UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Uno Studio Empirico sulla Generalizzabilità dei Risultati di Modelli di Deep Learning per l'Identificazione di Vulnerabilità

RELATORE

Prof. **Fabio Palomba**

Università degli studi di Salerno

CANDIDATO

Simone Della Porta

Matricola: 0512109134

Anno Accademico 2021-2022

Se ci aspettiamo che una macchina sia infallibile, essa non può essere anche intelligente

Alan Turing

Sommario

Con il continuo aumento dell'utilizzo delle tecnologie, e quindi del software, in tutte le aree della società, una vulnerabilità (un difetto di sicurezza, un problema tecnico o una debolezza riscontrata nei sistemi software) può potenzialmente causare danni significativi alle aziende e alla vita delle persone; infatti, gli attacchi informatici sono una minaccia continua per aziende, governi e utenti. Il tasso di cyberattacchi aumenta molto rapidamente e questi ultimi possono provocare ingenti perdite. Questi problemi sono principalmente causati dalle vulnerabilità del prodotto software. Per questo motivo e per una maggiore consapevolezza da parte degli utenti finali è in continuo aumento la richiesta di software sicuro. In questa tesi verrà trattato questo problema delle vulnerabilità facendo prima una panoramica generale sulle varie tipologie per poi concentrarsi su quelle software riportandone anche alcuni esempi. Dopodiché verranno analizzate le principali tecniche di individuazione di queste ultime e verranno discussi alcuni modelli esistenti per il loro rilevamento. In particolare però questa tesi sarà incentrata sui modelli di Deep Learning (DL) che preso in input un frammento di codice effettueranno una predizione indicando se il codice in input è vulnerabile o meno. Il progetto guida utilizzato per lo sviluppo di questa tesi è ReVeal nel quale viene fatto un confronto tra quattro tecniche pre-esistenti evidenziandone i problemi per poi proporre una propria soluzione. Dopo un'analisi approfondita di tale progetto si andrà ad effettuare uno studio empirico sulla generalizzabilità dei risultati ottenuti da tale modello cambiando l'architettura di rete neurale utilizzata.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

Questo capitolo riporta una panoramica generale del lavoro svolto per questa tesi

1.1 Contesto applicativo

Al giorno d'oggi la sicurezza dei dati è un aspetto cruciale durante lo sviluppo del software. Quando questa sicurezza non è garantita si parla di vulnerabilità. Con il continuo aumento dell'utilizzo delle tecnologie, e quindi del software, in tutte le aree della società, una vulnerabilità può potenzialmente causare danni significativi alle aziende e alla vita delle persone; infatti, gli attacchi informatici sono una minaccia continua per aziende, governi e utenti. Il tasso di cyberattacchi aumenta molto rapidamente e questi ultimi possono provocare ingenti perdite. Tali perdite sono legate alla grande mole di informazioni sensibili che transitano sulla rete ogni giorno. Queste informazioni possono essere *"rubate"* da hacker che sfruttano diversi tipi di vulnerabilità per sviluppare codice malevolo con il quale reperire tali informazioni. In particolare vengono sfruttate le vulnerabilità software per poter effettuare ciò.

Per i motivi sopra elencati e per una maggiore consapevolezza da parte degli utenti finali è in aumento la richiesta di software sicuro. Infatti, quello della sicurezza è un campo in continua fomentazione attualmente, anche perché, come mostra la Figura 1.1, il numero di attacchi informatici negli ultimi anni è aumentato in modo drastico. Per questo motivo c'è una crescente necessità di rendere il software sicuro e prevenire possibili attacchi provenienti dallo sfruttamento di vulnerabilità software.

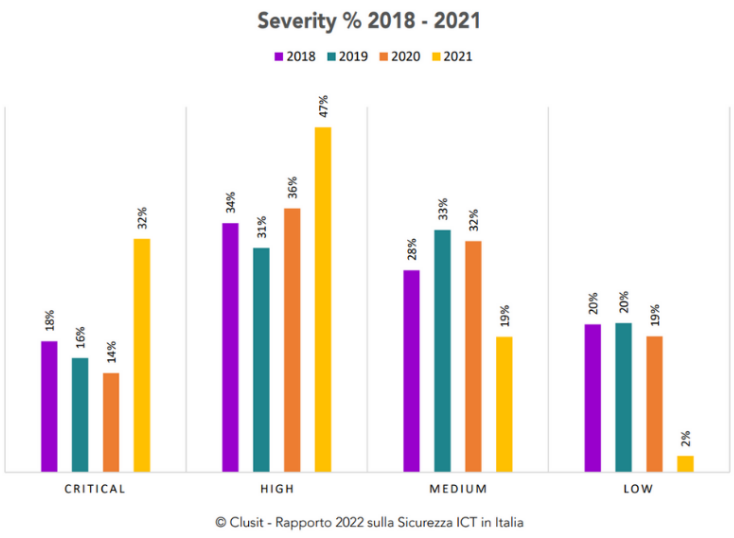

Figura 1.1: Statistiche attacchi informatici degli ultimi anno

1.2 Motivazioni e Obiettivi

Come riportato nella sezione precedente la sicurezza informatica è uno degli argomenti di maggiore interesse negli ultimi tempi. Per contrastare la crescita esponenziale degli attacchi informatici, osservata nella Figura 1.1, si sta cercando di sviluppare delle tecniche per individuare in modo automatico le vulnerabilità software. Esistono già alcune tecniche per effettuare ciò, ma esse soffrono di un elevato tasso di falsi positivi e falsi negativi, cioè queste tecniche sbagliano la predizione di un determinato frammento di codice etichettandolo come vulnerabile quando in realtà non lo è, primo caso, oppure come non vulnerabile quando in realtà lo è, secondo caso. I recenti progressi ottenuti nel **Deep Learning (DL)** hanno suscitato interesse nell'utilizzo di tali tecniche per il rilevamento automatico di vulnerabilità con elevata precisione.
In questa tesi viene trattato l'argomento delle vulnerabilità software ed in particolare si concentra l'attenzione su modelli automatici per l'individuazione di vulnerabilità basati su deep learning. L'obiettivo di questa tesi è quello di fare un'analisi preliminare dei modelli esistenti per andare ad effettuare successivamente uno studio empirico sulla generalizzabilità dei risultati ottenuti da tali modelli rispetto all'architettura di rete neurale utilizzata.

1.3 Risultati

Per ottenere i risultati dello studio sono state addestrare diverse varianti della pipeline proposta dagli autori del paper di riferimento utilizzando un'architettura di rete neurale diversa da quella utilizzata per tale progetto. I risultati ottenuti dallo studio evidenziano che l'utilizzo di architetture di rete neurale meno complesse dal punto di vista computazionale e più semplici da configurare portano ad un peggioramento delle prestazioni del modello sui medesimi dati. Tale risultato permette di concludere che, modelli basati su architetture semplici non sono utilizzabili in un contesto reale, quindi bisogna utilizzare modelli più complessi per poter ottenere risultati migliori. Inoltre c'è da affermare che neanche la pipeline proposta dagli autori ha prestazioni ottime, essa sicuramente ha prestazioni migliori rispetto alle varianti proposte, però può essere ulteriormente modificata per ottenere prestazioni migliori. Tali modifiche potrebbero consistere nell'andare ad utilizzare un'architettura di rete neurale più avanzata, come quella basata su transformers, ed inoltre potrebbe essere migliorato il dataset estendendolo a più linguaggi di programmazione in modo tale da generalizzare le prestazioni del modello a più linguaggi.

1.4 Struttura della tesi

Il Capitolo 2 riporta lo stato dell'arte ed i lavori presenti in letteratura sugli aspetti di ricerca per l'identificazione di vulnerabilità. In particolare viene dapprima fatta una panoramica generale sulle varie tipologie di vulnerabilità esistenti, per poi concentrarsi sulle vulnerabilità software, riportandone alcuni esempi. Successivamente vengono analizzate alcune tecniche esistenti in letteratura per la loro detection ed infine viene introdotta la detection delle vulnerabilità attraverso l'utilizzo del Deep Learning, facendo prima una panoramica generale sulle reti neurali.
Il Capitolo 3 riporta uno studio approfondito di quello che è il progetto su cui è incentrata questa tesi ovvero il lavoro svolto da Chakraborty et al.[1]. In questo capitolo vengono analizzati inizialmente 4 progetti esistenti proprio come fatto dagli autori del paper e poi viene analizzata la pipeline da loro proposta evidenziandone eventualmente alcune limitazioni. Successivamente viene fatto uno studio empirico sulla generalizzabilità dei risultati dei modelli di deep learning per la previsione di vulnerabilità rispetto all'architettura di rete utilizzata. A tal scopo vengono presentate alcune varianti della pipeline proposta che utilizzano una diversa architettura di rete neurale.
Il Capitolo 4 riporta i risultati ottenuti dall'esecuzione dei modelli sviluppati e riporta un confronto delle metriche ottenute per ciascun modello.
Infine il Capitolo 5 riporta una valutazione finale sullo studio empirico effettuato ed inoltre riporta la proposta di alcuni sviluppi futuri effettuabili per migliorare le prestazioni del modello.

CAPITOLO 2

Stato dell'arte

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nello studio dell'utilizzo delle reti neurali per l'identificazione di vulnerabilità software.

2.1 Cos'è una vulnerabilità

Al giorno d'oggi la sicurezza dei dati è un aspetto cruciale durante lo sviluppo del software. Quando questa sicurezza non è garantita si parla di **vulnerabilità**.
Con il continuo aumento dell'utilizzo delle tecnologie, e quindi del software, in tutte le aree della società, una vulnerabilità (un difetto di sicurezza, un problema tecnico o una debolezza riscontrata nei sistemi software) può potenzialmente causare danni significativi alle aziende e alla vita delle persone; infatti, gli attacchi informatici sono una minaccia continua per aziende, governi e utenti [2].
Il tasso di cyberattacchi aumenta molto rapidamente e questi ultimi possono provocare ingenti perdite. Per questo motivo e per una maggiore consapevolezza da parte degli utenti finali è in continuo aumento la richiesta di software sicuro. Per software sicuro si intende software che può continuare a funzionare in situazioni malevole, in particolare il software dovrebbe essere progettato in modo tale da essere resiliente ad attacchi esterni [3].
Nel contesto delle vulnerabilità è difficile determinare la natura di ognuna di esse in quanto possono essere dovute a vari fattori. Possiamo tuttavia distinguere alcune macro-categorie per catalogarle:

1. **Vulnerabilità software:** si verificano quando le applicazioni contengono errori o bug. Gli aggressori considerano il software difettoso come un'opportunità per attaccare il sistema sfruttando questi difetti (es. buffer overflow, race condition). Tali difetti vengono sfruttati per sviluppare metodi, noti anche come **exploit**, che vadano a favore di colui che li sviluppa. Si parla di **bug** invece per indicare un difetto per il quale ancora non è stato scoperto un exploit. Queste vulnerabilità possono essere dovute a malfunzionamenti ed errori di scrittura del codice. Le vulnerabilità del software rappresentano minacce alla sicurezza potenzialmente sfruttabili da soggetti esterni per causare perdita di dati, escalation di privilegi, race conditions e altri effetti indesiderati che possono influire sul codice sorgente [3].

2. **Vulnerabilità del firewall:** i firewall sono sistemi software e hardware che proteggono la rete interna dagli attacchi. Una vulnerabilità del firewall è un errore, un punto debole o un presupposto non valido effettuato durante la progettazione, l'implementazione o la configurazione del firewall che può essere sfruttato per attaccare la rete che il firewall dovrebbe proteggere.

3. **Vulnerabilità TCP/IP:** queste vulnerabilità sono legate ai protocolli dei vari livelli di

una rete. Questi protocolli potrebbero non disporre di funzionalità desiderabili su una rete sicura e potrebbero non contemplare un qualche problema legato alla sicurezza (es. attacchi ARP, attacchi di frammentazione etc.).

4. **Vulnerabilità del sistema operativo:** la sicurezza delle applicazioni in esecuzione dipende dalla sicurezza del sistema operativo. La minima negligenza da parte dell'amministratore di sistema può rendere vulnerabili i sistemi operativi.

5. **Vulnerabilità del server Web:** queste vulnerabilità sono causate da errori di progettazione e ingegnerizzazione o da un'implementazione errata di questi ultimi (es. sniffing, spoofing etc.).

6. **Vulnerabilità hardware:** si parla di vulnerabilità hardware quando una qualsiasi componente hardware del sistema può mettere a repentaglio il corretto funzionamento della macchina.

Discuteremo nel dettaglio delle ***vulnerabilità software***.

2.1.1 Vulnerabilità software

La classificazione delle vulnerabilità è il primo passaggio fondamentale nell'analisi delle vulnerabilità. Essa consente di migliorare la comprensione delle vulnerabilità attraverso la costruzione di un modulo di classificazione gerarchica [4].
Le vulnerabilità software sono catalogate nel ***National Vulnerability Database*** **(NVD)**, repository di dati di gestione della vulnerabilità basati su standard rappresentati tramite il Security Content Automation Protocol (SCAP) del governo degli Stati Uniti. L'NVD include database di riferimenti a elenchi di controllo di sicurezza, difetti software relativi alla sicurezza, configurazioni errate, nomi di prodotti e metriche di impatto [5].
Nel NVD le vulnerabilità vengono identificate attraverso un identificatore univoco chiamato **CVE ID**. Il programma ***Common Vulnerabilities and Exposures*** **(CVE**) è stato fondato nel 1999 ed è gestito dalla società ***MITRE***; è un dizionario o glossario di vulnerabilità che sono state identificate per basi di codice specifiche, come applicazioni software o librerie open source. Questo elenco consente alle parti interessate di acquisire i dettagli delle vulnerabilità facendo riferimento ad un identificatore univoco. Gli ID CVE sono assegnati principalmente dal MITRE, nonché da organizzazioni autorizzate note come ***CVE Numbering Authorities*** **(CNA)**, un gruppo internazionale di fornitori e ricercatori di numerosi paesi.
Il programma CVE è stato creato con l'obiettivo di diventare lo standard del settore nello

stabilire una linea di base per le vulnerabilità in modo tale che tutte le informazioni contenute nel progetto siano pubblicamente disponibili per qualsiasi parte interessata.
Ciascun CVE deve includere una descrizione fornita dal segnalante o creata utilizzando il modello facoltativo del CVE Assignment Team. Questa descrizione include il tipo di vulnerabilità (ad es. buffer overflow), il fornitore del prodotto e le basi di codice interessate. Il National Vulnerability Database (NVD) ha il compito di analizzare ogni CVE una volta che è stato pubblicato nell'elenco CVE, dopodiché è generalmente disponibile nel NVD entro un'ora [6].

Alcune note vulnerabilità sono:

- **Mydoom:** il peggior focolaio di virus informatici della storia, ha causato danni stimati a 38 miliardi di dollari nel 2004, ma il suo costo al netto dell'inflazione è in realtà di 52,2 miliardi di dollari. Conosciuto anche come **Novarg**, questo malware è tecnicamente un worm, diffuso principalmente tramite e-mail di massa. Ad un certo punto, il virus Mydoom era responsabile del 25% di tutte le email inviate. Mydoom ha prelevato gli indirizzi dalle macchine infette, quindi ha inviato copie di se stesso a quegli indirizzi. Ha anche collegato le macchine infette a una rete di computer chiamata botnet che ha eseguito attacchi DDoS (Distributed Denial of Service). Mydoom è ancora in circolazione oggi, generando l'1% di tutte le e-mail di phishing [7].
  Il virus colpisce il sistema operativo Windows modificando/aggiungendo chiavi di registro. Una volta che il virus inizia a modificare i registri, ottiene il controllo del sistema operativo avendo così la possibilità di aprire varie porte. L'attacco funziona in due step:
  - Viene installata una backdoor sulla porta 3127/tcp per consentire il controllo remoto del PC infetto (mettendo il proprio file SHIMGAPI.DLL nella directory system32 ed eseguendolo come un processo figlio di Windows Explorer);
  - Viene sferrato un attacco DDoS [8].
- **WannaCry:** è un ramsonware rilevato nel 2017 che si impossessa del computer (o dei file cloud) e li tiene in ostaggio [7]. Questo ramsonware sfrutta le vulnerabilità del protocollo di comunicazione **SMB** ***Server Message Block*** della Microsoft. Tale protocollo era utilizzato principalmente per la condivisione di file e servizi di stampa tra computer in rete. **SMBv1**,deprecato nel 2013, presenta una serie di vulnerabilità che consentono l'esecuzione di codice in modalità remota sul computer di destinazione. Anche se la

maggior parte di loro ha una patch disponibile e SMBv1 non è più installato di default a partire da Windows Server 2016, gli hacker stanno ancora sfruttando questo protocollo per lanciare attacchi devastanti. In particolare, ***EternalBlue*** sfrutta una vulnerabilità in SMBv1 e, appena un mese dopo la pubblicazione di EternalBlue, gli hacker lo hanno utilizzato per lanciare il famigerato attacco ransomware WannaCry [9]. Le vulnerabilità di SMBv1 sfruttate da WannaCry cono state catalogate nel NVD come CVE-2017-0280. Il ransomware WannaCry è penetrato nei computer di 150 paesi, causando enormi perdite di produttività poiché aziende, ospedali e organizzazioni governative che non pagavano sono state costrette a ricostruire i sistemi da zero. Il malware ha imperversato a macchia d'olio attraverso 200.000 computer in tutto il mondo. Si è fermato solo quando un ricercatore di sicurezza di 22 anni nel Regno Unito ha trovato un modo per disattivarlo [7].

- **CodeRed:** Osservato per la prima volta nel 2001, il virus informatico Code Red era un altro worm che è penetrato in 975.000 host. Mostrava le parole *"Hacked by Chinese!"* attraverso pagine Web infette ed è stato eseguito interamente nella memoria di ogni macchina. I costi finanziari sono fissati a 2,4 miliardi di dollari [7].
  Il 18 giugno 2001 l'eEye Digital Security ha segnalato l'esistenza di una vulnerabilità remote buffer overflow in tutte le versioni del software del **server Web IIS *(Internet Information Server)***. Utilizzando il processo di installazione di default dell'IIS vengono incluse parecchie estensioni ***Internet Service API* (ISAPI)** o collegate librerie dinamiche. *idq.dll* è una componente dell'Index Server (noto come Indexing Service in Windows 2000) e fornisce supporto per script amministrativi (file *.ida*) e Internet Data Query (file *.idq*).
  È stata rilevata una vulnerabilità di sicurezza in *idq.dll* (servizio di indicizzazione). Il filtro ISAPI non esegue un corretto "controllo dei limiti" sui buffer per i dati immessi dall'utente. Un attaccante remoto connesso al server può avviare un attacco che provocherà un overflow del buffer. Questo causerà un Denial of Service sul server o permetterà di introdurre un codice da eseguire sul target server. Tale codice in esecuzione nel contesto di sicurezza del sistema locale darebbe all'attaccante il completo controllo del server, consentendogli di eseguire praticamente qualsiasi azione sul server di destinazione che ha scelto, inclusa la modifica delle pagine Web, la riformattazione del disco rigido o l'aggiunta di nuovi utenti al gruppo di amministratori locali. Di seguito è elencato il funzionamento della prima versione di CodeRed:[10]

1. Il worm tenta di connettersi alla porta TCP 80 su un host selezionato a caso supponendo che il server web verrà trovato. Dopo essersi connesso con successo, l'host attaccante invia una richiesta HTTP GET alla vittima. Tale richiesta sfrutta la vulnerabilità di overflow del buffer causando l'esecuzione del worm sul sistema. Il worm non viene scritto su disco ma iniettato ed eseguito direttamente dalla memoria. L'inizio del pacchetto di attacco del worm sembra il seguente:

```
GET/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucb
d3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u
8b00%u531b%u53ff%u0078%u0000%u00=a
```

Figura 2.1: Inizio del pacchetto d'attacco

2. Una volta eseguito, il worm ricerca il file c:\notworm. Se il file esiste, il thread va in una sleep infinita. Se il file c:\notworm non esiste, vengono creati nuovi thread. Ogni thread può causare la generazione di un altro thread che causa continuamente la creazione di thread fino a 100 thread.
3. I successivi 99 thread tentano di sfruttare più sistemi prendendo di mira IP casuali, se la data è prima del giorno 20 del mese.
4. Il centesimo thread del codice del worm deturpa la home page del server Web se la lingua predefinita del sistema è l'inglese americano. Inizialmente il thread fa una sleep di 2 ore e poi lancia una funzione che risponde alle richieste HTTP. Quel collegamento viene quindi puntato al codice worm che produce la pagina web come mostrato dalla Figura 2.2. Le modifiche nella home page non vengono fatte modificando la home page nei file del disco fisico, ma vengono effettuate dal codice in memoria. Il collegamento dura per 10 ore e poi viene rimosso.

