla di classificazione multiclasse (**multiclass classification**). Ne è un esempio la classificazione di malware in una delle categorie del

paragrafo 2.2 *Tipologie di malware.*

Un altro aspetto fondamentale del supervised learning è l'analisi di regressione (**regression analysis**), che, come mostrato in Figura 3.3, punta a prevedere risultati su una scala continua piuttosto che categorica. In pratica, mentre la classificazione assegna etichette discrete, l'analisi di regressione stima valori continui.

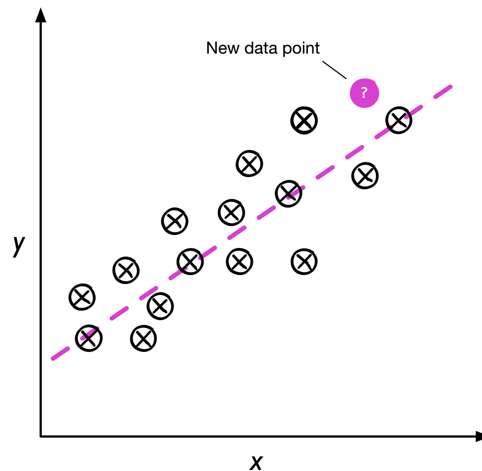

Figura 3.3: Esempio di regressione [1]

Nell'analisi di regressione, i termini **features** (caratteristiche) o variabili predittive fanno riferimento ai valori di input utilizzati per fare previsioni, mentre i valori di output sono comunemente detti **outcome** o variabili dipendenti. L'obiettivo dell'analisi di regressione è sviluppare un modello che possa accuratamente prevedere l'outcome in base alle variabili predittive, utilizzando una funzione che meglio approssima la relazione tra di essi. Questo tipo di modello è particolarmente utile in contesti dove si desidera stimare un risultato continuo.

3.3.2 Addestramento per rinforzo

Nel **reinforcement learning** (o *addestramento per rinforzo*), l'obiettivo è creare un **agente** capace di ottimizzare le sue prestazioni interagendo con un determinato ambiente. Queste interazioni spesso includono un **segnale di ricompensa** (*reward signal*), rendendo il reinforcement learning simile, in un certo senso, all'apprendimento supervisionato. Tuttavia, a differenza dell'apprendimento supervisionato, il feedback fornito all'agente è una valutazione di efficacia piuttosto che un'etichetta categorica. L'agente impara come interagire

con l'ambiente cercando di massimizzare il valore atteso di queste valutazioni, attraverso un processo di tentativi ed errori, comunemente detto **policy**.

3.3.3 Addestramento non supervisionato

Nel supervised learning, il modello viene addestrato su un set di dati in cui le risposte corrette, o etichette, sono già note. L'**unsupervised learning** (o *addestramento non supervisionato*), invece, si occupa di dati che non hanno etichette predefinite o una struttura nota. In questo tipo di apprendimento, gli obiettivi sono individuare strutture nascoste nei dati o estrarre informazioni significative, senza l'ausilio di etichette.

Il **clustering**, mostrato in Figura 3.4, è un metodo spesso utilizzato per l'analisi dei dati, che consiste nel raggruppare le informazioni in sottogruppi (**clusters**) senza la necessità di etichette preesistenti. Ogni cluster generato dall'analisi raggruppa elementi che presentano un certo livello di somiglianza tra loro e che sono distinti dagli elementi presenti in altri cluster.

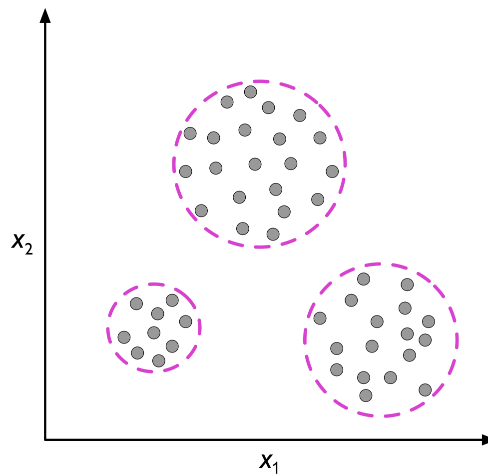

Figura 3.4: Esempio di clustering [1]

3.4 Reti feed-forward

Una rete neurale **feed-forward** è un grafo orientato aciclico pesato, con una struttura a strati (o layer), in cui:

- I nodi rappresentano neuroni artificiali

- Gli strati sono numerati
- Ogni nodo appartiene a un solo strato
- I nodi del primo strato non hanno archi entranti e il valore di uscita dei loro neuroni viene letto dall'esterno
- Ogni nodo, ad eccezione di quelli del primo strato, ha archi entranti provenienti solo dai nodi dello strato precedente
- I nodi dell'ultimo strato non hanno archi uscenti e il valore di uscita dei loro neuroni viene fornito all'esterno

Se non diversamente specificato, si assume che una rete neurale feed-forward abbia il massimo numero di connessioni che le consentono di mantenere una struttura a strati.

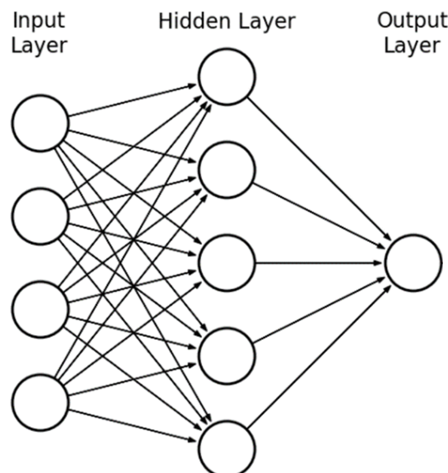

Figura 3.5: Rete neurale feed-forward

La Figura 3.5 mostra un esempio di rete neurale feed-forward. Si noti che il primo strato è detto **input layer**, l'ultimo **output layer**, mentre tutti gli altri strati intermedi sono solitamente chiamati **hidden layers**. Una rete neurale feed-forward è anche conosciuta come **perceptron** (o percettrone).

3.5 Single-Layer Perceptron

Un **Single-Layer Perceptron** (SLP) è una rete neurale feed-forward con n ingressi reali, nessuno strato nascosto e m uscite reali. La notazione (n, m)-SLP viene utilizzata per specificare i valori di n e m. Un (n, m)-SLP può essere

impiegato come approssimatore di una funzione $\mathbb{R}^n \to \mathbb{R}^m$ di cui si conoscono i valori per alcuni vettori, raccolti in un training set. Attraverso l'addestramento supervisionato, è possibile determinare i pesi del SLP in modo da approssimare efficacemente la funzione in esame (a tal proposito, una discussione più approfondita verrà fatta nel paragrafo 3.6.1 *Addestramento della rete*).

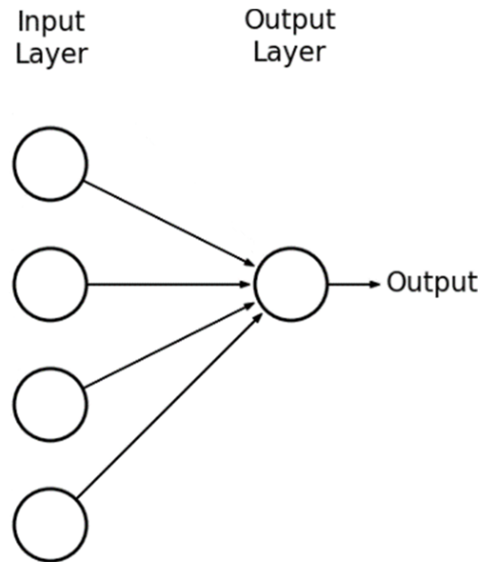

Figura 3.6: Single-Layer Perceptron con quattro neuroni nello strato di input e uno nello strato di output

Consideriamo un $(n, 1)$-SLP, come illustrato in Figura 3.6, e ipotizziamo che tutti i vettori in $\mathbb{R}^n$ utilizzati come ingressi del SLP siano del tipo $x = (x_1, ..., x_n)$. Poiché i neuroni nello strato di input non eseguono alcun tipo di computazione, in quanto le loro uscite contengono valori letti dall'esterno, l'analisi della rete si concentra sullo studio del neurone nello strato di uscita. In particolare, se si utilizza Heaviside come funzione di attivazione σ, e se $w \in \mathbb{R}^n$ e $b \in \mathbb{R}$ rappresentano, rispettivamente, il vettore dei pesi e il bias del neurone nello strato di output, allora, dato un vettore di ingresso x, la quantità z, detta **preattivazione**, sarà data da

$$z = \sum_{i=1}^{n} w_i x_i + b = w \cdot x + b \tag{3.6}$$

A questo punto, $\hat{y}$, che in questo caso corrisponde all'output della rete (vedi Figura 3.7), sarà

$$\hat{y} = \sigma(z) = \begin{cases} 1 & \text{se } z \geq 0 \\ 0 & \text{se } z < 0 \end{cases} \tag{3.7}$$

L'essenza del modello del single-layer perceptron risiede nell'emulare il comportamento di un singolo neurone biologico, che può essere attivo (1) o inattivo (0).

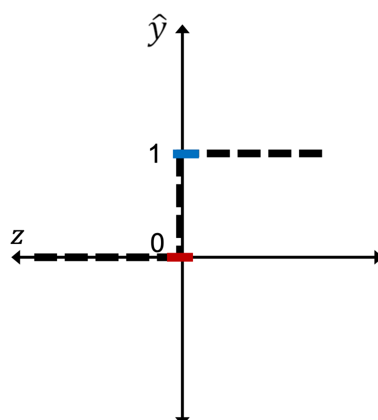

Figura 3.7: Output del neurone al variare della preattivazione [1]

È importante notare che, dati un bias e un vettore dei pesi opportuni, la correttezza del single-layer perceptron è garantita solo se le due classi sono linearmente separabili, ovvero se le due classi possono essere separate perfettamente da un confine decisionale lineare (vedi Figura 3.8).

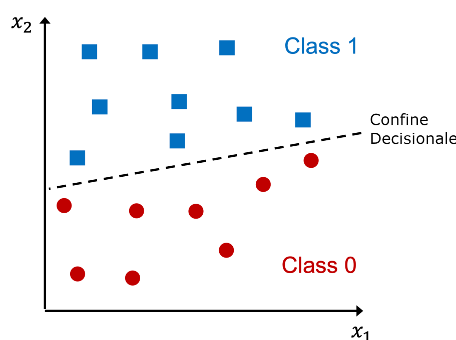

Figura 3.8: Classi linearmente separabili [1]

In altre parole, la correttezza del single-layer perceptron è garantita solo se esiste una retta (o, in generale, un iperpiano in uno spazio multidimensionale) che può separare perfettamente tutti gli elementi di una classe da quelli dell'altra.

3.6 Multi-Layer Perceptron

Per aumentare la potenza espressiva delle reti feed-forward è sufficiente aggiungere neuroni agli strati nascosti. Un **Multi-Layer Perceptron** (MLP) è una rete neurale feed-forward composta da uno o più ingressi reali, vari strati nascosti, ciascuno con un numero specifico di neuroni, e una o più uscite reali. Per semplificare l'analisi, ottenendo comunque risultati significativi, è utile concentrarsi su MLP con un solo strato nascosto e una sola uscita. In questo caso, si indica con $(n, m, 1)$-MLP un perceptron con n ingressi, un singolo strato nascosto con m neuroni e una sola uscita.

