UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Uno Studio Empirico sull'Impatto della Configurazione degli Hyper-parameter di modelli di Deep Learning per la Predizione di Vulnerabilità Software

RELATORE

Prof. **Fabio Palomba**

Università degli studi di Salerno

CANDIDATO

Rocco Iuliano

Matricola: 0512106804

Anno Accademico 2021-2022

"Dobbiamo avere il coraggio di seguire il nostro cuore e la nostra intuizione. In qualche modo, essi sanno che cosa vogliamo realmente diventare. Tutto il resto è secondario."

Steve Jobs

Sommario

La predisposizione ad essere attaccati affinché un agente esterno violi la nostra sicurezza, al giorno d'oggi è ormai consuetudine, infatti, nel campo del software, una grande quantità di dati e di richieste di determinati servizi che "viaggiano" in rete, da una parte all'altra dell'emisfero, molto spesso sono alterati o manipolati da utenti non autorizzati, i quali, con attacchi informatici, causano inevitabilmente il mal funzionamento dei servizi offerti da un determinato prodotto software. La causa principale di queste problematiche di sicurezza è senza dubbio da attribuire appunto alle vulnerabilità del prodotto software. In questa tesi, verrà considerato il problema delle vulnerabilità e ne verranno analizzate le varie tipologie per poi concentrarsi sulle vulnerabilità software. Dopodiché verranno analizzate le varie tecniche esistenti per l'analisi del codice che consentono l'individuazione delle vulnerabilità. In particolare, in questa tesi ci si focalizzerà sui modelli di deep learning (DL) che dato in input un codice, effettueranno una predizione indicando se il programma è vulnerabile o meno. Il progetto di DL analizzato è ReVeal nel quale vengono analizzati e confrontati gli attuali modelli di predizione delle vulnerabilità con il progetto in questione. Dopo un'attenta analisi di questo progetto, si andrà ad effettuare uno studio empirico sull'impatto che possono avere il settaggio degli hyper-parameter sulle performance dei modelli di deep learning per la predizione delle vulnerabilità.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

In questo capitolo si fornisce una panoramica del lavoro svolto in questa tesi.

1.1 Contesto applicativo

La predisposizione ad essere attaccati affinchè un agente esterno violi la nostra sicurezza, al giorno d'oggi è ormai consuetudine, infatti, nel campo del software, una grande quantità di dati e di richieste di determinati servizi che "viaggiano" in rete, da una parte all'altra dell'emisfero, molto spesso sono alterati o manipolati da utenti non autorizzati, i quali, con attacchi informatici, causano inevitabilmente il mal funzionamento dei servizi offerti da un determinato prodotto software. La causa principale di queste problematiche di sicurezza è senza dubbio da attribuire appunto alle vulnerabilità del prodotto software. A causa di queste vulnerabilità si è stimato che nel 2020, il costo annuale della criminalità informatica per l'economia globale è stato di 5,5 miliardi di euro, il doppio di quello del 2015 secondo le stime della Commissione europea [12]. Inoltre, l'*Associazione Italiana per la Sicurezza Informatica (Clusit)* ha effettuato un'analisi dei principali cyber attacchi noti a livello globale del periodo 2018-2020 e del primo semestre 2021, ed ha individuato 13014 attacchi di alta gravità che hanno causato danni economici, danni alla reputazione di aziende e persone e diffuso dati sensibili. Di questi attacchi, sono stati individuati 1874 nel 2020 e 1053 solo nel primo semestre 2021 e questo sta ad indicare come il tasso di attacchi informatici è in costante crescita, proprio come mostrato dal seguente grafico [1]:

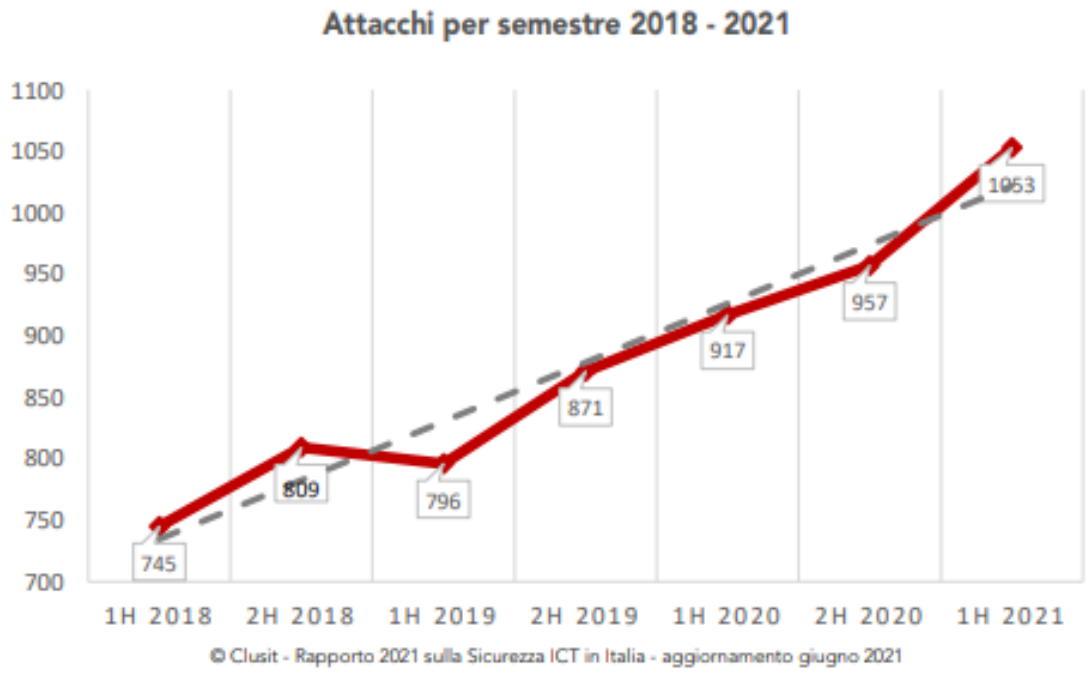

Figura 1.1: Analisi degli attacchi informatici per semestre nel periodo 2018-2020 effettuato da Clusit [1]

Inoltre, il rapporto "Second Annual Cost of Cyber Crime Study – Benchmark Study of U.S. Companies" pubblicato dal Ponemon Institute, mostra le tipologie di attacchi realizzati da parte degli hacker [2]:

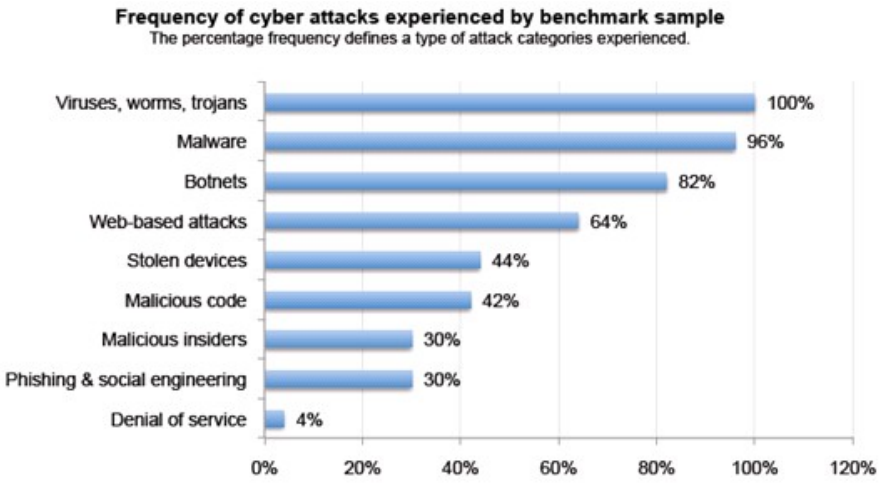

Figura 1.2: Tipologie di attacchi informatici [2]

Proprio per questi motivi c'è la necessità di rendere i sistemi informatici più sicuri e prevenire le vulnerabilità software in modo da ridurre i possibili attacchi informatici.

1.2 Motivazioni e Obiettivi

Come annunciato nella Sezione 1.1, c'è bisogno di prevenire i possibili attacchi informatici agendo sulla sicurezza dei prodotti software, ovvero realizzando codice privo di vulnerabilità. Per effettuare ciò, c'è bisogno di tool ad hoc per la loro individuazione ed attualmente esistono varie tecniche senza però risultati di rilievo. Proprio per questo, con l'ampia diffusione dell'intelligenza artificiale in vari settori e dalle sue ottime prestazioni in vari task, ultimamente ci si sta orientando al suo utilizzo in questo contesto. In particolar modo, nel campo della cyber security si sta diffondendo piano piano l'utilizzo di tecniche di deep learning (DL) che nell'ultimo decennio stanno riscontrando un enorme successo in vari settori. L'obiettivo di questa tesi è analizzare gli aspetti legati al deep learning ed effettuare uno studio empirico sull'impatto che possono avere gli hyper-parameter di una rete neurale nella predizione di vulnerabilità software.

1.3 Risultati

I risultati dello studio evidenziano come la modifica della configurazione dei parametri del modello di deep learning incidano sia sulla predizione che sulle performance. In particolar modo si ottengono lievi miglioramenti su alcune metriche di valutazione ma anche un inasprimento della loss function. In generale, sia il progetto ReVeal che le varianti proposte hanno un alto tasso di falsi positivi e bassa precision dovuti sia al problema che è di natura sbilanciato, sia alla tecnica utilizzata per il bilanciamento del dataset e/o dal non apprendimento delle feature che caratterizzano le istanze vulnerabili. In conclusione, è possibile migliorare le prestazioni del modello utilizzando tecniche di pre-processing dei dati più adatte ad analizzare il linguaggio di programmazione ed ampliando il dataset utilizzato per l'addestramento della rete neurale tramite dati del mondo reale.

1.4 Struttura della tesi

Nel Capitolo 2 vengono presentati i concetti relativi alle vulnerabilità, le varie tipologie ed i possibili rischi associati. Infine, si presenta lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nello studio dell'utilizzo di tecniche di deep learning per l'identificazione di vulnerabilità software. Nel Capitolo 3 viene analizzato il progetto ReVeal proposto dai ricercatori Chakraborty, Krishna, Ding e Ray illustrando i vari dettagli. Dopodichè ci si focalizza sull'impatto che possono avere gli hyper-parameter di un modello di deep learning in fase di predizione, proponendone uno studio empirico. Nel Capitolo 4 vengono riportati i risultati ottenuti dal progetto ReVeal e dalle varie varianti proposte, effettuandone anche un confronto. Infine, nel Capitolo 5 si conclude la tesi con una valutazione finale dello studio empirico effettuato e proponendo possibili sviluppi futuri per un miglioramento delle tecniche di deep learning per l'individuazione di vulnerabilità software.

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 di tecniche di deep learning per l'identificazione di vulnerabilità software.

2.1 Vulnerabilità dei prodotti software

La predisposizione ad essere attaccati affinchè un agente esterno violi la nostra sicurezza, al giorno d'oggi è ormai consuetudine, infatti, nel campo del software, una grande quantità di dati e di richieste di determinati servizi che "viaggiano" in rete, da una parte all'altra dell'emisfero, molto spesso sono alterati o manipolati da utenti non autorizzati, i quali, con attacchi informatici, causano inevitabilmente il mal funzionamento dei servizi offerti da un determinato prodotto software. La causa principale di queste problematiche di sicurezza è senza dubbio da attribuire appunto alle vulnerabilità del prodotto software. Delle suddette vulnerabilità ce ne sono diversi tipi:

1. **Vulnerabilità hardware**; quando una componente fisica della macchina in cui è ospitato il nostro applicativo non funziona correttamente e questo permette un attacco tramite un accesso fisico o remoto alla componente hardware in questione [13].

2. **Vulnerabilità software**; è un errore nella specifica, nello sviluppo o nella configurazione del software tale che la propria esecuzione comporta la violazione esplicita o implicita delle policy di sicurezza [14].

3. **Vulnerabilità ambientale**; riguarda l'area dove è collocata l'infrastruttura in cui è ospitato il nostro software ovvero, include tutti i possibili problemi ambientali di quella specifica area che possono danneggiare l'infrastruttura hardware [15].

4. **Vulnerabilità dei protocolli**; si verifica quando l'applicativo che stiamo utilizzando usa dei protocolli che non prevedono aspetti legati alla sicurezza.

Alcuni esempi di vulnerabilità:

1. **Vulnerabilità hardware**

    - **PACMAN**: è un attacco hardware progettato dai ricercatori dell' MIT nei confronti del processore M1 realizzato da Apple. Questo attacco sfrutta un bug software esistente e due vulnerabilità hardware, ovvero la vulnerabilità della corruzione di un'area di memoria e la vulnerabilità del canale laterale micro-architetturale. A partire dal bug software esistente, PACMAN può fornire supporto a un attacco software bypassando un importante meccanismo di sicurezza, ovvero il Pointer Authentication Code (PAC). PAC è una funzionalità di sicurezza che protegge l'integrità dei puntatori, ovvero aggiunge una firma crittografata ad essi. In questo

modo, quando i puntatori verranno utilizzati, si andrà a verificare il valore PAC e questo consentirà al sistema operativo di individuare e bloccare cambiamenti inaspettati che altrimenti potrebbero portare a perdite di dati o alla compromissione del sistema. Il bug software permette all'attaccante di poter compromettere il contenuto in una locazione di memoria che contiene informazioni importanti come il codice e i puntatori ai dati. La sua compromissione consente all'attaccante di poter modificare il flusso di controllo del programma ed eseguire codice malevolo. L'hacker, modificando la locazione di memoria, deve predire il valore PAC in modo che il sistema non blocchi l'esecuzione dell'attacco. L'attacco PACMAN si basa su due componenti: un oracolo PAC che consente di distinguere un valore corretto del PAC da uno errato su un puntatore scelto arbitrariamente e sul PACMAN gadget, ovvero una sequenza di codice che consente di eseguire due operazioni: verifica e trasmissione. L'attacco quindi, ha il seguente comportamento: utilizza l'area di memoria corrotta per poter effettuare ipotesi sul valore PAC, esegue il PACMAN gadget, ovvero effettua l'operazione di verifica che consiste nel controllare il valore PAC predetto e poi esegue l'operazione di trasmissione che consiste nel trasmettere il risultato di verifica tramite un canale laterale micro-architetturale in modo da poter essere a conoscenza del test effettuato [16].