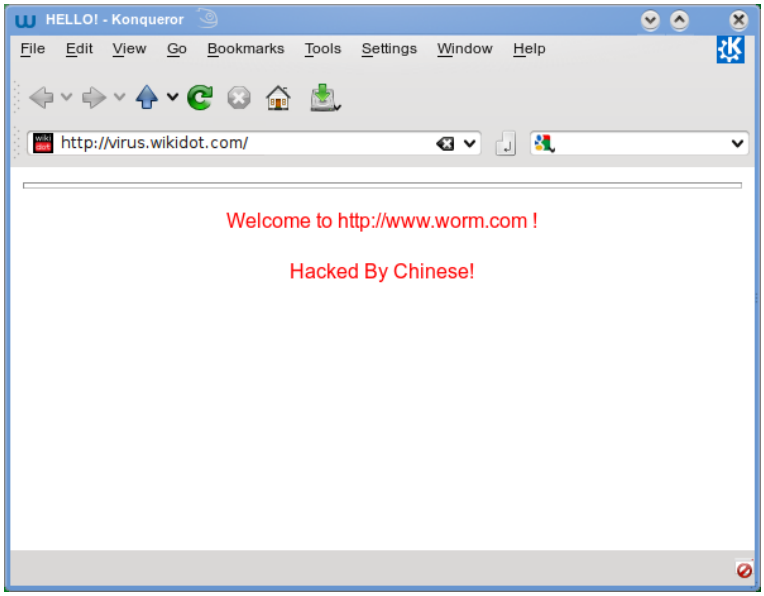

Figura 2.2: Esempio di pagina web infetta

5. Se la data è compresa tra il 20 e il 28, i thread attivi tentano un attacco DoS su un particolare indirizzo IP 198.137.240.91 (una volta era www.whitehouse.gov) inviando una grande quantità di dati spazzatura.
6. Se la data è maggiore del 28, i thread del worm vengono messi in una sleep infinita.

La vulnerabilità remote buffer overflow sfruttata da CodeRed è stata catalogata nel NVD come CVE-2001-0500.

- **Slammer:** il worm SQL Slammer è costato circa 750 milioni di dollari su 200.000 utenti nell'inizio del 2003. A differenza dei worm più comuni, che si propagano da computer a computer tramite un allegato di posta elettronica, SQL Slammer si propaga inviando pacchetti UDP ad una porta predefinita (1434) dei sistemi vittime [7].
  Slammer ha sfruttato una vulnerabilità di overflow del buffer in Microsoft SQL Server 2000 per propagarsi. Il worm non raggiunge il disco del computer infetto, infatti scompare al riavvio del server. D'altra parte, il server potrebbe essere reinfettato se il problema di sicurezza non viene risolto applicando le patch Microsoft. ***Worm:W32/Slammer*** è stato rilevato per la prima volta su Internet il 25 gennaio 2003 alle 05:30 GMT. Successivamente è stato rilevato nella maggior parte dei paesi del mondo. Tuttavia, ci sono segnalazioni non confermate di tracce di worm individuate già il 20 gennaio. Il worm genera enormi quantità di pacchetti di rete, sovraccaricando server e router e rallentando il traffico di rete. Secondo molti rapporti, fino a 5 dei 13 root nameserver Internet erano inattivi a causa di ciò. Questo worm non infetta affatto le macchine

tipiche degli utenti finali: infetta solo i computer che eseguono Microsoft SQL Server 2000 o MSDE 2000. Gli utenti finali potrebbero notare questo worm solo a causa della lentezza della rete. Il worm utilizza la porta UDP 1434 per sfruttare un buffer overflow nel server MS SQL. Le vulnerabilità sfruttate da SQL Slammer sono state catalogate nel NVD come CVE-2002-0649

- **Spring4Shell Vulnerability:** un altro noto esempio di vulnerabilità software è quello che ha interessato il framework ***Spring*** di Java. Il 29 marzo 2022 il blog sulla sicurezza cyberkendra ha pubblicato un post su una vulnerabilità zero-day **RCE *(Remote Code Execution)*** simile a Log4Shell (scoperta il 9 dicembre 2021 e catalogata nel NVD come CVE-2021-45046) nel framework ***Spring*** versioni da 5.3.0 a 5.3.18, da 5.2.0 a 5.2.20 e versioni precedenti [11].

  Una **zero-day vulnerability** (nota anche come 0-day) è una vulnerabilità del software precedentemente sconosciuta a coloro che dovrebbero essere interessati alla sua mitigazione. Fino a quando la vulnerabilità non viene mitigata, gli hacker possono sfruttarla per influire negativamente su programmi, dati, computer e rete. Un exploit che sfrutta uno zero-day è chiamato *exploit zero-day* o *attacco zero-day*. Il termine "zero-day" originariamente si riferiva al numero di giorni trascorsi da quando un nuovo software era stato rilasciato al pubblico, quindi *software zero-day* è stato ottenuto hackerando il computer di uno sviluppatore prima del rilascio [12].

  La vulnerabilità è stata soprannominata ***"SpringShell" (o "Spring4Shell")***, a causa del suo presunto significato paragonabile al famigerato *"Log4Shell"* (vulnerabilità RCE del framework Java Log4j2). Il 31 marzo è stata confermata dai manutentori del framework stesso ed è stata catalogata nel NVD come CVE-2022-22965.

  La vulnerabilità di SpringShell risiede nel meccanismo di `data binding` del framework, questo meccanismo consiste nel prendere i parametri dall'URL o dal corpo della richiesta e assegnarli agli argomenti della funzione o, in alcuni casi, agli oggetti Java. Quando si assegnano parametri di richiesta a oggetti Java, c'è un rischio intrinseco per la sicurezza poiché alcuni parametri "interni" dell'oggetto compilato non dovrebbero essere controllati esternamente (es. parametri `Class`, `ClassLoader` e `ProtectionDomain`). Spring contiene del codice specifico che non permette ai malintenzionati di assegnare valori a questi parametri interni, però sfortunatamente a partire da Java 9 gazie all'introduzione della nuova API **`class.getModule`** è possibile baypassare il codice protettivo di Spring ed è quindi possibile assegnare valori

arbitrari ai parametri interni degli oggetti, in particolare alle proprietà dell'attributo `ClassLoader`. Questa è la principale causa della vulnerabilità *Spring4Shell*.
Esistono molti modi per sfruttare la modifica del `ClassLoader` al fine di ottenere l'esecuzione di codice in modalità remota, ma l'exploit PoC *(Proof of Concept)* pubblicato originariamente ha scelto di sfruttare questa vulnerabilità di sicurezza con una tecnica specifica di Tomcat, cioè abusando di `AccessLogValve` per ottenere la sovrascrittura arbitraria dei file. L'exploit funziona in 2 fasi:

1. Sovrascrivere gli attributi `ClassLoader` specifici di Tomcat, per modificare il percorso del file di registro di accesso in un punto della web directory principale e modificare il modello di registro (i dati scritti) in un modello costante che contiene il codice webshell. Ciò fa sì che la JSP webshell venga eliminata dalla web root directory.
2. Invio di una richiesta di query alla webshell eliminata, per eseguire un comando di shell arbitrario.

Esempio di codice malevolo:

```
public class Greeting {
<%
java.io.InputStream in =
 ↪  Runtime.getRuntime().exec(request.getParameter("cmd")).getInputStream();
int a = -1;
byte[] b = new byte[2048];

while((a=in.read(b))!=-1) {
    out.println(new String(b));
}
%>
```

Poiché la vulnerabilità è nel meccanismo di associazione dei dati di Spring, solo le applicazioni Web che tentano di associare i parametri di richiesta ai POJO (Plain Old Java Objects) sono vulnerabili. Qualsiasi delle seguenti annotazioni, indica che un request handler potrebbe essere vulnerabile:

```
@RequestMapping
@GetMapping
@PostMapping
@PutMapping
```

```
@DeleteMapping
@PatchMapping
```

Un esempio di Controller Spring vulnerabile è il seguente:

```
@Controller
public class HelloController {
  @GetMapping("/greeting")
  public String helloWorld(@ModelAttribute SomeClass someClass, Model
   ↪  model) {
      //@ModelAttribute indica che verrà effettuato il data binding
      return "hello";
  }
}
```

La vulnerabilità è indotta dall'annotazione *@ModelAttribute* ma il request handler può essere scritto anche senza quest'annotazione ed è comunque vulnerabile, cioè:

```
@GetMapping("/greeting")
public String helloWorld(SomeClass someClass, Model model) {
    return "hello";
}
```

Ciò è dovuto al fatto che i tipi non semplici che non corrispondono a classi note vengono risolti come ***@ModelAttribute*** di default [11].

Tra gli attacchi alle web app, Remote Code Execution (RCE) è una delle minacce più dannose, infatti secondo Open Web Application Security Project (OWASP), RCE è il problema di sicurezza più diffuso dal luglio 2004 e quindi è stata classificata la minaccia numero uno nell'elenco dei problemi di sicurezza delle Web app [13].

2.2 Identificazione delle vulnerabilità

Con l'ampio utilizzo dei computer, il software diventa sempre più complicato e su larga scala, questo comporta anche un aumento dei problemi di sicurezza. La sicurezza software è la capacità del software di fornire i servizi richiesti quando è attaccato. C'è un grande interesse nei confronti del test di sicurezza perchè è considerato un mezzo importante per migliorare la sicurezza del software. Il test di sicurezza del software è il processo per identificare se le caratteristiche di sicurezza dell'implementazione del software sono coerenti con il design. I

test di sicurezza del software possono essere suddivisi in **test di sicurezza funzionali** e **test di vulnerabilità**.

Il **test di sicurezza funzionale** garantisce che le funzioni di sicurezza del software sono implementate in modo corretto e coerente con i requisiti di sicurezza specificati.

Il **test di vulnerabilità** consiste nello scoprire le vulnerabilità della sicurezza come un attaccante [14].

Per questo sono richieste tecniche di vulnerability detection da utilizzare durante la fase di sviluppo del software per individuare le vulnerabilità e prevenire situazioni critiche.

Le vulnerabilità software scoperte tardi nel ciclo di sviluppo del software sono più costose da risolvere rispetto a quelle scoperte nelle fasi iniziali dello sviluppo. Pertanto gli sviluppatori dovrebbero sfozarsi ad individuare il prima possibile le vulnerabilità [15].

Tuttavia non tutte le vulnerabilità vengono scoperte nel ciclo di sviluppo del software ma, nella maggior parte dei casi, le vulnerabilità vengono individuate durante il funzionamento del software.

Gli approcci esistenti per mitigare le minacce alle web app possono essere suddivisi in soluzioni lato client e lato server. Le soluzioni lato server hanno il vantaggio di poter scoprire una gamma più grande di vulnerabilità. Le tecniche possono essere ulteriormente classificate in analisi statica e analisi dinamica [16].

- **Analisi statica:** consiste nella scansione del codice sorgente o binario senza eseguirlo. Gli analyzer statici sono veloci e semplici da usare per individuare bug nel codice. Tuttavia gli analyzer statici non hanno una forte potenza predittiva infatti generano grosse quantità di falsi positivi e falsi negativi [17].

- **Analisi dinamica:** prevede l'esecuzione del codice per poter identificare le vulnerabilità. Gli analyzer dinamici sono più accurati e generano un numero inferiore di falsi positivi rispetto agli analizzatori statici. In controparte soffrono di un notevole overhead di tempo e richiedono casi di test di grandi dimensioni per garantire una certa confidenzialità nell'individuazione di bug di sicurezza. Gli analyzer dinamici strumentano il codice per produrre informazioni sui percorsi di esecuzione. Se viene rilevata una vulnerabilità, viene scritto un warning in un trace file [17].

La maggior parte degli approcci sono basati sull'analisi statica e/o dinamica, sul symbolic execution e sul fuzz-testing. Alcuni di essi sono anche implementati all'interno di strumenti automatizzati [3].

Un ***fuzz tester (o fuzzer)*** è uno strumento che genera in modo iterativo e casuale input con cui

testa un programma target. Nonostante appaia *"ingenuo"* rispetto a strumenti più sofisticati i ***fuzzer*** sono sorprendentemente efficaci. Di solito, l'obiettivo finale di un fuzzer è generare un input che provoca l'arresto anomalo del programma [18].
Il fuzzing viene generalmente condotto dopo lo sviluppo; è la tecnica che scopre di più le vulnerabilità[19].

2.3 Vulnerability prediction models

Il rilevamento automatico delle vulnerabilità di sicurezza è un problema fondamentale nella sicurezza dei sistemi. È noto che le tecniche tradizionali soffrono di alti tassi di falsi positivi e falsi negativi [1].
I recenti progressi in termini di potenza di calcolo, disponibilità di dati e nuovi algoritmi hanno portato a importanti scoperte nell'intelligenza artificiale (AI) e nel machine learnimg (ML) nell'ultimo decennio, sono sempre più sfruttati nei settori più diversi come industria, governo e commercio. In questo senso, l'ennesima promettente applicazione di AI/ML è l'analisi software automatizzata e intelligente con vari obiettivi come la previsione della vulnerabilità. Questo non solo riduce la necessità di avere esperti di dominio ma permette anche di semplificare ed automatizzare i processi necessari per le attuali tecniche di analisi della sicurezza [20].
I VPMs ***(Vulnerability Prediction Models)*** sono una delle numerose applicazioni dei modelli di predizione. Hanno lo scopo di classificare componenti software come vulnerabili/non vulnerabili con l'obiettivo finale di supportare il testing e la revisione del codice. I *VPMs* devono affrontare però alcuni problemi molto specifici a partire dalla disponibilità dei dati in quanto alla base dei modelli predittivi c'è l'apprendimento da una grande quantità di dati. Nel caso delle vulnerabilità è problematico in quanto queste, per la loro natura, sono poche; inoltre i dataset sono sbilanciati, ovvero le istanze vulnerabili sono nettamente inferiori rispetto a quelle non vulnerabili, di conseguenza si avrà un training set sbilanciato [21].
Con dataset sbilanciati i modelli generati possono essere sub-ottimali, infatti la maggior parte degli algoritmi di apprendimento assumono dataset bilanciati. In questi casi ci sono due alternative:

1. Applicare algoritmi che sono resistenti ai dataset sbilanciati.

2. Bilanciare i dati utilizzando tecniche di *sampling* prima di applicare su di essi l'algoritmo di data mining.

Le tecniche di sampling o balancing possono essere classificate in due gruppi:

- **Over-sampling:** gli algoritmi di over-sampling bilanciano la distribuzione tra le classi incrementando il numero di istanze della classe di minoranza.
- **Under-sampling:** gli algoritmi di under-sampling bilanciano la distribuzione tra le classi rimuovendo istanze dalla classe di maggioranza.

Le tecniche più semplici per i due gruppi sono ***Random Over-Sampling*** **(ROS)** e ***Random Under-Sampling*** **(RUS)**. Ovviamente il campionamento casuale non è il migliore, ci sono altre tecniche che utilizzano approcci più sofisticati per l'aggiunta o la rimozione di istanze dal dataset.
Una tecnica di *under-sampling* non randomica è la ***One-Sides Selection*** **(OSS)**, la quale invece di rimuovere casualmente le istanze dalla classe di maggioranza tenta di sottocampionare la classe di maggioranza cercando di rimuovere le istanze rumorose o al limite.
Una tecnica di *over-sampling* non randomica, invece, è la ***Synthetic Minority Oversampling Technique*** **(SMOTE)** che invece di duplicare le istanze, esplora le istanze della classe minoritaria e genera istanze sintetiche che *"assomigliano"* alle istanze vicine [22].
Sebbene il topic della previsione delle vulnerabilità sia un area di ricerca relativamente nuova (il primo tentativo noto è datato 2008) sono già state fatte numerose ricerche sulla capacità delle metriche software nell'indicare la presenza di vulnerabilità nei prodotti software. La maggior parte dei lavori di ricerca esistenti si è concentrata principalmente su metriche di coesione, accoppiamento e complessità ***(CCC)***. Diversi ricercatori si sono discostati dal comune trend proponendo delle proprie metriche e indagandone il potere predittivo nella previsione delle vulnerabilità. L'aggiunta di queste nuove metriche porta ad un miglioramento dell'***accuracy***,quindi è necessario introdurre metriche di misurazione aggiuntive e preferibilmente metriche legate alla sicurezza del software [23].
Le principali metriche di valutazione di un *Vulnerability Prediction Model* (o in generale per i modelli predittivi) sono:

- **Accuracy:** indica la percentuale di predizioni corrette effettuate dal modello.

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

- **Precision:** indica l'accuratezza delle predizioni, cioè il rapporto tra i veri positivi e veri positivi più falsi positivi, cioè:

$$precision = \frac{TP}{TP + FP}$$

- **Recall:** indica la percentuale di istanze positive che sono state correttamente predette dal modello, spesso questa metrica è chiamata anche *sensitivity o true positive rate (TRP)*.

$$recall = \frac{TP}{TP + FN}$$

Spesso *precision* e *recall* sono combinate in un unica metrica chiamata ***F1-score***, utilizzata in particolare se si ha bisogno di confrontare due modelli. Essa indica la media armonica di *precision* e *recall* ed in questo caso un modello otterrà un ***F1-score*** alto se lo sono anche *precision* e *recall* [24].

$$F_1 = \frac{TP}{TP + \frac{FN+FP}{2}}$$

Le variabili riportate in precedenza hanno il seguente significato:

- **TP** *(true positive)*, indica il numero di istanze della classe d'interesse per le quali il modello ha effettuato una predizione corretta.

- **FP** *(false positive)*, indica il numero di istanze che il modello ha considerato erroneamente come istanze della classe d'interesse.

- **TN** *(true negative)*, indica il numero di istanze che non appartenono alla classe d'interesse per le quali il modello ha effettuato una predizione corretta.

- **FN** *(false negative)*, indica il numero di istanze che appartengono alla classe d'interesse, ma, per le quali il modello ha effettuato una predizione sbagliata etichettandole come istanze appartententi ad un'altra classe.

2.3.1 Confronto tra alcuni modelli di Vulnerability Detection

In questa sezione vengono messi a confronto alcuni modelli di vulnerability detection. I modelli presi in esame sono i seguenti:

- **Modelli causali:** hanno bisogno di input sulla dimensione stimata, complessità, input qualitativi su test pianificati e requisiti di qualità. Il più grande vantaggio dei modelli causali è che possono essere applicati molto presto nel processo di sviluppo. Utilizzabili

per analizzare scenari what-if per stimare la qualità dell'output o il testing necessario per raggiungere gli obiettivi di qualità desiderati [25].

- **Opinione degli esperti:** c'è bisogno di esperienza di dominio (sviluppo software, test e valutazione di qualità). Questo è il modo più semplice e veloce per ottenere previsioni. L'incertezza sulle previsioni è elevata e le previsioni possono essere soggette a pregiudizi individuali [25].

- **Predizioni basate sull'analogia:** bisogna conoscere le caratteristiche del progetto e fare osservazioni da un gran numero di progetti storici. Questo modello è veloce e facile da usare, il progetto attuale è comparato rispetto al progetto precedente con la maggior parte delle caratteristiche simili. L'evoluzione del processo software, la catena di strumenti di sviluppo può portare all'inapplicabilità o a grandi errori di previsione [25].

- **Modelli di qualità costruttivi:** sono richieste stime delle dimensioni del software prodotto, attributi personali e di progetto, livello di rimozione delle vulnerabilità. Questo modello può essere utilizzato per prevedere il costo, la pianificazione o la densità residua delle vulnerabilità del software in fase di sviluppo. Richiede un grande sforzo per calibrare il modello [25].

- **Analisi di correlazione:** bisogna conoscere il numero di vulnerabilità riscontrate in una data iterazione; dimensione e test effort stimati possono essere utilizzate anche in modelli estesi. Questo metodo richiede una piccola quantità di dati in input che è disponibile dopo ogni iterazione. Esso fornisce regole semplici da usare che possono essere applicate rapidamente. Il modello può anche essere utilizzato per identificare i moduli che mostrano livelli alti/bassi di densità delle vulnerabilità e quindi consentire interventi precoci [25].

- **Modelli di regressione:** bisogna conoscere le metriche del software (o del modello) come misure delle differenti caratteristiche del software/modello; Un altro input possono essere le metriche di cambiamento. Questo modello usa le metriche del codice/modello effettivo, il che significa che le stime sono fatte sulla base dei dati effettivi del software sotto sviluppo. Può essere applicato solo quando il codice/i modelli sono già implementati e l'accesso al codice sorgente/modello è disponibile. La relazione tra le caratteristiche di input e di output nel modello di regressione può essere difficile da interpretare, non mappa relazioni casuali [25].

- **Modelli di classificazione:** questo modello ha gli stessi input del modello precedente. Simili ai modelli di regressione, questi possono essere utilizzati per la classificazione vulnerabile/non vulnerabile o per stimare il numero/densità delle vulnerabilità. Quando sono disponibili più dati i modelli improvvisano sulla loro accuracy regolando il valore dei loro parametri (imparare dall'esperienza). Mentre alcuni modelli come gli alberi decisionali sono facili da capire, gli altri possono agire come una scatola nera (ad esempio Reti Neurali) nelle quali il funzionamento interno non è esplicito [25].

- **Software Reliability Growth Models:** bisogna conoscere i dati di afflusso vulnerabili del software in sviluppo (modello del ciclo di vita) oppure il software in fase di test. Questo modello può utilizzare i dati sull'afflusso di vulnerabilità per fare previsioni sulle vulnerabilità o prevedere l'affidabilità del sistema software. I modelli di crescita dell'affidabilità sono utili anche per valutare la maturità/prontezza di rilascio del software. Questi modelli hanno bisogno di data point sostanziali per fare predizioni precise e stabili [25].