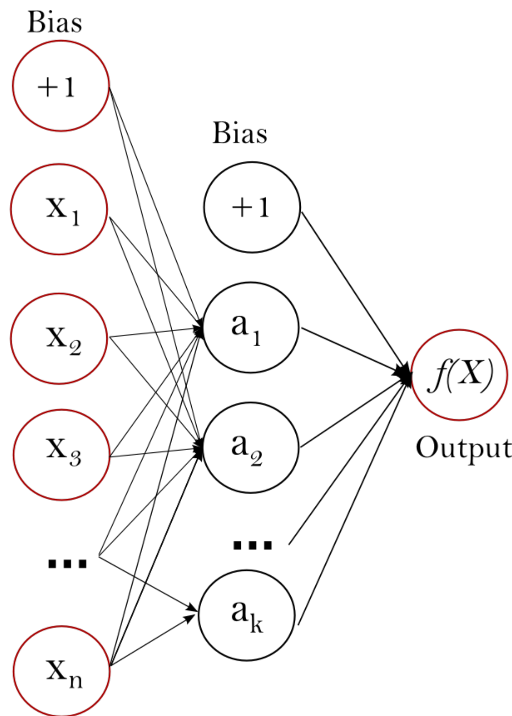

Figura 3.9: Multi-Layer Perceptron con uno strato nascosto [2]

Inoltre, per semplificare la notazione matematica, aggiungiamo ad ogni strato del perceptron, eccetto lo strato di output, un nodo il cui output è fissato al valore +1 e che non riceve archi entranti, come mostrato in Figura 3.9.

In questo modo, è possibile descrivere un neurone mediante un solo vettore dei pesi $\mathbf{w} = (w_i)_{i=1}^{n+1}$, dove w^{n+1} rappresenta il bias, detto anche **peso di bias**. La computazione del neurone diventa quindi

$$\hat{y} = \sigma(w \cdot x) \tag{3.8}$$

Dato un $(n+1, m+1, 1)$-MLP, l'obiettivo è utilizzare il MLP come approssimatore per una funzione $f : \mathbb{R}^{n+1} \to \mathbb{R}$ di cui si conoscano i valori per alcuni vettori. Quindi, se $w_j \in \mathbb{R}^{n+1}$ è il vettore dei pesi associato agli archi che collegano lo strato d'ingresso al j-esimo neurone dello strato nascosto, con $1 \leq j \leq m$, allora

$$\hat{h}_j = \sigma(w_j \cdot x) \tag{3.9}$$

In più, se $v \in \mathbb{R}^{m+1}$ è il vettore dei pesi associati agli archi che collegano lo strato nascosto con il neurone di uscita, l'uscita della rete sarà

$$\hat{y} = \sigma(v \cdot \hat{h}) \tag{3.10}$$

Quindi, scrivendo per esteso i prodotti scalari, l'output della rete diventa

$$\hat{y} = \sigma\left(\sum_{j=1}^{m+1} v_j \cdot \hat{h}_j\right) = \sigma\left(\sum_{j=1}^{m} v_j \cdot \sigma\left(\sum_{i=1}^{n} w_{j,i} \cdot x_i + w_{j,n+1}\right) + v_{m+1}\right) \tag{3.11}$$

3.6.1 Addestramento della rete

Come già anticipato, l'addestramento supervisionato permette di calcolare i pesi in grado di minimizzare la distanza media tra il valore calcolato dal perceptron $\hat{y}$ e il corrispondente valore della funzione da approssimare y. Si noti che, poichè l'addestramento è supervisionato, il valore della funzione è noto per tutti i vettori del training set.

Un vettore dei pesi che riesce a ridurre al minimo le distanze individuali tra i valori calcolati dal perceptron e i valori della funzione da approssimare, riesce anche a ridurre al minimo la distanza media tra questi valori. È quindi sufficiente concentrare l'attenzione sulla minimizzazione della distanza tra il valore calcolato dal perceptron e il corrispondente valore della funzione da approssimare.

Fissato uno dei vettori nel training set $x \in \mathbb{R}^{n+1}$, l'errore compiuto dal MLP nell'approssimazione della funzione f in x è dato da

$$\epsilon(w, x) = y - \hat{y} \tag{3.12}$$

Perciò, la capacità del perceptron di approssimare la funzione f in x è tanto migliore quanto $|\epsilon(w, x)|$ è piccolo. In altre parole, fissato un vettore x, il problema di individuare un vettore dei pesi w in grado di approssimare bene $f(x)$ può essere espresso mediante la ricerca di un w che minimizzi $|\epsilon(w, x)|$.

Per fare ciò, è possibile cercare il minimo dell'**errore quadratico** (che, in questo contesto, prende il nome di funzione di perdita o **loss function**)

$$e(w,x) = \frac{1}{2}\epsilon^2(w,x) \tag{3.13}$$

anziché di $|\epsilon(w,x)|$. Naturalmente, un w in grado di rendere minimo $e(w,x)$ è anche in grado di rendere minimo $|\epsilon(w,x)|$. In questo caso, utilizziamo l'**algoritmo di discesa del gradiente** per la ricerca del minimo di $e(w,x)$.

3.6.2 Discesa del gradiente

Consideriamo una funzione reale f definita su un insieme aperto $D \subseteq \mathbb{R}^n$. Per un punto $x \in D$ e un versore $v \in \mathbb{R}^n$ (ovvero, un vettore reale con norma euclidea pari a 1), la **derivata direzionale** della funzione f nella direzione v è definita come

$$\partial_v f(x) = \lim_{h \to 0} \frac{f(x+hv) - f(x)}{h} \tag{3.14}$$

La derivata direzionale rappresenta la velocità di crescita della funzione nella direzione considerata. Nel caso di funzioni reali di una sola variabile reale, la derivata direzionale coincide con la derivata ordinaria.

Data una funzione reale f definita su un insieme aperto $D \subseteq \mathbb{R}^n$, un punto $x \in D$ e un versore del tipo $v = (0, \dots, 1, \dots, 0)$, con il valore 1 in posizione $1 \le i \le n$, la derivata nella direzione v prende il nome di derivata parziale rispetto a x_i e si indica con

$$\frac{\partial f}{\partial x_i}(x) \quad \text{oppure} \quad \partial_i f(x) \tag{3.15}$$

Il calcolo delle derivate parziali può essere ridotto al calcolo delle derivate ordinarie. Infatti, poiché il limite del rapporto incrementale coinvolge una sola variabile, durante il calcolo di una derivata parziale è possibile trattare le altre variabili come costanti.

Il vettore formato dalle n derivate parziali di una funzione reale di n variabili reali è detto **gradiente** della funzione e si indica con

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_n}(x) \right) \tag{3.16}$$

Il gradiente $\nabla f(x)$ indica la direzione di massima velocità di crescita della funzione nel punto x. Di conseguenza, la quantità $-\nabla f(x)$, detta **antigradiente** della funzione, indica la direzione di massima velocità di decrescita della funzione nel punto x. Se $\nabla f(x) = 0$, allora la funzione è stazionaria nel punto x, che viene detto punto critico.

Algoritmo di discesa del gradiente

A questo punto, è possibile descrivere un algoritmo per la ricerca di un minimo locale di una funzione reale di $n \in \mathbb{N}^+$ variabili reali. L'algoritmo, noto come **discesa del gradiente**, è presentato nell'Algoritmo 1 e richiede quattro argomenti: la funzione f di cui si cerca un minimo locale, una prima approssimazione $x \in \mathbb{R}^n$ di uno dei minimi locali di f, il passo di discesa $\alpha \in \mathbb{R}^+$ e un valore $\delta \in \mathbb{R}^+$ per determinare quando la lunghezza del gradiente è sufficientemente piccola

Algoritmo 1 Algoritmo di discesa del gradiente

1: **function** GRADIENT_DESCENT(f, x, α, δ)
2: $\quad g \leftarrow \nabla f(x)$
3: $\quad$ **while** $||g|| > \delta$ **do**
4: $\quad\quad x \leftarrow x - \alpha \cdot g$
5: $\quad\quad g \leftarrow \nabla f(x)$
6: $\quad$ **return** x

L'algoritmo inizia con l'approssimazione x fornita e cerca un minimo locale in $\mathbb{R}^n$ spostandosi, a ogni iterazione, nella direzione dell'antigradiente della funzione. L'algoritmo termina quando x individua un punto critico della funzione, corrispondente quindi a un minimo locale. Si noti che il passo di discesa α rappresenta la velocità con cui l'algoritmo percorre la discesa nella direzione del gradiente invertito.

3.6.3 Algoritmo di backpropagation

A questo punto, applicando l'algoritmo di discesa del gradiente, che in questo contesto funge da **ottimizzatore**, alla funzione di errore quadratico 3.13, l'algoritmo termina restituendo i pesi che minimizzano l'errore del MLP.

In particolare, consideriamo il peso v_j, con $1 \leq j \leq m+1$, associato all'arco che collega il j-esimo nodo dello strato nascosto con lo strato di uscita

$$\frac{\partial e}{\partial v_j}(w) = \frac{1}{2} \cdot 2 \cdot \epsilon(w) \frac{\partial \epsilon}{\partial v_j}(w) \tag{3.17}$$

ma

$$\frac{\partial \epsilon}{\partial v_j}(w) = -\frac{\partial \hat{y}}{\partial v_j}(w) \tag{3.18}$$

e

$$\frac{\partial \hat{y}}{\partial v_j}(w) = \sigma'(v \cdot \hat{h}) \frac{\partial (v \cdot \hat{h})}{\partial v_j} = \sigma'(v \cdot \hat{h}) \hat{h}_j \tag{3.19}$$

A questo punto, è possibile esprimere la seguente regola per l'aggiornamento dei pesi che collegano lo strato nascosto con l'uscita

$$v_j \leftarrow v_j + \alpha(y - \hat{y})\sigma'(v \cdot \hat{h})\hat{h}_j \tag{3.20}$$

Per $1 \leq j \leq m+1$, assumendo che la componente $\hat{h}_{m+1}$ valga 1.

È importante notare che questa formula coincide con la formula di aggiornamento dei pesi del single-layer perceptron.

Si consideri ora $w_j \in \mathbb{R}^{n+1}$, il vettore dei pesi associato al j-esimo neurone dello strato nascosto, e quindi $1 \leq j \leq m$. La derivata dell'errore quadratico rispetto alla i-esima componente di w_j vale

$$\frac{\partial e}{\partial w_{j,i}}(w) = \frac{1}{2} \cdot 2 \cdot \epsilon(w) \frac{\partial \epsilon}{\partial w_{j,i}}(w) \tag{3.21}$$

ma appunto

$$\frac{\partial \epsilon}{\partial w_{j,i}}(w) = -\frac{\partial \hat{y}}{\partial w_{j,i}}(w) \tag{3.22}$$

e

$$\frac{\partial \hat{y}}{\partial w_{j,i}}(w) = \sigma'(v \cdot \hat{h}) \frac{\partial (v \cdot \hat{h})}{\partial w_{j,i}} \tag{3.23}$$

ma ora, a contrario di quanto accaduto per lo strato di output

$$\frac{\partial (v \cdot \hat{h})}{\partial w_{j,i}} = v_j \sigma'(w_j \cdot x) x_i \tag{3.24}$$

Quindi, la regola di aggiornamento dei pesi $w_{j,i}$ associati agli archi che collegano i neuroni di ingresso con il j-esimo neurone dello strato nascosto è

$$w_{j,i} \leftarrow w_{j,i} + \alpha(y - \hat{y})\sigma'(v \cdot \hat{h}) v_j \sigma'(w_j \cdot x) x_i \tag{3.25}$$

con $1 \leq j \leq m$ e $1 \leq i \leq n$.