2. **Vulnerabilità dei protocolli**

   - **Padding Oracle On Downgraded Legacy Encryption (POODLE)**: è un attacco pubblicato nell'ottobre del 2014 nei confronti del protocollo SSL che sfrutta due fattori: il primo riguarda alcuni server che per interoperare con sistemi legacy supportano SSL 3.0. Il secondo fattore è la vulnerabilità di SSL 3.0 che è legata ai byte di padding. L'attacco avviene durante la fase di handshake tra il client e il server, ovvero quando il client e il server concordano gli algoritmi e i protocolli supportati da entrambi e da utilizzare durante la comunicazione. In questa fase viene decisa anche la versione di SSL da utilizzare. La vulnerabilità di SSL 3.0 è la tecnica Cipher Block Chaining (CBC) in quanto, un messaggio per essere inviato, viene diviso in blocchi di lunghezza fissata e ognuno di essi verrà cifrato tramite l'algoritmo di cifratura concordato nella fase di handshake. Se il messaggio ha una lunghezza che non è multipla della lunghezza del blocco fissata, allora, nel suddividere il messaggio in blocchi, otterremo dei blocchi di dimensione più piccola rispetto a quella fissata e quindi sarà aggiunto del padding in modo da

raggiungere questa lunghezza. Quando i blocchi che compongono il messaggio giungeranno al server, esso non considererà il padding ma verificherà solo la lunghezza di quest'ultimo e il Message Authentication Code (MAC), quindi, non verificherà se il padding è stato modificato. L'attacco POODLE consiste nell'applicare la tecnica di man-in-the-middle nella fase di handshake, ovvero l'hacker si posiziona nel mezzo della comunicazione tra client e server intercettandone i messaggi. L'attaccante, quindi, svolgerà il ruolo del server in modo da forzare il client ad utilizzare la versione 3.0 di SSL, dopodichè sfrutta la vulnerabilità di CBC, ovvero: può decifrare un messaggio criptato inviato dal client modificando i byte di padding e verificando la risposta del server per questo messaggio modificato. In media dopo 256 richieste, il server accetterà il messaggio modificato e l'attaccante può decifrare il primo byte del messaggio. Dopodichè l'hacker procederà nel decifrare i byte successivi modificando nuovamente la dimensione del body e del percorso del messaggio ma shiftando ogni volta l'header. Grazie a questo attacco, l'hacker può scoprire dati sensibili contenuti nel messaggio inviato dal client al server senza la necessità di conoscere il tipo di cifrario o la password utilizzata [17] [18].

3. **Vulnerabilità ambientale**: il 20/07/2022 Google e Oracle sono stati costretti a chiudere i loro data center di Londra a causa delle alte temperature raggiunte. Questi grandi centri di calcolo contengono molti computer i quali generano molto calore e quindi richiedono un sistema di raffreddamento ad hoc in modo tale da non danneggiare le componenti hardware. La chiusura del data center per entrambe le aziende è dovuto ad un malfunzionamento delle unità di raffreddamento e dalle alte temperature raggiunte nella città, questo ha comportato la sospensione di alcuni servizi [19].

4. **Vulnerabilità software**

   - **SQL injection attack**: è una tecnica utilizzata per attaccare gli applicativi che interagiscono con un database relazionale utilizzando il linguaggio SQL. L'attacco consiste nell'iniettare una stringa SQL all'interno di un form di una pagina web o nell'URL e inoltrare la richiesta al server. Se lato server non vengono applicati opportuni controlli sull'input del client o adottate opportune tecniche di difesa, il server eseguirà la stringa SQL malevola, consentendo all'attaccante di poter accedere a dati sensibili e poterli modificare e/o cancellare in modo persistente sul database con il quale l'applicativo attaccato interagisce.

- **Morris worm**: è stato uno dei primi worm creato da uno studente del Cornell University, Robert Tappan Morris da cui deriva il nome del virus. Un worm è un malware che è in grado di replicare se stesso tramite la rete e installa una backdoor sul computer infettato in modo da poterne avere l'accesso e poter effettuare altre operazioni malevole. Questo worm fu creato per fini accademici ovvero per stabilire la dimensione di Internet, quindi quanti computer sono collegati alla rete e non per causare danni. Il malware si poteva replicare tramite 3 fattori:

  (a) la vulnerabilità del sistema di mail dei sistemi UNIX, ovvero la possibilità di poter accedere ad esso tramite una backdoor e senza richiedere l'autenticazione;

  (b) fingerd: è un programma che consente di trovare informazioni relative al computer e la sua vulnerabilità è l'overflow del buffer;

  (c) rexec: è un applicativo che permette di eseguire altri programmi su una macchina remota

  Non tutti i sistemi supportavano questi applicativi ma molti sostenevano SMTP, quindi il sistema di mail che era il veicolo principale con il quale il Morris worm riusciva a replicarsi. Prima di installarsi sul pc, il worm verificava se la macchina fosse già infetta ma una volta su 7 esso si installava lo stesso e questo portò all'esaurimento delle risorse del computer infetto fino a spegnersi. Inoltre, il malware tentava di bucare il meccanismo di autenticazione dell'account dell'utente cercando di indovinare la password, ovvero sfruttava: un dizionario interno di password; una lista locale di parole della macchina infetta (/usr/dict/words) e informazioni legate all'account dell'utente. Una volta che il Morris worm riusciva ad installarsi correttamente sul computer, cercava di ottenere sia gli indirizzi IP di altre macchine da dover infettare e sia gli IP dei seguenti host: i vicini fidati dell'amministratore o dell'utente, i vicini locali della macchina infettata selezionandoli casualmente e il gateway. Si è stimato che questo malware sia riuscito a contaminare circa 6000 computer e secondo la US Government Accountability Office (GAO) avrebbe causato danni per circa 10.000.000 di dollari [20].

- **ESET vulnerability**: ESET è una software house che produce vari prodotti software tra cui antivirus. Il 18 novembre 2021 ESET scoprì una potenziale vulnerabilità software che riguardava i suoi antivirus per Windows; permetteva all'attaccante di ottenere il livello di privilegio più alto, ovvero NT AUTHORITY\SYSTEM. L'attac-

cante deve possedere il seguente privilegio: SeImpersonatePrivilege che consente di utilizzare in modo improprio l'interfaccia di scansione antimalware di Windows (AMSI) in modo da poter ottenere il privilegio NT AUTHORITY\SYSTEM. AMSI è indipendente dal fornitore di antivirus e consente ai servizi e agli applicativi di potersi integrare con l'antimalware installato sul pc [21]. La difficoltà di questo attacco è possedere il privilegio di SeImpersonatePrivilege che è posseduto come default dagli utenti del gruppo Administrators locale. Se l'attaccante riesce ad ottenere questi permessi può realizzare attacchi molto pericolosi sulla macchina minacciata in quanto può svolgere qualsiasi azione. Questa vulnerabilità è riconosciuta come CVE-2021-37852 nel National Information Security Vulnerability Database ed è stata risolta grazie agli aggiornamenti rilasciati dalla stessa ESET [22].

- **Shellshock vulnerability**: La shell fu creata per il progetto GNU, usato nel sistema operativo UNIX e successivamente anche nei sistemi Linux, macOS e Windows. Questa vulnerabilità è classificata come CVE-2014-6271 nel National Information Security Vulnerability Database ed è presente in GNU Shell 4.3 e nelle versioni precedenti. Il bug esiste perchè la bash GNU elabora le stringhe dichiarate dopo la definizione di una funzione nel valore delle variabili d'ambiente. Questa problema riguarda molti server in quanto sono basati su sistemi UNIX, un hacker, quindi, tramite un tool di analisi può individuare sistemi soggetti a questa vulnerabilità e può far eseguire qualsiasi codice che egli voglia [23].
- **Emotet**: è un trojan malware che infetta i dispositivi tramite l'invio di una mail di phishing contenente in allegato un file o un link che consente l'installazione del malware stesso sulla macchina. Una mail di phishing è una mail in cui l'attaccante finge di essere un'organizzazione in modo da farla sembrare legittima ed ha l'intento di farsi fornire dall'utente credenziali o altre informazioni. Un trojan, invece; è un malware nascosto all'interno di un altro programma o file, apparentemente legittimo e quando l'utente esegue o installa il programma, attiva anche il trojan al suo interno che consente all'attaccante di avere il controllo remoto della macchina. Nel caso di Emotet, una volta che l'utente apre il file della mail o equivalentemente clicca il link, il malware inizia la propria installazione. Dopodichè Emotet scaricherà ed eseguirà il payload, ovvero le azioni che deve compiere il malware, dal server C2C (command-and control) e poi preleverà quante più informazioni possibili sulla vittima della macchina infetta ed invia queste

informazioni al server C2C. Tutte le operazioni eseguite dal malware utilizzano meccanismi di offuscamento in modo tale che la sua esecuzione venga nascosta all'utente e ai sistemi di antivirus. Quando il computer viene infettato, esso farà parte della botnet, ovvero una rete di pc infetti dal malware che vengono controllati da remoto dal botnet herders, ovvero l'hacker che sta gestendo l'attacco. Queste macchine vengono chiamate bot o zombie in quanto possono essere utilizzate dal botnet herders per far partire degli attacchi all'insaputa dell'utente che possiede il pc infetto. Emotet viene considerato un Mummy Spider in quanto grazie ad esso possono essere installati ed eseguiti sulla macchina infetta ulteriori malware. Un'altra caratteristica fondamentale di questo malware è il polimorfismo, ovvero esso può mutare ogni volta che si installa sulla macchina e questo rende ancora più difficile la sua individuazione da parte degli antivirus. Infatti, secondo gli esperti di G DATA, sono state registrate più di 33 mila varianti di Emotet solo nel 2019. Infine, esso per potersi replicare sulla rete non richiede un'azione da parte dell'uomo ma utilizza la lista dei contatti della mail della vittima attaccata e inoltra se stesso a quest'ultimi. In questo modo la mail non risulta come una mail spam in quanto proviene da una fonte legittima e questo aumenta la probabilità che anche il destinatario venga infettato. La US-CERT ha dichiarato che il malware Emotet è il malware più distruttivo e costoso, infatti l'azienda di cybersecurity CrowdStrike ha affermato che debellare il malware dalla macchina infetta costi un milione di dollari[24].

É fondamentale, quindi saper individuare la vulnerabilità, scoprirne la causa e successivamente risolverla. Sfortunatamente è difficile determinare quali tool e tecniche esistenti bisogna utilizzare per l'analisi del software e quando il loro utilizzo è appropriato [25].

2.2 Vulnerabilità software: ciclo di vita

Precedentemente abbiamo affermato che una vulnerabilità software è un pericoloso bug che consente all'attaccante di poter violare la confidenzialità, l'integrità o la disponibilità del sistema. Ogni vulnerabilità software ha un ciclo di vita, ovvero un insieme di stati distinti tra loro e caratterizzati dagli eventi relativi alla sua scoperta, divulgazione, sfruttamento e risoluzione ed ogni fase ha un certo livello di rischi associati. Analizziamo ogni fase del ciclo di vita:

1. **Scoperta** : questa fase inizia quando la vulnerabilità viene individuata dal fornitore del prodotto software, da un hacker o da analisti software di terze parti. Il rischio più alto in questa fase è se la vulnerabilità viene rilevata da un hacker.

2. **Divulgazione** : questa fase inizia quando la vulnerabilità viene resa pubblica a tutti e la divulgazione può essere effettuata dalle stesse entità della fase precedente. In questo caso i rischi sono più alti rispetto alla prima fase in quanto, un team di hacker può iniziare a elaborare un attacco che sfrutta la vulnerabilità in questione, realizzando così il zero-day attack. Questo attacco viene così denominato perchè gli sviluppatori hanno avuto zero giorni per poter risolvere la falla prima che qualcuno potesse sfruttarla.

3. **Sfruttamento o exploitation** : è la fase in cui gli hacker sfruttano la vulnerabilità per creare danni. Il tempo che intercorre tra la prima exploitation e quando il numero di sistemi vulnerabili si riduce fino a diventare insignificante è chiamato finestra di vulnerabilità.

4. **Risoluzione o patch** : questa fase inizia quando il fornitore del prodotto software rilascia un aggiornamento che risolve il bug in questione.

Il ciclo di vita della vulnerabilità termina quando tutti gli utenti avranno installato la patch rilasciata dal fornitore per risolvere il bug [26].

2.3 Vulnerabilità software: analisi

Le vulnerabilità software vengono memorizzate e rese note tramite il Common Vulnerabilities and Exposures (CVE) e il National Vulnerability Database (NVD). Si è notato che il numero di vulnerabilità divulgate tramite quest'ultimi, negli ultimi anni è aumentato a dismisura, infatti siamo passati da 4600 vulnerabilità nel 2010 a 8000 nel 2014 ed infine a più di 17000 nel 2017. É importante quindi, saper individuare le vulnerabilità prima che il software venga messo in commercio. Proprio per queste ragioni sono stati proposti vari metodi per l'individuazione delle vulnerabilità e sono:

1. **Analisi statica**: Un processo di valutazione del sistema, ovvero della sua struttura e del suo contenuto che non richiede alcuna esecuzione del codice. Questa tecnica, quindi, prevede l'analisi del codice in ordine di scoperta gli errori al proprio interno. Il problema di questa tipologia di analisi è che richiede molto tempo per essere applicata ed inoltre chi ispeziona il codice deve avere una buona conoscenza pregressa sull'individuazione

delle vulnerabilità. Per l'analisi statica sono stati sviluppati vari tool che consento di automatizzare parzialmente il processo di analisi, questo perchè è sempre richiesto l'intervento dell'uomo per valutare e verificare i risultati ottenuti [14].

2. **Analisi Fuzzing**: questa tecnica è ispirata al problema che un tempo colpiva le applicazioni modem, ovvero esse fallivano a causa di input casuali dovuti al rumore delle linee telefoniche. Così Miller nel 1990, propose questa nuova tecnica chiamata Fuzzing che permise di individuare e prevenire il problema, essa consisteva nell'inviare stream di caratteri randomici. Miller definì questa metodologia come una tecnica di testing randomico in quanto riusciva a coprire un gran numero di casi di boundary test perchè testava l'applicativo anche con dati non validi. L'analisi Fuzzing ha il seguente comportamento: genera prima i dati di input semi-validi, poi fornisce in input questi dati all'applicativo da testare ed infine osserva il suo comportamento [14].