La seguente tabella riassume in modo schematico quanto riportato in precedenza:

Metodo	Input richiesti	Vantaggi e limiti
Modelli causali	Dimensione stimata, requisiti di qualità	Possono essere applicati molto presto nel processo di sviluppo
Opinione degli esperti	Esperienza di dominio	Metodo più semplice e veloce. Incertezza sulle previsioni elevata
Predizioni basate sull'analogia	Caratteristiche del progetto e osservazioni da progetti storici	Veloce e facile da utilizzare. L'evoluzione del software può portare all'inapplicabilità o a grandi errori di previsione
Modelli di qualità costruttivi	Stima dimensione del software prodotto, livello di rimozione delle vulnerabilità	Può essere utilizzato in fase di sviluppo per prevedere la densità residua delle vulnerabilità. Richiede grande sforzo per la calibrazione del modello
Analisi di correlazione	Numero vulnerabilità riscontrate in un iterazione	Richiede una piccola quantità di dati in input, fornisce regole semplici da utilizzare

Modelli di regressione	Metriche del software	Applicabile solo quando il codice è già implementato. Relazioni tra caratteristiche di input e output difficili da interpretare
Modelli di classificazione	Metriche del software	Utilizzati per la classificazione vulnerabile/non vulnerabile o per stimare la densità delle vulnerabilità. Alcuni modelli sono facili da capire, altri possono agire come una scatola nera
Software Reliability Growth Models	Dati di afflusso vulnerabili del software in sviluppo o software sotto test	Utilizzano i dati per fare predizioni sulle vulnerabilità o prevedere l'affidabilità del sistema. Hanno bisogno di data point sostanziali per fare buone predizioni

Tabella 2.1: Confornto tra alcuni modelli di Vulnerability Detection

2.4 Vulnerability Detection attraverso l'utilizzo del Deep Learning

Il deep learning (DL) è una branca del machine learning (ML) che si basa sull'utilizzo delle reti neurali per simulare il trasferimento dati che avviene nel nostro cervello.

2.4.1 Neural Networks

Le ***Artificial Neural Networks*** **(ANNs)** sono il cuore del ***Deep Learning*** **(DL)**.
Esse sono state introdotte nel 1943 dal neurofisiologo *Warren McCulloch* e dal matematico *Walter Pitts*. Nel loro articolo *"A Logical Calculus of Ideas Immanent in Nervous Activity"* presentarono un modello computazionale semplificato di come i neuroni biologici degli animali lavorano insieme per eseguire computazioni complesse usando la logica proposizionale. Questa era la prima architettura di rete neurale artificiale **(ANNs)**.
In principio le ANNs non ebbero molto successo in quanto le risorse che si avevano a disposizione erano scarse, invece adesso questa tecnologia sta riscontrando numerosi successi in diversi ambiti e questo è dovuto:

- Alla grande mole di dati a disposizione ed alla qualità di questi ultimi.
- Al grande incremento di potenza computazionale delle GPU che hanno reso possibile l'addestramento di grosse reti neurali in tempi ragionevoli.

- Al miglioramento degli algoritmi di apprendimento.
- Allo sviluppo di piattaforme open source e librerie che permettono di implementare un modello di rete neurale in modo semplificato.

Le reti neurali sono inspirate all'architettura del cervello umano per costruire macchine intelligenti, sono composte da unità chiamate **nueroni** che sono collegati tra di loro tramite i **dendriti**. Prima di parlare di neuroni atrificiali è opportuno dare un rapido sguardo a quello che sono i neuroni biologici.
Un neurone è composto da un corpo cellulare che contiene il nucleo e molte delle componenti più complesse della cellula, inoltre contiene molte ramificazioni chiamate ***dendriti*** più un'estensione molto lunga chiamata ***assone***. L'***assone*** può essere poche volte più lungo del corpo della cellula oppure può essere fino a decine di migliaia di volte più lungo. L'estremità dell'assone si divide in molti rami chiamati ***telodendri*** e alla punta di questi rami ci sono delle minuscole strutture chiamate ***terminazioni sinaptiche*** o semplicemente ***sinapsi*** che sono connesse ai dendriti di altri neuroni. Tramite le **sinapsi** i neuroni ricevono piccoli impulsi elettrici chiamati **segnali** da altri neuroni. Quando un neurone riceve un numero sufficiente di segnali da altri neuroni in pochi millisecondi emette i propri segnali. I neuroni singolarmente si comportano in modo piuttosto semplice, ma essi sono organizzati in vaste reti di miliardi di neuroni, ogni neurone è tipicamente connesso a migliaia di altri neuroni [24].

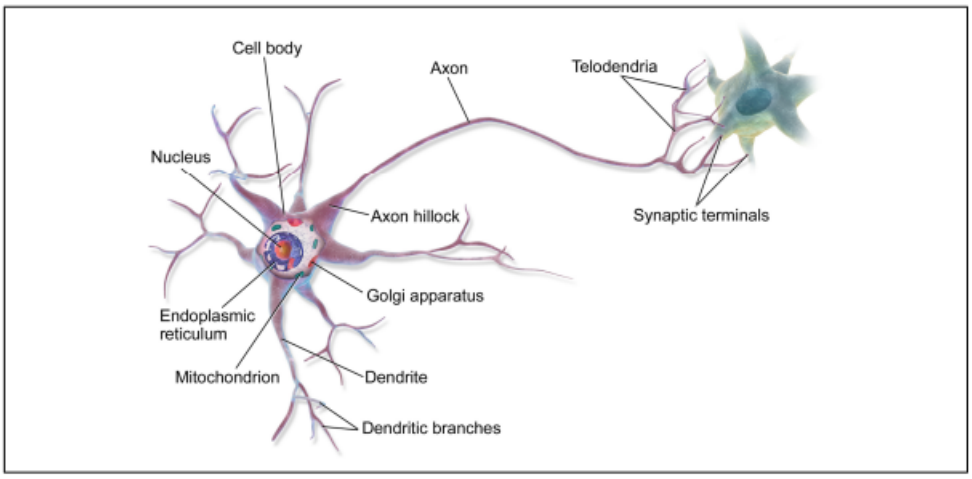

Figura 2.3: Neurone biologico

I neuroni artificiali *(chiamati anche **percettroni**, **unità** o **nodi**)* sono gli elementi più semplici di una rete neurale. Sono ispirati dai neuroni biologici del cervello umano, si può considerare

un neurone artificiale come un modello matematico ispirato a un neurone biologico. In particolare:

- Un neurone biologico riceve i suoi **segnali** in ingresso da altri neuroni attraverso i **dendriti**. Allo stesso modo, un percettrone riceve i suoi dati da altri percettroni attraverso **neuroni di input** che prendono numeri.
- I punti di connessione tra dendriti e neuroni biologici sono chiamati **sinapsi**. Allo stesso modo, le connessioni tra input e percettroni sono chiamate **pesi**. Misurano il livello di importanza di ogni input. Infatti ad ogni arco della rete è associato un peso.
- In un neurone biologico, il nucleo produce un **segnale di uscita** basato sui segnali forniti dai dendriti. Allo stesso modo, il nucleo in un percettrone esegue alcuni calcoli basati sui valori di input e produce un **output**.
- In un neurone biologico, il segnale di uscita viene trasportato dall'assone. Allo stesso modo, l'assone in un percettrone è il valore di output che sarà l'input per i successivi percettroni del layer successivo [26].

Un neurone artificiale di solito viene rappresentato nel seguente modo:

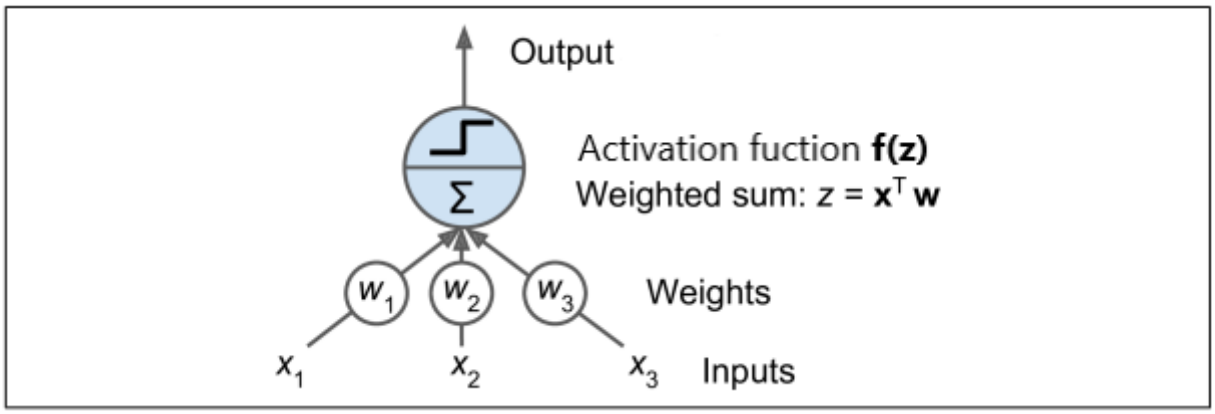

Figura 2.4: Neurone artificiale

La seguente immagine mostra l'analogia tra neurone biologico e neurone artificiale:

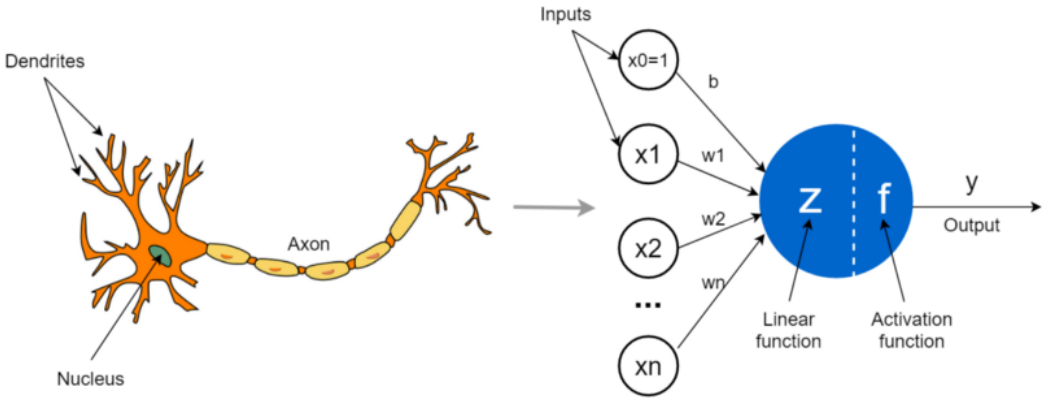

Figura 2.5: Analogia tra neurone biologico e neurone artificiale

Le ***ANNs*** sono versatili, potenti e scalabili, queste caratteristiche le rendono ideali per affrontare molti task di machine learning di grande complessità [24].
Una rete neurale può essere vista come un grafo in cui i nodi rappresentano i neuroni e gli archi rappresentano i dendriti/sinapsi che permettono l'interconnessione tra neuroni. I nodi del grafo sono divisi in layer in senso verticale proprio come lo schema proposto da *McCulloch e Pitts* del nostro cervello.

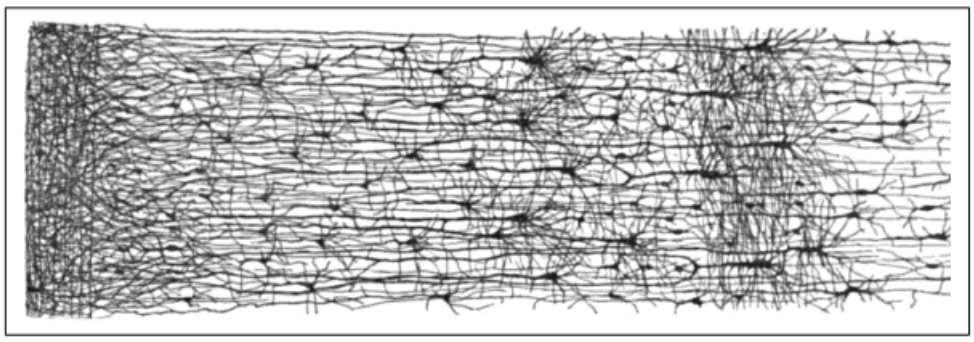

Figura 2.6: Struttura di una rete neurale biologica

I nodi dei vari layer vengono utilizzati per la classificazione e la predizione dei dati forniti in ingresso alla rete. Ogni nodo ha un peso che viene considerato durante l'elaborazione delle informazioni da un layer al layer successivo.
Il primo layer di una rete neurale è chiamato **input layer**, i layer centrali vengono chiamati layer nascosti o **hidden layer** e il layer finale (ultimo layer) viene chiamato **output layer** ed è il layer che restituirà la previsione del modello su un dato input. In una rete neurale si possono avere zero (architettura ***Perceptron***) o più hidden layer. Ogni nodo ha un valore

compreso tra 0 e 1 e prende il nome di **valore di attivazione del nodo**.
Il valore di attivazione di un determinato nodo del layer x è dato dalla somma pesata degli input del neurone in questione. Dato che questa somma può superare il range [0,1] si utilizza una ***funzione di attivazione*** che permette di trasformare il valore ottenuto dalla somma pesata in un valore compreso tra 0 e 1. Il valore ottenuto viene fornito in input al livello successivo.
L'accuracy di una rete neurale è definita dal tipo di **funzione di attivazione** utilizzata. Le funzioni di attivazione più comuni sono quelle non lineari. Se la funzione di attivazione in una rete neurale non è definita il segnale di output è una semplice funzione lineare che è un polinomio di grado uno. Una rete neurale senza funzione di attivazione agisce come un ***modello di regressione lineare*** con prestazioni e potenza limitate nella maggior parte dei casi. È auspicabile che una rete neurale non impari e calcoli solo una funzione lineare ma esegua compiti molto più complicati. Pertanto, dobbiamo applicare una funzione di attivazione per rendere la rete dinamica e aggiungere la capacità di estrarre informazioni complicate dai dati e rappresentare mappature funzionali non lineari, casuali e contorte tra input e output.
Una caratteristica importante di una funzione di attivazione è che deve essere differenziabile, in modo da poter implementare una strategia di ottimizzazione della **back propagation** per calcolare gli errori o le perdite rispetto ai pesi ed eventualmente ottimizzarli utilizzando **Gradient Descend** o qualsiasi altra tecnica di ottimizzazione per ridurre gli errori. Esempi di funzione di attivazione sono: Binary Step Function,Sigmoid, ReLU, Leaky ReLU, Parametrized ReLU, etc. [27].
In sostanza una funzione di attivazione ha il compito di decidere se attivare o meno un neurone di un determinato layer in base al valore che esso fornisce in output.
La predizione di una rete su un determinato input è data dal nodo dell'**output layer** che ha il **valore di attivazione** più alto.
Un classico esempio di architettura di rete neurale è la ***Multi-Layer Perceptron* (MLP)**, dove ogni layer tranne quello di output ha un **bias neuron** (che da sempre 1 come output) e ogni nodo di un layer è collegato a tutti i nodi del layer successivo *(dense layer)*. Quando una rete neurale ha molti *hidden layer* viene chiamata ***Deep Neural Network* (DNN)**.Per l'addestramento delle MLP viene utilizzato il metodo del ***Gradient Descent*** [24].

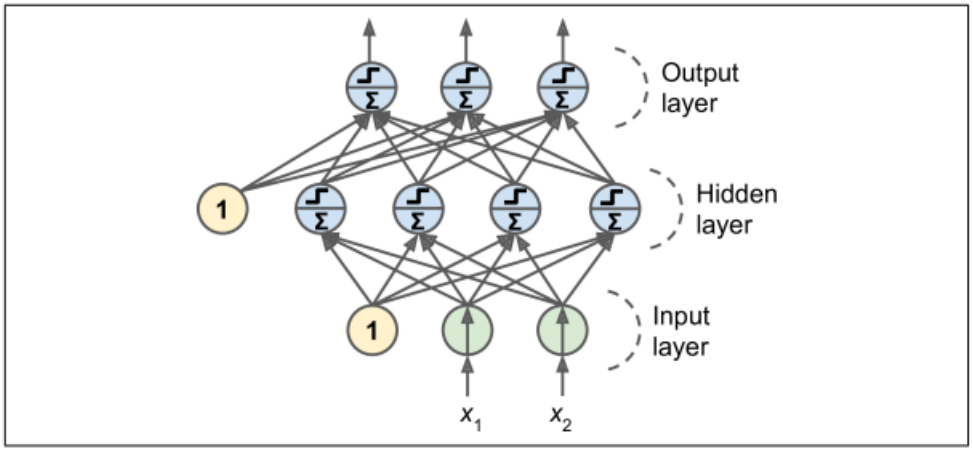

Figura 2.7: Architettura ***Multi-Layer Perceptron***

Gradient Descent è un algoritmo di ottimizzazione molto generico in grado di trovare la soluzione ottima ad una vasta gamma di problemi. L'idea generale di Gradient Descent è di modificare i parametri del modello (θ) in modo iterativo per ridurre al minimo la loss function, la Figura 2.8 ne rappresenta il funzionamento. Questi parametri non sono altro che i pesi delle connessioni della rete neurale. Ogni step di quest'algoritmo viene chiamato **epoca** ed il loro numero dipende dal **learning rate**. Gradient descent calcola come cambia la loss function modificando un parametro θ_j per volta [24].

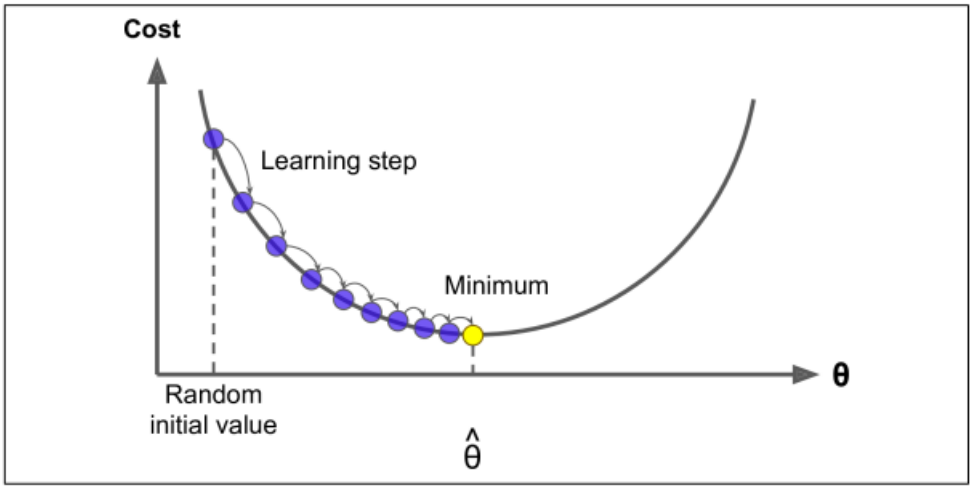

Figura 2.8: Funzionamento Gradient Descent

Scegliere un buon valore per l'iperparametro **learning rate** determina il tempo di convergenza del modello, infatti, tale valore va ad impattare sul numero di fasi/epoche dell'algoritmo. Se il learning rate è troppo basso, allora l'algoritmo dovrà passare attraverso molte iterazioni per convergere, il che richiederà molto tempo. D'altro canto, se il learning rate è troppo alto, potrebbe far oscillare l'algoritmo da una parte all'altra della curva, avendo in

tal caso la possibilità di poter raggiungere anche un punto più alto nella curva rispetto al precedente. Questo potrebbe far divergere l'algoritmo, con valori sempre più grandi, non riuscendo a trovare una buona soluzione [24]. Tali osservazioni sono riportate dalle Figure 2.9 e 2.10.