In sintesi, dopo aver applicato il MLP al vettore di input per ottenere l'uscita mediante un processo chiamato **forward propagation**, si valuta l'errore e lo si ripartisce sui vari strati, uno strato alla volta, tramite un processo chiamato **backward propagation**. Complessivamente, questa procedura rappresenta l'algoritmo di addestramento di una rete neurale. L'Algoritmo 2 mostra l'algoritmo di addestramento per un $(n+1, m+1, 1)$-MLP e richiede i seguenti parametri:

- T: il training set, formato da coppie (x, y)

- α: il coefficiente di apprendimento
- δ: la tolleranza accettata sull'errore medio
- ρ: il massimo numero di epoche

Algoritmo 2 Algoritmo di addestramento per un $(n+1, m+1, 1)$-MLP

1: **function** MLP_LEARN(T, α, δ, ρ)
2: randomize $v \in \mathbb{R}^{m+1}$
3: **for** $1 \leq j \leq m$ **do**
4: randomize $w_j \in \mathbb{R}^{n+1}$
5: **for** $1 \leq r \leq \rho$ **do**
6: $\overline{\epsilon} \leftarrow 0$
7: **for** $(x, y) \in T$ **do**
8: $z \leftarrow v \cdot y$
9: **for** $1 \leq j \leq m$ **do**
10: $b \leftarrow w_j \cdot x$
11: **for** $1 \leq i \leq n+1$ **do**
12: $w_{j,i} \leftarrow w_{j,i} + \alpha(y - \hat{y})\sigma'(v \cdot \hat{h})v_j\sigma'(w_j \cdot x)x_i$
13: **for** $1 \leq i \leq m+1$ **do**
14: $v_j \leftarrow v_j + \alpha(y - \hat{y})\sigma'(v \cdot \hat{h})\hat{h}_j$
15: $\overline{\epsilon} \leftarrow \overline{\epsilon} + \frac{|y - \sigma(z)|}{|T|}$
16: **if** $\overline{\epsilon} < \delta$ **then**
17: **return** w
18: **return** w

Come già detto, l'algoritmo determina un vettore di pesi in grado di approssimare una funzione di cui si conoscono alcuni valori quando:

- L'errore medio $\overline{\epsilon}$ calcolato su tutto il training set è minore della tolleranza
- Viene raggiunto il numero massimo di epoche, dove un'epoca non è altro che un'iterazione completa dell'algoritmo su tutto il training set

Si noti che le condizioni di terminazione dell'algoritmo di backpropagation non coincidono con quelle dell'algoritmo di discesa del gradiente.

L'Algoritmo 2 può essere facilmente esteso per realizzare un (n, h, m)-MLP, per approssimare una funzione $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Per farlo, è sufficiente disporre di un vettore dei pesi per ognuna delle m uscite richieste.

3.6.4 Teorema di approssimazione universale

Concludiamo il discorso introducendo il teorema di approssimazione universale, che, in modo informale, può essere enunciato come segue:

Data una certa tolleranza $\epsilon \in \mathbb{R}^+$ nell'approssimazione di una funzione f continua su un insieme compatto (un insieme chiuso e limitato), è possibile trovare un MLP con uno strato nascosto che possa approssimare la funzione f entro la tolleranza richiesta.

Una definizione più formale è invece la seguente:

Dato $n \in \mathbb{N}^+$, sia $D \subset \mathbb{R}^n$ un insieme compatto e sia $f : S \to \mathbb{R}$ una funzione reale di più variabili reali continua nell'insieme aperto $S \subseteq \mathbb{R}^n$, con $D \subset S$. Sia $\sigma : \mathbb{R} \to \mathbb{R}$ una funzione continua, monotona strettamente crescente e limitata. Per ogni $\epsilon \in \mathbb{R}^+$ esiste un $m \in \mathbb{N}^+$ tale che la funzione

$$h(x) = \sum_{j=1}^{m} v_j \cdot \sigma\left(\sum_{i=1}^{n} w_{j,i} \cdot x_i + w_{j,n+1}\right) + v_{m+1} \tag{3.26}$$

soddisfa la condizione

$$\max_{x \in D} |f(x) - h(x)| < \epsilon \tag{3.27}$$

In questo caso, si noti che viene utilizzata una funzione di attivazione identità per il calcolo dell'uscita.

3.7 Reti neurali e classificatori

Per $n \in \mathbb{N}^+$, il problema di classificazione dei vettori di $S \subseteq \mathbb{R}^n$ consiste nell'associare ogni vettore di S a una delle classi appartenenti a un numero finito e noto di classi disgiunte. Se $m \in \mathbb{N}^+$ è il numero di classi, un classificatore è una funzione f che mappa S in $[1, \ldots, m] \subseteq \mathbb{N}$.

Per risolvere il problema di classificazione, si utilizza comunemente un MLP (Multilayer Perceptron) con n ingressi e m uscite, addestrato in modo supervisionato mediante campioni che associano vettori di S a vettori del tipo $(0, \ldots, 1, \ldots, 0)$, dove il valore 1 è posizionato nella posizione corrispondente alla classe del campione.

Per stabilire la classe del vettore di input, una volta noto il vettore di output, si utilizzano normalmente due metodi. Il primo metodo prevede di scegliere

arbitrariamente un indice corrispondente a una delle componenti massime del vettore calcolato dalla rete. La classe stimata è data da

$$k = \arg \max_{1 \leq i \leq m} \hat{y}_i \tag{3.28}$$

dove il vettore di uscita della rete è $\hat{y} = (\hat{y}_i)_{i=1}^m$.

In alternativa, dato $\hat{y}$, si può calcolare un nuovo vettore s mediante la funzione

$$s = \text{softmax}(\hat{y}) = \left(\frac{e^{\hat{y}_i}}{\sum_{j=1}^m e^{\hat{y}_j}} \right) \tag{3.29}$$

Il vettore s ha tutte le componenti comprese tra 0 e 1, e la somma di tutte le componenti è pari a 1. Pertanto, le componenti di s possono essere interpretate come probabilità.

Capitolo 4

Progettazione

Ultimata la presentazione dei concetti teorici alla base del progetto, è possibile definire le tecniche utilizzate per risolvere il problema del riconoscimento e della classificazione dei malware. La pipeline progettata per l'analisi dei file PE verrà presentata nel seguente modo: la prima parte del capitolo si concentrerà sul riconoscimento dei malware, descrivendo un modello progettato per leggere e analizzare il contenuto dell'intestazione del file. La seconda parte affronterà invece il problema della classificazione del file, basata sull'analisi del codice disassemblato.

4.1 Dipendenze del sistema

Il progetto utilizza la libreria **pytorch** per la costruzione e l'addestramento di reti neurali, **pandas** (`pd`) e **numpy** (`np`) per la manipolazione dei dati, **matplotlib** e **seaborn** per visualizzare i risultati ottenuti, **scikit-learn** per la prototipazione e il preprocessing dei dati, **tqdm** per la creazione di barre di avanzamento e **joblib** per la serializzazione degli oggetti.

4.2 Riconoscimento di malware

Il formato Portable Executable prevede che l'intestazione contenga diverse informazioni essenziali per il caricamento e l'esecuzione del software nel sistema operativo, come il tipo di macchina per cui è stato compilato il codice o la struttura delle sezioni del file. Come sarà mostrato di seguito, analizzandone il contenuto è possibile classificare i file come malware o benigni. In particolare, questa parte del progetto si basa sulla costruzione di un modello utilizzando il dataset CLaMP (Classification of Malware with PE headers) [10], che contiene gli header di 5184 campioni.

4.2.1 Caricamento e pre-elaborazione dei dati

Innanzitutto, è necessario caricare in memoria il contenuto del dataset CLaMP, attraverso la creazione di un `DataFrame` (vedi Algoritmo 3).

Algoritmo 3 Caricamento e pre-elaborazione del dataset CLaMP

```
data: pd.DataFrame = pd.read_csv("./data/ClaMP/ClaMP_Raw-5184.csv")
data = data.drop('e_res', axis=1)
data = data.drop('e_res2', axis=1)
data = data.drop('BaseOfData', axis=1)

X: np.ndarray = data.iloc[:, :52].values
y: np.ndarray = data.iloc[:, 52].values
```

Durante la fase di preprocessing, è possibile eliminare alcune colonne superflue. In particolare, vengono rimosse le colonne `e_res`, `e_res2` e `BaseOfData`, poiché i campi corrispondenti non sono più presenti nell'intestazione delle versioni più recenti del formato (PE32+).

Successivamente, si procede alla separazione delle variabili indipendenti, le **feature**, dalle variabili dipendenti, le **etichette** di classe, che il modello cercherà di prevedere. A questo punto, è possibile verificare la suddivisione del dataset in malware e file benigni. Come mostrato in Figura 4.1, il dataset risulta essere piuttosto bilanciato.

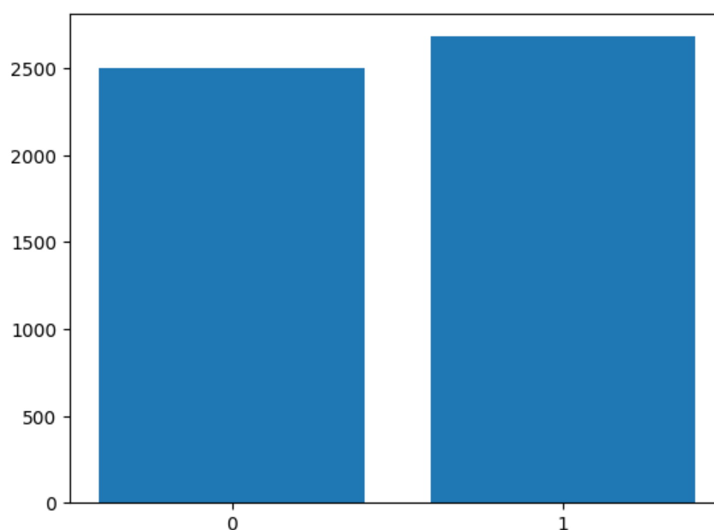

Figura 4.1: Quantità di file benigni (0) e malware (1) in CLaMP

4.2.2 Generazione del seed

Per garantire che i risultati dell'addestramento siano il più possibile riproducibili, nell'Algoritmo 4 viene impostato un valore specifico come seed per i generatori pseudo-casuali utilizzati nel seguito.

Algoritmo 4 Generazione del seed

```
seed = 32
torch.manual_seed(seed)
np.random.seed(seed)
```

4.2.3 Suddivisione del dataset in training e test set

Dopo aver pulito e preparato il dataset, questo viene suddiviso nei set di training e di test. In particolare, il dataset viene diviso in modo che l'80% dei dati venga utilizzato per l'addestramento, mentre il 20% restante viene riservato per il testing.