3. **Analisi dinamica**: è una tecnica che richiede l'esecuzione dell'applicativo da testare. Per fare ciò essa richiede una test suite, ovvero un insieme di casi di test che devono essere eseguiti sull'applicativo in esame in modo da verificare se esso contiene bug. I tool di analisi dinamica, a fine test, producono un file (test incident report) contenete informazioni inerenti all'esito dell'esecuzione del codice testato per ogni caso di test, ovvero viene riportato un warning se è stato riscontrato un comportamento del programma non previsto o il chrash del programma [27].

Nella Tabella 2.1 sono stati riportati i vantaggi e gli svantaggi di ogni tecnica precedentemente illustrate:

Tecniche	Vantaggi	Svantaggi
Analisi statica	Non richiede l'esecuzione del codice; Permette di individuare molti errori nel codice prima che il software venga rilasciato; Facilmente integrabile nel modello di sviluppo del prodotto software	Il tester deve avere una buona conoscenza pregressa sull'individuazione delle vulnerabilità; I tool semi automatici hanno un alto tasso di falsi positivi perchè non comprendono la semantica del codice; Richiede l'intervento dell'uomo; Non riesce a individuare errori di configurazione o dovuti all'ambiente in cui è collocato il software; Non riesce a individuare errori di progettazione
Analisi dinamica	Produce un file contenente tutti gli errori riscontrati durante il test; Permette di testare l'applicativo su determinati input; Minor numero di falsi positivi e falsi negativi grazie all'esecuzione del codice	Richiede un alto tasso di test case in modo da avere un'alta confidenza nel rilevamento delle vulnerabilità; Richiede molto tempo dovuto all'esecuzione del software su tutti i casi di test specificati; Ha un alto tasso di falsi negativi
Fuzzing	É facile da comprendere e da applicare; É automatizzato; Non genera falsi positivi; Genera un gran numero di boundary test	Basso livello di generalizzazione in quanto gli input vengono generati casualmente; Alto tasso di falsi positivi

Tabella 2.1: Vantaggi e svantaggi delle tecniche di individuazione delle vulnerabilità

Un nuovo approccio per l'individuazione delle vulnerabilità sono i **Vulnerability Prediction Models (VPMs)** dove viene utilizzato il machine learning per predire se un determinato software è vulnerabile. Un software è un insieme di programmi e dati associati ed è quindi impossibile controllare tutte le sue componenti a causa di risorse limitate. I VPMs classificano

le componenti del software in due classi: vulnerabili e non vulnerabili e questo fornisce un ottimo supporto per i team di sviluppo in quanto devono verificare solo la presenza di vulnerabilità nei file etichettati come vulnerabili dal modello. Un Vulnerability Prediction Model viene addestrato su un **training set** che rappresenta un insieme di istanze che nel nostro caso rappresentano codice non vulnerabile e codice con vulnerabilità conosciute. I dati contenuti nel training set vengono raccolti da database pubblici come NVD o da repository pubblici delle software house che contengono le vulnerabilità per ogni componente software riscontrate in passato. In realtà, i dati non vengo collezionati all'interno del training set così come sono state memorizzate dalle fonti precedentemente citate ma viene effettuato un processo chiamato **feature selection** che consiste nell'estrazione delle feature del software in analisi, ovvero si estraggono le caratteristiche fondamentali che sono più correlate al problema in esame. Queste caratteristiche, quindi, andranno a caratterizzare le istanze del training set. Per poter individuare le vulnerabilità tramite i VPMs è fondamentale effettuare un'attenta fase di **feature selection** e scegliere accuratamente il modello di machine learning da utilizzare. Le istanze del training set, quindi, rappresentano una componente software che indicheremo con S_i e per ognuna di esse è associato un **vettore delle feature** che indichiamo con f_i e la **variabile dipendente** V_i. Il vettore f_i rappresenta l'insieme di **variabili indipendenti**, ovvero l'insieme delle feature che caratterizzano l'istanza, mentre V_i rappresenta l'etichetta che indica se questa istanza è vulnerabile o meno. Una volta che il modello è stato addestrato ed ha appreso le feature, esso potrà effettuerà predizioni su nuove istanze basandosi su ciò che ha appreso; ovviamente prima di poter utilizzare il modello, esso deve essere prima testato. Per fare ciò, viene utilizzato il **test set** che rappresenta un'insieme di istanze S_i che non sono presenti nel training set. Queste istanze, però, rispetto a quelle del dataset di addestramento, vengono fornite in input al modello omettendo la variabile dipendente V_i associata. In questo modo, dopo che il modello ha effettuato le predizioni su ogni istanza del test set, si verifica la correttezza di quest'ultime e si definiscono le metriche di valutazione del modello [3]. Le metriche di valutazione sono [5]:

- **Accuracy**: indica il numero totale di predizioni corrette:
$$\text{accuracy} = \frac{TP+TN}{TP+TN+FN+FP}$$
- **Precision**: indica il numero di volte in cui il modello ha predetto correttamente le istanze di una data classe che può assumere la variabile dipendente sul numero di predizioni corrette ed errate per quella classe:
$$\text{precision} = \frac{TP}{TP+FP}$$

- **Recall**: indica il numero di volte che il machine learner ha predetto in modo corretto le istanze di una data classe rispetto a tutte le istanze del dataset etichettate in quel modo:

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

Le variabili riportate nelle formule precedenti hanno il seguente significato:

- **TP** indica i "true positive", ovvero il numero di istanze di una data classe per il quale il machine learnear ha effettuato una predizione corretta;
- **TN** indica i "true negative", ovvero il numero di istanze per il quale il machine learnear ha correttamente non assegnato una data classe;
- **FP** indica i "false positive", ovvero il numero di istanze che il machine learnear ha assegnato erroneamente a una data classe;
- **FN** indica i "false negative", ovvero il numero di istanze di una data classe per il quale il modello ha stabilito erroneamente che non appartengono a quella stessa classe.

Esistono due tipologie di VPM:

1. **Vulnerability prediction model basati sulle metriche del software**: in questa tipologia di VPM, le feature delle istanze del dataset sono le metriche del software, ovvero le caratteristiche del software espresse in valori numerici come ad esempio la sua dimensione, espressa come numero di linee di codice o come numero di funzioni contenute in esso. Le metriche del software vengono create durante le varie fasi del ciclo di vita dello sviluppo del software e una volta che vengono definite, gli si associa la variabile dipendente con il corrispettivo valore (vulnerabile o non), creando così l'istanza all'interno del dataset che rappresenta il software in questione. Il processo di creazione dell'istanza nel dataset, il processo di addestramento e di test del modello è rappresentato nell'immagine seguente:

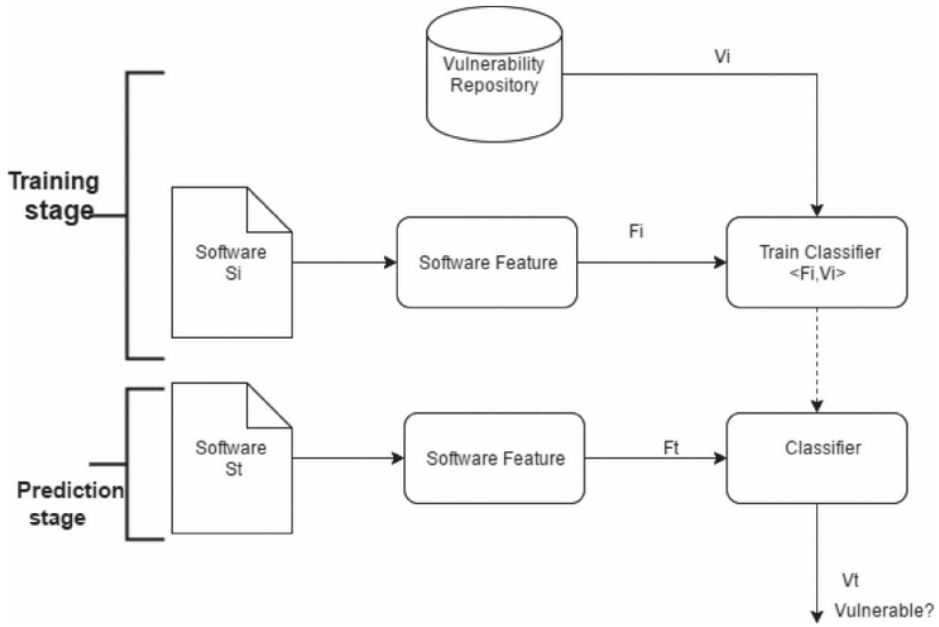

Figura 2.1: Creazione istanza con le metriche del software [3]

É fondamentale scegliere accuratamente le metriche del software, in quanto una scelta sbagliata potrebbe far si che le metriche non impattino sulla predizione effettuata dal modello. Esse, infatti, cambiano in base alle assunzioni effettuate:

- Dato che il codice viene modificato molte volte durante il proprio ciclo di sviluppo, questo aumenta la probabilità che contenga vulnerabilità. Basandosi, quindi, su queste caratteristiche, l'ipotesi effettuata è che i file vulnerabili hanno più codice rispetto ai file non vulnerabili, quindi le metriche utilizzate saranno: numero di cambiamenti effettuati sul file, numero totale di linee di codice aggiunte, numero totale di linee di codice modificate, numero totale di linee di codice recentemente aggiunte;
- Il software può essere realizzato da diversi team di sviluppo e una scarsa coesione tra i team o comunicazioni errate o scarsa gestione, può aumentare la probabilità di presenza di vulnerabilità nel software. Basandosi, quindi, su queste ipotesi, i file contenenti vulnerabilità hanno più probabilità di essere realizzati da vari team, quindi le metriche sono legate all'attività dello sviluppatore;
- **Grafo delle dipendenze del software**: questo grafo viene creato mettendo in relazione i vari elementi del software che sono le variabili, le funzioni, le classi ecc. Per poter individuare le relazioni tra questi elementi si utilizza principalmente

l'analisi statica del codice e la specifica di progettazione del software. Con questa tecnica, viene realizzato un grafo diretto, chiamato **grafo delle dipendenze dei membri (MDG)**. Esso è realizzato tramite le variabili e le funzioni utilizzate nelle componenti del software [3]. Come mostrato in Figura 2.2 il grafo è composto da: nodi rotondi che rappresentano le funzioni, nodi quadrati che rappresentano i dati, linee continue che rappresentano la chiamata dalla sorgente alla destinazione e da linee tratteggiate che rappresentano il punto di ritorno. Le varie linee, sono etichettate con il numero di dati trasferiti e il tipo di comunicazione (in-process e out-process) [4]. Dopodichè, a partire dal MDG, si realizza il **grafo delle dipendenze delle componenti (CDG)** andando a raggruppare tutti i nodi membri della stessa componente software in un unico nodo nel CDG. Infine si aggiungono ulteriori informazioni a questo grafo in modo che il modello possa poi effettuare le predizioni [3].

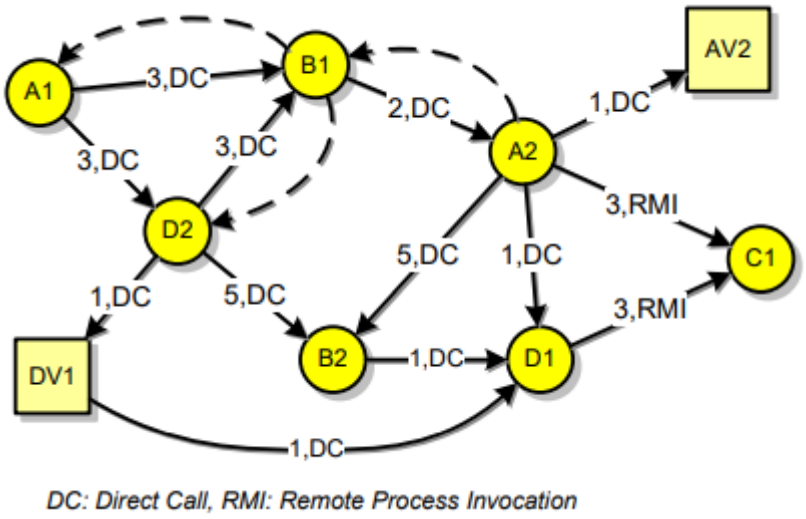

Figura 2.2: grafo delle dipendenze dei membri (MDG) [4]

2. **Vulnerability prediction model basati sull'analisi del testo**: questa tecnica utilizza il text mining o estrazione del testo, ovvero, si effettua un'analisi del codice del software e ogni sua componente è rappresentata da un insieme di termini individuati al suo interno e alle loro frequenze. Questi termini, vengono chiamati **token**. Per esempio, nel caso di un'applicazione Android, le sue componenti sono i file Java. Ogni file è rappresentato da un insieme di token e dalle loro corrispettive frequenze. Per effettuare questa conversione, si utilizza un metodo che sfrutta i delimitatori che possono essere ad esempio: operatori matematici e logici, lo spazio, la punteggiatura ecc. Nelle figure

di seguito riportate viene mostrato come avviene la conversione della componente software in un insieme di token:

```
/* The HelloWorldApp class prints "Hello World!" */
class HelloWorldApp {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Figura 2.3: Esempio file Java che stampa Hello Word [3]

.