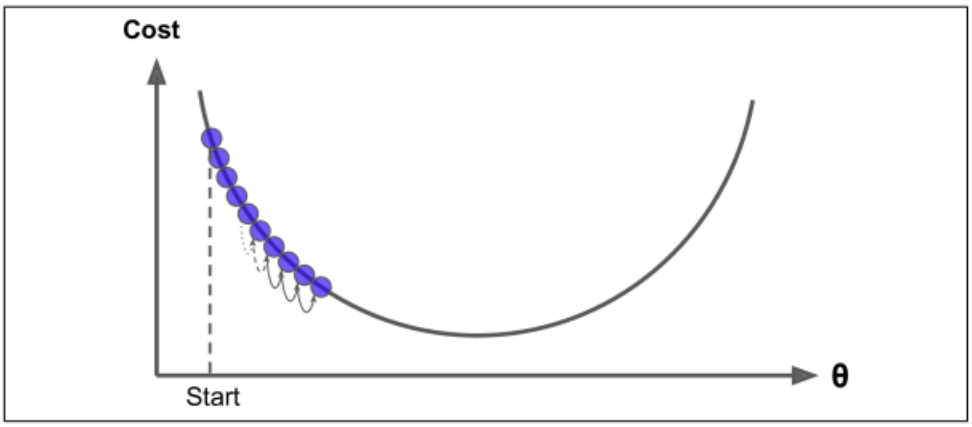

Figura 2.9: Comportamento Gradient Descent con learning rate basso

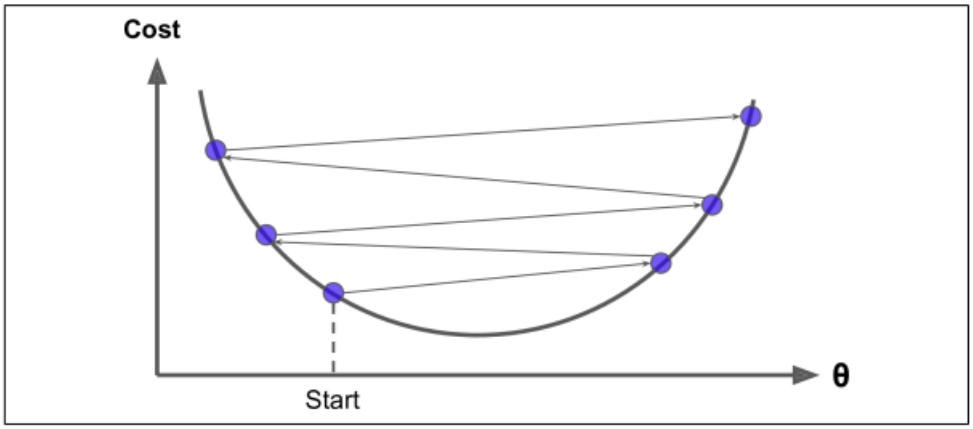

Figura 2.10: Comportamento Gradient Descent con learning rate alto

Inoltre non tutte le funzioni di costo sono regolari, potrebbero esserci buchi, creste, plateau e tutti i tipi di "terreni irregolari", rendendo molto difficile la convergenza al minimo. La Figura 2.11 mostra un esempio. In tal caso se l'inizializzazione random avvia l'algoritmo da sinistra, esso convergerà in un minimo locale. Se invece inizia a destra, impiega molto tempo per attraversare il plateau e se il limite al numero di iterazioni è troppo basso l'algoritmo non raggiungerà mai il minimo globale [24].

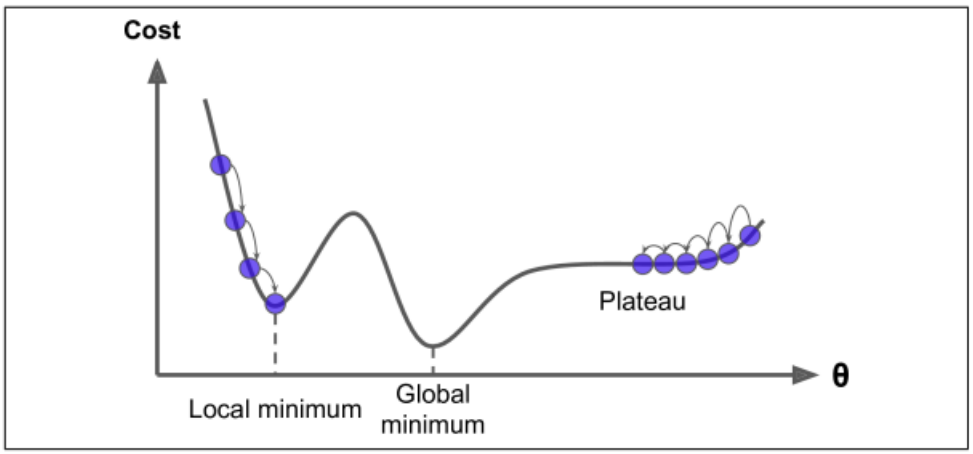

Figura 2.11: Comportamento Gradient Descent con una funzione di costo non regolare

Ci sono diverse tipologie di **Gradient Descent** tra cui:

- **Batch Gradient Descent**, tale tecnica consiste nel fornire l'intero training set in input alla rete neurale e dopodiché viene calcolata la derivata parziale della loss function in base al ogni parametro θ_j in θ per ogni epoca. Il principale svantaggio di questa tecnica è che richiede molto tempo, soprattutto se il training set è di grandi dimensioni [24].

- **Stochastic Gradient Descent**, in tale tecnica viene selezionata una singola istanza in modo casuale dal training set e solo sulla base di questa singola istanza viene calcolato il gradiente. Ovviamente questo tipo di approccio rende la tecnica molto più veloce rispetto al Batch Gradient Descent poiché ha pochi dati da manipolare ad ogni epoca. Questo rende possibile l'addestramento su dataset di grandi dimensioni. D'altra parte, a causa della sua natura stocastica (cioè casuale), questo algoritmo è molto meno regolare di Batch Gradient Descent; infatti, nella ricerca del minimo globale "toccherà" vari punti della curva in modo irregolare. Inoltre quando l'algoritmo si avvicina al minimo continua la sua esecuzione spostandosi su e giù sulla curva e questo porta ad avere dei parametri finali buoni ma non ottimali. Questo problema può essere risolto tramite l'utilizzo del *Simulated Annealing* [24].

- **Mini-batch Gradient Descent**, questa tecnica è simile al Batch Gradient Descent, la differenza sostanziale sta nel fatto che ad ogni epoca calcola il gradiente su una piccola parte di dati del training set, chiamata **mini-batch** che viene scelta in modo randomico. Questa tecnica rispetto allo Stochastic Gradient Descent ha un andamento più regolare e quindi riesce ad avvicinarsi di più al punto di minimo della loss function. D'altra parte è più difficile che questo tipo di algoritmo riesca ad uscire da ottimi locali [24].

Durante l'addestramento di una rete si possono riscontare due tipi di errore:

1. **Vanishing Gradient**, durante la fase di backpropagation il gradiente d'errore viene inoltrato dall'output layer all'input layer per aggiornare i pesi delle connessioni. Durante questa fase può capitare che il gradiente diventa sempre più piccolo e questo comporta che l'algoritmo non aggiornerà i pesi delle connessioni dei layer vicini al layer di input. Questo problema porta l'algoritmo a non convergere mai verso una buona soluzione [24].

2. **Exploding Gradient**, rappresenta l'esatto opposto, cioè, il gradiente man mano che attraversa i vari layer cresce sempre di più, il che porta a fare modifiche anomale dei pesi delle connessioni [24].

2.4.2 Vulnerability Detection attraverso l'utilizzo del Deep Learning

Come riportato nella Sezione 2.3 le tecniche tradizionali producono un alto tasso di falsi positivi/falsi negativi e per questo motivo questi strumenti sono rimasti inaffidabili, lasciando un sovraccarico significativo per gli sviluppatori.
Progressi recenti nel ***Deep Learning* (DL)**, in particolare in domini come la ***computer vision*** e il ***Natural Language Processing* (NLP)**, hanno suscitato interesse nell'utilizzo del DL per rilevare le vulnerabilità automaticamente con elevata precisione. Infatti diversi studi recenti hanno dimostrato risultati molto promettenti ottenendo un ***accuracy*** fino al 95%.
La generalizzabilità di un modello di DL è spesso limitata dal ***bias*** implicito del dataset, che viene spesso introdotto durante il processo di generazione/etichettatura del dataset e pertanto influisce allo stesso modo sia sui dati di test che su quelli di addestramento. Questo bias tende a consentire ai modelli di DL di ottenere una precision elevata nei dati di test apprendendo caratteristiche altamente specifiche per quel dataset, in questo modo il modello ha buoni risultati con i dati di training ma non ha buoni risultati su nuovi dati, questo porta il modello in una situazione di *overfitting* [1].
Dato che studi recenti hanno rivelato che l'utilizzo del DL per rilevare vulnerabilità fornisce risultati promettenti il prossimo capitolo sarà incentrato su quanto fatto da Chakraborty et al. [1].

CAPITOLO 3

Progettazione

Questo capitolo illustra il lavoro svolto da Chakraborty et al.[1] evidenziandone eventuali limitazioni.

3.1 Lavoro svolto da Chakraborty et al.

Nel paper in questione viene fatta dapprima una panoramica sui DLVP *(Deep Learning-based Vulnerability Predictor)* per poi fare un confronto tra tecniche e dataset esistenti evidenziandone i problemi, per andare a proporre infine una propria soluzione ed un proprio dataset. Nelle seguenti sezioni viene fatta un'analisi di quanto riportato dagli autori del paper.

3.1.1 DLVP (Deep Learning-based Vulnerability Predictor)

I metodi di **DLVP** mirano a rilevare vulnerabilità nel software di destinazione apprendendo diversi pattern di vulnerabilità da un dataset di training. L'approccio più famoso di DLVP è basato su tre step: raccolta dei dati, creazione del modello e valutazione. In primo luogo vengono raccolti i dati per il training e viene scelto un modello appropriato. Dopodiché il modello viene addestrato per minimizzare la loss function. Infine il modello viene utilizzato nel mondo reale; per testare l'efficacia del modello ne vengono valutate le performance su esempi di test mai visti, cioè istanze del problema che non sono presenti nel training set.
I predittori di vulnerabilità basati sul Deep Learning apprendono i pattern di codice da un training set (D_{train}) dove le istanze sono etichettate con vulnerabile o neutra. Dato un elemento di codice (x) e la corrispondente etichetta vulnerabile/neutra (y), l'obiettivo del modello è apprendere le caratteristiche che massimizzano il probabilità $p(y|x)$ rispetto ai parametri del modello θ. In pratica addestrare un modello significa apprendere il setting ottimale dei parametri θ^* tale che:

$$\theta^* = argmax_\theta \prod_{(x,y)\in D_{train}} p(y|x,\theta)$$

In primo luogo un elemento di codice (x^i) viene trasformato in un vettore a valori reali $h^i \in \mathbb{R}^n$ che è una rappresentazione compatta di (x^i). Il modo in cui un elemento di codice x^i viene trasformato nel vettore h^i dipende dalle specifiche del modello. Dopo tale operazione il vettore a valori reali h^i viene trasformato in uno scalare $\hat{y} \in [0,1]$ che rappresenta la probabilità dell'elemento di codice x^i di essere vulnerabile. In generale, questa trasformazione e il calcolo della probabilità è ottenuto attraverso il **feed forward layer** e il **softmax layer** del modello. Un **feed forward layer** è un layer per il quale le connessioni dei nodi non creano loop, questi layer (ma in generale le reti di questo tipo) sono chiamati così perché le informazioni fluiscono solo in modo diretto, non ci sono collegamenti che permettono la backpropagation delle informazioni. Lo scopo di questi layer è quello di approssimare funzioni [28]. Il **softmax**

layer svolge un ruolo importante all'interno della rete in quanto si occupa del calcolo della **loss function** utilizzata per ottimizzare i parametri θ della rete neurale [29]. La funzione softmax è una funzione che trasforma un vettore di K valori reali in un vettore di K valori reali che si sommano a 1. I valori di input possono essere positivi, negativi, zero o maggiori di uno, ma softmax li trasforma in valori compresi tra 0 e 1, in modo che possano essere interpretati come probabilità. La funzione softmax è talvolta chiamata **softargmax function** o **multi class logistic regression**. Questo perché softmax è una generalizzazione della regressione logistica che può essere utilizzata per la classificazione multi classe e la sua formula è molto simile alla funzione sigmoidea utilizzata per la regressione logistica. La funzione softmax può essere utilizzata solo quando le classi sono mutualmente esclusive. Molte reti neurali multi-layer terminano in un penultimo layer che genera punteggi di valore reale che non sono adeguatamente ridimensionati e con i quali potrebbe essere difficile lavorare. In questo caso softmax è molto utile perché converte i punteggi in una distribuzione di probabilità normalizzata, che può essere mostrata ad un utente o utilizzata come input per altri sistemi. Per questo motivo è consuetudine aggiungere una funzione softmax come strato finale della rete neurale [30]. La formula di softmax è la seguente:

$$\sigma(\vec{h})_i = \frac{e^{h_i}}{\sum_{j=1}^{K} e^{h_j}}$$

Le variabili hanno il seguente significato:

- $\vec{h}$, vettore di input per la funzione softmax, composto da $(h_0, ..., h_K)$.
- h_i, rappresentano gli elementi del vettore di input per la funzione softmax e possono assumere qualsiasi valore reale che sia positivo, negativo, zero o maggiore di uno.
- e^{h_i}, rappresenta la funzione esponenziale standard che viene applicata a ciascun elemento del vettore di input. Questo fornisce un valore positivo superiore a 0, che sarà molto piccolo se l'input è negativo e molto grande se l'input è grande. Tuttavia, non è ancora fissato nell'intervallo (0,1) che è ciò che è richiesto da una probabilità.
- $\sum_{j=1}^{K} e^{h_j}$, rappresenta il termine di normalizzazione, garantisce che tutti i valori di output della funzione vengano sommati a 1 e che ciascuno sia compreso nell'intervallo (0,1), costituendo così una distribuzione di probabilità valida.
- K, rappresenta il numero di classi.

In genere, per le attività di classificazione binaria come la previsione di vulnerabilità, i parametri ottimali del modello vengono appresi riducendo al minimo la **cross-entropy loss**

function. Tale funzione penalizza la discrepanza tra la probabilità prevista dal modello e la probabilità effettiva (0 per neutrale, 1 per vulnerabile).

3.1.2 Dataset esistenti

Per addestrare un VPM (Vulnerability Prediction Model) c'è bisogno ovviamente di un dataset di codice etichettato come vulnerabile o neutrale. Le istanze vulnerabili dovrebbero essere in un numero tale da permettere al modello di imparare da esse. I ricercatori utilizzano varie fonti di dati per l'addestramento dei DVLP. A seconda di come questi campioni di codice vengono raccolti e come vengono annotati vengono classificati in:

- **Dati sintetici**, gli esempi di codice vulnerabile e le annotazioni vengono create artificialmente. In questo caso gli esempi vengono sintetizzati utilizzando pattern vulnerabili noti. Questi dataset erano originariamente progettati per la valutazione di tool per la predizione di vulnerabilità basati sull'analisi statica e sull'analisi dinamica.
- **Dati semi-sintetici**, in questo tipo di dataset uno tra il codice e l'annotazione è derivato artificialmente. Sebbene questi dataset sono più complessi rispetto a quelli sintetici, non permettono di catturare tutta la complessità delle vulnerabilità del mondo reale, il che è dovuto a semplificazioni ed isolamenti.
- **Dati reali**, in questo caso sia il codice che la corrispettiva annotazione sono derivate da fonti reali.

Con questo tipo di dataset si possono avere due tipi di problemi:

1. **L'origine dei dati e la loro annotazione non sono realistiche.**
   La Figura 3.1 compara i dataset esistenti sulle vulnerabilità in termini di realismo del codice (asse x) e della strategia di annotazione di tale codice (asse y).

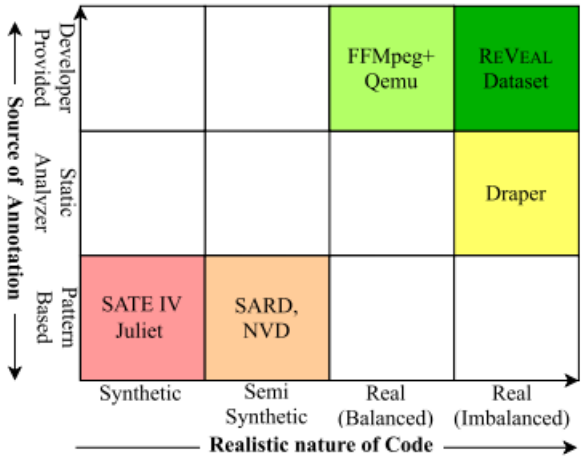

Figura 3.1: Confronto dataset esistenti

Un modello addestrato su un dataset sintetico, ovvero quelli contenenti esempi di codice semplici e non realistici, sarà limitato a rilevare solo quei pattern semplici che raramente si verificano nelle vulnerabilità della vita reale. Si consideri un tipico esempio di buffer overflow mostrato in Figura 3.2, esso è stato utilizzato da *VulDeePecker* [31] e da *SySeVR* [32] (modelli esistenti presi in considerazione). Anche se è un buon esempio, le vulnerabilità nel mondo reale non sono così semplici o isolate come tale esempio.

```
1  void action(char *data) const {
2  // FLAW: Increment of pointer in the loop will cause
3  // freeing of memory not at the start of the buffer.
4     for (; *data != '\0'; data++){
5        if (*data == SEARCH_CHAR){
6           printLine("We have a match!");
7           break;
8        }
9     }
10    free(data);
11 }
```

Figura 3.2: Esempio di buffer overflow

In contrapposizione la Figura 3.3 mostra un esempio di buffer overflow del kernel Linux. Anche se in questo caso la soluzione è semplice da trovare, la fonte della vulnerabilità richiede una comprensione approfondita della semantica delle diverse componenti del codice come le variabili e le funzioni.

```
1  static void eap_request(
2        eap_state *esp, u_char *inp, int id, int len) {
3    ...
4    if (vallen < 8 || vallen > len) {
5         ...
6         break;
7    }
8  /* FLAW: 'rhostname' array is vulnerable to overflow.*/
9  -   if (vallen >= len + sizeof (rhostname)){
10 +   if (len - vallen >= (int)sizeof (rhostname)){
11       ppp_dbglog(...);
12       MEMCPY(rhostname, inp + vallen,
             sizeof(rhostname) - 1);
13       rhostname[sizeof(rhostname) - 1] = '\0';
14       ...
15    }
16    ...
17 }
```

Figura 3.3: Esempio di buffer overflow del kernel Linux

Un modello addestrato su esempi semplici come quello in Figura 3.2 non riesce a ragionare su esempi complessi come quello riportato in Figura 3.3. Inoltre, un qualsiasi modello basato su dati annotati da un analizzatore statico erediterebbe tutti gli svantaggi dell'analisi statica come gli alti tassi di falsi positivi e di conseguenza sarebbe fortemente condizionato.

2. **La distribuzione di esempi vulnerabili e neutrali non è realistica.**
   Nel dataset più realistico che è stato considerato (FFMPeg+Qemu), il rapporto tra istanze vulnerabili e neutrali è di circa 45-55 percento. Tuttavia tale distribuzione non riflette una distribuzione nel mondo reale del codice vulnerabile. In tal caso gli esempi di codice neutrale superano di gran lunga quelli vulnerabili. Inoltre tale dataset non discrimina tra codice vulnerabile e neutrale ma bensì annota gli esempi di codice come vulnerabile o meno analizzando il messaggio di commit di quel codice. Un modello addestrato su un tale dataset potrebbe non funzionare bene in uno scenario di caso d'uso realistico in cui è necessario differenziare la funzione vulnerabile da tutte le altre funzioni neutre.

3.1.3 Modelli esistenti

La selezione del modello dipende principalmente dalle informazioni che si vogliono incorporare. Le scelte più popolari per DLVP sono modelli token-based o graph-based e i

dati di input (codice) vengono pre-processati di conseguenza.

- **Modelli token-based**, in questi modelli il codice viene considerato come una sequenza di token. I modelli esistenti basati su token utilizzano diverse architetture di reti neurali come ad esempio una **Convolutional Neural Network (CNN)** o modelli basati su Random Forest. Per questi modelli token-based relativamente semplici la lunghezza della sequenza di token è un fattore importante per l'impatto sulle prestazioni in quanto è difficile per i modelli ragionare su lunghe sequenze. Per far fronte a questo problema *VulDeePecker* e *SySeVR* estraggono porzioni di codice. La motivazione dietro lo slicing è che non tutte le righe di codice sono di egual importanza per la previsione della vulnerabilità. Perciò, invece di considerare l'intero codice, solo una parte di esso, estratta da "punti d'interesse" come chiamate API, indicizzazione di array, utilizzo del puntatore e così via, vengono presi in considerazione per la previsione della vulnerabilità mentre il resto viene omesso.

- **Modelli graph-based**, questi modelli considerano il codice come un grafo ed incorporano diverse dipendenze sintattiche e semantiche. Per la predizione di vulnerabilità possono essere utilizzati diversi tipi di grafi sintattici (es. Abstarct Syntax Tree) e semantici (es. Control Flow Graph, Data Flow Graph).

Entrambi i modelli devono affrontare il problema dell'*esplosione del vocabolario*, cioè il numero dei possibili identificatori (nomi di variabili, nomi di funzioni, nomi di costanti) può essere virtualmente infinito e i modelli devono ragionare su questi identificatori. Un modo comune per risolvere questo problema è sostituire i token con nomi astratti. Ad esempio, *VulDeePecker* sostituisce la maggior parte dei nomi delle variabili e delle funzioni con nomi simbolici (VAR1, FUNC1, VAR2 etc.).
Gli input previsti per tutti i modelli sono vettori con valori reali comunemente noti come embedding. Ci sono diversi modi per incorporare token nei vettori.

- Uno di questi è usare un **embedding layer** che viene addestrato congiuntamente con l'attività di previsione delle vulnerabilità. Tale layer viene utilizzato principalmente nelle applicazioni relative all'elaborazione del linguaggio naturale come la modellazione del linguaggio, ma può essere utilizzato anche per altre attività che coinvolgono le reti neurali. Mentre si affrontano problemi di NLP (Natural Language Processing) si può utilizzare un layer di embedding che viene addestrato insieme alla rete. L'**embedding layer** consente di convertire ogni parola in un vettore di lunghezza e dimensione fissata. Il vettore risultante è un dense vector che ha valori reali invece di soli 0 e 1. La

lunghezza fissata del word vector aiuta a rappresentare in un modo migliore le parole con dimensioni ridotte. In questo modo il dense layer lavora come una tabella di ricerca dove le parole sono le chiavi della tabella mentre i dense word vector sono i valori [33].