Algoritmo 5 Suddivisione di CLaMP in training e test set

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=seed, stratify=y)
```

La divisione viene realizzata utilizzando la funzione `train_test_split` di scikit-learn. Questa funzione garantisce che la suddivisione sia casuale, contribuendo a prevenire bias nei dati di addestramento o di test.

4.2.4 Standardizzazione delle features

Le feature vengono standardizzate utilizzando lo `StandardScaler` di scikit-learn, come mostrato in Algoritmo 6, uno strumento che normalizza i dati assicurando che abbiano media uguale a zero e deviazione standard pari a uno.

Algoritmo 6 Standardizzazione delle features di CLaMP

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Prima di tutto si applica il metodo `fit_transform` al training set: questo metodo calcola la media e la deviazione standard per ciascuna feature, poi standardizza i dati sottraendo la media calcolata da ogni valore e dividendo il risultato per la deviazione standard.

Per quanto riguarda i dati di test, si utilizza il metodo `transform`. Invece di ricalcolare media e deviazione standard, questo metodo applica i valori precedentemente determinati dal set di addestramento per standardizzare i dati di test. Tale approccio garantisce che entrambi i set di dati, addestramento e test, siano trasformati in modo consistente, fornendo una base equa per l'addestramento del modello e la successiva valutazione delle sue prestazioni.

4.2.5 Creazione dei data loader

Innanzitutto, come mostrato in Algoritmo 7, i vettori che contengono i dati destinati all'addestramento e al testing vengono trasformati in tensori.

In informatica, un **tensore** è una struttura dati che generalizza i concetti di scalare, vettore e matrice a dimensioni superiori. In particolare, un tensore è una collezione di valori reali organizzati in una griglia multi-dimensionale, in cui ogni dimensione è detta **ordine** (o grado) del tensore (e.g. uno scalare è un tensore di ordine 0, un vettore è un tensore di ordine 1, mentre una matrice è un tensore di ordine 2).

Algoritmo 7 Creazione dei tensori

```
1 X_train = torch.tensor(X_train, dtype=torch.float32)
2 X_test = torch.tensor(X_test, dtype=torch.float32)
3 y_train = torch.tensor(y_train, dtype=torch.float32)
4 y_test = torch.tensor(y_test, dtype=torch.float32)
```

Successivamente, si procede alla creazione di due `TensorDataset`, strumenti essenziali per accorpare le feature e le etichette di classe in un unico oggetto. Questi vengono poi utilizzati per istanziare due `DataLoader`, oggetti iterabili che automatizzano la suddivisione dei dati in batch durante l'addestramento e la valutazione del modello (vedi Algoritmo 8).

I **batch** sono gruppi di dati processati simultaneamente dal modello, permettendo non solo un utilizzo più efficiente delle risorse computazionali tramite la parallelizzazione delle operazioni, ma anche una migliore generalizzazione del modello, dal momento che ogni iterazione considera una maggiore varietà di esempi.

Algoritmo 8 Creazione dei Dataset e DataLoader

```
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
```

Il parametro `batch_size` definisce la dimensione di ogni batch, indicando il numero di campioni di addestramento impiegati per calcolare il gradiente in ogni step di ottimizzazione.

Per quanto riguarda la configurazione dei `DataLoader`, il parametro `shuffle` impostato a `True` nel `train_loader` assicura che i dati vengano mischiati prima di ogni epoca di addestramento, favorendo un apprendimento più robusto e prevenendo la memorizzazione dell'ordine di presentazione dei dati da parte del modello. Al contrario, `shuffle=False` nel `test_loader` stabilisce che i dati di test rimangano invariati durante la fase di valutazione, poiché in questo contesto la mischiatura non incide sulla misurazione delle performance del modello.

4.2.6 Definizione del modello

L'Algoritmo 9 mostra la definizione del $(52, 32, 16, 1)$-MLP utilizzato per il riconoscimento di malware

Algoritmo 9 $(52, 32, 16, 1)$-MLP utilizzato per il riconoscimento di malware

```
import torch.nn as nn

class BinaryMLP(nn.Module):
    def __init__(self):
        super(BinaryMLP, self).__init__()

        self._input__h_1 = nn.Linear(52, 32)
        self._h_1__h_2 = nn.Linear(32, 16)
        self._h_2__output = nn.Linear(16, 1)

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self._input__h_1(x))
        x = self.relu(self._h_1__h_2(x))
        x = self._h_2__output(x)
        return x
```

Si noti che la funzione di attivazione utilizzata dai neuroni negli strati nascosti è la ReLU, mentre lo strato di output non viene attivato.

4.2.7 Funzione di loss e ottimizzatore

Come descritto nel Paragrafo 3.6.1 *Addestramento della rete*, è necessario definire una funzione di loss per misurare l'errore compiuto dal modello e un algoritmo di ottimizzazione da utilizzare per aggiornare i parametri (pesi e bias) dei neuroni durante l'addestramento.

Binary Cross Entropy

La binary cross entropy (BCE) è la funzione di loss comunemente utilizzata per problemi di classificazione binaria. In particolare, la BCE misura la differenza tra la distribuzione di probabilità prevista dal modello e la distribuzione di probabilità reale dei dati di addestramento. La formula della BCE è la seguente

$$BCE(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{4.1}$$

Dove:

- N è il numero di campioni
- y_i è l'etichetta vera del i-esimo campione (0 o 1)
- $\hat{y}_i$ è la probabilità predetta per il i-esimo campione

L'obiettivo della BCE è quindi minimizzare la differenza tra le probabilità previste $\hat{y}_i$ e le etichette reali y_i, penalizzando maggiormente le previsioni meno sicure.

Adam (Adaptive Moment Estimation)

Adam (Adaptive Moment Estimation) è un algoritmo di ottimizzazione utilizzato per addestrare reti neurali, che adatta dinamicamente i tassi di apprendimento, o learning rate (simile al passo di discesa descritto nel paragrafo 3.6.2 *Discesa del gradiente*), dei parametri del modello.

Il funzionamento di Adam inizia con il calcolo del gradiente della funzione di perdita rispetto ai parametri del modello a ogni passo di addestramento. Successivamente, Adam mantiene in memoria due stime: la media dei gradienti passati (primo momento) e la varianza dei gradienti passati (secondo momento), ossia il gradiente al quadrato.

Poiché le stime iniziali dei momenti sono basate su un numero ridotto di dati, Adam applica una correzione del bias per migliorare l'accuratezza delle stime. Infine, i parametri del modello vengono aggiornati utilizzando le stime dei momenti. La media dei gradienti aiuta a indirizzare la discesa, mentre la varianza permette di adattare il tasso di apprendimento per ciascun parametro, rendendo l'algoritmo più robusto e stabile.

Adam è quindi in grado di adattarsi dinamicamente alle caratteristiche del problema, convergendo più velocemente e in modo più stabile rispetto a molti altri algoritmi di ottimizzazione. L'Algoritmo 10 mostra la definizione della rete, della funzione di perdita e dell'ottimizzatore utilizzati.

L'Algoritmo 10 mostra la definizione della rete, della funzione di perdita e dell'ottimizzatore utilizzati per il riconoscimento di malware.

Algoritmo 10 Definizione della rete, della loss function e dell'ottimizzatore

```
1 model = BinaryMLP()
2 loss_fn = torch.nn.BCEWithLogitsLoss()
3 optimizer = optim.Adam(model.parameters(), lr=0.01)
```

4.2.8 Addestramento del modello

Il processo di addestramento supervisionato, mostrato in Algoritmo 11, inizia impostando il numero di epoche di addestramento, `num_epochs`. Questo significa che il modello verrà iterativamente addestrato attraverso `num_epochs` iterazioni complete del dataset. Durante queste iterazioni, vengono monitorate e registrate due metriche fondamentali: la loss e la precisione, memorizzate in due liste separate, `loss_hist` e `accuracy_hist`, inizialmente riempite con valori zero, corrispondenti a ciascuna epoca di addestramento.

L'addestramento del modello si svolge attraverso un ciclo che copre tutte le epoche specificate. Per ogni epoca, il modello elabora il dataset in batch. Per ogni batch di dati, il modello effettua previsioni basate sull'output della funzione **sigmoide** applicata al neurone di output. La correttezza di queste previsioni viene valutata calcolando la perdita attraverso la funzione di loss, che misura la differenza tra i valori predetti e i valori reali. Dopo il calcolo della perdita, il modello aggiorna i suoi parametri interni utilizzando l'algoritmo Adam.

Dopo l'elaborazione di ciascun batch, le statistiche cumulative di perdita e precisione vengono aggiornate. La perdita per il batch corrente viene moltiplicata per il numero di esempi nel batch per ottenere un contributo ponderato alla perdita totale dell'epoca. Analogamente, la precisione viene calcolata confrontando le previsioni con i valori reali per determinare il numero di previsioni corrette.

Al termine di ciascuna epoca, le statistiche cumulative di perdita e precisione sono normalizzate dividendo per il numero totale di esempi nel dataset. Questo passaggio fornisce una media della perdita e della precisione per l'epoca, che può essere utilizzata per monitorare e valutare l'efficacia dell'addestramento.

Algoritmo 11 Addestramento del modello

```
1  num_epochs = 128
2  loss_hist = [0] * num_epochs
3  accuracy_hist = [0] * num_epochs
4
5  for epoch in tqdm(range(num_epochs)):
6      model.train()
7      total_loss = 0
8      correct_predictions = 0
9      total_samples = 0
10
11     for inputs, labels in train_loader:
12         # Forward pass
13         pred = model(inputs)
14         loss = loss_fn(pred, labels.unsqueeze(1))
15
16         # Backward pass and optimization
17         loss.backward()
18         optimizer.step()
19         optimizer.zero_grad()
20
21         # Calculate total loss
22         total_loss += loss.item() * labels.size(0)
23
24         # Calculate total accuracy
25         with torch.no_grad():
26             prediction = torch.sigmoid(pred)
27             prediction = (prediction >= 0.5).float()
28             correct_predictions += (prediction == labels.unsqueeze(1)).sum().
       item()
29             total_samples += labels.size(0)
30
31     # Calculate average loss and accuracy for the epoch
32     loss_hist[epoch] = total_loss / total_samples
33     accuracy_hist[epoch] = correct_predictions / total_samples
```

4.2.9 Analisi dell'addestramento

Per valutare i progressi e le prestazioni del modello durante la fase di addestramento, viene generato un grafico che mostra l'andamento della funzione di perdita (loss) e dell'accuratezza (accuracy) in funzione delle epoche.

Il grafico della loss (Figura 4.2) rivela come la rete migliori la sua capacità di ridurre l'errore tra le previsioni e le etichette reali con il passare delle epoche.

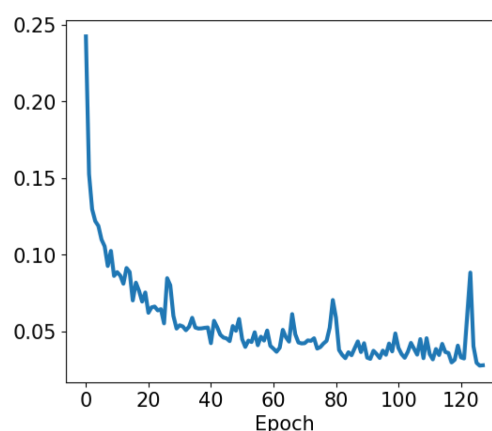

Figura 4.2: Grafico che mostra l'andamento della funzione di perdita durante il processo di addestramento

Il grafico dell'accuratezza (Figura 4.3), invece, mostra la percentuale di classificazioni corrette rispetto al totale.

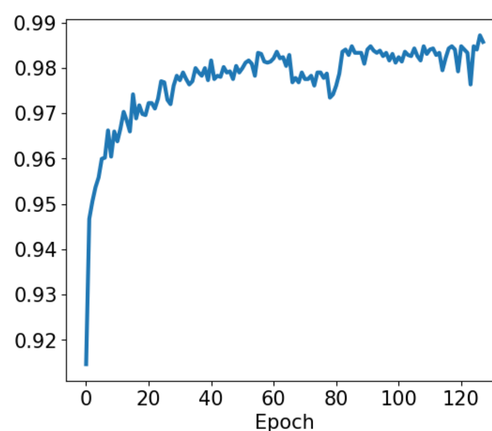

Figura 4.3: Grafico che mostra la precisione del modello durante il processo di addestramento

4.2.10 Valutazione del modello

Dopo aver completato la fase di addestramento, il modello viene testato utilizzando il set di test, al fine di valutarne le prestazioni in scenari non incontrati durante l'addestramento. L'Algoritmo 12 mostra il processo di testing del modello.

Algoritmo 12 Valutazione del modello