```
args: 1, class: 2, Hello: 2, HelloWorldApp: 2,
main: 1, out: 1, println: 1, prints: 1, public: 1,
static: 1, String: 1, System: 1, The: 1, void: 1,
World: 2
```

Figura 2.4: Trasformazione file Java in token [3]

In questa tecnica, quindi, la variabile dipendente rappresenta sempre lo stato di vulnerabilità mentre le variabili indipendenti sono i token e la loro frequenza. Quando il modello viene testato, gli viene fornito in input un codice sorgente che verrà convertito in un insieme di token e le corrispettive frequenze, dopodichè il modello effettuerà la predizione. Il problema di questa tecnica è il grande numero di token del linguaggio di programmazione con cui è realizzato il software, infatti non tutti i token vengono utilizzati in un solo programma poichè utilizzandoli tutti, la realizzazione del modello risulterebbe più complessa. É importante, quindi, scegliere attentamente i token da utilizzare [3].

Nella seguente tabella vengono riportati i vantaggi e gli svantaggi di entrambe le tecniche:

VPM basate su	Vantaggi	Svantaggi
metriche del software	Le metriche possono essere estrapolate durante il ciclo di sviluppo del software	Le metriche del software scelte possono non impattare sulla predizione [3]; Bisogna effettuare delle ipotesi sulle quali poi verranno definite le metriche
text mining	Non c'è bisogno di effettuare nessuna assunzione inerente l'impatto che possono avere alcune feature sulle vulnerabilità software [28]; Le feature vengono estratte direttamente dal codice sorgente	Elevato numero di token [3]; L'apprendimento potrebbe non riuscire a creare feature significative [28];

Tabella 2.2: Vantaggi e svantaggi dei VPMs

Nella ricerca condotta da James Walden, Jeff Stuckman e Riccardo Scandariato [29], si evince, dal confronto tra le due tecniche che i VPMs basati su text mining rendo meglio rispetto ai VPMs basati sulle metriche del software.
Abbiamo quindi mostrato che le tradizionali tecniche per l'individuazione delle vulnerabilità sono soggette a un alto tasso di falsi negativi e falsi positivi. Recenti studi hanno mostrato grandi progressi in quest'ambito tramite l'utilizzo del Deep Learning (DP), specialmente nel settore della computer vision e del natural language processing (NLP) con un alto livello di accuracy, ovvero il 95% [11]. Prima di trattare questa nuova tecnica nella sezione successiva, introduciamo alcuni aspetti legati al Deep Learning.

2.4 Deep Learning

Il Deep learning (DP) è una branca del machine learning (ML) che si basa sulle reti neurali ed ha l'obiettivo di far simulare al computer il trasferimento di informazioni che avviene nel nostro cervello. Questa scienza, al giorno d'oggi, sta riscontrando un enorme successo in vari settori come: il campo medico, l'automazione industriale, settore automobilistico, il natural language processing (NLP) ecc. I motivi per i quali il deep learning ha riscontrato questo enorme successo sono dovuti:

- in parte dalla grande mole di dati contenuti nei dataset che diventano sempre più grandi e facilmente accessibili;
- dalle moderne architetture delle GPU che hanno consentito di ridurre notevolmente il tempo di esecuzione rispetto alle CPU;
- ed infine dallo sviluppo di piattaforme open source come TensorFlow che ha reso più semplice lo sviluppo di una rete neurale.

In realtà, il deep learning non è una nuova invenzione ma ben sì essa esiste sin dal 1940 [6]. Nel 1943, infatti, il neurofisiologo Warren McCulloch e il matematico Walter Pitts scrissero un articolo accademico intitolato "A Logical Calculus of Ideas Immanent in Nervous Activity", dove presentarono un semplice modello computazionale di come i neuroni biologici, all'interno di un cervello di un animale potevano interoperare per realizzare computazioni complesse usando proposizioni logiche. Questo modello fu la prima rete neurale realizzata. Purtroppo, la ricerca nel campo del deep learning è stata molto altalenante negli anni successivi a causa di vari fattori, come la mancanza di dati, la mancanza di potenza di calcolo ecc. Oggi però, grazie ai vari fattori precedentemente illustrati, il deep learning sta ottenendo grandi risultati [5]. Una rete neurale, quindi, cerca di simulare il nostro sistema nervoso che è composto da neuroni. La struttura di un neurone biologico è riportata nella Figura 2.5 ed è la seguente:

1. Un **nucleo** al centro del corpo cellulare;
2. Le fibre minori che si ramificano a partire dal nucleo si chiamano **dendriti**;
3. L'estensione più lunga è chiamata **assone**;
4. All'estremità dell'assone partono varie ramificazioni chiamati **terminali sinaptici** che sono collegate ai dendriti di un altro neurone. In questo modo i neuroni sono connessi tra loro [5].

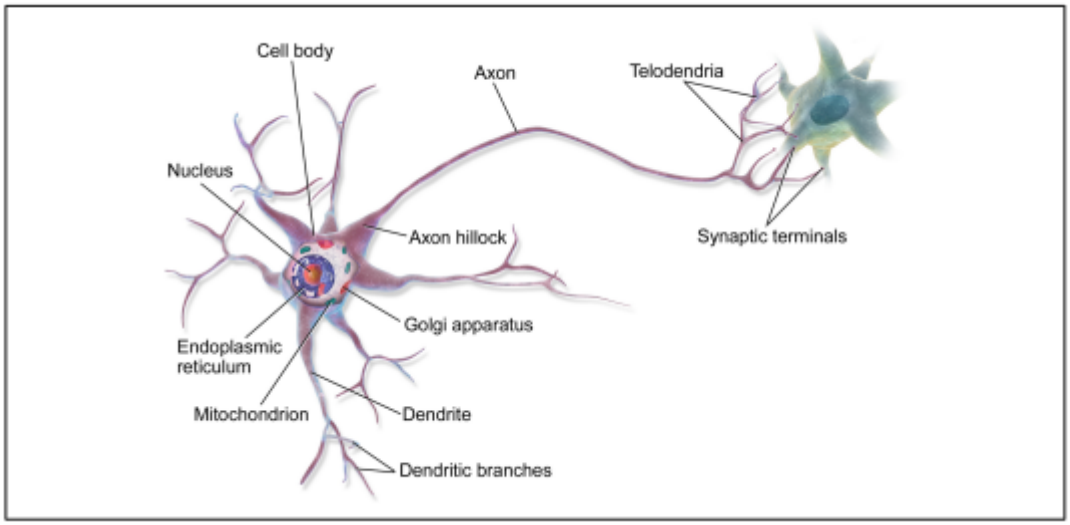

Figura 2.5: Struttura di un neurone biologico [5]

I neuroni interagiscono tra loro tramite l'invio di un impulso elettrico e quando un neurone riceve un certo numero di impulsi inizierà anch'esso a emettere segnali. All'interno del nostro cervello ci sono miliardi di neuroni e ognuno è collegato in media con altri migliaia di neuroni, formando così, una vasta rete di neuroni biologici. Questa rete, grazie a varie ricerche; è stata mappata come in Figura 2.6, mostrando che i neuroni sono organizzati in layer consecutivi [5].

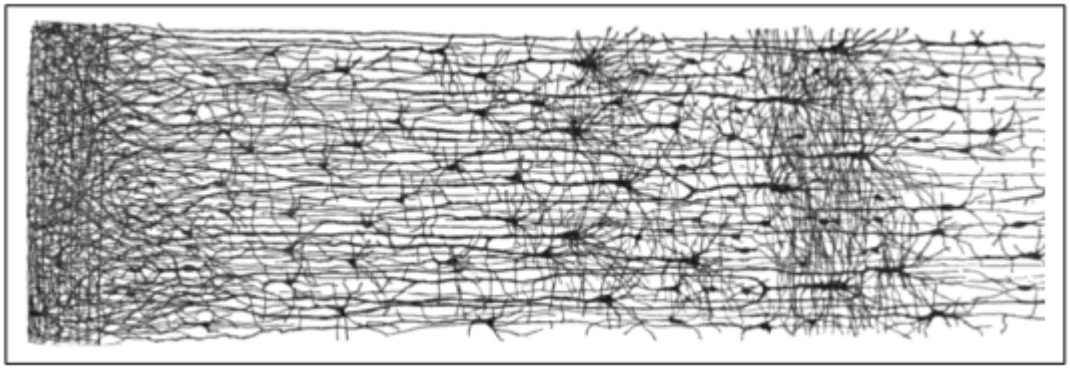

Figura 2.6: Rete neurale biologica [5]

La rete neurale artificiale proposta da McCulloch e Pitts cerca proprio di emulare questa struttura biologica e il comportamento dei neuroni; infatti, la rete è formata da nodi e archi che rappresentano rispettivamente i neuroni e gli assoni. I neuroni artificiali possono avere uno o più archi entranti che rappresentano l'input binario e un arco uscente che rappresenta l'output binario. Il neurone artificiale attiva il suo output quando un certo numero dei suoi input sono attivi. McCulloch e Pitts hanno mostrato che grazie a una rete di neuroni artificiali

è possibile computare una qualsiasi proposizione logica [5].

Per poter spiegare le varie caratteristiche di una rete neurale artificiale, introduciamo il **Percettrone**. Esso è una rete neurale realizzata nel 1957 da Frank Rosenblatt e si basa su un tipo di neurone artificiale chiamato **threshold logic unit (TLU)** oppure **linear threshold unit (LTU)**. Questi neuroni, rispetto a quelli della rete proposta da McCulloch e Pitts, hanno input (x_i) e output numerici e su ogni loro arco entrante è associato un peso (w_i). L'output di un TLU sarà l'applicazione della funzione di attivazione sulla somma pesata dei suoi archi entranti [5]. Questi neuroni vengono rappresentati come in Figura 2.7, mentre in Figura 2.8 viene mostrata l'analogia con il neurone biologico.

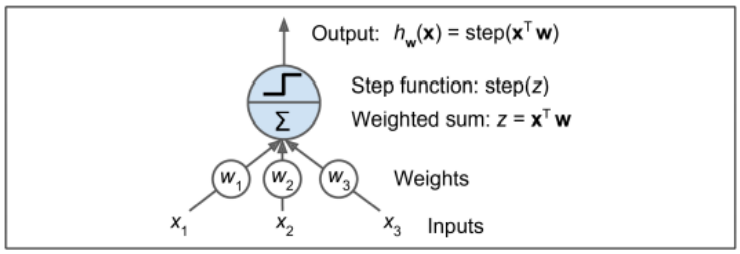

Figura 2.7: Threshold logic unit (TLU) [5]

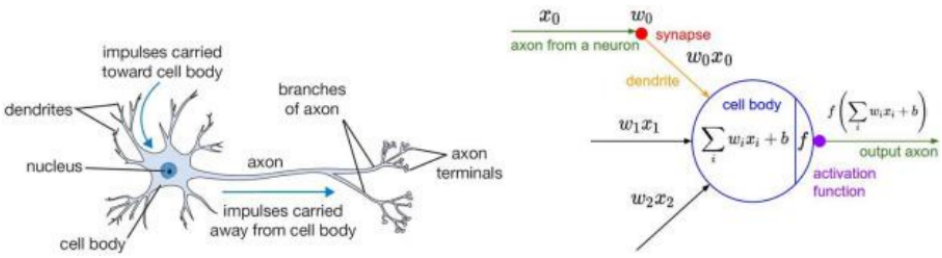

Figura 2.8: Analogia tra neurone biologico e neurone artificiale [6]

La rete neurale **Multi Layer Perceptron (MLP)**, quindi; è un grafo in cui:

- i nodi sono i neuroni TLU;
- gli archi rappresentano gli assoni che connettono i vari TLU e su ogni arco è associato un peso w_i;
- i neuroni sono disposti in layer verticali proprio per rispecchiare la rete biologica (Figura 2.6);
- i neuroni all'interno di un stesso layer non sono collegati tra loro ma solo con i neuroni del layer precedente e successivo;

- per ogni layer, tranne quello di output, è presente un neurone speciale chiamato **neurone di bias** che permette di introdurre una feature di bias, ovvero indica una soglia oltre la quale il neurone si attiva. In questo caso, l'output che esso genera è sempre pari a 1;
- il primo layer viene chiamato **input layer** in quanto contiene neuroni chiamati **neuroni di input** che consentono di inoltrare il valore a tutti i TLU. Infatti, questi neuroni inoltrano qualsiasi valore ricevuto;
- l'ultimo layer è chiamato **output layer** che fornisce la predizione della rete. In questo strato abbiamo un numero di neuroni pari al numero di valori che può assumere la variabile dipendente, questo perchè ognuno di essi rappresenta un valore che può assumere la variabile target;
- I layer che si trovano tra l'input layer e l'output layer vengono chiamati **hidden layer**. Quando una rete neurale ha un hidden layer composto da molti layer, viene chiamata **Deep Neural Network (DNN)**.

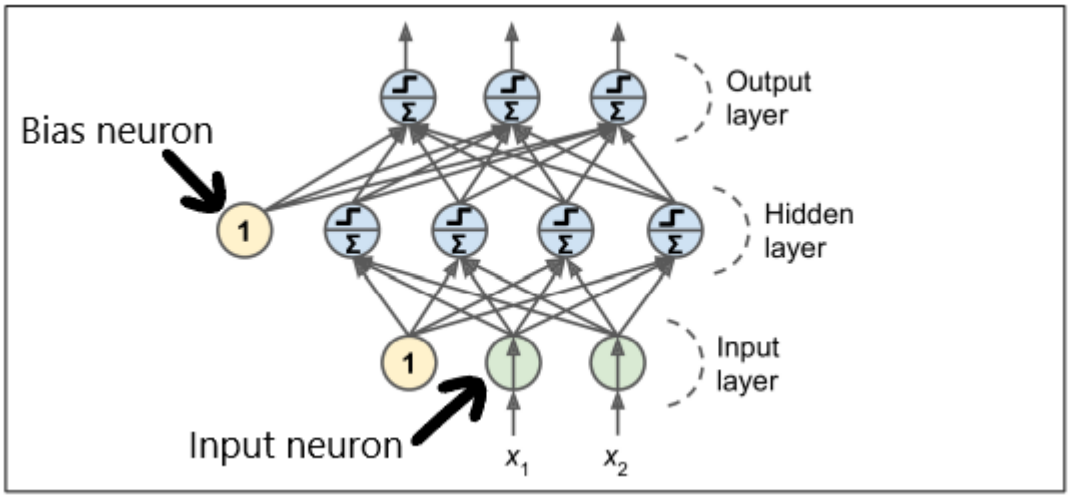

Figura 2.9: Multi-Layer Perceptron [5]

Ogni TLU di questa rete, applica alla somma pesata dei suoi input una funzione chiamata **funzione di attivazione**. Questa funzione, così come accade nei neuroni biologici, ha il compito di attivare o meno il neurone artificiale, ovvero converte la somma pesata calcolata dal TLU in un nuovo range di valori e questo sarà l'output del neurone. La tipologia di funzione di attivazione utilizzata nella rete MLP è la **step function** e più precisamente la Heaviside step function o la funzione segno (sign function) [5]:

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

Figura 2.10: funzioni di attivazione [5]

In termini matematici, le operazioni effettuate dai TLU all'interno di un layer x, possono essere viste come il prodotto tra una matrice e un vettore, dove:

- La **matrice** W è l'insieme dei pesi degli archi tra il layer x-1 e il layer x. Essa ha un numero di righe pari al numero di neuroni del layer x-1 e un numero di colonne pari al numero di neuroni del layer x;
- Il **vettore** X è l'insieme delle attivazioni dei nodi del layer x-1.

Questo prodotto fornisce in output un nuovo vettore che verrà sommato al vettore B il quale contiene i valori di bias. Dalla somma di questi due vettori otteniamo un altro nuovo vettore (Z) al quale verrà applicata la funzione di attivazione ϕ. Il risultato finale ottenuto, quindi, sarà l'attivazione di ogni neurone del layer x: $\phi(XW + B)$

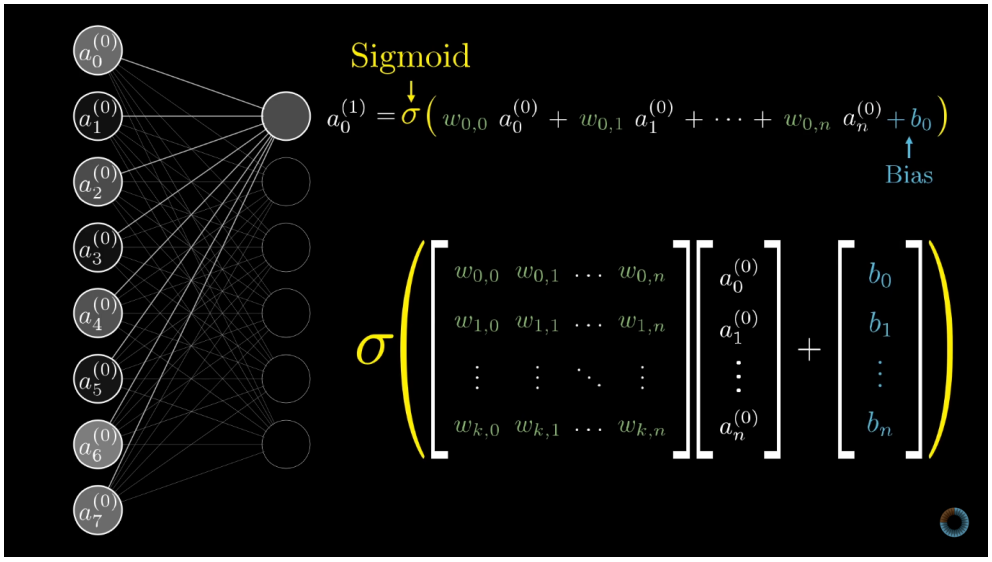

Figura 2.11: Operazioni eseguite all'interno di un layer della rete [7]

In questo modo, l'attivazione dei nodi appartenenti a un dato layer influenza l'attivazione dei nodi del layer successivo; proprio come avviene in una rete neurale biologica. Ma ora, come fa una rete neurale artificiale a fornire una predizione, data una certa istanza? Una volta data in input l'istanza alla rete, la sua predizione sarà il valore della variabile target associata

al neurone del layer di output che ha l'attivazione più alta rispetto agli altri neuroni. I valori prodotti dai nodi del layer di output indicano la probabilità che l'istanza presa in input dal modello faccia parte della classe associata a quel nodo.

2.5 Reti neurali: addestramento

[5] Una volta definita l'architettura di una rete neurale, per poterla addestrare per un determinato task, si inizializzano tutti i pesi degli archi con valori casuali, dopodichè si applica l'algoritmo di **Backpropagation**. Questo algoritmo è composto da due fasi:

1. **Fase di Forward**: In questa fase, ogni istanza del training set viene fornita in input alla rete neurale, ovvero, viene processata dai vari layer del modello fino ad arrivare all'ouput layer dal quale otterremo la predizione della rete. Sulla base degli esiti forniti dal modello, si definisce la **loss function** o **funzione di perdita** che confronta le predizioni effettuate dalla rete con i valori effettivi della variabile target; ovvero indica l'errore effettuato dal modello sul training set;

2. **Fase di Backward**: In questa fase, si calcola quanto ha contribuito all'errore ogni connessione a partire dall'output layer fino all'input layer. L'obiettivo è apprendere il gradiente della loss function rispetto agli attuali pesi della rete, sfruttando la **chain rule** del calcolo differenziale (ovvero la derivata composta). Questo gradiente verrà propagato all'indietro, fino all'input layer e successivamente sarà utilizzato per aggiornare i pesi del modello;

Durante l'addestramento, l'obiettivo principale è minimizzare la loss function e modificare i pesi delle connessioni tra i neuroni ed il bias, in modo da correggere la rete. Per fare ciò si utilizza la tecnica del **Gradient Descent** e nel caso del Multi Layer Perceptrons, l'aggiornamento della rete avviene tramite questa formula:

$$W_{i,j} = W_{i,j} + \eta(y_j - \hat{y}_j)X_i$$

- $W_{i,j}$ è il peso dell'arco tra il neurone i-esimo del layer x-1 e il neurone j-esimo del layer x;
- X_i è l'i-esimo valore di input dell'istanza di training erogato dal neurone del layer x-1;
- $\hat{y}_j$ rappresenta la predizione effettuata dal modello;
- y_j rappresenta il valore che ci aspettavamo di ottenere dal modello su questa istanza del training set, quindi il valore della variabile target;

- η rappresenta l'iperparametro chiamato **learning rate** o velocità di apprendimento che indica quanto velocemente il modello dovrebbe cambiare in base all'errore effettuato.

É importante che i pesi delle connessioni tra i vari hidden layer siano settati in modo randomico, altrimenti la rete non verrà addestrata correttamente. Infatti, se per esempio settiamo tutti i pesi e i bias a 0 otterremo dei neuroni identici e questo fa si che la backpropagation tratterà tutti i neuroni allo stesso modo e quindi rimarranno uguali. Dopo aver applicato l'algoritmo di backpropagation, si applica la tecnica di **Gradient Descent** che ha l'obiettivo di minimizzare la loss function modificando i parametri del modello (θ) tramite il gradiente come mostrato in Figura 2.12. Questi parametri, sono i pesi delle connessioni della rete neurale e il bias. Ogni step di questa tecnica viene chiamata **epoca** e il loro numero dipende dal learning rate. Il Gradient Descent, quindi, calcola come cambia la loss function modificando un parametro alla volta, ovvero calcola la derivata parziale della loss function in base a ogni parametro θ_j in θ.

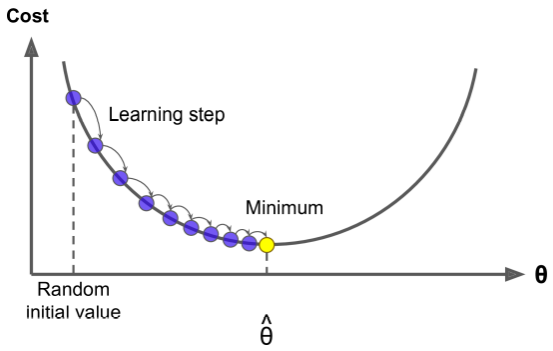

Figura 2.12: Rappresentazione grafica del Gradient Descent [5]

É fondamentale scegliere accuratamente il valore del learning rate, in quanto andrà ad impattare sulle fasi del Gradient Descent; infatti se assumesse un valore molto basso, l'algoritmo impiegherebbe più tempo nel far convergere la funzione. Allo stesso tempo, se il parametro assumesse un valore molto alto potrebbe far oscillare l'algoritmo da una parte all'altra della curva raggiungendo eventualmente un punto più alto rispetto al punto precedente, cioè ottenere un valore della funzione più alto rispetto a quello ottenuto nella precedente epoca. Questa osservazione è rappresentata in Figura 2.13 e 2.14.

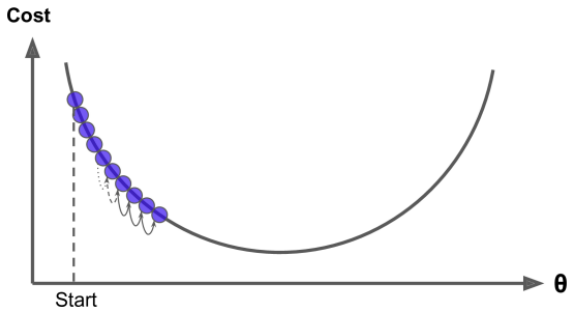

Figura 2.13: Gradient Descent con learning rate basso [5]

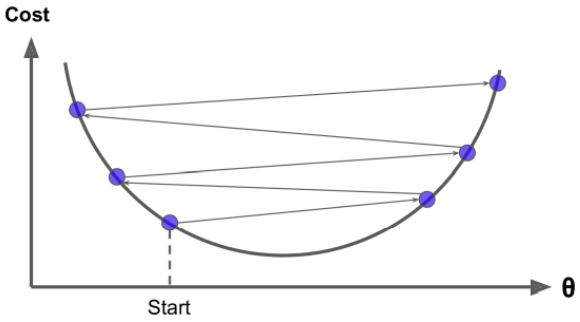

Figura 2.14: Gradient Descent con learning rate alto [5]

Ci sono varie tipologie di Gradient Descent e sono:

1. **Batch Gradient Descent**: Questa tecnica consiste nel fornire in input alla rete neurale l'intero training set, dopodichè viene calcolata la derivata parziale della loss function in base a ogni parametro θ_j in θ ad ogni epoca. Lo svantaggio di questa tecnica è che richiede una complessità temporale molto elevata sopratutto quando il dataset di addestramento è molto grande;

2. **Gradient Descent Stocastico**: In questa tecnica, viene scelta a ogni epoca, un'istanza casuale dal training set e quindi viene calcolato il gradiente solo sulla base di questa istanza. Rispetto al Batch Gradient Descent, questa tecnica è molto più veloce e riesce ad uscire da ottimi locali durante la ricerca del valore minimo della loss function. Il Gradient Descent stocastico, però, è meno regolare nel minimizzare la funzione, perchè nella ricerca del valore minimo, può "toccare" vari punti della curva in modo irregolare. Infine, l'algoritmo, una volta essersi avvicinato al minimo, continua la propria esecuzione e questo comporta una soluzione finale dei parametri non ottimale. Questo problema lo si può risolvere tramite la tecnica di Simulated Annealing;

3. **Mini-Batch Gradient Descent**: Con questa tecnica, a ogni epoca, si considera un piccolo insieme di istanze del training set scelte in modo randomico che viene chiamato mini-

bacth. Su questo insieme, viene calcolato il gradiente. Il Mini-Batch Gradient Descent rispetto alla tecnica precedente, riesce ad avvicinarsi di più al punto di minimo della loss function ma difficilmente riesce ad uscire da ottimi locali.

Durante l'addestramento si possono riscontrare due tipologie di errore:

- **Vanishing gradient**: Durante la fase di backward, il gradiente viene inoltrato dall'output layer verso l'input layer. Può capitare però che nell'inoltrare all'indietro il gradiente, esso può diventare sempre più piccolo e questo non permette al Gradient Descent di aggiornerà i pesi delle connessioni dei layer più vicini all'input layer;
- **Exploding gradient**: è l'esatto opposto del vanishing gradient, ovvero, il gradiente man mano che attraversa i vari layer cresce vertiginosamente e questo comporta una modifica anomala dei pesi delle connessioni durante il Gradient Descent.

Questi problemi sono causati da vari fattori come ad esempio la maggiore varianza dell'output di ogni layer rispetto alla varianza del suo input. Questo problema fa si che nell'attraversare i vari layer della rete, la varianza continua ad aumentare a tal punto che la funzione di attivazione si satura nei layer più prossimi all'output layer. Un'altra causa potrebbe essere la scelta della funzione di attivazione utilizzata nel modello. In generale, le reti neurali sono soggette a questo problema del gradiente instabile, in quanto i vari layer possono apprendere con velocità molto diverse.

CAPITOLO 3

Progetto ReVeal e studio empirico

In questo capitolo viene discusso ed analizzato il lavoro effettuato da Chakraborty, Krishna, Ding, Ray per l'individuazione delle vulnerabilità software tramite il deep learning ed infine si effettua uno studio empirico sull'impatto che possono avere gli hyper-parameter sulle prestazioni della rete neurale

3.1 Deep Learning based Vulnerability Detection: Are We There Yet?

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, Baishakhi Ray [11], nel loro articolo accademico, hanno prima analizzato, poi studiato e testato i quattro principali modelli esistenti di rete neurale per l'individuazione delle vulnerabilità software (DLVP DL-based Vulnerability Predictors), per poi proporne una soluzione. Questi modelli di deep learning hanno una granularità a livello di funzione, ovvero, dato in input un codice, essi individuano un'eventuale vulnerabilità all'interno di una funzione. Per poterli realizzare, vengono effettuate le seguenti fasi:

1. **Raccolta dei dati (data understending)**: viene collezionata una grande quantità di dati da poter utilizzare per la realizzazione del dataset che verrà poi diviso in due parti: training set e test set. Il training set verrà utilizzato per addestrare il modello ed il test set per valutare la bontà della rete neurale;

2. **Realizzazione del modello (data modeling)**: Viene selezionato e realizzato il modello più opportuno per il problema in esame, dopodichè si addestra il modello sul training set e si cerca di minimizzare la loss function;

3. **Valutazione (evaluation)**: si testa il modello sul test set e si definiscono le metriche di valutazione. Sulla base di quest'ultime, si stabilisce se la rete neurale è in linea con gli obiettivi di business.

Per verificare se questi modelli esistenti hanno buone prestazioni così come dichiarato, Chakraborty e gli altri, hanno realizzato un proprio dataset di vulnerabilità, dove le istanze rappresentano codice di reali progetti software, ovvero Chromium e Debian. Questi due progetti sono **open-source**, ovvero progetti il cui codice è reso disponibile a tutti affinchè possano essere studiati e modificati da persone che hanno competenze nel settore e tutto ciò ha lo scopo di migliorare il prodotto software. Realizzando un dataset, a partire da codice del mondo reale, si ha la possibilità di addestrare e testare la rete neurale in uno scenario più realistico. Le istanze collezionate da Ray e gli altri, sono state etichettate in due modi: vulnerabili o non vulnerabili. Nel loro articolo, hanno evidenziato che i modelli esistenti di deep learning preaddestrati, quando vengono testati su dati reali, hanno un grande calo di prestazioni rispetto a quella dichiarata, ovvero circa del 73%. Inoltre, Chakraborty e gli altri, hanno provato anche ad addestrare questi modelli sul loro dataset riscontrando comunque

un calo di prestazioni. Effettuando un'attenta analisi su questi approcci esistenti, sono stati rilevati vari problemi:

1. **Modelli inappropriati**: i modelli basati su token non considerano l'importanza delle dipendenze semantiche, mentre i modelli basati su grafi non prestano attenzione all'aumento della separazione tra classi vulnerabili e non;

2. **Apprendere feature non rilevanti**: si è evidenziato che i modelli attuali apprendono caratteristiche dei dati che non impattano sull'individuazione delle vulnerabilità. Questo è dovuto alla falsificazione del dataset utilizzato, in quanto sono state aggiunte feature artificiali;

3. **Istanze duplicate**: sia nel training set che nel test set utilizzati nella maggior parte dei modelli esistenti, ci sono dati duplicati e questo comporta la falsificazione dei risultati ottenuti;

4. **Dati sbilanciati**: nei modelli esistenti, non viene risolto il problema dello sbilanciamento tra le istanze appartenenti alle due classi che può assumere la variabile target. In questo modo, la rete neurale sarà in grado di individuare più facilmente la classe di maggioranza ma non la classe di minoranza.

In questo articolo [11], gli autori hanno prestato maggiore attenzione alla fase di collezione dei dati e alla progettazione del modello, sia per l'analisi degli approcci esistenti sia per la loro soluzione proposta. Nel primo caso, infatti, hanno classificato gli attuali dataset in base a: come sono stati realizzati i dati e come sono stati classificati ottenendo la seguente suddivisione:

- **Dataset artificiali**: contiene esempi di codice vulnerabile creati appositamente dall'uomo. Questi tipi di dataset sono stati creati per testare i modelli di analisi statica e dinamica. Proprio per questo motivo, i DLVP ereditano da queste due tecniche anche gli svantaggi, come per esempio un alto tasso di falsi positivi.

- **Dataset semi artificiali**: in questa categoria, la creazione delle istanze o la classificazione di quest'ultime viene effettuata dall'uomo. Un esempio è il dataset DRAPER che contiene codice open source etichettato tramite un'analisi statica;

- **Dataset reali**: in questa tipologia, le istanze e la loro classificazione provengono dal mondo reale.

Dopo aver definito il dataset da utilizzare, si procede con la scelta del modello, dove la sua selezione dipende anche dal tipo di informazioni che si vogliono utilizzare per il problema in esame. Successivamente alla scelta della rete neurale, viene applicato un pre-processing alle istanze del dataset in base al modello selezionato. I DLVP più utilizzati sono:

- **Modelli basati su token**: Per questa categoria, sono state applicate e confrontate diverse architetture di reti neurali e si è mostrato che la lunghezza della sequenza di token che rappresenta il codice (e quindi un'istanza), impatta notevolmente sulle prestazioni del modello, in quanto esso impiega molto tempo per poterla elaborare. Per poter risolvere questo problema, si effettua un'ulteriore pre-processing ai dati del dataset, ovvero, si estrapolano dal programma solo alcune porzioni di linee di codice, in quanto non tutte le istruzioni sono rilevanti per la predizione (ciò che si considera sono l'uso dei puntatori, chiamate a funzioni offerte da API, ecc...). Un'assunzione che viene effettuata in questa categoria di modelli è quella di considerare i token linearmente dipendenti tra loro e questo solo sulla base della loro dipendenza lessicale senza considerare la dipendenza semantica;

- **Modelli basati su grafi**: In questa tecnica, il codice è rappresentato tramite grafi in modo da poter rappresentare sia le dipendenze sintattiche che quelle semantiche. Si possono utilizzare varie tipologie di grafi, per esempio: Abstract Syntax Tree (AST) che rappresenta la composizione sintattica del codice e Code Property Graph (CPG) che è realizzato a partire dall'AST. I modelli basati su grafi, rispetto alla precedente tipologia, hanno un carico computazionale più elevato ed inoltre non hanno ottime prestazioni quando vengono utilizzati in ambienti con risorse vincolate.

Un problema che affligge entrambe le tecniche è il **vocabulary explosion**, ovvero, il numero di possibili identificatori nel codice possono essere infiniti ed il modello dovrà elaborarli tutti. Questi identificatori sono i nomi delle variabili, delle funzioni, dei puntatori ecc. Per poter risolvere questo problema, si rimpiazzano queste parole con nomi astratti, ad esempio le variabili diventano: "VAR1, VAR2, ecc..." e le funzioni diventano: "FUNC1, FUNC2, ecc". Come precedentemente illustrato, la fase di training ha l'obiettivo di minimizzare la loss function ed anche di far apprendere alla rete neurale le caratteristiche dei dati che consentono di massimizzare la seguente probabilità condizionata: $p(y|x)$, rispetto ai parametri del modello (θ). Cosa rappresentano le variabili utilizzate nella probabilità? La variabile x, rappresenta l'insieme di variabili indipendenti che caratterizzano l'input, mentre y, indica il valore che può assumere la variabile target. Questo rappresenta quindi la probabilità condizionata che

una data istanza, sia vulnerabile (y), sapendo che essa ha queste determinate caratteristiche (x) oppure che non è vulnerabile sapendo che ha queste feature. La probabilità è definita sulla base dei parametri θ del modello che vengono migliorati man mano durante la fase di addestramento (2.5), ovvero:

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

La funzione **argmax**, fornisce in output i valori da assegnare ai parametri θ del modello in modo da massimizzare la probabilità condizionata [5], D_{train} rappresenta il dataset di addestramento e θ^* rappresenta il valore ottimale da assegnare ai parametri del modello. La rete neurale, infatti, nell'effettuare la predizione, fornisce in output un valore $\hat{y}$ che indica la probabilità con la quale l'input appartiene a quella determinata classe; quindi, il modello deve apprendere quelle feature che impattano significativamente sulla probabilità stessa. Prima di addestrare la rete neurale, come precedentemente affermato, viene effettuato un pre-processing sui dati in base al modello utilizzato. In questo caso, l'istanza x^i viene trasformata in un vettore h^i composto da valori reali ($h^i \in R^n$ dove n è il numero di feature dell'input). L'input fornito alla rete neurale, quindi, non sarà il codice vero e proprio ma bensì il vettore h^i che lo rappresenta e che viene chiamato **embeddings**. Questa trasformazione da codice a vettore può essere applicata in vari modi:

- **embedding layer**: è un layer appartenente all'architettura della rete neurale che viene anch'esso addestrato per il problema in esame;
- **external word embedding tool**: sono applicativi esterni al modello di deep learning che consentono di trasformare il codice nel corrispettivo vettore. Un esempio è Word2Vec o Code2Vec.

In generale, i dataset utilizzati per i modelli di predizione di vulnerabilità software, presentano il problema di dati sbilanciati, ovvero il numero di istanze appartenenti alla classe di codice non vulnerabile rispetto al numero di istanze della classe vulnerabile è nettamente superiore. Addestrando, quindi, una rete neurale su un dataset che presenta questo tipo di problema, si ottiene un modello che è più bravo a riconoscere codice privo di vulnerabilità rispetto a codice vulnerabile. Tutto ciò è legato anche al problema in esame, in quanto esso è di natura sbilanciato.

Come precedentemente anticipato, dopo aver addestrato il modello sul training set, vi è la fase di valutazione, in cui si testa la rete sul test set e si definiscono le metriche di valutazione. C'è da notare che anche i dati del test set vengono sottoposti alla stessa fase di pre-processing

effettuata sui dati di addestramento. Chakraborty e gli altri, fanno osservare che le prestazioni ottenute dagli attuali modelli, vengono definite solo sulla base degli esiti ottenuti sul loro test set e questo non ci permette di ottenere informazioni su come si comporta il modello nel mondo reale. Infine, ci sono pochi studi, con i quali viene mostrato che i modelli realizzati riescono ad individuare vulnerabilità anche su istanze del mondo reale ma non fanno chiarezza sul tasso di falsi positivi e negativi ottenuti. Per poter realizzare un buon modello, capace di tracciare le vulnerabilità anche nella realtà, bisogna effettuare un'attenta analisi e collezione dei dati, ovvero realizzare un dataset che rispecchi il più possibile il mondo reale. Proprio per questo motivo, il dataset definito dagli autori dell'articolo accademico, contiene istanze di codice prelevato da due progetti software: Chromium e Linux Debian Kernel. In questo modo, hanno collezionato due tipologie di vulnerabilità, ovvero quelle legate al browser e quelle legate al sistema operativo. I file selezionati da questi due progetti, sono quei file vulnerabili per i quali è stata già pubblicata la corrispettiva risoluzione, detta anche **patch**. Per fare ciò, hanno selezionato tutte le patch etichettate tramite la parola sicurezza. Dato che il modello realizzato ha un livello di granularità a livello di funzione, le funzioni contenute all'interno di questi file vengono suddivise in due categorie:

- funzioni vulnerabili per le quali è stata realizzata la patch;
- funzioni prive di bug e che quindi, non sono state modificate dalla patch.

Per quanto riguarda la prima categoria, è stata memorizzata sia la versione vulnerabile che la versione corretta, memorizzando anche i cambiamenti inerenti ai file header, ovvero le librerie utilizzate nel codice. La versione contenente bug è stata etichettata all'interno del dataset come vulnerabile, mentre la versione corretta è stata etichettata come non vulnerabile. Infine, anche la seconda categoria è stata etichettata come non vulnerabile e viene riportata due volte all'interno del dataset. Nell'articolo, gli autori, hanno illustrato il loro progetto ReVeaL che ha consentito di individuare vulnerabilità software nella realtà. In questo progetto, durante il pre-processing dei dati, è stata applicata la tecnica di **representation learning** che consente di individuare la rappresentazione dell'input necessaria affinchè si abbia una netta e migliore separazione tra le classi che può assumere la variabile target. Questa tecnica, inoltre, permette di evitare l'applicazione del **feature engineering**, ovvero la fase in cui si selezionano le caratteristiche del problema più rilevanti per la predizione. Questo progetto, si suddivide in due fasi:

1. **Feature extraction**: In questa fase, si trasforma il codice in un grafo, ovvero il **Code Property Graph (CPG)** che consente di rappresentare sia le informazioni sintattiche

che semantiche. Questo grafo viene realizzato sulla base del **control flow graph**, del **data-flow graph**, del **dependency graph** e dall'AST. Il primo, rappresenta il flusso di controllo, cioè i cammini di esecuzione che possono essere intrapresi dal programma durante la sua esecuzione. Il secondo, invece, rappresenta il flusso dei dati all'interno del programma, ovvero illustra da dove provengono i dati e dove vengono utilizzati. [8] Il CPG, come mostrato in Figura 3.1, è un grafo diretto e unisce queste quattro rappresentazioni del codice in un'unica rappresentazione a grafo, dove i nodi rappresentano il codice mentre gli archi rappresentano le relazioni tra i pezzi di codice. Ogni nodo può contenere varie informazioni come: id, tipo di istruzione, posizione dell'istruzione nel codice, ecc. Gli archi sono etichettati con il tipo di relazione come: USE, VARIABILITY, ecc. Il CPG rappresenta:

- L'AST del codice tramite nodi quadrati che sono collegati tra loro tramite gli archi grigi che rappresentano la relazione padre-figlio;
- I quadrati blu rappresentano le istruzioni del codice;
- Il nodo grigio ENTRY ed il nodo nero EXIT rappresentano il punto di inizio e fine del codice che è rappresentano nel control flow graph;
- Gli archi di colore nero rappresentano gli archi del control flow graph;
- I nodi verdi rotondi indicano il flusso dei dati. Essi possono avere due tipi di archi entranti: arco etichettato con **USE**, quando i dati che rappresentano vengono utilizzati oppure un arco etichettato con **DEF**, quando i dati che rappresentano vengono modificati;