- Un'altra opzione è quella di utilizzare un tool esterno, ad esempio Word2Vec, per creare vettori rappresentativi per ogni token. *VulDeePecker* e *SySeVR* utilizzano Word2Vec per trasformare i loro token in vettori. *Devign* [34] (un'altro modello esistente considerato), al contrario, usa Word2Vec per trasformare i token di codice concreti in vettori reali. Word2vec è un insieme di modelli che sono utilizzati per produrre word embedding, il cui pacchetto fu originariamente creato in linguaggio `C` da Tomas Mikolov. Word2vec è una semplice rete neurale a due layer progettata per elaborare il linguaggio naturale, l'algoritmo richiede in ingresso un corpus e restituisce un insieme di vettori che rappresentano la distribuzione semantica delle parole nel testo. Per ogni parola contenuta nel corpus, in modo univoco, viene costruito un vettore in modo da rappresentarla come un punto nello spazio multidimensionale creato. In questo spazio le parole saranno più vicine se riconosciute come semanticamente più simili [35].

Una volta scelto un modello, viene eseguito il pre-processing appropriato sul training set, dopodiché il modello è pronto per essere addestrato riducendo al minimo una loss function. La maggior parte degli approcci esistenti ottimizzano i parametri del modello riducendo al minimo alcune varianti della cross-entropy loss function. Con tali modelli si possono però avere dei problemi, tra cui:

1. **I modelli basati su token mancano di rappresentatività sintattica**
   Tali modelli presuppongono che i token siano linearmente dipendenti l'uno dall'altro, e quindi, sono presenti solo dipendenze lessicali tra i token, mentre le dipendenze semantiche vengono perse, anche se esse spesso svolgono ruoli importanti nella previsione delle vulnerabilità. Per incorporare un po' di informazioni semantiche *VulDeePecker* e *SySeVR* estraggono sezioni di codice da punti potenzialmente interessanti. Le vulnerabilità del mondo reale sono molto più complesse e richiedono ragionamenti su control flow, data flow, relazioni di dominanza e altri tipi di dipendenze tra gli elementi di codice. D'altronde, i modelli basati su grafi, in generale, sono molto più costosi dei modelli token-based e non funzionano bene in un ambiente con risorse limitate.

2. **I modelli attuali sono "fragili"**
   Un altro problema con gli approcci esistenti è che, sebbene i modelli addestrati imparino

a discriminare gli esempi di codice vulnerabile e neutrale, il paradigma di training non si concentra esplicitamente su di essi aumentando il divario tra gli esempi vulnerabili e neutrali. Infine i modelli soffrono di uno sbilanciamento dei dati tra codice vulnerabile e codice neutrale, infatti nel mondo reale la proporzione degli esempi vulnerabili rispetto a quelli neutri è estremamente bassa nel dataset. Quando un modello viene addestrato con un tale dataset sbilanciato, tende ad essere distorto dagli esempi neutri (bias implicito del dataset).

3.1.4 Approcci di valutazione esistenti

Per comprendere l'applicabilità di un modello per la rilevazione delle vulnerabilità nel mondo reale, bisogna dapprima valutarlo. In molti casi il modello viene valutato sul test set. Gli esempi di test subiscono lo stesso pre-processing che viene fatto sui dati di training, dopodiché il modello prevede la vulnerabilità di questi esempi di test. Questo approccio di valutazione fornisce una stima di come il modello può funzionare se viene utilizzato per rilevare le vulnerabilità nel mondo reale.
Un problema con questo tipo di approccio è che **l'ambito di valutazione è limitato**. Infatti tutti gli approcci esistenti valutano le loro prestazioni utilizzando il proprio test set. Tale strategia di valutazione non fornisce una panoramica completa dell'applicabilità dei modelli per altri esempi del mondo reale. Tutto quello che si può apprendere da tale valutazione intra-dataset è quanto bene si adatta l'approccio utilizzato al dataset. Sebbene ricerche riportano alcuni casi di studio limitati sull'individuazione di vulnerabilità in progetti reali, questi progetti non fanno luce sui falsi positivi e falsi negativi ottenuti. Il numero di falsi positivi e falsi negativi è direttamente correlato allo sforzo dello sviluppatore nella predizione di vulnerabilità.

3.1.5 Collezione dei dati per ReVeal

Per affrontare le limitazioni dei dataset esistenti, è stato creato un dataset più robusto che comprende dati del mondo reale, questo dataset è **ReVeal**. Esso è stato creato tramite il monitoraggio delle vulnerabilità di due progetti open source: il kernel di Debian e Chromium (progetto open source di Chrome). Sono stati selezionati questi progetti perché:

- Sono due progetti pubblici popolari e ben mantenuti con una grande storia evolutiva.
- Rappresentano due importanti domini (OS e browser) che esibiscono diversi problemi di sicurezza.

- Entrambi i progetti hanno molti rapporti di vulnerabilità pubblicamente disponibili.

Per la costruzione del dataset sono stati raccolti i problemi già risolti, ovvero per i quali esistono patch disponibili pubblicamente. Per Chromium i dati sono stati raccolti dal proprio repository di bug Bugzilla, mentre per il kernel di Debian i dati sono stati raccolti dal Debian security tracker. L'identificazione di problemi relativi alle vulnerabilità viene effettuata scegliendo le patch etichettate con "sicurezza". Questo meccanismo di identificazione è ispirato all'identificazione dei problemi di sicurezza delle tecniche presenti in letteratura che sono state analizzate in precedenza. L'approccio proposto in letteratura filtra le commit che non hanno parole chiave relative alla sicurezza. Per ogni patch sono state estratte le corrispondenti versioni vulnerabili e fixed (cioè vecchia versione vulnerabile e nuova versione) del codice sorgente e dell'intestazione dei file `C/C++` che sono stati modificati nella patch. Le versioni precedenti delle funzioni che sono state modificate nella patch sono state annotate come **vulnerable** e le versioni fixed (versione dopo la patch) sono state invece annotate come **clean**. Inoltre le funzioni che non sono coinvolte nella patch sono state annotate come **clean**. Annotando il codice in tal modo si simula uno scenario di previsione delle vulnerabilità reale dove un modello imparerebbe ad individuare le funzioni vulnerabili nel contesto di tutte le altre funzioni. Inoltre, mantenere la variante fixed della funzione vulnerabile aiuta il modello ad apprendere la natura della patch di correzione della vulnerabilità. Un esempio di strategia di raccolta dei dati viene illustrato nella Figura 3.4, in questo caso si hanno due versioni di un file `.c`, la versione $k-1$ del file ha una vulnerabilità che è stata corretta nella successiva versione k.

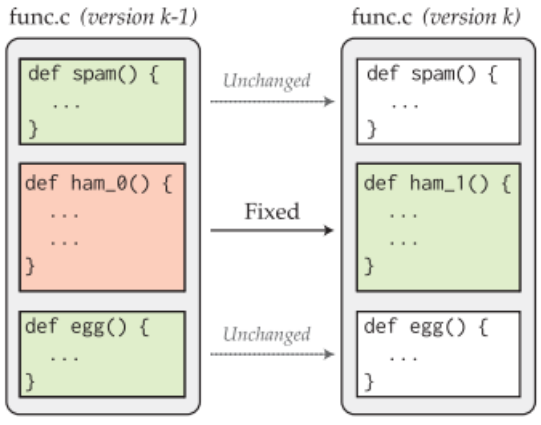

Figura 3.4: Esempio stategia di raccolta dei dati

Nel dataset vengono incluse sia la funzione `ham_0()` che la funzione `ham_1()` e vengono annotate rispettivamente come vulnerable e clean.

La Figura 3.5 mostra una percentuale cumulativa delle commit di correzione delle vulnerabilità in relazione al numero di funzioni vulnerabili corrette. È stato osservato che nella maggior parte dei casi le patch cambiano un numero piccolo di funzioni. Nell'80% dei casi, le modifiche riguardano 4 funzioni o meno.

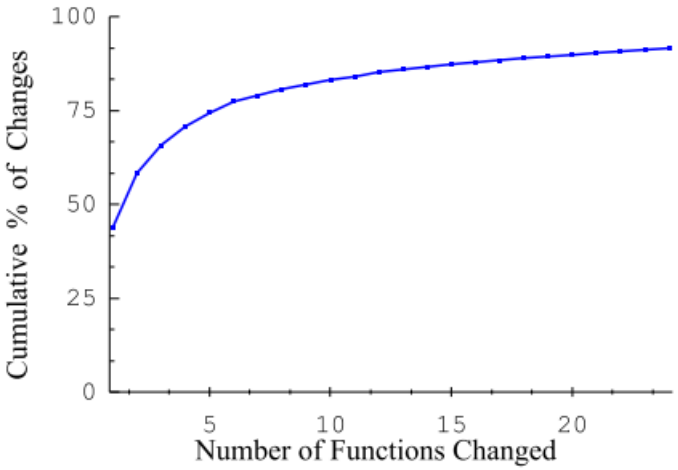

Figura 3.5: Percentuale delle commit in relazione al numero di funzioni vulnerabili corrette

La Tabella 3.1 riassume i dettagli dei dataset esistenti in letteratura valutati e del dataset che è stato creato (ReVeal).

Dataset	#Programmi	%Vuln	Granularità	Tipo modello	Modello
SATE IV Juliet	11,896	45.00	Funzione	Token	CNN+RF
SARD	9,851 14,000	31.00 13.41	Slice Slice	Token Token	BLSTM BGRU
NVD	840 1,592	31.00 13.41	Slice Slice	Token Token	BLSTM BGRU
Draper	1,274,366	6.46	Funzione	Token	CNN+RF
FFMPeg+Qemu	22,361	45.02	Funzione	Grafo	GGNN
ReVeal	18,169	9.16	Funzione	Grafo	GGNN+MLP+Triplet Loss

Tabella 3.1: Dettagli dataset esistenti

3.1.6 Costruzione del modello

Dopo aver raccolto i dati si passa alla costruzione del modello, la Figura 3.6 riporta la pipeline utilizzata per il progetto ReVeal. Essa opera fondamentalmente in due fasi, **feature extraction** (fase 1) e **addestramento del modello** (fase 2). Nella prima fase il codice viene tradotto in un graph-embedding. Nella seconda fase viene addestrato un representation learner sulle feature estratte per apprendere una rappresentazione che più demarca gli esempi vulnerabili da quelli neutrali.

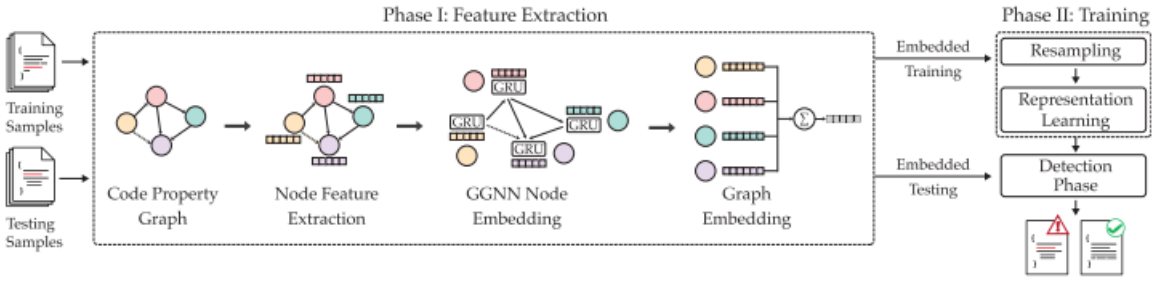

Figura 3.6: Pipeline del progetto ReVeal

Di seguito vengono riportate nel dettaglio le due fasi dalle quali è formata la pipeline di ReVeal.

(Fase 1) Feature Extraction

L'obiettivo di questa fase è convertire il codice in un vettore di feature compatto e di lunghezza uniforme mantenendo le informazioni semantiche e sintattiche. Per far fronte al problema della mancanza di rappresentatività sintattica riscontrata nei modelli token-based esistenti riportato nella Sezione 3.1.3, per ReVeal il feature vector viene estratto utilizzando una rappresentazione su grafo del codice.
Per estrarre le caratteristiche sintattiche e semantiche dal codice viene generato un **Code Property Graph (CPG)**. Il CPG è una rappresentazione particolarmente utile del codice originale poiché offre una rappresentazione combinata e succinta del codice costituita da elementi del control-flow e del data-flow graph oltre all'AST (Abstract Syntax Tree) ed al Program Dependency Graph (PDG). Ciascuno degli elementi riportati in precedenza offre contesto aggiuntivo sulla struttura semantica complessiva del codice.
Di seguito vengono riportate alcune informazioni sui grafi appena citati:

- **Control-flow graph**, modella il flusso di controllo tra i blocchi di base in un programma. Un blocco di base è una sequenza di operazioni che vengono sempre eseguite insieme,

a meno che un'operazione non sollevi un'eccezione. Un **CFG** è un grafo orientato, $G = (V, E)$. Ogni nodo $n \in V$ corrisponde a un blocco di base. Ogni arco $e = (n_i, n_j) \in E$ corrisponde ad un possibile trasferimento di controllo dal blocco n_i al blocco n_j. Un CFG ha un nodo per ogni blocco di base e un arco per ogni possibile trasferimento di controllo tra blocchi. Esso fornisce una rappresentazione grafica dei possibili percorsi del flusso di controllo a runtime. La Figura 3.7 riporta un esempio di tale grafo per un ciclo while. L'arco da `stmt1` all'intestazione del ciclo crea un loop [36].

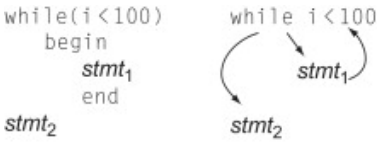

Figura 3.7: Esempio di Contol-flow Graph

- **Data-flow Graph**, è un modello di un programma senza condizionali. In un linguaggio di programmazione di alto livello un segmento di codice senza condizionali, con un solo punto di ingresso e di uscita, è noto come blocco di base. Un **data flow graph** è un insieme di archi e nodi in cui i nodi sono luoghi in cui vengono assegnate o utilizzate le variabili e gli archi mostrano la relazione tra i luoghi in cui viene assegnata una variabile e dove viene successivamente utilizzato il valore assegnato. Un data flow graph è generalmente disegnato come nella Figura 3.8. In questo caso, le variabili non sono rappresentate in modo esplicito da nodi. Invece, gli archi sono etichettati con le variabili che rappresentano. Di conseguenza, una variabile può essere rappresentata da più di un arco. Tuttavia, gli archi sono diretti e tutti gli archi per una variabile devono provenire da un'unica fonte. Viene utilizzato questo modulo per la sua semplicità e compattezza.

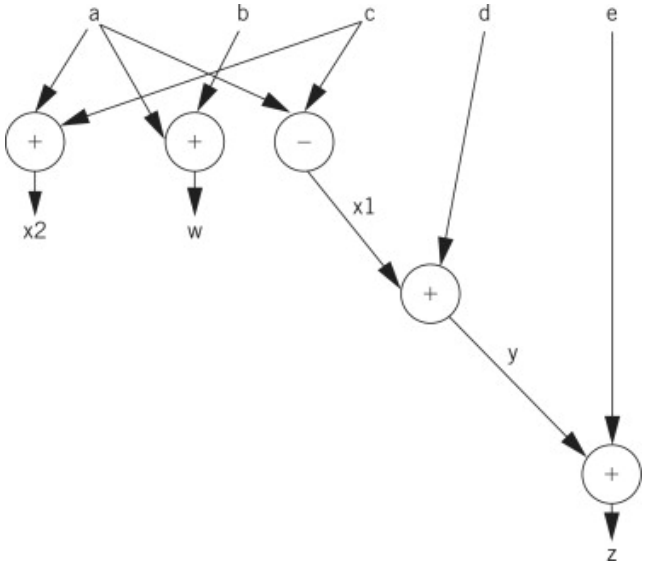

Figura 3.8: Esempio di Data-flow Graph

Prima di disegnare tale grafo in alcuni casi è necessario fare una piccola fase di pre-processing sui dati che consiste nell'andare a sostituire le assegnazioni multiple con assegnazioni singole in modo tale che per una variabile venga effettuata una singola assegnazione. L'avere una singola assegnazione per ogni variabile permette di avere un data flow graph aciclico, questa caratteristica è importante per molti tipi di analisi che possono essere effettuati su tale grafo. Infatti il grafo riportato in precedenza rappresenta il codice in Figura 3.9 nel quale le assegnazioni `x1`, `x2` corrispondono in realtà all'assegnazione di una singola variabile `x`, tale operazione è stata fatta per avere un data flow graph aciclico [37].

```
w = a+b;
x1= a−c;
y = x1+d;
x2 = a+c;
z = y+e;
```

Figura 3.9: Blocco di base associato al control-flow graph precedente

- **Abstract Syntax Tree**, include l'importante struttura sintattica del programma omettendo tutte le caratteristiche non terminali che non sono necessarie per comprendere quella struttura. A causa dei suoi legami con la sintassi del linguaggio di programmazione, un AST conserva rappresentazioni concise per la maggior parte delle astrazioni nel linguaggio di programmazione d'origine [38].

- **Progran Dependency Graph**, per un blocco b, il suo program dependency graph $G = (V, E)$ ha un nodo per ogni operazione in b. Un arco diretto collega due nodi $n1$ e $n2$ se $n2$ utilizza il risultato di $n1$ [39].

- **Code Property Graph**, questo grafo combina proprietà degli Abstract Syntax Tree, dei Control-flow Graph e dei Program Dependence Graph in un'unica struttura. Questa visione globale del codice consente di modellare elegantemente modelli comuni di vulnerabilità mediante attraversamenti di grafi. Simile ad una query, un attraversamento del grafo *"passa sopra"* il code property graph ed esamina la struttura del codice, il flusso di controllo e le dipendenze dei dati associate a ciascun nodo. Questo accesso congiunto a diverse proprietà del codice consentono di creare modelli concisi per diversi tipi di difetti e quindi aiuta a controllare grandi quantità di codice per le vulnerabilità [40].

Formalmente, un CPG è indicato come $G = (V, E)$, dove V rappresenta i vertici (o nodi) del grafo ed E rappresenta gli archi. Ogni vertice v nel CPG è composto dal tipo di vertice (ad esempio, `ArithmeticExpression`, `CallStatement` etc.) e un frammento del codice originale. Per codificare le informazioni sui tipi, viene utilizzato un vettore di codifica one-hot indicato da T_v. Per codificare il frammento di codice nel vertice, viene utilizzato un embedding word2vec indicato con C_v. Successivamente, per creare il vertex embedding, vengono concatenati T_v e C_v in una notazione vettoriale congiunta per ogni vertice. L'attuale incorporamento dei vertici non è adeguato poiché considera ogni vertice isolatamente. Mancano quindi informazioni sui suoi vertici adiacenti e, di conseguenza, sulla struttura del grafo. Questo può essere affrontato assicurando che ogni embedding di vertice riflette sia le sue informazioni che quelle dei suoi vicini. Per questo scopo vengono utilizzate le **Gated Graph Neural Network (GGNN)**.
Prima di parlare delle **GGNN** è opportuno introdurre le **Graph Neural Network (GNN)**. Le **GNN** sono un'architettura di rete neurale generale definita secondo una struttura a grafo $G = (V, E)$. I nodi $v \in V$ assumono valori univoci da $1, ..., |V|$ e gli archi sono delle coppie $e = (v, v') \in VxV$. Il vettore del nodo (o rappresentazione del nodo o embedding del nodo) per il nodo v viene indicato con $h_v \in \mathbb{R}^D$. I grafi possono ache contenere label $l_v \in 1, ..., L_V$ per ogni nodo v e label o tipi di archi $l_e \in 1, ..., L_E$ per ogni arco. Viene fatto un overload di questa notazione e viene impostato $h_s = h_v | v \in S$ dove S è un insieme di nodi e $l_s = l_e | e \in S$ dove S è un insieme di archi. Le GNN mappano i grafi in output in due step:

1. Innanzitutto, c'è una fase di propagation che calcola le rappresentazione dei nodi per ciascun nodo.

2. Come seconda cosa, un modello di output $o_v = g(h_v, l_v)$ effettua il mapping tra la rappresentazione dei nodi e le label corrispondenti in un output o_v per ogni nodo $v \in V$.