```
model.eval()
with torch.no_grad():

    TP = 0
    TN = 0
    FP = 0
    FN = 0

    for inputs, labels in test_loader:
        outputs = model(inputs)
        outputs = torch.sigmoid(outputs)

        for label, output in zip(labels, outputs.data):
            prediction = np.where(output >= 0.5, 1,0)

            if prediction == 1 and label == 1:
                TP += 1
            elif prediction == 0 and label == 1:
                FN += 1
            elif prediction == 1 and label == 0:
                FP += 1
            else:
                TN += 1
```

Prima di iniziare la valutazione, il modello viene posto in modalità `eval`. Questo passaggio è cruciale perché modifica il comportamento di alcuni moduli specifici per adattarli alla fase di test anziché a quella di addestramento. In questo modo, il modello può esibire un comportamento coerente e ottimizzato per la valutazione. Inoltre, durante la valutazione, il calcolo dei gradienti è disabilitato per risparmiare memoria e accelerare il processo, dato che l'aggiornamento dei pesi del modello non è necessario in questa fase. Il processo di valutazione procede iterando su un insieme di dati di test, caricati in batch.

Per ciascun file del batch, il modello prevede l'etichetta di classe del file basandosi sull'output della funzione **sigmoide** applicata al neurone di output. In particolare, se questo valore è maggiore o uguale a 0.5, la rete prevede la classe 1 (malware); altrimenti, viene prevista la classe 0 (file benigno). Successivamente, queste previsioni vengono confrontate con le etichette reali per determinare il numero di previsioni corrette.

Durante la valutazione si mantengono aggiornati quattro diversi contatori: veri positivi (**TP**, True Positives), veri negativi (**TN**, True Negatives), falsi positivi (**FP**, False Positives) e falsi negativi (**FN**, False Negatives).

Questi contatori, introdotti per completezza, verranno però utilizzati nel Capitolo 5 per mostrare le prestazioni della rete.

4.3 Classificazione di malware

Come già discusso, il processo di classificazione dei malware può essere complesso, poiché i malware di nuova generazione spesso appartengono a più categorie. In questo progetto di tesi, la classificazione dei malware si basa sul dataset della Microsoft Malware Classification Challenge (BIG 2015) [10], una competizione organizzata da Microsoft con l'obiettivo di migliorare la classificazione del malware attraverso tecniche avanzate di machine learning e analisi dei dati.

Il dataset BIG 2015 contiene una serie di file malware noti, tutti in formato Portable Executable, rappresentanti nove diverse famiglie di malware. Ogni file malware è possiede un Id, un valore hash che lo identifica univocamente, e una Classe, un numero intero che rappresenta la famiglia di appartenenza del malware. Per ciascun file, viene fornito il contenuto del file binario in formato esadecimale (senza l'intestazione per garantirne la sterilità). Viene inoltre fornito un registro contenente varie informazioni sui metadati estratti dal file binario, come chiamate di funzioni, stringhe, ecc. Questo manifest è stato generato utilizzando lo strumento di disassemblaggio IDA.

4.3.1 Famiglie di malware

Di seguito sono riportate le caratteristiche principali delle famiglie di malware presenti nel dataset BIG 2015:

- **Ramnit** (0): I malware ramnit [11] sono un tipo di trojan ampiamente diffuso, tanto che nel 2015 avevano già infettato circa 3,2 milioni di dispositivi Windows [12]. L'obiettivo principale di questo tipo di file è il furto delle credenziali per l'online banking. Questi malware si comportano principalmente come worm, grazie alla loro capacità di diffondersi tramite unità removibili. Tuttavia, possiedono anche funzionalità da downloader, consentendo l'installazione di malware aggiuntivo.

- **Lollipop** (1): I malware lollipop [11] sono principalmente adware, progettati per visualizzare annunci indesiderati sui browser web. Questi malware possono anche reindirizzare i risultati dei motori di ricerca, monitorare le attività sul PC, scaricare applicazioni e inviare informazioni del computer a un hacker.

- **Kelihos_ver3** (2): I malware kelihos_ver3 [11] sono noti per distribuire messaggi di spam via posta elettronica. Questi messaggi spesso contengono collegamenti ipertestuali a programmi di installazione di malware. Questa categoria di malware spesso comunica con server remoti per inviare informazioni sensibili o per scaricare ed eseguire ulteriori file malevoli.

- **Vundo** (3): I malware vundo [11] sono trojan noti per far comparire principalmente finestre popup o annunci indesiderati. Spesso possiedono la capacità di raccogliere informazioni sull'utente, e le versioni più recenti possono agire anche come ransomware.

- **Simda** (4): I malware simda [11] sono trojan progettati principalmente per sottrarre le password degli utenti. Possono fornire a terzi accesso e controllo remoto (backdoor) sul PC, consentendo così il furto delle password e la raccolta di informazioni sul sistema.

- **Tracur** (5): Tracur [11] è una famiglia di trojan che può reindirizzare le ricerche sul web. Questo viene fatto per generare entrate per gli autori del malware attraverso frodi pubblicitarie online. I trojan dirottano i collegamenti dei risultati di ricerca dai motori di ricerca e li reindirizzano a pagine web diverse.

- **Kelihos_ver1** (6): I malware kelihos_ver1 [11] sono trojan che agiscono come downloader, fornendo una backdoor per l'accesso remoto al sistema infetto.

- **Obfuscator.ACY** (7): Obfuscator.ACY [11] è un termine generico utilizzato per descrivere malware che impiegano tecniche di offuscamento, rendendo difficile la loro classificazione in una categoria specifica.

- **Gatak** (8): I malware gatak [11] sono trojan che si fingono aggiornamenti per applicazioni legittime. Questi malware consentono agli attaccanti di controllare da remoto il sistema infetto e rubare informazioni sensibili.

La Figura 4.4 mostra la quantità di malware per famiglia in BIG 2015.

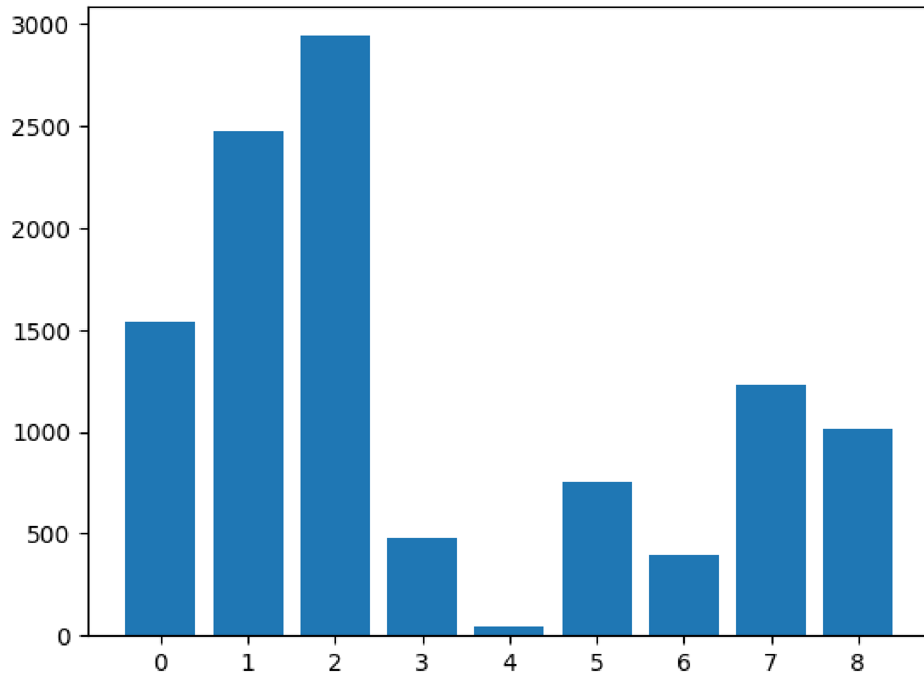

Figura 4.4: Distribuzione dei malware in BIG 2015

4.3.2 Estrazione delle features

Sono stati sviluppati diversi approcci per la risoluzione del problema della classificazione dei malware in BIG 2015. Questa sezione del progetto si basa sull'analisi del manifest generato dal disassembler IDA, ed in particolare alla parte del codice assembly del programma.

L'analisi consiste infatti nel contare il numero delle seguenti istruzioni persenti nel codice assembly: call, add, cdq, cld, cli, cmc, cmp, cwd, daa, dd, dec, dw, endp, faddp, fchs, fdiv, fdivr, fistp, fld, fstp, fword, fxch, imul, in, inc, ins, jb, je, jg, jl, jmp, jnb, jno, jo, jz, lea, mov, mul, not, or, out, outs, pop, push, rcl, rcr, rep, ret, rol, ror, sal, sar, sbb, scas, shl, shr, sidt, stc, std, sti, stos, sub, test, wait, xchg e xor.

Questi dati sono stati poi aggregati nel dataset utilizzato per l'addestramento e il testing della rete dedicata alla classificazione.

4.3.3 Operazioni preliminari

Seguendo un processo analogo a quello descritto nel paragrafo 4.2 *Riconoscimento di malware*, il dataset generato precedentemente viene caricato in un

DataFrame. Da questo vengono estratte le feature e le etichette di classe, che vengono standardizzate mediante lo StandardScaler e utilizzate per la creazione dei set di training e di test. Il dataset viene di nuovo suddiviso in modo tale che l'80% dei dati venga utilizzato per l'addestramento, mentre il restante 20% è riservato per il testing. I vettori contenenti i dati vengono successivamente trasformati in tensori, che sono poi utilizzati per la creazione dei TensorDataset e dei DataLoader.

4.3.4 Definizione del modello

L'Algoritmo 13 mostra la definizione del $(66, 32, 16, 9)$-MLP utilizzato per la classificazione dei malware

Algoritmo 13 $(66, 32, 16, 9)$-MLP utilizzato per la classificazione di malware

```
import torch.nn as nn


class MultiClassMLP(nn.Module):
    def __init__(self):
        super(MultiClassMLP, self).__init__()

        self._input__h_1 = nn.Linear(66,32)
        self._h_1__h_2 = nn.Linear(32, 16)
        self._h_2__output = nn.Linear(16, 9)