```
1  /* File : aSndr.c */
2  #include "drvUtls.h"
3  #include "valCmp.h"
4  #include "Sndr.h"
5
6  #ifdef analogueSender
7      int sendASignal(int *signal) {
8          int var = utilFunction(&signal);
9          doSomethingImportant(var);
10         return var;
11     }
12 #endif
```

```
42  /* File : drvUtls.c */
43  int utilFunction(int *i) {
44      /* some useful utilities */
45      return i;
46  }
```

```
538 /* File : cnvrt.c */
539 #ifdef analogueSender
540 int convertASignal(int *signal) {
541     /* some converting is done */
542     return signal;
543 }
544 #endif
```

```
698 /* File : valCmp.h */
699 #ifdef otherFeature
700     #define doSomethingImportant(var) var*2
701 #endif
```

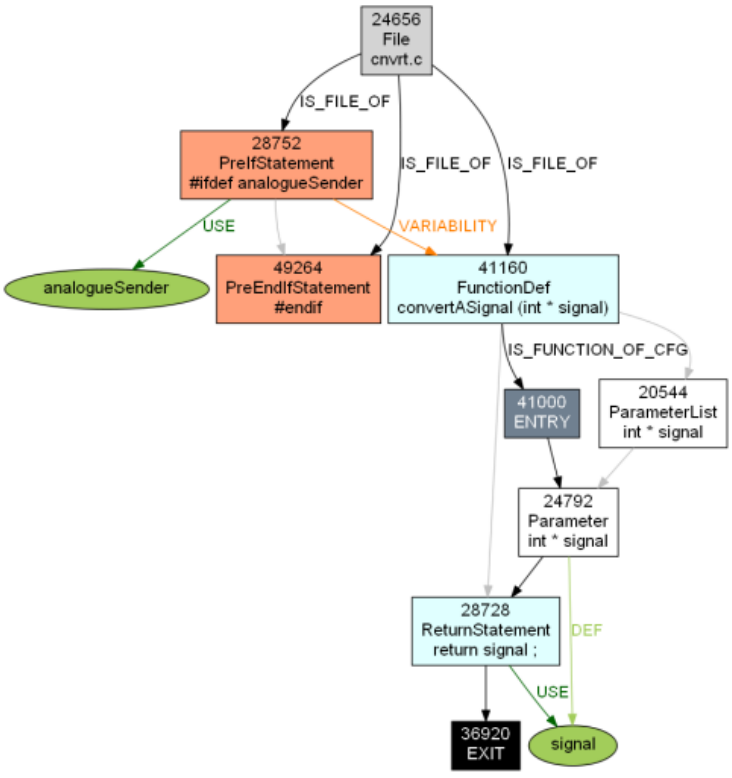

Figura 3.1: CPG del codice precedente. [8]

[11] Nell'articolo di Chakraborty e gli altri, il CPG è così composto:

(a) Insieme di nodi V, dove ogni nodo contiene due informazioni: un vertex type che rappresenta il tipo d'informazione contenuta dal nodo (ad esempio chiamata a funzione, espressione aritmetica) ed il frammento di codice. Il tipo di informazione viene codificato tramite un **one-hot encoding vector** (T_v) mentre il codice viene codificato tramite il tool Word2Vec (C_v). Questi due concetti verranno spiegati in seguito.

(b) Insieme di archi E che rappresentano le relazioni tra le porzioni di codice.

One-hot encoding vector è il metodo naive che permette di trasformare le parole in vettori, contando la frequenza di queste parole nel document. Le colonne della matrice risultanti rappresentano le parole mentre le righe rappresentano i documenti (i document contenuti nel corpus).
[30] Gli autori nell'articolo accademico, per poter trasformare il codice in un vettore embeddings, utilizzano la libreria **GENSIM**, ovvero una libreria di Python open-source che consente di realizzare questa trasformazione includendo sia informazioni sintattiche e semantiche in modo efficiente. Gli algoritmi contenuti in questa libreria, sono capaci di individuare automaticamente la struttura semantica di un testo analizzando le co-occorrenze statistiche dei pattern. Inoltre, essi sono algoritmi non supervisionati, cioè non viene richiesta nessuna operazione da parte dell'uomo per addestrare il modello, ovvero le istanze del dataset non sono etichettate. Gli elementi utilizzati in Gensim sono:

- **Document**: rappresenta un oggetto contenente del testo;
- **Corpus**: rappresenta un insieme di document. Esso viene utilizzato sia nella fase di addestramento che nella fase di test del modello. Nella fase di training il corpus consente al modello di individuare e apprendere argomenti e temi comuni, mentre nella fase di test, permette di verificare le performance del modello. Dato che un corpora dovrà risiedere in memoria principale (RAM) e può contenere molti document, per avere una migliore gestione della memoria, Gensim preleva un document alla volta dal file contenente il corpora e lo trasferisce in RAM;
- **Vettori**: sono una rappresentazione matematica dei documenti;
- **Modelli**: è un algoritmo che ha il compito di trasformare la rappresentazione dei vettori in un'altra rappresentazione.

Dopo aver memorizzato il corpus, vengono effettuate alcune operazioni di pre-processing in modo da semplificare il dataset. In genere, in questo step, si applica la tecnica di **stopword removal**, ovvero si andranno ad eliminare le parole che non sono rilevanti per comprendere la semantica del testo come ad esempio alcune parole inglesi comunemente usate (come "the") ed inoltre vengono rimosse le parole che appaiono una sola volta nel corpus. Un'altra operazione di normalizzazione del testo è quella di riportare tutte le parole in minuscolo. Sempre in questo step si andranno a tokenizzare i dati, dividendo così i vari document in parole. Il delimitatore comunemente utilizzato per fare ciò è lo spazio. Dopo aver definito i token, si andrà a calcolare la loro frequenza. Infine, ad ogni token viene assegnato un ID intero univoco, definendo così un **vocabolario** che contiene questo mapping. Per definire la rappresentazione latente del corpus bisogna utilizzare una tecnica che ci consente di rappresentare i document in un formato matematico in modo da poterli manipolare. Una tecnica potrebbe essere quella di rappresentare i document tramite i **vettori delle feature (feature vector)**. Questi vettori risiedono in uno spazio chiamato **feature space** che è uno spazio n-dimensionale dove n dipende dal numero di feature che caratterizzano le istanze all'interno del dataset. Un approccio può essere il **modello bag-of-word**, dove i document vengono rappresentati da un vettore le cui componenti rappresentano le frequenze delle parole del vocabolario nel document in questione. Per esempio, se il vocabolario è il seguente: [coffee milk sugar spoon] ed il document è "coffee milk coffee", esso sarà rappresentato da questo vettore: [2, 1, 0, 0]. La dimensione del vettore, quindi, è il numero di elementi contenuti nel vocabolario. Una caratteristica di questo modello è che ignora l'ordine dei token nel document codificato. Quindi, grazie al vocabolario di dimensione n, possiamo trasformare il document tokenizzato in un vettore di dimensione n. Possiamo inoltre, rappresentare un document tramite un **vettore bag-of-word o sparse vector**, dove gli elementi di questo vettore sono delle coppie così definite: id del token presente nel vocabolario, frequenza del token nel document. Le parole del vocabolario che non sono contenute nel document, vengono rappresentate implicitamente da uno 0 nella rappresentazione vettoriale del document. Come precedentemente illustrato, l'intero corpus deve risiedere in memoria principale e dato che in genere, contiene migliaia di document, Gensim consente di memorizzare il corpus in un file e caricare in RAM un document alla volta in modo da ottimizzare l'uso della memoria. Una volta trasformato l'intero corpus in formato vettoriale si può applicare su di esso un modello che consente di ottenere un'altra rappresentazione. Il modello, quindi, rappresenta quel processo

che consente di passare da uno spazio vettoriale ad un altro. Quest'ultimo imparerà i dettagli di questa trasformazione tramite i dati di addestramento, ovvero il corpus. Per ottimizzare l'uso della memoria, anche il modello addestrato può essere memorizzato in memoria di massa e successivamente può essere riportarlo in RAM per continuare l'addestramento o eseguire altre operazioni. C'è da notare che il modello non andrà a trasformare l'intero corpus quando verrà invocato, poichè ne implica la sua intera presenza in RAM, quindi la trasformazione avverrà quando itereremo sul corpus.
Dopo aver addestrato il modello si possono effettuare varie operazioni come: trasformare ed indicizzare l'intero corpus in modo da poter applicare query di similarità, ovvero verificare la similarità di un dato document con tutti i document nel corpus. Le query di similarità consentono di verificare la similarità tra 2 document o tra un determinato document e un corpus. Nel secondo caso, l'output della query di similarità sarà un insieme di coppie dove il primo elemento rappresenta il numero del document nel corpus, mentre il secondo elemento indica la percentuale di similarità che ha il document corrente del corpus con il document preso in input. Una misura di similarità che può essere utilizzata è la somiglianza del coseno (**cosine similarity**) che assume valori nel range [-1,1], dove 1 indica che i due elementi sono simili mentre -1 indica che sono opposti. Questa misura può essere utilizzata solo nei casi in cui i vettori rappresentano una distribuzione di probabilità. I motivi per i quali si vuole trasformare la rappresentazione vettoriale dei document in un'altra rappresentazione sono:

- Possibilità di mostrare una struttura nascosta nel corpus ed individuare le relazioni tra le parole, in modo da poter descrivere i document in modo migliore;
- Avere una rappresentazione più compatta dei document. Questo rende il modello più efficiente in quanto vengono impiegate meno risorse ed inoltre viene ridotto il rumore.

Tra i modelli offerti dalla libreria Gensim, gli autori, hanno utilizzato il **modello Word2Vec** in cui viene addestrata, valutata ed utilizzata una rete neurale realizzata da dei ricercatori di Google. Riportiamo in seguito alcuni parametri importanti del modello:

- **Sentences**: rappresenta una lista di token (token definiti sul corpus) ed è iterabile in modo da ottimizzare la memoria. Tramite questo parametro andremo ad addestrare la rete, senza il modello non sarà inizializzato;

- **Corpus-file**: può essere utilizzato al posto del parametro precedente e consente di avere delle prestazioni migliori. In questo modo passeremo l'intero corpus;
- **Vector-size**: indica la dimensione dei vettori che andranno a rappresentare il testo;
- **Min-count**: rappresenta una soglia della frequenza delle parole. Tramite questo parametro tutte le parole che hanno una frequenza al di sotto di questa soglia non verranno considerate;
- **Sg**: può assumere valore 0 o 1, se vale 1 vuol dire che l'algoritmo di addestramento sarà skip-gram e 0 per CBOW;
- **Hs**: può assumere valore 0 o 1: se vale 1 vuol dire che verrà utilizzata hierarchical softmax durante l'addestramento, se vale 0 e il parametro negative ha un valore diverso da 0 allora verrà applicato il negative sampling;
- **Negative**: se assume un valore maggiore di 0 allora verrà applicato il negative sampling;
- **Ns-exponent**: è l'esponente utilizzato per il negative sampling. Se vale 1 allora applicherà la tecnica in proporzione alla frequenza, se vale 0 tratta le parole equamente, se assume valori negativi allora riduce maggiormente le parole con bassa frequenza rispetto a quelle con frequenza maggiore;
- **Alpha**: indica il valore iniziale del learning rate;
- **Min-alpha**: durante l'addestramento del modello, il learning rate decrescerà linearmente e questo parametro indica il valore che verrà raggiunto;
- **seed**: è il seme per il generatore di numeri casuali, in quanto inizialmente i vettori delle parole sono inizializzati con l'hash della concatenazione tra la parola e il seme;
- **max-vocab-size**: indica la quantità massima di memoria RAM occupabile durante la definizione del vocabolario;
- **sample**: indica la soglia per la quale verranno eliminate casualmente delle parole ad alta frequenza;
- **epochs**: indica il numero di iterate che si devono effettuare sul corpus. Questo parametro viene chiamato anche iter. Vengono quindi eseguiti iter+1 passi sulla sentences dove il primo memorizza le parole e la loro frequenza in modo da realizzare un dizionario interno strutturato ad albero mentre dal secondo step in poi si addestra la rete neurale;

- **sorted-vocab**: può assumere un valore pari a 0 o ad 1. Se vale 1 vuol dire che il vocabolario verrà ordinato in ordine decrescente in base alla frequenza delle parole e dopodichè verrà assegnato l'id;
- **compute-loss**: è un booleano che se vale true allora verrà calcolato e memorizzato il valore della loss function;

Gli algoritmi di apprendimento che possono essere utilizzati per questo modello sono: continuous bag-of-words e continuous skip-gram. Come precedentemente illustrato, nel parametro **hs** del modello Gensim possiamo scegliere se applicare la hierarchical softmax o il negative sampling. Analizziamo la **Hierarchical softmax**:
[9] essa è una variante della funzione d'attivazione softmax ma che richiede un tempo nettamente inferiore per la valutazione. Questa funzione viene applicata all'output layer dove ogni nodo j ci dice la probabilità con la quale la parola in input i è uguale alla parola j-esima del vocabolario. La hierarchical softmax utilizza un albero binario multi layer (**Huffman tree**) dove le foglie rappresentano le parole del vocabolario e per ogni nodo foglia esiste un unico cammino che ci permette di raggiungerle a partire dalla radice dell'albero. Inoltre, su ogni arco è associata una probabilità, quindi la probabilità di una parola è calcolata come il prodotto delle probabilità degli archi che si trovano sul path per raggiungere il nodo foglia che rappresenta la parola in questione. In quest'albero ci sono dei nodi intermedi che rappresentano i parametri interni, ovvero la **probability mass**. Ognuno di questi nodi, quindi, rappresenta la probabilità relativa ai suoi nodi figli. Ai nodi intermedi è associato un vettore $v'_{n(w,j)}$ che lo rappresenta. Si decompone l'output layer in un albero binario perchè questo consente di ridurre la complessità per ottenere la probabilità distribuita da $O(V)$ a $O(\log V)$. Un esempio è mostrato in Figura 3.2 dove $n(w,j)$ indica il j-esimo nodo del percorso che parte dalla radice ed arriva al nodo foglia w, $v'_{n(w,j)}$ è la rappresentazione vettoriale del nodo intermedio j-esimo, $[[x]]$ è una funzione così definita:

$$[\![x]\!] = \begin{cases} 1 & \text{se } x \text{ è true} \\ -1 & \text{altrimenti} \end{cases}$$

h è l'output prodotto dall'hidden layer e $ch(n(w,j))$ è il figlio sinistro del nodo $n(w,j)$. La probabilità per la parola w è calcolata nel seguente modo:

$$p(w = w_0) = \prod_{j=1}^{L(w)-1} \sigma([[n(w,j+1) = ch(n(w,j))]] \cdot v'^{T}_{n(w,j)} h)$$

dove w_0 rappresenta la parola in considerazione e σ è la funzione sigmoidea. Per raggiungere un nodo foglia, a partire dalla radice, dobbiamo definire per ogni nodo intermedio la probabilità di andare al figlio sinistro o al figlio destro. Quindi la probabilità per andare al figlio sinistro del nodo intermedio n è:

$$p(n, left) = \sigma(v_n'^T \cdot h)$$

mentre la probabilità di andare al figlio destro è:

$$p(n, right) = 1 - \sigma(v_n'^T \cdot h) = \sigma(-v_n'^T \cdot h)$$