Tutti i parametri vengono appresi utilizzando la gradient based optimization. La più grande differenza tra le **GNN** e le **GGNN** è che per le **GGNN** vengono utilizzate le **Gated Recurrent Units (GRU)** [41], vengono *"srotolate"* le dipendenze in un numero fisso di step T e viene utilizzata la backpropagation nel tempo per calcolare i gradienti. Ciò richiede abbastanza memoria ma rimuove la necessità di parametri vincolanti per assicurare la convergenza. Nelle GNN non ha senso inizializzare la rappresentazione dei nodi in quanto il vincolo della contraction map assicura che il fixed point sia indipendente dalle inizializzazioni. Questo però non è il caso delle GGNN che consentono di incorporare le etichette dei nodi come input aggiuntivi. Per distinguere queste etichette dei nodi usate come input da quelle introdotte in precedenza, esse vengono chiamate **node annotations** e viene utilizzato un vettore x per denotare queste annotazioni.
Le GGNN supportano le attività di **node selection** rendendo $o_v = g(h_v^{(T)}, x_v)$ per ogni nodo $v \in V$ di output e applicando una funzione **softmax** sullo score di tali nodi [42].
I **feature vector** di tutti i nodi del grafo (x) insieme agli archi (E) compongono l'input della GGNN. Per ogni vertice del **CPG**, la **GGNN** assegna una **Gated Recurrent Unit (GRU)** che aggiorna il vertex embedding corrente assimilando l'embedding dei vicini. Formalmente:

$$x'_v = GRU(x_v, \sum_{(u,v)\in E} g(x_u)$$

$GRU(\cdot)$ è una **Gated Recurrent Function**, x_v è l'embedding del vertice corrente v e $g(\cdot)$ è una funzione di trasformazione che assimila l'embedding di tutti i vertici vicini v. x'_v è la rappresentazione trasformata dalla GGNN dell'embedding originale x_v del vertice v.
Lo step finale del pre-processing è quello di aggregare tutti gli embedding dei vertici x'_v per creare un singolo vettore che rappresenti l'intero CPG denotato da x_g, cioè:

$$x_g = \sum_{v\in V} x'_v$$

Il risultato della pipeline presentata finora è una rappresentazione vettoriale m-dimensionale del codice sorgente. Per effettuare il pre-training della GGNN viene aggiunto un classification layer in cima alla feature extraction della GGNN. Questo meccanismo di training è simile a quello utilizzato in *Devign*, tale meccanismo viene utilizzato anche da *Russell et al.* [43] (ulteriore modello esistente considerato).

(Fase 2) Training

Nei dati del mondo reale, il numero di campioni neutri (istanze negative) supera di gran lunga gli esempi vulnerabili (istanze positive) come viene mostrato nella Tabella 3.1. Se non affrontato, questo problema introduce una distorsione indesiderata nel modello limitandone le prestazioni predittive. Inoltre i feature vector estratti da esempi vulnerabili e neutrali esibiscono una significante sovrapposizione nello spazio delle feature. Questo rende difficile distinguere gli esempi neutrali da quelli vulnerabili. Ovviamente l'addestramento di un modello di DL senza tener conto di questa sovrapposizione lo rende soggetto a scarse prestazioni predittive. Per mitigare questo problema è stato proposto un approccio in due step. In primo luogo viene utilizzato il **re-sampling** per bilanciare il rapporto tra istanze vulnerabili e neutrali nei dati di training. Il **re-sampling** consiste nel fare **over-sampling** sulla classe di minoranza oppure **under-sampling** sulla classe di maggioranza [44].
Successivamente viene addestrato un **representation learner** sul training set bilanciato per apprendere una rappresentazione che più distingue le istanze vulnerabili da quelle neutrali.

1. **Bilanciamento del dataset**
   Per gestire lo sbilanciamento dei dati viene utilizzata la tecnica **Synthetic Minority Oversampling Technique (SMOTE)**. La tecnica SMOTE effettua un sotto-campionamento della classe di maggioranza mentre effettua un super-campionamento della classe di minoranza affinché tutte le classi abbiamo la stessa frequenza. Nel caso della previsione delle vulnerabilità, la classe minoritaria è solitamente quella vulnerabile. SMOTE ha dimostrato di essere efficace in un certo numero di domini e dataset sbilanciati. Durante il super-campionamento, SMOTE seleziona un'istanza vulnerabile e trova k vicini vulnerabili. Dopodiché costruisce un'istanza sintetica della classe minoritaria interpolando tra sé e uno dei suoi vicini. Durante il sotto-campionamento, SMOTE rimuove casualmente istanze neutre dal training set. Questo processo è ripetuto fino a raggiungere un equilibrio tra le istanze vulnerabili e quelle neutre.

2. **Addestramento del representation learner**
   Il graph embedding del codice vulnerabile e neutrale delle istanze alla fine della Fase 3.1.6 tendono a mostrare un elevato grado di sovrapposizione nello spazio delle feature. Questo rende i modelli "fragili" come evidenziato in precedenza nella Sezione 3.1.3. Questo effetto e illustrato nella Figura 3.10. Come si può notare non c'è una distinzione chiara tra le istanze vulnerabili (denotate con $+$) e le istanze neutrali (denotate con $\circ$). Questa mancanza di separazione rende particolarmente difficile addestrare un modello

di ML. Per migliorare le prestazioni predittive, è stato cercato un modello che potesse proiettare le caratteristiche dallo spazio originale non separabile in uno **spazio latente** che offre una migliore separabilità tra campioni vulnerabili e neutri.

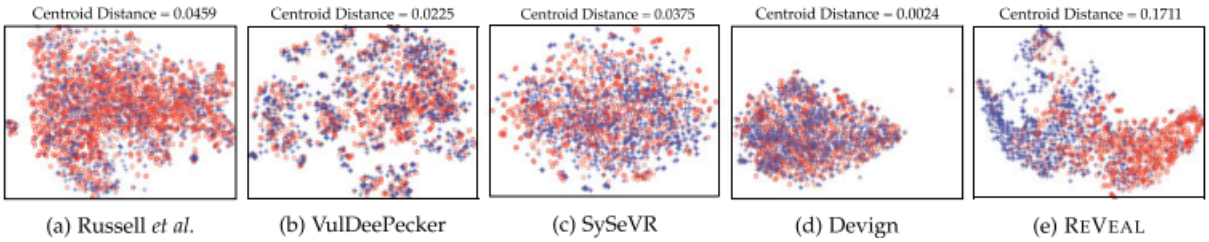

Figura 3.10: Plot delle istanze dei dataset

Per questo motivo viene utilizzato un **representation learner**, progettato per la trasformazione di un vettore di feature in input (x_g) in una rappresentazione latente denotata con $h(x_g)$. Esso è formato da tre gruppi di layer, l'input layer (x_g), un set di layer intermedi parametrizzati da θ denotati con $f(\cdot, \theta)$ ed un layer di output denotato con $\hat{y}$. Il representation learner proposto lavora prendendo in input il graph embedding originale (x_g) e passandolo attraverso i layer intermedi $f(\cdot, \theta)$. I layer intermedi proiettano l'embedding originale in uno spazio latente $h(x_g)$. Infine l'output layer utilizza le feature nello spazio latente per predire la vulnerabilità come $\hat{y} = \sigma(W * h(x_g) + b)$, dove σ rappresenta la funzione softmax, h_g è lo spazio latente, W e b rappresentano rispettivamente i pesi del modello e il bias.
Per massimizzare la separazione tra le istanze vulnerabili e neutrali nello spazio latente, viene utilizzata la **triplet loss function** come funzione di perdita. La triplet loss function è composta da tre loss function individuali: **cross entropy loss** L_{CE}, **projection loss** L_p e **regularization loss** L_{reg}. Essa è data da:

$$L_{trp} = L_{ce} + \alpha * L_p + \beta * L_{reg}$$

α e β sono due iperparametri che indicano il contributo della projection loss e della regularizarion loss rispettivamente.
La prima componente della triplet loss è la **cross entropy loss** utilizzata per penalizzare le classificazioni errate. La **cross entropy loss** aumenta quando la probabilità prevista diverge dal valore effettivo dell'etichetta. Essa viene calcolata come:

$$L_{CE} = -\sum \hat{y} \cdot log(y) + (1 - \hat{y}) \cdot log(1 - y)$$

In questo caso y è il valore effettivo dell'etichetta e $\hat{y}$ rappresenta il valore predetto.
La seconda componente della triplet loss è la **projection loss** e viene utilizzata per

quantificare quanto bene la rappresentazione latente può separare gli esempi vulnerabili e neutrali. Una rappresentazione latente è considerata utile se tutti gli esempi vulnerabili nello spazio latente sono vicini e simultaneamente sono lontani da tutti gli esempi neutri. Di conseguenza viene definita la loss function L_p che viene ottenuta come:

$$L_p = |\mathbb{D}(h(x_g), h(x_{same})) - \mathbb{D}(h(x_g), h(x_{diff})) + \gamma|$$

Qui $h(x_{same})$ è la rappresentazione latente di un'istanza appartenente alla stessa classe di x_g mentre $h(x_{diff})$ è la rappresentazione latente di un'istanza appartenente ad una classe diversa dalla classe di x_g. Inoltre γ è un iperparametro utilizzato per definire un limite inferiore di separazione. Infine, $\mathbb{D}(\cdot)$ rappresenta la distanza del coseno tra due vettori ed è data da:

$$\mathbb{D}(v_1, v_2) = 1 - \left| \frac{v_1 \cdot v_2}{||v_1|| * ||v_2||} \right|$$

Se la distanza di due esempi che appartengono alla stessa classe è grande oppure se la distanza tra due esempi che appartengono a classi diverse è piccola allora L_p ha un valore alto per indicare una soluzione sub-ottimale.

L'ultima componente della triplet loss è la **regularization loss** L_{reg} che viene utilizzata per limitare la grandezza della rappresentazione latente $(h(x_g))$. È stato osservato che, in diverse iterazioni, la rappresentazione latente $h(x_g)$ dell'input x_g tende ad aumentare arbitrariamente di grandezza. Tale aumento arbitrario impedisce al modello di convergere. Pertanto viene utilizzata una regularization loss L_{reg} per penalizzare rappresentazioni latenti $(h(x_g))$ di grandezza maggiore. La regularization loss è ottenuta come:

$$L_{reg} = ||h(x_g)|| + ||h(x_{same})|| + ||h(x_{diff})||$$

Con la triplet loss function, ReVeal addestra il proprio modello per ottimizzare i parametri θ, W, b minimizzando suddetta funzione.

3.1.7 Valutazione

Per comprendere le prestazioni di un modello, ricercatori e sviluppatori hanno bisogno di comprendere le prestazioni di un modello rispetto ad un set di istanze note. Come riportato nella Sezione 3.1.4 le attuali strategie di valutazione sono limitate sia nell'ambito che nella scelta delle metriche di valutazione. Ci sono due aspetti importanti da considerare durante la valutazione e sono: le **metriche di valutazione** e la **procedura di valutazione**.

1. **Formulazione del problema e metriche di valutazione**
   La maggior parte degli approcci formula il problema come un problema di classificazione, dove, dato un esempio di codice, il modello deve effettuare una predizione binaria indicando se il codice è vulnerabile o meno. Questa formulazione si basa sul fatto che ci sia un numero sufficiente di esempi su cui addestrare il modello. La classificazione per lo studio trattato avviene a livello di funzione e non su slice di codice come avviene per altri modelli esistenti in letteratura analizzati. Il modello ottenuto è stato valutato secondo quattro metriche di valutazione ben note: accuracy, precision, recall e F1-score.

2. **Procedura di valutazione**
   Dato che i modelli dipendono fortemente dalla casualità, per rimuovere l'eventuale bias creato da questa casualità, vengono effettuate 30 esecuzioni del modello. Ad ogni esecuzione il dataset viene splittato randomicamente in training, validation e test set con 70, 10 e 20 percento delle istanze del dataset rispettivamente.

3.1.8 Risultati Empirici

L'obiettivo degli approcci DVLP è quello di essere in grado di prevedere le vulnerabilità nel mondo reale. I dataset esistenti su cui vengono addestrati i modelli esistenti in letteratura, contengono esempi semplicistici che rappresentano le vulnerabilità del mondo reale. Pertanto, in teoria, questi modelli dovrebbero essere utilizzabili per rilevare le vulnerabilità nel mondo reale. Ci sono due possibili scenari per poter utilizzare questi modelli:

1. Effettuare il training dei modelli pre-esistenti con il proprio dataset originario per poi andare ad utilizzare tali modelli addestrati sulle vulnerabilità del mondo reale.

2. Ricostruire i modelli addestrandoli su dataset del mondo reale e poi successivamente utilizzarli per la previsione di vulnerabilità.

I risultati che si ottengono da questi utilizzi non sono ottimali, infatti, gli approcci esistenti falliscono nella previsione delle vulnerabilità del mondo reale. Il fallimento di questi modelli è dovuto ad una serie di problemi tra cui:

- **Duplicazione dei dati**, le tecniche di pre-processing come slicing e tokenization introducono un gran numero di duplicati nel training e test set. La presenza di questi duplicati nel training set potrebbe portare il modello ad apprendere feature irrilevanti. Idealmente un modello di DL dovrebbe essere addestrato e testato su un dataset in cui il 100% degli esempi è unico.

- **Sbilanciamento dei dati**, i dati del mondo reale nella maggior parte dei casi contengono molte più istanze neutrali rispetto a quelle vulnerabili. Un modello addestrato su un tale dataset sbilanciato potrebbe essere notevolmente influenzato a predire le nuove istanze come istanze della classe di maggioranza (bias implicito dei dati).

- **Learning di feature irrilevanti**, per costruire un buon modello è importante capire quali feature utilizza il modello per effettuare la predizione. Un buon modello dovrebbe dare maggiore importanza alle feture relative a esempi di codice vulnerabile.

- **Mancanza di separazione tra le classi**, gli approcci esistenti traducono il codice sorgente in un vettore numerico di feature che può essere utilizzato per addestrare il modello. L'efficacia dei VPM dipende da quanto separabili sono i feature vector delle due classi.

Ciò che è stato osservato è che gli approcci esistenti hanno numerose limitazioni, in quanto, introducono la duplicazione dei dati, non gestiscono lo sbilanciamento dei dati, non apprendono informazioni semantiche, le classi mancano di separabilità. Per far fronte a questi problemi riscontrati nei modelli pre-esistenti è stato sviluppato ReVeal che affronta i suddetti problemi nel seguente modo:

- Per affrontare la **duplicazione dei dati**, durante il preprocessing converte le istanze nel corrispondete CPG dei cui vertici viene fatto l'embedding con una GGNN e l'aggregazione con un'aggregation function.

- Per affrontare lo **sbilanciamento dei dati** viene utilizzata la tecnica SMOTE per bilanciare il dataset.

- Per affrontare la **mancanza di separabilità delle classi** viene utilizzato un representation learner che impara automaticamente come riequilibrare i feature vector di input in modo tale che le classi vulnerabili e neutrali siano separate al massimo.

3.2 Limitazioni del progetto ReVeal e implicazioni pratiche

Dopo un'analisi approfondita del progetto ReVeal, fatta nella Sezione 3.1 e del codice Python, messo a disposizione per poter effettuare il pre-processing dei dati e per costruire ed addestrare il modello di Deep Learning da loro sviluppato, si può evincere che viene utilizzato un modello con un'architettura complessa, il quale richiede un overhead di tempo e spazio di memoria per effettuare l'addestramento. Per far fronte a questa limitazione individuata si è deciso di effettuare uno studio empirico sulla generalizzabilità delle architetture dei modelli

di previsione di vulnerabilità rispetto ai risultati ottenuti. Tale studio viene riportato nella sezione successiva.

3.3 Studio empirico sulla generalizzabilità dei modelli rispetto ai risultati ottenuti

Nell'ottica delle considerazioni fatte nella sezione precedente si è deciso di costruire ed addestrare una rete **Multi Layer Perceptron (MLP)**, che è un architettura di rete neurale più semplice rispetto a quella utilizzata in ReVeal, per poi in seguito mettere a confronto i risultati ottenuti e fare le opportune considerazioni. Quindi l'obiettivo finale di tale studio è quello di capire se, effettivamente, la predizione di vulnerabilità necessita di avere modelli complessi di deep learning oppure se, con algoritmi meno complessi da un punto di vista computazionale e più semplici da configurare si ottengono risultati diversi da quelli ottenuti con tali modelli complessi.
L'architettura di una rete **MLP** è stata brevemente descritta nella Sezione 2.4.1. Di seguito vengono riportate le API della libreria Python `sklearn` che permettono di istanziare una MLP:

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,),
↪  activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto',
↪  learning_rate='constant', learning_rate_init=0.001, power_t=0.5,
↪  max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False,
↪  warm_start=False, momentum=0.9, nesterovs_momentum=True,
↪  early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999,
↪  epsilon=1e-08, n_iter_no_change=10, max_fun=15000)
```

Gli iperparametri più importanti e quelli utilizzati per creare un setting quanto più simile a ReVeal sono:

- **`hidden_layer_sizes`**, rappresenta il numero di neuroni presenti in un hidden layer, di default è settato a 100.
- **`n_layers`**, indica il numero di hidden layer della rete, di default ha valore 1.
- **`activation`**, indica la funzione di attivazione utilizzata, di default viene utilizzata la funzione di attivazione **ReLu**.

- **`solver`**, rappresenta l'ottimizzatore, ovvero colui che si occupa di ottimizzare i pesi delle connessioni per calcolare il vettore θ^*, di default viene utilizzato l'ottimizzatore **Adam**, utilizzato anche nel progetto ReVeal.

- **`alpha`**, indica la *"forza"* del termine di regolarizzazione L2, tale termine viene diviso per la dimensione dell'esempio quando viene sommato alla loss. Come default assume il valore 0.0001.

- **`batch_size`**, indica la dimensione del minibatch, il suo valore di default è "auto", ciò vuol dire che `batch_size=min(200, n_samples)`.

- **`max_iter`**, indica il numero massimo di epoche che possono essere fatte prima che il modello converga, di default questo parametro è settato a 200.

- **`early_stopping`**, permette si settare l'interruzione anticipata dell'esecuzione nel caso in cui i risultati non migliorano dopo un certo numero di iterazioni. Se tale parametro è impostato a `true` il modello metterà automaticamente da parte il 10% dei dati di training per utilizzarli come validation e terminerà l'addestramento quando il validation score non migliora dopo `n_iter_no_change` epoche successive. Tale parametro di default è impostato a `false`.

- **`n_iter_no_change`**, indica il massimo numero di epoche dopo le quali l'addestramento del modello termina se quest'ultimo non migliora rispetto al modello migliore ottenuto in precedenza. Di default questo parametro è impostato a 10.

Nell'ambito dello studio condotto si è deciso di implementare tre varianti della rete considerata:

1. Una prima variante addestrata con la configurazione di default degli iperparametri della rete.

2. Una seconda variante con un setting simile al setting di ReVeal (simile perché alcuni parametri definiti per la rete più complessa non sono configurabili per una MLP), quindi impostando:

   - La dimensione dei minibatch `batch_size` a 128.
   - Il numero massimo di iterazioni effettuate senza ottenere. miglioramenti `n_iter_no_change` al valore 5
   - L'iperparametro `alpha=0.5`

- Il numero di neuroni dell'hidden layer e il numero di hidden layer `hidden_layer_sizes=(256,1)`.
- Il numero massimo di iterazioni `max_iter=100`.
- L'iperparametro `early_stopping=true` per permettere l'arresto anticipato del modello.

3. Una terza variante con una configurazione personalizzata impostando:
   - La dimensione del minibatch `batch_size=200`.
   - Il numero massimo di iterazioni effettuate senza ottenere miglioramenti `n_iter_no_change=15`.
   - Il numero di neuroni dell'hidden layer e il numero di hidden layer `hidden_layer_sizes=(512,4)`.

Prima di creare ed addestrare le varianti suddette è stato addestrato il modello ReVeal fornito dai proprietari del progetto. L'addestramento del modello avviene mediante l'esecuzione del file `api_test.py`, di cui viene riportato di seguito il codice, eseguendo nel terminale il comando `python api_test.py -dataset chrome_debian/imbalanced -features ggnn;`

```
import argparse
import json
import numpy
import os
import sys
import torch
from representation_learning_api import RepresentationLearningModel
from sklearn.model_selection import train_test_split
from baseline_svm import SVMLearningAPI

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--dataset', default='chrome_debian/balanced',
                        choices=['chrome_debian/balanced',
                        ↪  'chrome_debian/imbalanced', 'chrome_debian',
                        ↪  'devign'])
    parser.add_argument('--features', default='ggnn', choices=['ggnn',
    ↪  'wo_ggnn'])
    parser.add_argument('--lambda1', default=0.5, type=float)
    parser.add_argument('--lambda2', default=0.001, type=float)
    parser.add_argument('--baseline', action='store_true')
    parser.add_argument('--baseline_balance', action='store_true')
    parser.add_argument('--baseline_model', default='svm')
```

```
    parser.add_argument('--num_layers', default=1, type=int)
    numpy.random.rand(1000)
    torch.manual_seed(1000)
    args = parser.parse_args()
    dataset = args.dataset
    feature_name = args.features
    parts = ['train', 'valid', 'test']
    if feature_name == 'ggnn':
        if dataset == 'chrome_debian/balanced':
            ds = '../../data/after_ggnn/chrome_debian/balance/v3/'
        elif dataset == 'chrome_debian/imbalanced':
            ds = '../../data/after_ggnn/chrome_debian/imbalance/v6/'
        elif dataset == 'devign':
            ds = '../../data/after_ggnn/devign/v6/'
        else:
            raise ValueError('Invalid Dataset')
    else:
        if dataset == 'chrome_debian':
            ds = '../../../../chrome_debian/full_graph/v1/graph_features/'
        elif dataset == 'devign':
            ds = '../../../../devign/full_graph/v1/graph_features/'
        else:
            raise ValueError('Invalid Dataset')
    assert isinstance(dataset, str)
    output_dir = 'results_test'
    if args.baseline:
        output_dir = 'baseline_' + args.baseline_model
        if args.baseline_balance:
            output_dir += '_balance'

    if not os.path.exists(output_dir):
        os.mkdir(output_dir)
    output_file_name = output_dir + '/' + dataset.replace('/', '_') + '-' +
    ↪  feature_name + '-'
    if args.lambda1 == 0:
        assert args.lambda2 == 0
        output_file_name += 'cross-entropy-only-layers-'+ str(args.num_layers) +
        ↪  '.tsv'
    else:
        output_file_name += 'triplet-loss-layers-'+ str(args.num_layers) +
        ↪  '.tsv'
    output_file = open(output_file_name, 'w')
    features = []
    targets = []
    for part in parts:
        json_data_file = open(ds + part + '_GGNNinput_graph.json')
        data = json.load(json_data_file)
        json_data_file.close()
        for d in data:
            features.append(d['graph_feature'])
            targets.append(d['target'])
```

```
        del data
    X = numpy.array(features)
    Y = numpy.array(targets)
    print('Dataset', X.shape, Y.shape, numpy.sum(Y), sep='\t', file=sys.stderr)
    print('=' * 100, file=sys.stderr, flush=True)
    for i in range(30):
        train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.2)
        print(train_X.shape, train_Y.shape, test_X.shape, test_Y.shape, sep='\t',
        ↪  file=sys.stderr, flush=True)
        if args.baseline:
            model = SVMLearningAPI(True, args.baseline_balance,
            ↪  model_type=args.baseline_model)
        else:
            model = RepresentationLearningModel(
                lambda1=args.lambda1, lambda2=args.lambda2, batch_size=128,
                ↪  print=True, max_patience=5, balance=True,
                num_layers=args.num_layers
            )
        model.train(train_X, train_Y)
        results = model.evaluate(test_X, test_Y)
        print(results['accuracy'], results['precision'], results['recall'],
        ↪  results['f1'], sep='\t', flush=True,
              file=output_file)
        print(results['accuracy'], results['precision'], results['recall'],
        ↪  results['f1'], sep='\t',
              file=sys.stderr, flush=True, end=('\n' + '=' * 100 + '\n'))
    output_file.close()
    pass
```

Dopo aver addestrato il modello ReVeal è stata definita un'altra classe PyThon `mlp.py` per poter implementare le varianti proposte in precedenza.

Variante 1

Per la prima variante come detto in precedenza viene definita una MLP lasciando invariata la configurazione di default degli iperparametri. Di seguito viene riportato il codice della classe sviluppata a tal scopo.