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self._input__h_1(x))
        x = self.relu(self._h_1__h_2(x))
        x = self._h_2__output(x)
        return x
```

Si noti che anche in questo caso la funzione di attivazione utilizzata dai neuroni negli strati nascosti è la ReLU, mentre lo strato di output non viene attivato.

4.3.5 Funzione di loss e ottimizzatore

Ancora una volta definiamo una funzione di loss per misurare l'errore compiuto dal modello e un algoritmo di ottimizzazione da utilizzare per aggiornare i parametri dei neuroni durante l'addestramento.

Cross Entropy

In questo caso, dal momento che la classificazione non è binaria, ma multiclasse, la funzione di loss utilizzata è la cross entropy.

La **cross entropy** è una funzione di perdita più generale che misura la differenza tra due distribuzioni di probabilità. Viene utilizzata principalmente nei problemi di classificazione multi-classe. In questo contesto, la cross entropy misura quanto le previsioni del modello si discostano dalle etichette reali. La formula della cross entropy per una singola istanza con una distribuzione di probabilità reale p e una distribuzione di probabilità prevista q è:

$$CE(p, q) = -\sum_i p_i \log(q_i) \tag{4.2}$$

Dove p_i è la probabilità reale per la classe i e q_i è la probabilità prevista per la classe i. Questa formula viene sommata su tutte le classi del problema.

Adam

Anche per questa rete, l'algoritmo di ottimizzazione utilizzato in fase di addestramento è Adam.

L'Algoritmo 14 mostra la definizione della rete, della funzione di perdita e dell'ottimizzatore utilizzati per la classificazione dei malware.

Algoritmo 14 Definizione della rete, della loss function e dell'ottimizzatore

```
1 model = MultiClassMLP()
2 loss_fn = nn.CrossEntropyLoss()
3 optimizer = optim.Adam(model.parameters(), lr=0.01)
```

4.3.6 Addestramento del modello

Ancora una volta, l'addestramento del modello si svolge attraverso un ciclo che copre tutte le epoche specificate (vedi Algoritmo 15). Per ogni epoca, il modello elabora il dataset in batch. Per ogni batch di dati, il modello effettua previsioni basate sull'output della funzione **argmax** applicata ai neuroni di output. La correttezza di queste previsioni viene valutata calcolando la perdita attraverso la funzione di loss, che misura la differenza tra i valori predetti e i valori reali. Dopo il calcolo della perdita, il modello aggiorna i suoi parametri interni utilizzando l'algoritmo Adam.

Dopo l'elaborazione di ciascun batch, le statistiche cumulative di perdita e

precisione vengono aggiornate seguendo gli stessi principi del paragrafo precedente.

Algoritmo 15 Addestramento del modello

```
num_epochs = 128
loss_hist = [0] * num_epochs
accuracy_hist = [0] * num_epochs

for epoch in tqdm(range(num_epochs)):
    model.train()
    total_loss = 0
    correct_predictions = 0
    total_samples = 0

    for inputs, labels in train_loader:
        # Forward pass
        pred = model(inputs)
        loss = loss_fn(pred, labels.long())

        # Backward pass and optimization
        loss.backward()  # calcolo i gradienti
        optimizer.step()  # aggiornamento dei pesi
        optimizer.zero_grad()

        # Calculate total loss
        total_loss += loss.item() * labels.size(0)

       # Calculate total accuracy
        with torch.no_grad():
            prediction = torch.argmax(pred, dim=1)
            correct_predictions += (prediction == labels).sum().item()
            total_samples += labels.size(0)

    # Calculate average loss and accuracy for the epoch
    loss_hist[epoch] = total_loss / total_samples
    accuracy_hist[epoch] = correct_predictions / total_samples
```

4.3.7 Analisi dell'addestramento

Per valutare i progressi e le prestazioni del modello durante la fase di addestramento, viene generato un grafico che mostra l'andamento della funzione di perdita (loss) e dell'accuratezza (accuracy) in funzione delle epoche.

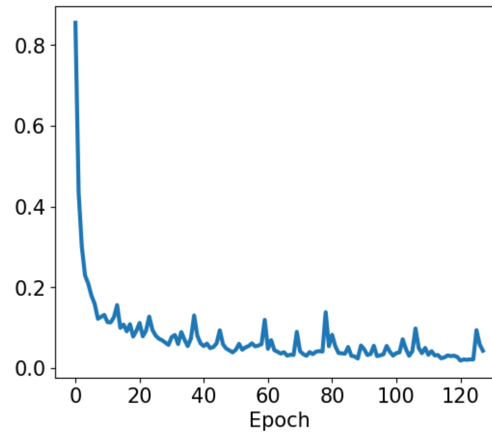

Figura 4.5: Grafico che mostra l'andamento della funzione di perdita durante il processo di addestramento

Il grafico della loss (Figura 4.5) rivela come la rete migliori la sua capacità di ridurre l'errore tra le previsioni e le etichette reali con il passare delle epoche. Il grafico dell'accuratezza (Figura 4.6), invece, mostra la percentuale di classificazioni corrette rispetto al totale.

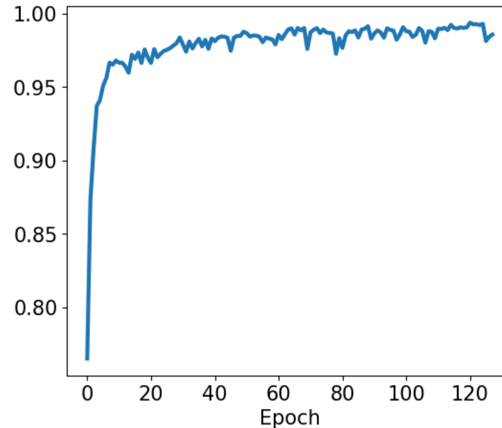

Figura 4.6: Grafico che mostra la precisione del modello durante il processo di addestramento

4.3.8 Valutazione del modello

Dopo aver completato la fase di addestramento, il modello viene testato utilizzando il set di test, al fine di valutarne le prestazioni in scenari non incontrati

durante l'addestramento. L'Algoritmo 16 mostra il processo di testing del modello.

Algoritmo 16 Valutazione del modello

```
model.eval()
with torch.no_grad():

    top_3_corr = 0
    test_predictions = []
    test_labels = []

    all_out_probs = []

    for inputs, labels in test_loader:
        outputs = model(inputs)
        outputs = torch.nn.functional.softmax(outputs, dim=1)

        m, predictions = torch.max(outputs.data, 1)


        test_predictions.extend(predictions.numpy())
        test_labels.extend(labels.numpy())
        all_out_probs.extend(outputs.numpy())

        for label, output in zip(labels, outputs.data):
            top_conf, topk_pred = torch.topk(output, 3, dim=0)

            if label in topk_pred:
                top_3_corr += 1
```

In questo caso, per ciascun file del batch, il modello prevede l'etichetta di classe basandosi sull'output della funzione **softmax** applicata ai neuroni di output. In particolare, viene scelta come etichetta quella della classe con la probabilità più alta. Queste previsioni verranno poi confrontate con le etichette reali per determinare il numero di previsioni corrette.

È importante notare che viene definita la variabile `top_3_corr` per verificare se la classe corretta si trova tra le tre classi con le probabilità più alte.

Capitolo 5

Risultati

In questa sezione verranno analizzati i risultati ottenuti dai modelli durante la fase di testing. Inoltre, saranno discussi i risultati del test dell'intero sistema sul dataset DikeDataset [13], utilizzando una tecnica nota come **zero-shot classification**, che consiste nel valutare un modello su campioni che non sono stati osservati durante l'addestramento.

5.1 Strumenti per l'analisi

Per valutare le prestazioni di una rete neurale, vengono utilizzate diverse metriche a seconda del tipo di problema (e.g., classificazione, regressione, ecc.). In particolare, in questa fase del progetto, le metriche utilizzate sono l'**accuracy**, la **precision**, la **recall** e la **confusion matrix**.

Inoltre, completando le definizioni introdotte nel paragrafo 4.2.10 *Valutazione del modello*:

- True Positives (TP) rappresenta il numero di osservazioni correttamente predette come positive,
- True Negatives (TN) rappresenta il numero di osservazioni correttamente predette come negative,
- False Positives (FP) rappresenta il numero di osservazioni erroneamente predette come positive (anche detti Errori di Tipo I),
- False Negatives (FN) rappresenta il numero di osservazioni erroneamente predette come negative (anche detti Errori di Tipo II).

5.1.1 Accuracy

L'accuracy (accuratezza) rappresenta la quantità di predizioni corrette (sia positive che negative) rispetto al totale delle osservazioni. E' quindi definita come

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

5.1.2 Precision

La precision (precisione) rappresenta la quantità di predizioni positive corrette rispetto al totale delle predizioni positive. Si calcola quindi come

$$\text{precision} = \frac{TP}{TP + FP} \tag{5.2}$$

In altre parole, rappresenta la qualità delle predizioni positive.

5.1.3 Recall

La recall (sensibilità) rappresenta la quantità di osservazioni positive reali che sono state correttamente identificate dal modello. E' quindi definita come

$$\text{recall} = \frac{TP}{TP + FN} \tag{5.3}$$

In altre parole, misura la capacità del modello di trovare tutti i positivi.

5.1.4 Confusion Matrix

La confusion matrix (matrice di confusione) è uno strumento fondamentale nell'ambito del machine learning per valutare le prestazioni di un modello di classificazione. In pratica, la matrice di confusione è una tabella che permette di visualizzare e confrontare le predizioni fatte dal modello rispetto ai valori reali.

La matrice di confusione è fondamentale per comprendere non solo l'efficacia generale di un modello, ma anche per analizzare come si comporta in termini di errori specifici, come la tendenza a predire falsi positivi o falsi negativi.

5.2 Riconoscimento di malware

Per quanto riguarda il problema del riconoscimento di malware, la confusion matrix generata dal testing della rete del paragrafo 4.2 *Riconoscimento di malware* è mostrata in Figura 5.1

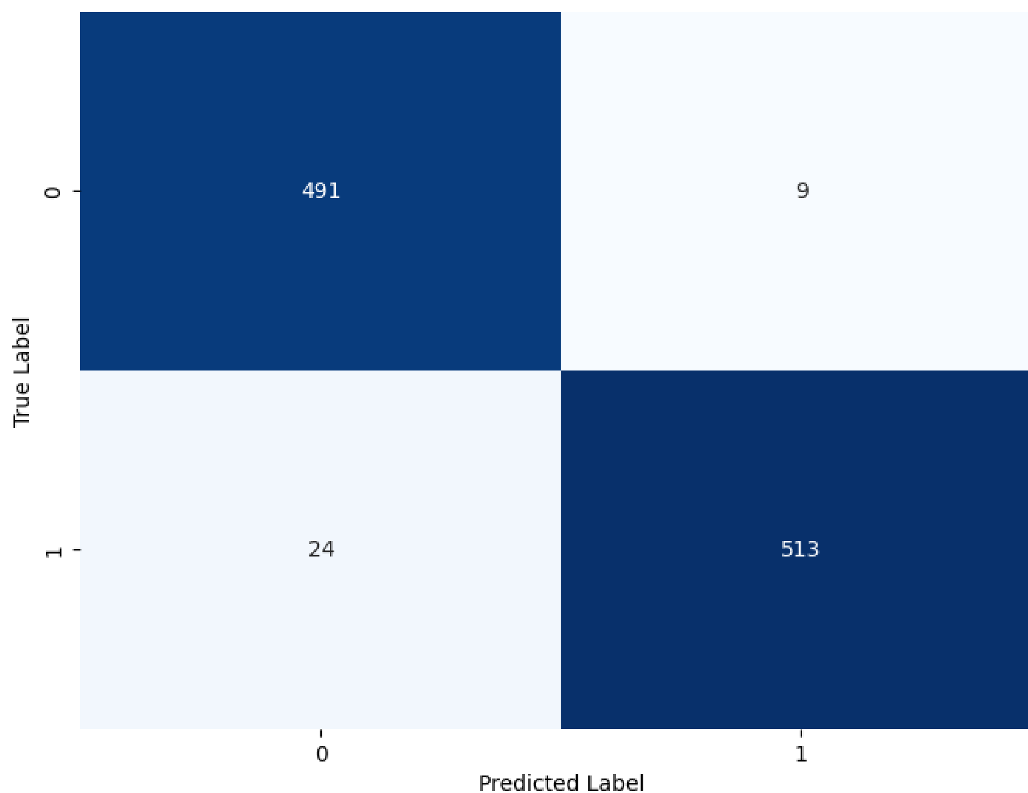

Figura 5.1: Confusion matrix generata dal modello per il riconoscimento di malware

Basandosi sulle formule precedentemente definite, si ricava che:

1. L'**accuracy** del modello in fase di test è del 96.82%

2. La **precision** del modello in fase di test è del 98.28%

3. La **recall** del modello in fase di test è del 95.53%

Possiamo quindi affermare che la rete, nonostante esegua un'analisi di tipo statico, dimostra ottime capacità nell'individuare malware sconosciuti. Ha infatti identificato correttamente 513 nuovi malware su 537 presentati, superando i limiti delle analisi statiche tradizionali basate su firme. Inoltre, la quantità di falsi positivi rispetto al totale delle predizioni positive è piuttosto bassa (9 su 522). Tuttavia, come indicato dalla matrice di confusione e dalla recall calcolata, la rete tende a generare più falsi negativi rispetto ai falsi positivi.

5.3 Classificazione di malware

Per quanto riguarda invece il problema della classificazione di malware, la confusion matrix generata dal testing della rete del paragrafo 4.3 *Classificazione di malware* è mostrata in Figura 5.2

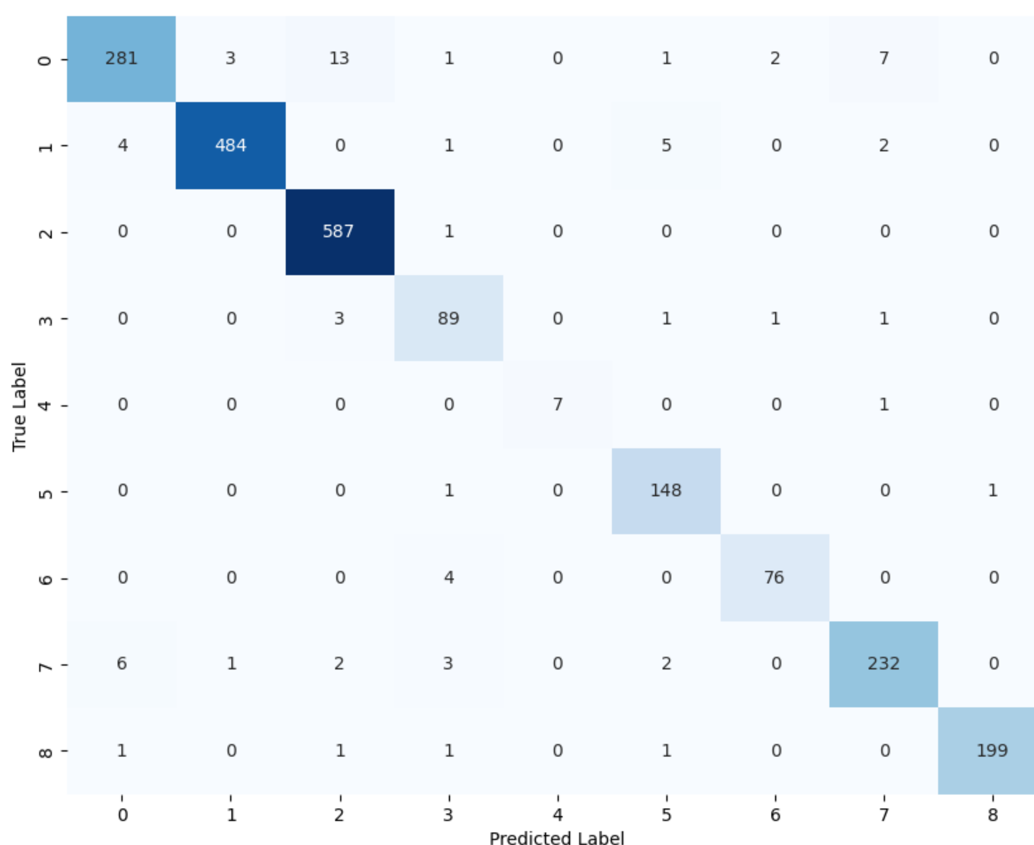

Figura 5.2: Confusion matrix generata dal modello per la classificazione di malware

In questo caso, prima di presentare i risultati del testing, è necessario definire brevemente il concetto di **top-3 accuracy**, basato su quello di accuracy, che in questo contesto chiameremo **top-1 accuracy**.

La top-3 accuracy rappresenta la percentuale di volte in cui la classe corretta dell'elemento testato si trova tra le prime tre classi previste dal modello. Questa metrica è utile quando è accettabile che la previsione corretta sia tra le prime opzioni suggerite dal modello, anche se non è la prima in assoluto. È particolarmente rilevante in applicazioni dove un insieme ristretto di suggerimenti può essere comunque utile all'utente.

In questo caso, i risultati sono invece i seguenti:

1. La **top-1 accuracy** del modello in fase di test è del 96.73%

2. La **top-3 accuracy** del modello in fase di test è del 98.85%

5.4 Analisi di DikeDataset

Per valutare le prestazioni del sistema in un ambiente sconosciuto, la pipeline è stata testata utilizzando i file del DikeDataset. Questo set di dati etichettati comprende file PE e OLE, un formato utilizzato nelle applicazioni Word, Excel e PowerPoint, sia benigni che dannosi.

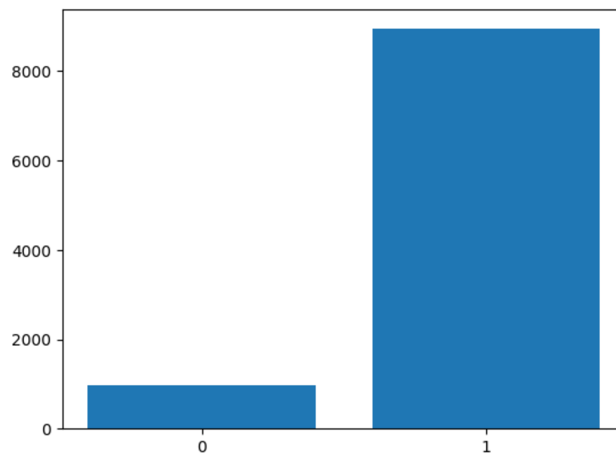

Figura 5.3: Quantità di file benigni (0) e malware (1) in DikeDataset

Innanzitutto, sono stati estratti gli header dei 9902 file PE contenuti in DikeDataset attraverso la libreria **pefile** di Python. La Figura 5.3 mostra la quantità di malware e file benigni elaborati.

Successivamente, è stato applicato il modello per il riconoscimento di malware ai file estratti, ottenendo i seguenti risultati:

1. L'**accuracy** del modello è del 84.19%

2. La **precision** del modello è del 99.58%

3. La **recall** del modello è del 82.83%

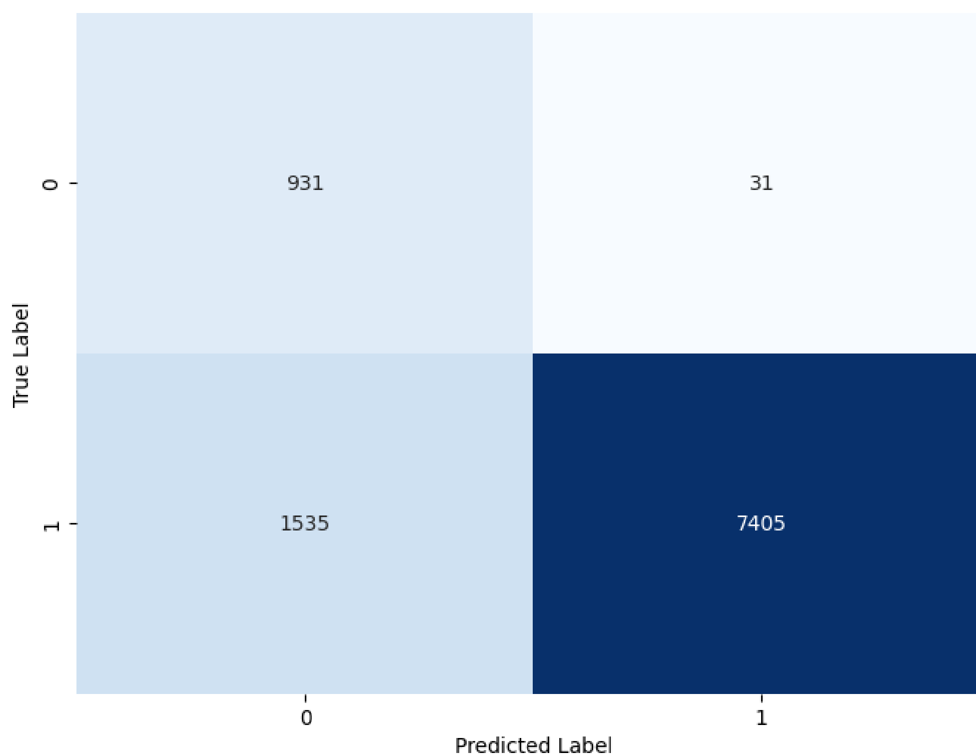

Figura 5.4: Confusion matrix generata dal testing su DikeDataset

Si noti che l'accuracy, pur rimanendo alta, è diminuita rispetto ai test effettuati su CLaMP. La Figura 5.4 mostra la matrice di confusione risultante.

È importante sottolineare che, come spiegato nel paragrafo 2.3.2 *Tecniche di offuscamento*, i malware spesso utilizzano tecniche come il polimorfismo o il metamorfismo, che modificano il contenuto del file e rendono inefficaci questo tipo di analisi. Questo significa che i file contenuti nel DikeDataset potrebbero essere radicalmente diversi rispetto a quelli su cui il modello è stato addestrato e testato. Nonostante ciò, la capacità del modello di predire la pericolosità dei file non ne risente eccessivamente. Bisogna anche considerare che il calo delle performance potrebbe essere semplicemente dovuto alla maggiore numerosità di DikeDataset rispetto a CLaMP.

A questo punto, dopo aver disassemblato i file maligni utilizzando la libreria **capstone** di Python, è stato applicato il modello per la classificazione ai dati estratti.

La Figura 5.5 mostra i risultati ottenuti.

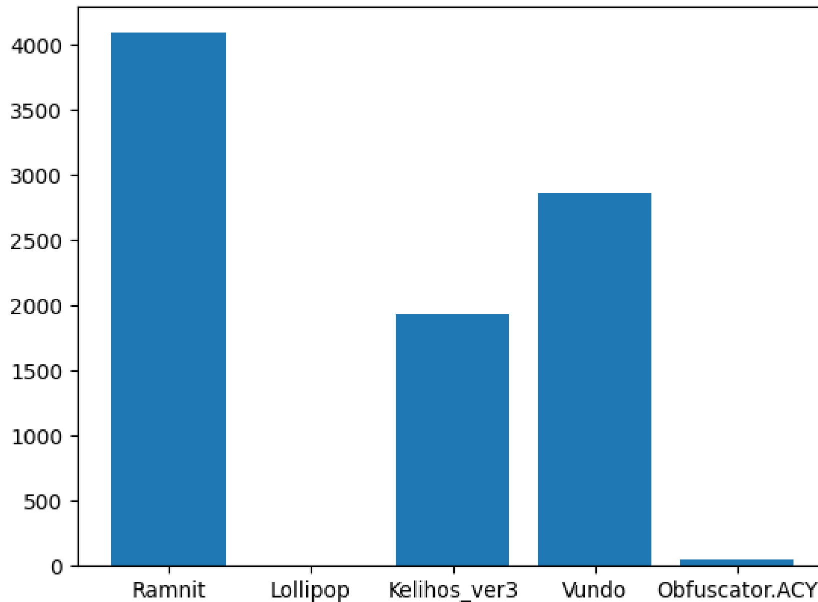

Figura 5.5: Distribuzione predetta per i malware di DikeDataset

In questo caso, non è possibile fare un paragone con la classificazione presente in DikeDataset, poiché le tecniche utilizzate per l'assegnazione delle etichette non corrispondono. Infatti, sulla base della presentazione fatta nel paragrafo 4.3.1 *Famiglie di malware*, è possibile assegnare a ciascuna famiglia le seguenti etichette:

- **Ramnit**: Worm, Trojan, Downloader, Spyware
- **Lollipop**: Adware, Downloader, Spyware
- **Kelihos ver3**: Trojan, Downloader, Backdoor
- **Vundo**: Trojan, Spyware, Ransomware
- **Simda**: Trojan, Downloader, Backdoor
- **Tracur**: Trojan
- **Kelihos ver1**: Trojan, Downloader, Backdoor
- **Obfuscator.ACY**: Generico
- **Gatak**: Trojan, Backdoor

mentre in DikeDataset i malware sono classificati perlopiù come trojan generici.

Capitolo 6

Conclusioni

Concludiamo discutendo i risultati ottenuti e i possibili sviluppi futuri del progetto.

6.1 Discussione dei risultati