```
import sys
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix,
↪  ConfusionMatrixDisplay
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score, precision_score, recall_score,
↪  f1_score
from sklearn.neural_network import MLPClassifier
```

```
from imblearn.over_sampling import SMOTE
class MLPLearningAPI(BaseEstimator):
    def __init__(self,print,balance,model_type='mlp'):
        super(MLPLearningAPI,self).__init__()
        self.print=print
        self.balance=balance
        self.model_type=model_type
        if self.print:
            self.output_buffer = sys.stderr
        else:
            self.output_buffer = None
        pass

    def fit(self, train_x, train_y):
        self.train(train_x, train_y)

    def train(self, train_x, train_y):
        import warnings
        warnings.filterwarnings('ignore')
        self.model=MLPClassifier()
        if not self.balance:
            full_X, full_Y = train_x, train_y
        else:
            full_X, full_Y = self.rebalance(train_x, train_y)
        if self.output_buffer is not None:
            print('Fitting ' + self.model_type + ' model',
            ↪  file=self.output_buffer)
        self.model.fit(full_X, full_Y)
        if self.output_buffer is not None:
            print('Training Complete', file=self.output_buffer)

    def predict(self, test_x):
        if not hasattr(self, 'model'):
            raise ValueError('Cannot call predict or evaluate in untrained model.
            ↪  Train First!')
        return self.model.predict(test_x)

    def predict_proba(self, test_x):
        if not hasattr(self, 'model'):
            raise ValueError('Cannot call predict or evaluate in untrained model.
            ↪  Train First!')
        return self.model.predict_proba(test_x)

    def evaluate(self, text_x, test_y,counter):
        if not hasattr(self, 'model'):
            raise ValueError('Cannot call predict or evaluate in untrained model.
            ↪  Train First!')
        predictions = self.predict(text_x)
        return {
            'accuracy': accuracy_score(test_y, predictions)*100,
            'precision': precision_score(test_y, predictions)*100,
```

```
            'recall': recall_score(test_y, predictions)*100,
            'f1': f1_score(test_y, predictions)*100,
        }

    def score(self, text_x, test_y):
        if not hasattr(self, 'model'):
            raise ValueError('Cannot call predict or evaluate in untrained model. Train First!')
        scores = self.evaluate(text_x, test_y)
        return scores['f1']

    def rebalance(self, _x, _y):
        smote = SMOTE(random_state=1000)
        return smote.fit_resample(_x, _y)
        pass
```

Per addestrare tale modello viene eseguita la classe `api_test.py` modificata nel seguente modo:

```
...
for i in range(30):
        train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.2)
        print(train_X.shape, train_Y.shape, test_X.shape, test_Y.shape, sep='\t', file=sys.stderr, flush=True)
        if args.baseline:
            model = SVMLearningAPI(True, args.baseline_balance, model_type=args.baseline_model)
        else:
           model=MLPLearningAPI(print=True,balance=True)
...
```

Variante 2

Per la seconda variante, ovvero quella che definisce una MLP con un setting degli iperparametri simile a quello di ReVeal, è stata riutilizzata la classe `mlp.py` definita in precedenza per la prima variante ridefinendo solamente il costruttore in questo modo:

```
...
def __init__(self, print, balance, model_type='mlp', alpha=0.5, hidden_layer_sizes=(256,1), batch_size=64, max_iter=100, early_stopping=True, n_iter_no_change=20):
        super(MLPLearningAPI, self).__init__()
        self.print = print
        self.balance = balance
        self.model_type = model_type
```

```
        self.alpha=alpha
        self.hidden_layer_sizes=hidden_layer_sizes
        self.batch_size=batch_size
        self.max_iter=max_iter
        self.early_stopping=early_stopping
        self.n_iter_no_change=n_iter_no_change

        if self.print:
            self.output_buffer = sys.stderr
        else:
            self.output_buffer = None
        pass
...
```

Dopo aver definito il modello in tal modo esso può essere addestrato utilizzando la classe `api_test.py` modificata nel seguente modo:

```
...
for i in range(30):
        train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.2)
        print(train_X.shape, train_Y.shape, test_X.shape, test_Y.shape, sep='\t',
        ↪  file=sys.stderr, flush=True)
        if args.baseline:
            model = SVMLearningAPI(True, args.baseline_balance,
            ↪  model_type=args.baseline_model)
        else:
          model=MLPLearningAPI(print=True, balance=True, batch_size=128,
          ↪  n_iter_no_change=5)
...
```

Variante 3

Per l'ultima variante sviluppata è stata riutilizzata la classe `mlp.py` definita per la Variante 3.3 ed è solo stato modificata la classe `api_test` come riportato di seguito per poter addestrare il modello:

```
...
for i in range(30):
        train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.2)
        print(train_X.shape, train_Y.shape, test_X.shape, test_Y.shape, sep='\t',
        ↪  file=sys.stderr, flush=True)
        if args.baseline:
            model = SVMLearningAPI(True, args.baseline_balance,
            ↪  model_type=args.baseline_model)
        else:
```