In generale, le tecniche più utilizzate per il riconoscimento di malware sconosciuti sono quelle di tipo dinamico, come l'analisi comportamentale. Tuttavia, poiché queste tecniche richiedono l'esecuzione del malware, devono essere condotte in un ambiente sicuro, sotto la supervisione di un esperto, e richiedono un tempo considerevole per l'elaborazione di ogni file. Di conseguenza, questo tipo di analisi non può essere eseguita sui dispositivi degli utenti finali, ma è necessario ricorrere ad analisi di tipo statico. Queste ultime, infatti, non comportano l'esecuzione dei file, sono più rapide, ma risultano quasi inefficaci contro i malware sconosciuti.

Per quanto riguarda il riconoscimento di malware, la tecnica proposta è un'analisi di tipo statico, che può quindi essere applicata anche sui dispositivi degli utenti. Questa tecnica si è dimostrata efficace nel riconoscimento di malware sconosciuti e, grazie alla sua rapidità (i 9902 file del DikeDataset sono stati elaborati in circa $0,4$ secondi, stima calcolata utilizzando la funzione `perf_counter()` della libreria **time** di Python), può essere utilizzata come supporto all'interno degli antivirus commerciali.

Per quanto riguarda la classificazione dei malware, la tecnica proposta si è dimostrata efficace nel suddividerli in diverse famiglie. Tuttavia, il modello presentato è utile solo per definire una somiglianza tra il file esaminato e una delle famiglie presenti nel dataset originale. Per una classificazione più accurata, è necessario affidarsi a tecniche di analisi di tipo comportamentale.

6.2 Sviluppi futuri

Per migliorare l'efficacia dei due metodi, è fondamentale aggiungere nuovi campioni ai dataset utilizzati. A tal proposito, una versione estesa di CLaMP è disponibile presso `https://github.com/alessio-russo/Malware-Detection-and-Classification`, contenente 15086 campioni (rispetto ai 5184 dell'originale), estratti da CLaMP e da DikeDataset. Inoltre, nello stesso repository, sono disponibili gli script utilizzati per estrarre gli header e il codice disassemblato.

Sarebbe inoltre interessante cercare di costruire una rete unica che combini le conoscenze di entrambi i modelli. Un'opzione potrebbe essere quella di creare un MLP con 10 neuroni in input (il neurone di output del riconoscitore e i nove neuroni di output del classificatore) e 10 neuroni in output (corrispondenti alle nove classi di BIG 2015 più la classe che identifica i software benigni).

Bibliografia

[1] S. Raschka, Y. Liu, and V. Mirjalili, *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, 2022.

[2] "Neural network models (supervised)," accessed: Giu. 21, 2024. [Online]. Available: https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[3] Ömer Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[4] S. Morgan, "2023 cybersecurity almanac: 100 facts, figures, predictions and statistics," *Cybercrime Magazine*, 2023, accessed: Giu. 12, 2024. [Online]. Available: https://cybersecurityventures.com/cybersecurity-almanac-2023

[5] "Windows 10: Powering the world with one billion monthly active devices," 2020, accessed: Giu. 12, 2024. [Online]. Available: https://news.microsoft.com/apac/2020/03/17/windows-10-powering-the-world-with-one-billion-monthly-active-devices

[6] "Virustotal statistics," accessed: Giu. 12, 2024. [Online]. Available: https://www.virustotal.com/gui/stats

[7] S. Talukder and Z. Talukder, "A survey on malware detection and analysis tools," *International Journal of Network Security & Its Applications (IJNSA)*, vol. 12(2), p. 37–57, 2020.

[8] F. Cohen, "Computer viruses: theory and experiments," *Computers & security*, vol. 6, no. 1, pp. 22–35, 1987.

[9] D. M. Chess and S. R. White, "An undetectable computer virus," in *Proceedings of Virus Bulletin Conference*, vol. 5, no. 4. Orlando, 2000, pp. 409–422.

[10] "Microsoft malware classification challenge (big 2015)," accessed: Giu. 21, 2024. [Online]. Available: https://www.kaggle.com/c/malware-classification

[11] "Microsoft security intelligence," accessed: Giu. 22, 2024. [Online]. Available: https://www.microsoft.com/en-us/wdsi

[12] J. Zorabedian, "Europol takedown of ramnit botnet frees 3.2 million pcs from cybercriminals' grasp," 2015, accessed: Giu. 22, 2024. [Online]. Available: https://news.sophos.com/en-us/2015/02/27/europol-takedown-of-ramnit-botnet-frees-3-2-million-pcs-from-cybercriminals-grasp/

[13] "Dikedataset," accessed: Giu. 22, 2024. [Online]. Available: https://github.com/iosifache/DikeDataset

[14] "Formato pe," 2024, accessed: Giu. 16, 2024. [Online]. Available: https://learn.microsoft.com/it-it/windows/win32/debug/pe-format

Ringraziamenti

Desidero esprimere i miei più sinceri ringraziamenti a tutti coloro che mi hanno accompagnato nel raggiungimento di questo importante traguardo.

Un sentito grazie va a tutti i miei amici e a tutte le persone che mi hanno voluto bene e sostenuto durante questi anni. La vostra presenza e il vostro supporto hanno reso più leggero questo duro percorso. In particolare, vorrei ringraziare Riccardo. Il tuo incoraggiamento costante e la tua amicizia sono stati fondamentali nei momenti di difficoltà.

Appendice A

Portable Executable

Il formato Portable Executable (PE) [14] è un tipo di file utilizzato per eseguibili, file oggetto, librerie condivise e driver di dispositivi nelle versioni a 32-bit e 64-bit del sistema operativo Microsoft Windows. Il file PE contiene varie informazioni essenziali per il caricamento e l'esecuzione del software nel sistema operativo:

- **Intestazione.** Fornisce i metadati essenziali sul file, come il tipo di macchina per cui è stato compilato il codice e la struttura delle sezioni del file.

- **Sezioni.** Possono includere codice eseguibile, dati, risorse, informazioni di importazione ed esportazione e altre informazioni essenziali per l'esecuzione e la gestione del programma.

- **Tabella delle importazioni.** Elenca le funzioni di altre DLL di cui il programma necessita per funzionare.

- **Tabella delle esportazioni.** Elenca le funzioni che il file mette a disposizione per altri programmi o DLL.

- **Risorse.** Icone, menu e stringhe di dialogo utilizzate dall'applicazione.

Il formato PE è un diretto discendente del formato **COFF** (Common Object File Format), utilizzato in molti altri sistemi operativi per memorizzare informazioni eseguibili. Nel contesto di Windows, il formato PE supporta funzionalità specifiche come il caricamento dinamico di librerie e la gestione avanzata della memoria, essenziali per il funzionamento delle moderne applicazioni Windows.

Intestazione

L'header PE è una parte cruciale del file, contenente metadati e informazioni strutturali che descrivono come il sistema operativo deve caricare ed eseguire il file stesso. L'header è suddiviso in diverse sezioni, ciascuna con scopi specifici. Di seguito sono riportati i principali campi di ogni sezione:

DOS Header

- **e_magic**: I primi due byte che contengono la firma MZ, indicando che si tratta di un file eseguibile.
- **e_lfanew**: Un puntatore all'inizio dell'header PE, che segue immediatamente il DOS header.

Questo header serve principalmente per la compatibilità all'indietro con le applicazioni DOS.

PE Header

Segue il DOS header e inizia con la firma 'PE\0\0' (50 45 00 00 in esadecimale). Il PE header è composto da:

a. File Header

- **Machine**: Specifica l'architettura della macchina per cui il file è stato compilato.
- **NumberOfSections**: Il numero di sezioni che seguono l'header.
- **TimeDateStamp**: Data e ora della creazione del file.
- **PointerToSymbolTable** e **NumberOfSymbols**: Utilizzati per il debug.
- **SizeOfOptionalHeader**: Dimensione dell'Optional Header.
- **Characteristics**: Flag che indicano le caratteristiche del file (e.g se è un'applicazione a finestre GUI o a console).

b. Optional Header

Nonostante il nome, per i file PE è obbligatorio. Contiene informazioni cruciali per il caricamento e l'esecuzione:

- **Magic**: Indica se il file è a 32 bit o a 64 bit.
- **AddressOfEntryPoint**: L'indirizzo relativo del punto di entrata del codice eseguibile.
- **ImageBase**: L'indirizzo di base preferito in memoria per il caricamento del file eseguibile.
- **SectionAlignment** e **FileAlignment**: Allineamento delle sezioni in memoria e nel file su disco.
- **SizeOfImage**: Dimensione totale dell'immagine caricata in memoria.
- **SizeOfHeaders**: Dimensione totale degli header nel file.
- **Subsystem**: Il sottosistema richiesto per l'esecuzione, ad esempio Windows GUI o console.
- **DLLCharacteristics**: Attributi specifici per le DLL.
- **StackReserveSize**, **StackCommitSize**, **HeapReserveSize**, **HeapCommitSize**: Dimensioni prenotate e commit per stack e heap.

Section Headers

Dopo l'Optional Header, ci sono gli header di sezione, uno per ogni sezione del file. Questi header descrivono le caratteristiche delle sezioni individuali, come il nome della sezione, la dimensione, gli indirizzi virtuali, e le autorizzazioni di accesso (lettura, scrittura, esecuzione).