```
8           model=MLPLearningAPI(print=True, balance=True, batch_size=200,
            ↪  n_iter_no_change=15, hidden_layer_sizes=(512,4))
9   ...
```

CAPITOLO 4

Risultati

Questo capitolo illustra i risultati ottenuti dall'esecuzione dei vari modelli ed il confronto tra le metriche di valutazione di tali modelli.

4.1 Risultati ottenuti

Nelle sezioni seguenti vengono riportati i risultati ottenuti dall'esecuzione delle diverse varianti del progetto.

4.1.1 Risultati Reveal

Dall'esecuzione del progetto ReVeal sono stati ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.3655	84.159	1.755	83.4159	85.1706
Precision	33.1793	32.601	3.67	31.3021	34.9722
Recall	69.5037	69.143	6.737	66.1083	72.8456
F1-score	44.7435	44.9	2.132	43.5233	54.6552

Tabella 4.1: Metriche ReVeal

Siccome l'addestramento del modello viene ripetuto per 30 volte, per evitare errori di predizione dovuti alla casualità, si è deciso di calcolare la media dei risultati ottenuti nelle suddette iterazioni. Tale media coinvolge le metriche del miglior modello ottenuto per la determinata iterazione. Inoltre questi risultati sono stati plottati anche graficamente attraverso l'utilizzo di un `boxplot()` della libreria PyThon `matplotlib`.

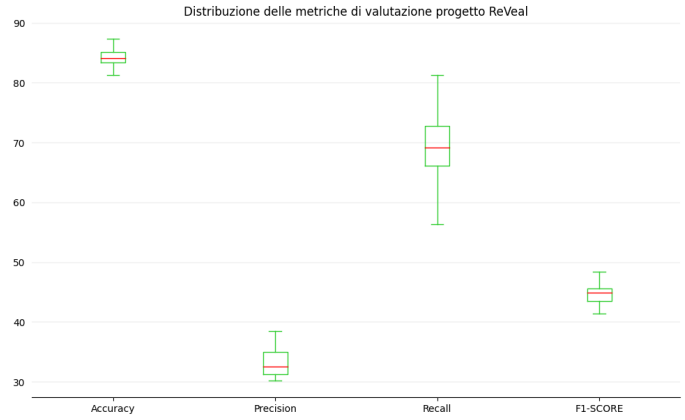

Figura 4.1: Plot delle metriche di ReVeal

La Figura 4.2 riporta la matrice di confusione ottenuta dal miglior modello considerando come metrica di valutazione la F1-score sulle 30 iterazioni.

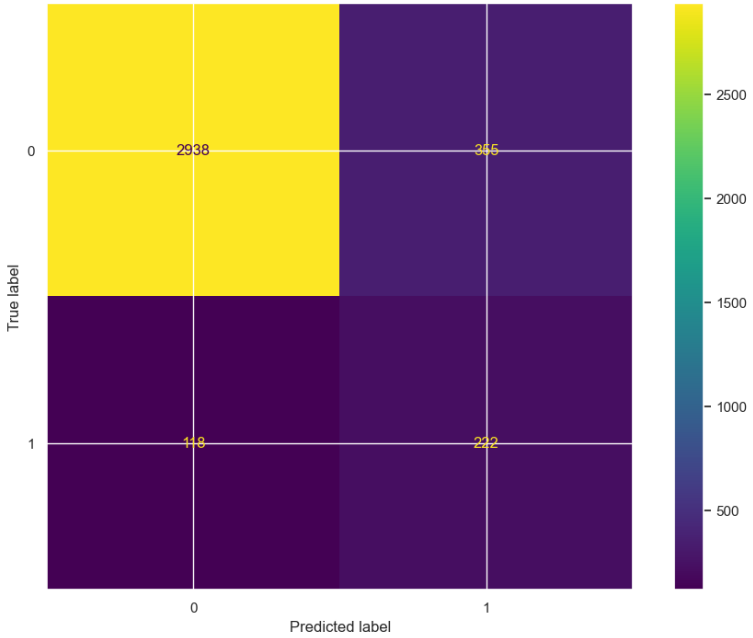

Figura 4.2: Matrice di confusione del miglior modello ottenuto per ReVeal

4.1.2 Risultati prima variante

Dall'esecuzione della prima variante, ovvero quella che implementa una MLP con il setting degli iperparametri di default sono stati ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	85.1330	85.003	0.743	84.7620	85.5049
Precision	26.30	26.464	1.935	25.6812	27.6165
Recall	35.0495	34.815	5.481	32.4161	37.8968
F1-score	29.9676	30.351	3.049	28.4404	31.4898

Tabella 4.2: Metriche prima variante

Plottando graficamente tali risultati si ottiene:

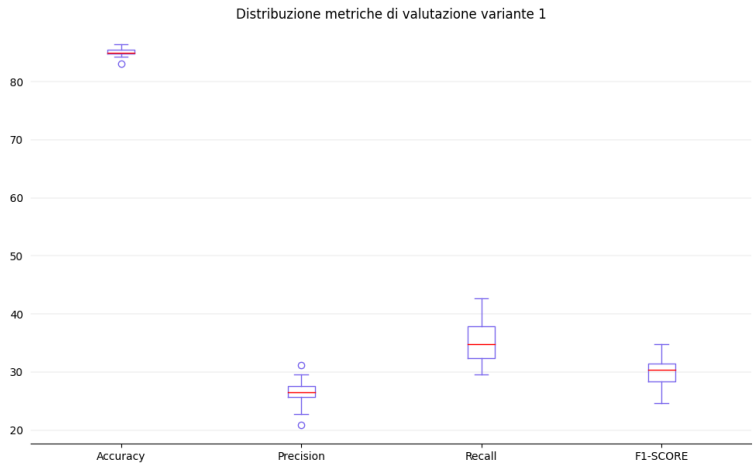

Figura 4.3: Plot metriche prima variante

La Figura 4.4 riporta la matrice di confusione ottenuta dal miglior modello considerando come metrica di valutazione la F1-score sulle 30 iterazioni.

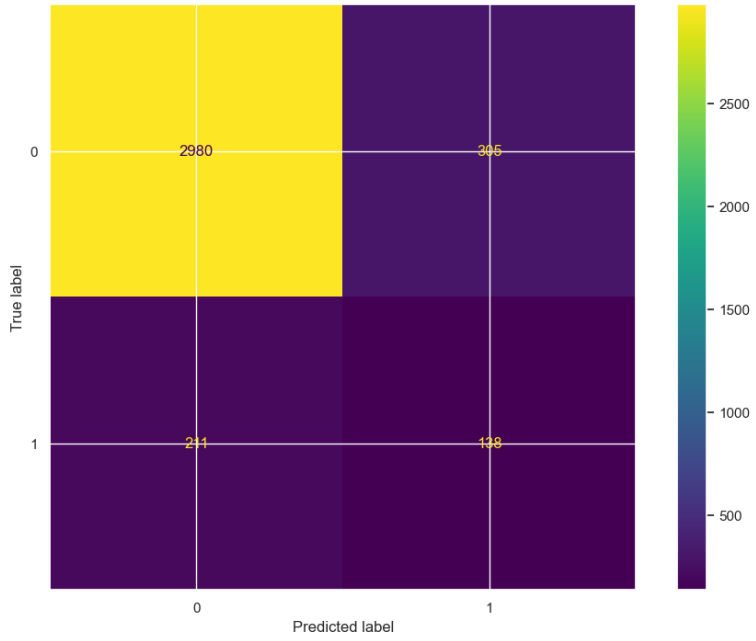

Figura 4.4: Matrice di confusione del miglior modello ottenuto per la prima variante

4.1.3 Risultati seconda variante

Dall'esecuzione della variante con una configurazione degli iperparametri della MLP simile a quella fornita per ReVeal sono stati ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	85.3247	85.237	1.218	84.7207	85.9383
Precision	26.8152	27.031	2.09	25.7302	27.8198
Recall	34.8038	35.215	2.868	33.6745	36.5427
F1-score	30.2396	30.282	1.742	29.4258	31.1676

Tabella 4.3: Metriche seconda variante

Plottando graficamente i risultati si ottiene:

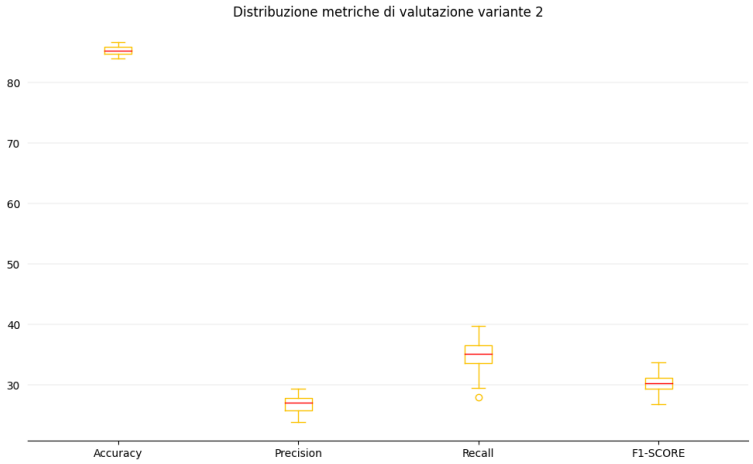

Figura 4.5: Plot metriche seconda variante

La Figura 4.6 riporta la matrice di confusione ottenuta dal miglior modello considerando come metrica di valutazione la F1-score sulle 30 iterazioni.

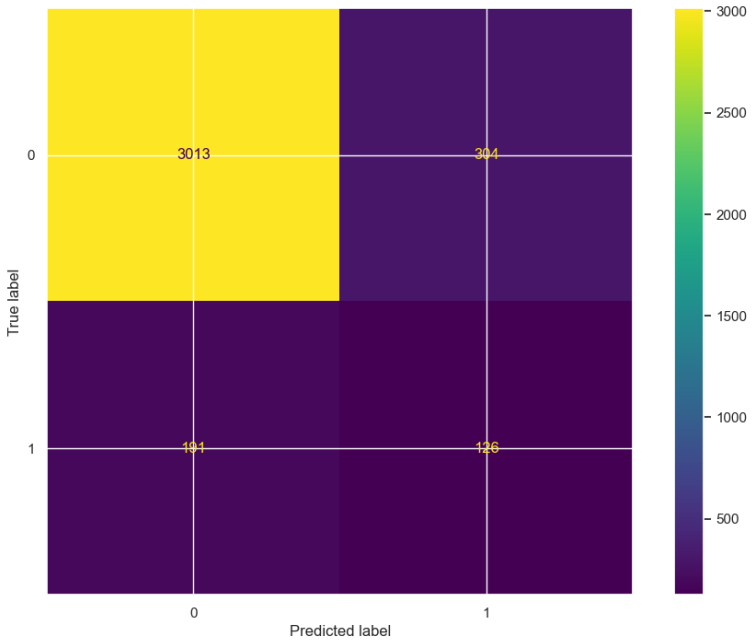

Figura 4.6: Matrice di confusione del miglior modello ottenuto per la seconda variante

4.1.4 Risultati terza variante

Dall'esecuzione dell'ultima variante proposta ovvero quella con una configurazione dei parametri personalizzata sono stati ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	85.2532	85.305	0.777	84.8583	85.6357
Precision	26.6531	26.661	1.96	25.6304	27.5900
Recall	36.4504	36.763	3.809	34.1993	38.0080
F1-score	30.7444	30.829	1.879	29.8943	31.7736

Tabella 4.4: Metriche terza variante

Plottando graficamente si ottiene:

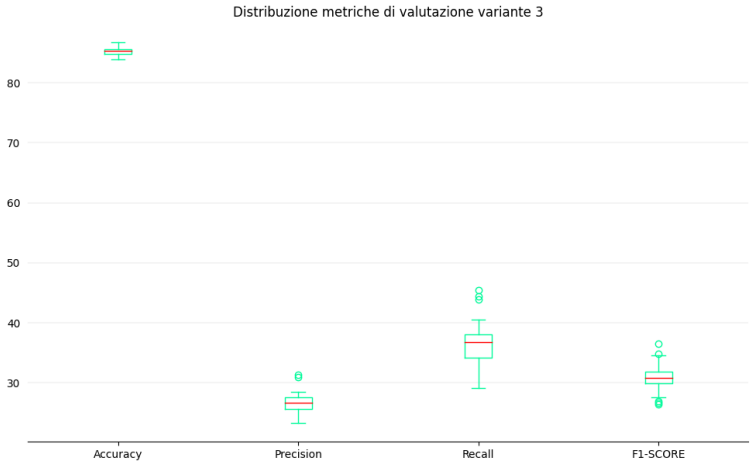

Figura 4.7: Plot metriche terza variante

La Figura 4.8 riporta la matrice di confusione ottenuta dal miglior modello considerando come metrica di valutazione la F1-score sulle 30 iterazioni.

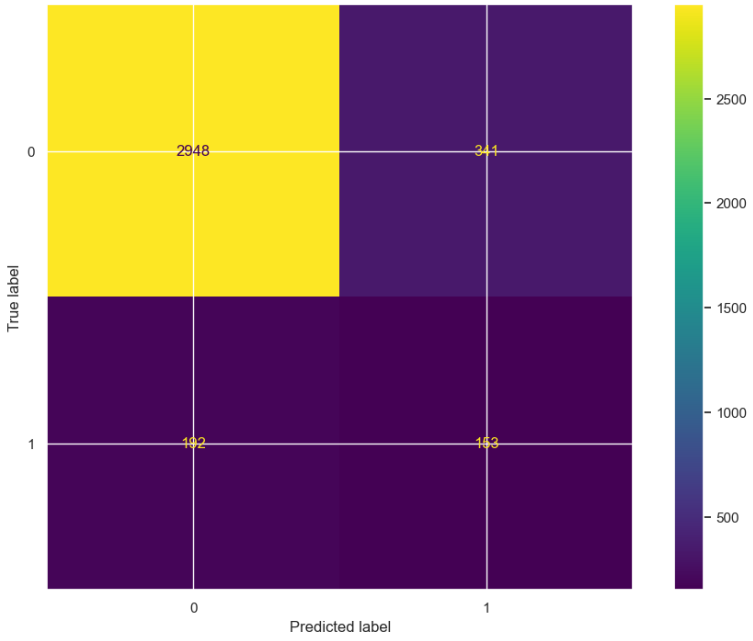

Figura 4.8: Matrice di confusione del miglior modello ottenuto per la terza variante

4.2 Confronto varianti

Dopo un'attenta analisi dei risultati ottenuti dall'esecuzione delle varianti del modello proposto, si possono osservare i seguenti risultati:

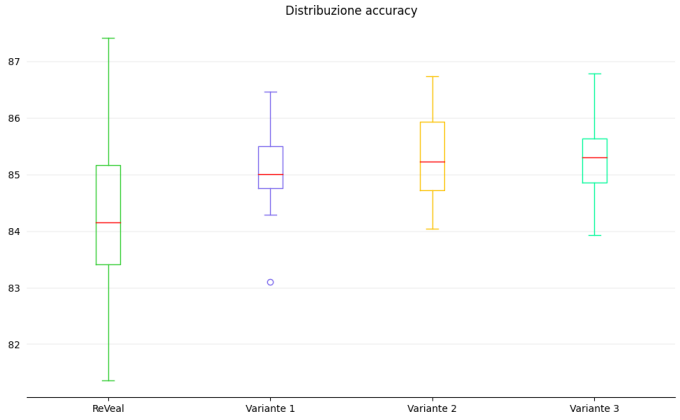

Figura 4.9: Confronto accuracy diverse implementazioni

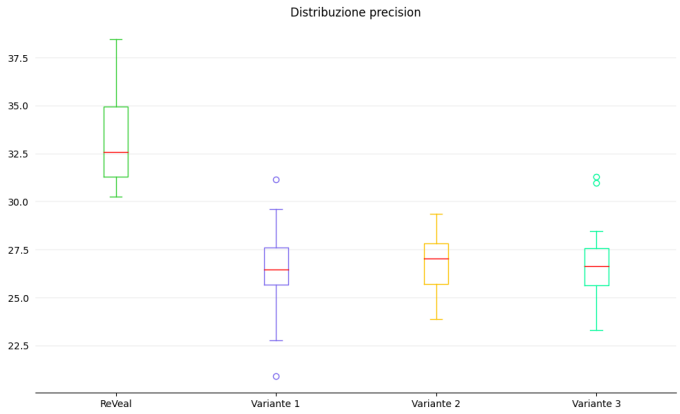

Figura 4.10: Confronto precision diverse implementazioni

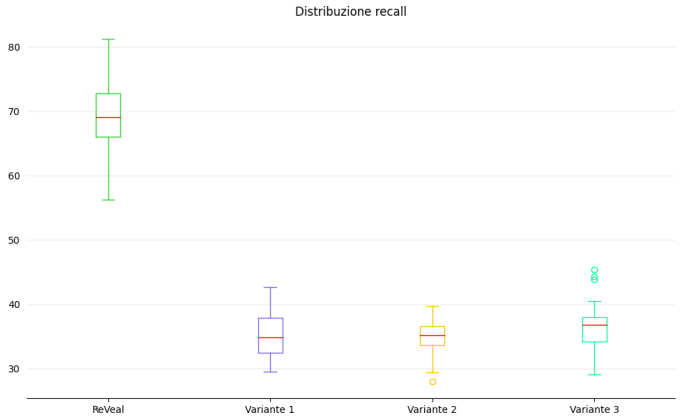

Figura 4.11: Confronto recall diverse implementazioni

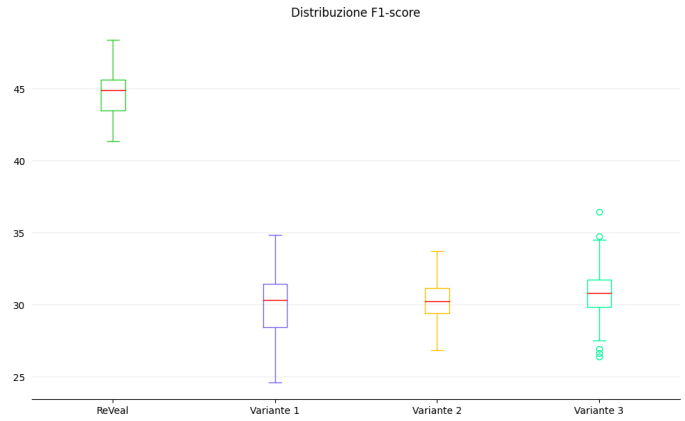

Figura 4.12: Confronto F1-score diverse implementazioni

Dalla Figura 4.9 si può evincere che i modelli proposti hanno un'accuracy più alta rispetto alla pipeline ReVeal, infatti per quanto riguarda ReVeal è stata ottenuta una mediana per l'accuracy sulle 30 iterazioni di 84.159 mentre per le varianti proposte questo valore è sugli 85, come si può osservare nelle Tabelle 4.1, 4.2, 4.3, 4.4. Però, non bisogna concentrare l'attenzione solo sull'accuracy, infatti come si può osservare dalle Figure 4.10, 4.11, 4.12, i modelli proposti presentano una precision, una recall ed un F1-score molto più basse rispetto a quanto ottenuto per ReVeal. Infatti si possono fare le seguenti considerazioni:

- Per quanto riguarda la precision in ReVeal è stato ottenuta una mediana di 32.601 mentre per le varianti proposte essa si aggira intorno ai 26-27, tali risultati sono esposti nelle Tabelle 4.1, 4.2, 4.3, 4.4. Questo indica che i modelli proposti hanno un alto tasso di falsi positivi come si può osservare dalle Figure 4.4, 4.6, 4.8. In questo caso però, come si può osservare dalla Figura 4.2, anche ReVeal ha ottenuto un alto tasso di falsi positivi, anzi, anche più alto rispetto alle varianti, però non bisogna soffermare l'attenzione sulla singola iterazione in quanto i valori riportati in precedenza fanno riferimento al complessivo delle 30 iterazioni.

- Per quanto riguarda la recall in ReVeal è stata ottenuta una mediana di 69.143, mentre, per le varianti proposte, questo valore si aggira intorno ai 34-36, come mostrato nelle Tabelle 4.1, 4.2, 4.3, 4.4. Questo evidenzia che i modelli proposti hanno un alto tasso di falsi negativi. Ciò vuol dire che tali modelli hanno una maggiore propensione a classificare come non vulnerabili le istanze anche se non lo sono. Infatti come si può notare dalle Figure 4.4, 4.6, 4.8 le varianti proposte hanno un tasso di falsi negativi più alto rispetto a quello di ReVeal riportato nella Figura 4.2

- Per quanto riguarda la F1-score per la pipeline ReVeal si è ottenuta una mediana di 44.9 mentre per le altre proposte questo valore si aggira intorno ai 30, questo perché la F1-score è ottenuta come la media armonica di precision e recall. Dato che sia precision che recall sono basse per le varianti proposte si ha anche una F1-score bassa.

Resoconto risultati ottenuti

Volendo fare un resoconto dei risultati ottenuti, si può concludere che, le varianti proposte migliorano solo in termini di accuracy rispetto al progetto ReVeal. Per quanto riguarda le altre metriche, ovvero precision, recall e F1-score, per le varianti proposte, esse assumono un valore inferiore rispetto a ReVeal. Questo implica che le varianti proposte peggiorano in termini di prestazioni ottenendo un alto tasso di falsi positivi e negativi, c'è da notare però, che anche per ReVeal si ottiene ul alto tasso di falsi positivi e negativi, però, tutto sommato, tale modello si comporta in modo migliore rispetto alle varianti proposte.

Ciò conferma quello che ci si aspettava di ottenere da tale studio, ovvero che, con architetture di rete neurale meno complesse dal punto di vista computazionale e più semplici da configurare si ottengono risultati sub-ottimali.

CAPITOLO 5

Conclusioni

Questo capitolo riporta le opportune considerazioni da fare dopo aver condotto uno studio empirico sulla generalizzabilità dei risultati di modelli di deep learning per l'identificazione di vulnerabilità

5.1 Valutazioni finali

Come è stato osservato nel capitolo precedente, le varianti proposte non forniscono buoni risultati. Infatti esse hanno solamente l'accuracy più alta rispetto alla pipeline ReVeal, ma per il resto delle metriche, sono stati ottenuti risultati nettamente inferiori rispetto a quelli ottenuti con ReVeal. Questo evidenzia che non conviene scegliere architetture meno complesse dal punto di vista computazionale e più semplici da configurare in quanto, la loro semplicità, porta ad avere modelli di predizione che sbagliano nella maggior parte dei casi e tali modelli non sono utilizzabili in un contesto reale.
C'è da notare però che neanche il progetto ReVeal è ottimale in quanto:

- Si è ottenuta una mediana della precision di 32.601, come riportato dalla Tabella 4.1, che è comunque bassa, questo implica che anche ReVeal ha un alto tasso di falsi positivi, infatti il miglior modello ottenuto ha predetto come vulnerabili 355 istanze che non lo sono, tale risultato è riportato nella Figura 4.2.

- Si è ottenuta una mediana della recall di 69.143, come riportato dalla Tabella 4.1, che non è male come risultato però può essere comunque migliorato. Infatti come si può notare dalla Figura 4.2 il numero si falsi negativi per il miglior modello sulle 30 iterazioni, scelto in base al valore dell'F1-score, ammonta a 118.

- Infine come riportato dalla Tabella 4.1, è stata ottenuta una mediana dell'F1-score di 44.9 che è abbastanza bassa, questo valore è influenzato dal basso valore di precision del modello.

In conclusione si può affermare che la pipeline ReVeal è sicuramente migliore rispetto alle varianti proposte, come previsto, però non e del tutto ottimale. Infatti è possibile apportare modifiche al modello per migliorarne le prestazioni, alcune di queste modifiche vengono riportate nella sezione successiva.

5.2 Sviluppi futuri

Per migliorare ulteriormente le performance della pipeline proposta in futuro si potrebbe optare di cambiare l'architettura di rete neurale dall'attuale representation learner ad una rete basata su transformer, che sono una delle architetture di rete più recenti. I transformer sono stati introdotti nel 2017 da un team di Google Brain e vengono sempre più utilizzati per i problemi di Natural Language Processing (NLP) sostituendo le Recurrent Neural Network

(RNN). Un **transformer** è un modello di machine learning che adotta il meccanismo del self-attention, assegnando un peso differente a ciascuna parte dei dati in input. Vengono utilizzati principalmente nel campo del NLP e della Computer Vision (CV). I transformer, come le RNN, sono progettati per elaborare dati sequenziali, però a differenza delle RNN elaborano l'intero input tutto in una volta. Il meccanismo dell'attention fornisce il contesto per qualsiasi posizione nella sequenza di input, cioè, se per esempio l'input è una frase del linguaggio naturale il transformer non ha la necessità di elaborare una parola per volta. Questo permette una maggiore parallelizzazione e quindi riduce il tempo di addestramento. La parallelizzazione aggiuntiva permette di effettuare l'addestramento su dataset di grandi dimensioni. Ciò ha portato allo sviluppo di sistemi pre-trained addestrati su dataset di grandi dimensioni come **BERT** (Bidirectional Encoder Representations for Transformers) e **GPT** (Generative Pre-trained Transformer) [45]. La versione più recente di GTP è la versione **GPT-3** introdotta nel maggio 2020 ma non è stata ancora rilasciata pubblicamente e quindi per ora viene ancora utilizzata la versione GTP-2.

Un'ulteriore miglioramento che potrebbe essere apportato in futoro è quello di estendere il modello a più linguaggi di programmazione, infatti attualmente il dataset a disposizione contiene solo vulnerabilità di codice *C*. É opportuno estendere il dataset con altri esempi di codice reale provenienti da altri progetti sviluppati in altri linguaggi di programmazione in modo da generalizzare i risultati ottenuti dal modello non al singolo linguaggio, come in questo caso, ma a più linguaggi di programmazione. Per fare ciò c'è bisogno di più progetti open source con rapporti di vulnerabilità pubblici che permettono l'acquisizione del codice vulnerabile e del codice fixato, come è avvenuto per i progetti open source presi in considerazione da ReVeal, per permettere la costruzione di un dataset più vasto che permetta al modello di predire anche vulnerabilità di altri linguaggi di programmazione.

Ringraziamenti

Chi mi conosce sa che sono di poche parole e che l'affetto sono abituato a dimostrarlo in altro modo, però, volevo sfruttare queste ultime pagine di tesi per ringraziare un po' di persone.
Innanzitutto ringrazio il mio relatore, il prof. Fabio Palomba per avermi guidato nella stesura di questa tesi e per avermi spronato a trovare soluzioni quando io una soluzione non la vedevo. Sempre disponibile e disposto ad infondere le sue conoscenze, grazie a lui ho capito cosa mi piace realmente fare e cosa voglio fare nella vita.
Ringrazio i miei genitori per aver reso possibile tutto ciò, perché senza di loro oggi non sarei qui. Grazie per essermi stati vicino, anche se a modo vostro, per avermi supportato e sopportato in questi tre anni e per non avermi fatto mancare mai niente, anche nei momenti un po' più bui.
Ringrazio mio fratello Vincenzo perché è grazie a lui che è nata la mia passione per l'informatica, ricordo quando da piccolo passavamo intere giornate vicino a quel vecchio pc a giocare a Ghotic, di quando un po' più grande lo guardavo, anche di domenica, progettare componenti di aerei. Ricordo però anche di quando studiava per i sui esami di ingegneria aerospaziale ed io gli rubavo la tavola periodica degli elementi perchè ero troppo affascinato da essa. P.S. Alla fine si sono rivelati più affascinanti i computer.
Ringrazio mio fratello Antonio per avermi sopportato in questi tre anni, per tutte quelle sere passate a studiare e ripetere fino a tardi per l'esame che avevo a breve.
Ringrazio Ferdinando che, anche se ci conosciamo da pochi anni (solo 3 alla fine) è come se ci conoscessimo da una vita, sempre pronto e disponibile a dare consigli e conforto nei momenti più difficili e a spronarmi quando qualcosa non va ed inizio a demoralizzami.
Ringrazio Rocco, compagno di viaggio con cui ho condiviso la maggior parte delle giornate passate a studiare, studiare e ancora studiare, a smadonnare su quel codice che non voleva

funzionare (e tu mi dicevi ci vediamo a dicembre), ma, a volte anche a ridere e cazzeggiare.
Ringrazio Alfonso, Sissi (anche se ad un certo punto ci ha abbandonati e lasciati in balia del nostro destino), ringrazio Francesco Peluso, Antonio Maddaloni, Antonio De Lucia, Luigi Sica, Fancesco Ciccone, Francesco Vece, Gerardo Di Pascale, Manuel di Matteo, Giacinto Adinolfi ed altri colleghi che adesso mi sfuggono per aver trascorso con me questi anni.
Ultimo ma non meno importante ringrazio i miei amici di uscite Rossella, Venere, Christian, Federica etc. per aver condiviso con me momenti di spensieratezza e divertimento dopo aver passato intere giornate a studiare.
Infine dedico questo traguardo anche a me stesso, pensando che 3 anni fa non contavo nemmeno di superare gli esami del primo semestre ed invece adesso sono qui dopo tre anni esatti a ripensare a quel momento mentre tiro le ultime somme della tesi.
P.S. più che un traguardo è solo un grande inizio.

Bibliografia

[1] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021. (Citato alle pagine 4, 16, 29 e 30)

[2] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017. (Citato a pagina 6)

[3] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?," *Journal of Systems and Software*, p. 111283, 2022. (Citato alle pagine 6 e 15)

[4] B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang, "Automatic classification for vulnerability based on machine learning," in *2013 IEEE International Conference on Information and Automation (ICIA)*, pp. 312–318, IEEE, 2013. (Citato a pagina 7)

[5] "National vulnerability database (nvd)." Disponibile al link : `https://www.nist.gov/programs-projects/national-vulnerability-database-nvd`. (Citato a pagina 7)

[6] "Cves and the nvd process." Disponibile al link : `https://nvd.nist.gov/general/cve-process`. (Citato a pagina 8)

[7] "The top 10 worst computer viruses in history." Disponibile al link : `https://www.hp.com/us-en/shop/tech-takes/top-ten-worst-computer-viruses-in-history`. (Citato alle pagine 8, 9 e 11)

[8] "Mydoom." Disponibile al link : `https://it.wikipedia.org/wiki/Mydoom`. (Citato a pagina 8)

[9] "Get a quick win in the battle against ransomware by disabling smbv1." Disponibile al link : `https://blog.netwrix.com/2021/11/30/what-is-smbv1-and-why-you-should-disable-it/`. (Citato a pagina 9)

[10] A. Tham, "What is code red worm," *Retrieved August*, vol. 31, p. 2016, 2001. (Citato a pagina 9)

[11] "Springshell (spring4shell) zero-day vulnerability cve-2022-22965 : All you need to know." Disponibile al link : `https://jfrog.com/blog/springshell-zero-day-vulnerability-all-you-need-to-know/`. (Citato alle pagine 12 e 14)

[12] "Zero-day (computing)." Disponibile al link : `https://en.wikipedia.org/wiki/Zero-day_(computing)`. (Citato a pagina 12)

[13] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 652–661, IEEE, 2013. (Citato a pagina 14)

[14] G. Tian-yang, S. Yin-Sheng, and F. You-yuan, "Research on software security testing," *International Journal of Computer and Information Engineering*, vol. 4, no. 9, pp. 1446–1450, 2010. (Citato a pagina 15)

[15] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 97–106, IEEE, 2011. (Citato a pagina 15)

[16] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 6–pp, IEEE, 2006. (Citato a pagina 15)

[17] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, pp. 343–350, IEEE, 2006. (Citato a pagina 15)

[18] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, 2018. (Citato a pagina 16)

[19] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 fourth international conference on multimedia information networking and security*, pp. 152–156, IEEE, 2012. (Citato a pagina 16)

[20] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150672–150684, 2020. (Citato a pagina 16)

[21] M. Jimenez, *Evaluating vulnerability prediction models*. PhD thesis, University of Luxembourg, Luxembourg, 2018. (Citato a pagina 16)

[22] J. Riquelme, R. Ruiz, D. Rodríguez, and J. Moreno, "Finding defective modules from highly unbalanced datasets," *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, vol. 2, no. 1, pp. 67–74, 2008. (Citato a pagina 17)

[23] M. Siavvas, D. Kehagias, and D. Tzovaras, "A preliminary study on the relationship among software metrics and specific vulnerability types," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 916–921, IEEE, 2017. (Citato a pagina 17)

[24] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2019. (Citato alle pagine 18, 22, 24, 25, 26, 27, 28 e 29)

[25] M. K. Thota, F. H. Shajin, P. Rajesh, *et al.*, "Survey on software defect prediction techniques," *International Journal of Applied Science and Engineering*, vol. 17, no. 4, pp. 331–344, 2020. (Citato alle pagine 19 e 20)

[26] "The concept of artificial neurons (perceptrons) in neural networks." Disponibile al link :`https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc`. (Citato a pagina 23)

[27] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *towards data science*, vol. 6, no. 12, pp. 310–316, 2017. (Citato a pagina 25)

[28] "Feedforward neural network: Its layers, functions, and importance." Disponibile al link :`https://www.analyticsvidhya.com/blog/2022/01/feedforward-neural-network-its-layers-functions-and-importance/`. (Citato a pagina 31)

[29] R. Hu, B. Tian, S. Yin, and S. Wei, "Efficient hardware architecture of softmax layer in deep neural network," in *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pp. 1–5, IEEE, 2018. (Citato a pagina 32)

[30] "What is the softmax function?." Disponibile al link : `https://deepai.org/machine-learning-glossary-and-terms/softmax-layer`. (Citato a pagina 32)

[31] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018. (Citato a pagina 34)

[32] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021. (Citato a pagina 34)

[33] "Understanding embedding layer in keras." Disponibile al link : `https://medium.com/analytics-vidhya/understanding-embedding-layer-in-keras-bbe3ff1327ce`. (Citato a pagina 37)

[34] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019. (Citato a pagina 37)

[35] "Word2vec." Disponibile al link : `https://it.wikipedia.org/wiki/Word2vec`. (Citato a pagina 37)

[36] "Control-flow graph." Disponibile al link : `https://www.sciencedirect.com/topics/computer-science/control-flow-graph`. (Citato a pagina 42)

[37] "Data flow graph." Disponibile al link : `https://www.sciencedirect.com/topics/computer-science/data-flow-graph`. (Citato a pagina 43)

[38] "Abstract syntax tree." Disponibile al link : `https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree`. (Citato a pagina 43)

[39] "Dependence graph." Disponibile al link : `https://www.sciencedirect.com/topics/computer-science/dependence-graph`. (Citato a pagina 44)

[40] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, IEEE, 2014. (Citato a pagina 44)

[41] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014. (Citato a pagina 45)

[42] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015. (Citato a pagina 45)

[43] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762, IEEE, 2018. (Citato a pagina 45)

[44] Y. Nie, A. S. Zamzam, and A. Brandt, "Resampling and data augmentation for short-term pv output prediction based on an imbalanced sky images dataset using convolutional neural networks," *Solar Energy*, vol. 224, pp. 341–354, 2021. (Citato a pagina 46)

[45] "Transformer (machine learning model)." Disponibile al link : `https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)`. (Citato a pagina 72)