Quindi, dato in input la rappresentazione vettoriale di una parola (più nello specifico la parola elaborata dall'hidden layer), esso viene moltiplicato con il vettore associato al nodo radice dell'albero e da questo prodotto vettoriale otteniamo uno scalare al quale applichiamo la funzione sigmoidea in modo da ottenere un valore compreso tra 0 ed 1. Il valore 0 indica vai a sinistra mentre 1 significa vai a destra. Una volta raggiunto il nodo figlio, si ripete il processo fino a raggiungere un nodo foglia. Questo nodo foglia rappresenta una parola del vocabolario e moltiplicando la probabilità di ogni arco del percorso effettuato, otteniamo la probabilità che la parola in input è simile alla parola rappresentata dal nodo foglia in questione. Questo processo viene effettuato per ogni nodo dell'output layer della rete, quindi per ogni parola del vocabolario. Una caratteristica della funzione hierarchical softmax è che la somma delle probabilità di ogni parola del vocabolario è pari a 1.

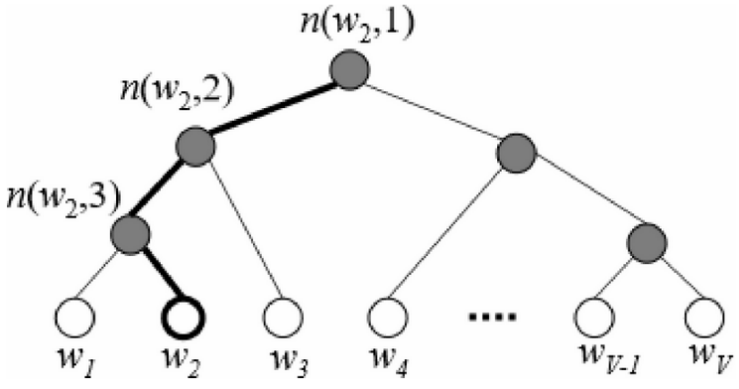

Figura 3.2: Albero binario utilizzato nella hierarchical softmax [9]

Dopo aver definito il CPG, per ogni vertice, si concatenano i due vettori, T_v e C_v in un unico vettore. In questo modo, ogni nodo rappresenta solo i dati in esso contenuti ma

non ha alcuna informazione sui nodi ad esso adiacenti e sulla struttura del grafo. Ogni vertice, quindi, dovrà esporre sia i propri dati che quelli dei suoi nodi adiacenti e per realizzare ciò, gli autori hanno utilizzato un **Gated Graph Neural Networks (GGNN)**. [31] I GNN sono delle architetture generali per le reti neurali e sono definite sulla base della struttura di un grafo $G = (V, E)$. Esso prende in input un grafo che può essere sia un grafo diretto che indiretto. Nel caso di grafo diretto le notazioni utilizzare sono:

- (v, v') rappresenta l'arco diretto che parte dal nodo v e raggiunge il nodo v' (v $\rightarrow$ v');
- la rappresentazione del nodo v viene chiamata node vector o node embedding e viene indicata con h_v;
- i nodi e gli archi possono essere etichettati e vengono denotati rispettivamente con: l_v e l_e;
- IN(v) è una funzione che restituisce tutti i nodi padre del nodo v;
- OUT(v) è una funzione che restituisce l'insieme di nodi raggiungibili tramite un arco dal nodo v;
- NBR(v) restituisce l'insieme dei vicini del nodo v;
- CO(v) restituisce tutti gli archi entranti ed uscenti dal nodo v.

L'obiettivo principale dei GNN, è mappare il grafo di input nell'output tramite due fasi:

(a) **fase di propagation**: è un processo iterativo dove viene computata la rappresentazione di ogni nodo h_v. Inizialmente h_v è settato in modo arbitrario e verrà man mano aggiornato tramite la seguente operazione finché il modello non converge:

$$h_v^{(t)} = f^* \left(l_v, l_{CO(v)}, l_{NBR(v)}, h_{NBR(v)}^{(t-1)} \right)$$

La funzione $f^*()$ può essere decomposta in due sommatorie in base al tipo di arco, ovvero arco entrante o uscente dal nodo:

$$f^* \left(l_v, l_{CO(v)}, l_{NBR(v)}, h_{NBR(v)}^{(t)} \right) = \sum_{v' \in IN(v)} f \left(l_v, l_{(v',v)}, l_{v'}, h_{v'}^{(t-1)} \right) + \sum_{v' \in OUT(v)} f \left(l_v, l_{(v,v')}, l_{v'}, h_{v'}^{(t-1)} \right)$$

Questa funzione può essere una funzione lineare di h_v o una rete neurale. Le altre formule utilizzate nella fase di propagation sono:

$$h_v^{(1)} = \left[x_v^T, 0\right]^T \qquad r_v^t = \sigma\left(W^r a_v^{(t)} + U^r h_v^{(t-1)}\right)$$
$$a_v^{(t)} = A_{v:}^T \left[h_1^{(t-1)T} ... h_{|V|}^{(t-1)T}\right]^T + b \qquad \widetilde{h_v^{(t)}} = tanh\left(W a_v^{(t)} + U\left(r_v^t \odot h_v^{(t-1)}\right)\right)$$
$$z_v^t = \sigma\left(W^z a_v^{(t)} + U^z h_v^{(t-1)}\right) \qquad h_v^{(t)} = (1 - z_v^t) \odot h_v^{(t-1)} + z_v^t \odot \widetilde{h_v^{(t)}}$$

La matrice A indica come sono collegati i nodi nel grafo ed ha una struttura di sparsità che rappresenta gli archi del grafo, mentre i valori all'interno di essa dipendono dal tipo di archi e dalla loro direzione. A_v sono 2 colonne prese dalla matrice A_{out} e dalla matrice A_{in} in corrispondenza del nodo v. La prima formula rappresenta l'inizializzazione di h_v, la seconda rappresenta la fase in cui si fanno fluire le informazioni tra i nodi tramite gli archi. Le altre formule rappresentano l'aggiornamento del GRU che permette di aggiornare l'h_v di ogni nodo incorporando le informazioni degli altri e le rappresentazioni dei nodi degli step precedenti. Z ed R rappresentano rispettivamente il gate di aggiornamento e di reset, mentre σ rappresenta la funzione sigmoidea logistica e $\odot$ è il simbolo che rappresenta la moltiplicazione;

(b) **output model**: Viene definito un output model per nodo che è una funzione differenziabile ed ha il compito di mappare la rappresentazione dei nodi e le rispettive etichette nell'output o_v ($o_v = g(h_v, l_v)$). Questo mapping può essere lineare oppure effettuato tramite una rete neurale. L'apprendimento per questo modello avviene tramite l'algoritmo Almeida-Pineda che si occupa di addestrare il modello dalla fase di propagazione fino alla fase in cui il modello converge.

[11] L'input di questa rete neurale, quindi, sarà il grafo CPG, dove ad ogni suo nodo verrà assegnato un **gated recurring unit (GRU)** che consente di aggiornare le informazioni del vertice, includendo i dati dei suoi nodi vicini. Quest'operazione di aggiornamento, avviene nel seguente modo:

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

x_v rappresenta i dati contenuti nel nodo v del CPG, la funzione $g()$ (**transformation function**) consente di incorporare le informazioni di tutti i vicini u del nodo corrente v ed infine x_v' rappresenta il nuovo valore dei dati del vertice v dopo aver applicato quest'operazione. L'ultima fase di pre-processing da applicare sul grafo, è incorporare

tutti gli x'_v in un unico nodo x_g che rappresenta l'intero grafo CPG di partenza. Per effettuare ciò viene utilizzata un **aggregation function** che nel nostro caso è una semplice sommatoria:

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

Il risultato finale ottenuto è una rappresentazione vettoriale m-dimensionale delle feature del codice originale. Gli step di questo pre-processing dei dati è mostrato in Figura 3.4. Infine, gli autori fanno notare che l'addestramento della GGNN è avvenuto in un contesto supervisionato e che un representation learner, in un contesto non supervisionato, potrebbe apprendere una rappresentazione dei dati migliore.

2. **Addestramento**: In questa seconda fase del progetto, si addestra la rete neurale sui dati di training, sui quali è stato applicato il pre-processing. Come già detto precedentemente, il problema è di natura sbilanciato e questo permette di avere un dataset con maggiori esempi di codice non vulnerabile rispetto agli esempi vulnerabili. Questo problema non può passare inosservato, in quanto produrrebbe un modello che è più abile ad individuare codici privi di bug che codici vulnerabili. Inoltre, effettuando un'analisi sui dati, si è notato che nello **spazio delle feature**, c'è una sovrapposizione tra le due tipologie di istanze che vengono rappresentate da un vettore delle feature. Questa sovrapposizione, indica che le due categorie hanno dei punti in comune che rendono difficile la loro distinzione per il modello. Per poter risolvere questi problemi, gli autori dell'articolo, hanno applicato due tecniche sui dati:

   (a) **SMOTE**: è una tecnica di **data balancing** che consente di bilanciare le classi all'interno del dataset. Essa ricade nella categoria di **oversampling** che rappresenta l'insieme di tecniche che generano un certo numero di istanze casuali per la classe di minoranza del dataset. La tecnica SMOTE è l'acronimo di Synthetic Minority Over-sampling Technique e consiste nell'eliminare istanze casuali della classe di maggioranza e allo stesso tempo realizza istanze artificiali per la classe di minoranza in modo da bilanciare il set di dati. Per creare le istanze sintetiche, SMOTE procede nel seguente modo: seleziona casualmente un'istanza della classe di minoranza ed applica alle sue feature delle modifiche in modo da generare un insieme chiamato insieme dei vicini dell'istanza scelta. Dopodichè, da quest'insieme, individua i k neighbors più vicini e realizza l'istanza artificiale, andando ad interpolare l'esempio scelto precedentemente con uno dei k vicini selezionato casualmente.

(b) **Riaddestramento del representation learning model**: Questa fase permette di risolvere il problema legato alla sovrapposizione dei vettori delle feature degli esempi vulnerabili e non. L'obiettivo è realizzare un modello di machine learning capace di trasformare lo spazio delle feature in uno spazio che consente una maggiore separazione tra le classi, ovvero il **latent space**. Per fare ciò, gli autori hanno utilizzato il Multi Layer Perceptron (MLP 2.4) che permette di trasformare il vettore x_g, ottenuto alla fine della fase di feature extraction, in una rappresentazione latente $h(x_g)$. [10] Il **latent space** o **spazio latente** consente di rappresentare i dati in modo compresso, ovvero consente di semplificare la rappresentazione dei dati in modo da poterli elaborare ed analizzare (quindi poter individuare dei pattern e similarità tra le istanze). Questa compressione ci permette di focalizzarci solo sulle feature fondamentali e di poter graficare, ad esempio, le istanze su un piano 3D. In genere ogni volta che il modello viene addestrato, esso riduce la dimensionalità dell'input per poi riportarla alle dimensioni iniziali durante la fase di output. Nel nostro caso, invece, il modello dovrà realizzare la rappresentazione latente dell'input per poi fornire la predizione in output.

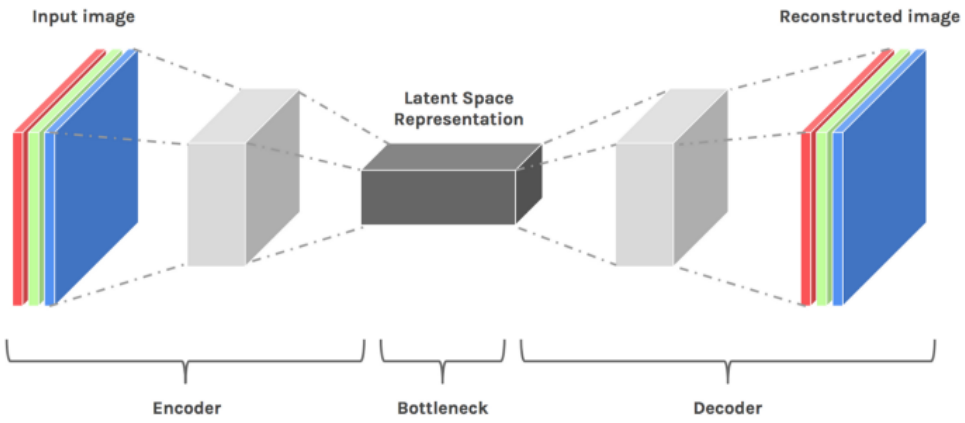

Figura 3.3: Latente space [10]

Nello spazio latente vengono compresse solo le feature che più caratterizzano le istanze mentre le informazioni meno rilevanti vengono eliminate. Quindi un punto nello spazio latente contiene le informazioni necessarie per poter rappresentare l'istanza originale. Inoltre, i vettori che sono simili tra loro saranno vicini nello spazio latente.

[11] Il compito dell'hidden layer del MLP è quello di proiettare il vettore x_g nello spazio latente $h(x_g)$ dopodichè l'output layer, sulla base delle feature di

questo spazio, fornisce la predizione sulla vulnerabilità. La funzione di attivazione utilizzata in questa rete neurale è la **funzione softmax**, ovvero una funzione logistica che converte l'input k-dimensionale in un altro vettore k-dimensionale dove le sue componenti sono comprese tra 0 e 1 e la loro somma è pari 1.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} conj = 1, ..., K$$

Le componenti di questo vettore indicano la probabilità con la quale l'input z faccia parte della classe rappresentata dalla componente, ovvero la classe che può assumere la variabile target. Come affermato nella Sezione 2.5, durante l'addestramento, si ha l'obiettivo di minimizzare la loss function, in modo da migliorare le performance del modello. In questo articolo, gli autori hanno utilizzato come loss function la **triplet loss** che è frequentemente utilizzata nel representation learning, in quanto consente di massimizzare la separazione tra le classi che può assumere la variabile dipendente. La triplet loss function è composta da tre funzioni di perdita che sono:

- **cross entropy loss** L_{CE}: permette di penalizzare il modello quando effettua degli errori nella classificazione delle istanze, infatti la funzione cresce all'aumentare della discrepanza tra la predizione effettuata dalla rete ($\hat{y}$) e l'effettiva classificazione dell'istanza (y).

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

- **project loss** L_p: indica quant'è accurata la separazione tra le classi che può assumere la variabile target nello spazio latente. Questa rappresentazione è considerata ottimale quando le istanze della stessa classe sono vicine tra loro e allo stesso tempo lontano dalla istanze appartenenti a un'altra classe.

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

  $h(x_{same})$ è la rappresentazione latente di un'istanza appartenente alla stessa classe di x_g, mentre $h(x_{diff})$ è la rappresentazione latente di un esempio che non appartiene alla stessa classe di x_g. Infine, γ è un iperparametro che indica la distanza minima che ci deve essere tra le classi, mentre $D()$ è il coseno della distanza (**cosine distance**) tra due vettori (ovvero 2 istanze) che viene calcolato nel seguente modo:

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

Nota che il cosine distance è diverso dal cosine similarity ma sono correlati tra loro, infatti `cosine distance = 1 - cosine similarity` in quanto se due vettori sono uguali allora la distanza tra essi è 0. La project loss tenderà a crescere se due istanze appartenenti alla stessa classe sono lontane tra loro o quando due istanze appartenenti a classi diverse sono vicine nello spazio latente. Questa crescita indica che la rappresentazione latente non è ideale.

- **regularization loss** L_{reg}: consente di moderare la crescita della rappresentazione latente dell'input x_g, questo perchè si è evidenziato che su diverse iterazioni, la rappresentazione latente $h(x_g)$ tende a crescere arbitrariamente. Se non si provvede a contenere questa crescita, si otterrà un modello che non riuscirà a convergere. Questa funzione, quindi, ha l'obiettivo di penalizzare le rappresentazioni latenti che hanno una grandezza maggiore, tramite questa formula:

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

L'obiettivo finale, quindi, è minimizzare la triplet loss function che è data da:

$$L_{trp} = L_{CE} + \alpha \cdot L_p + \beta \cdot L_{reg}$$

α e β sono due iperparametri che indicano il contributo della project loss e della regularization loss.

Infine, la fase di addestramento viene interrotta quando la **F1-score** sul test set non aumenta dopo 50 iterazioni di addestramento per il GGNN e dopo 5 iterazioni per il representation learner. La F1-score è la media armonica definita sulla precision e sulla recall. La **media armonica** è una particolare media statistica cioè il rapporto tra il numero di elementi considerati e la somma tra i reciproci (n^{-1}) di questi elementi. Essa è il reciproco duale della media aritmetica ed è definita in questo modo:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Quindi, per la F1-score diventa:

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} = 2 \cdot \frac{recall \cdot precision}{recall + precision}$$

Il range di valori che può assumere la F1-score è tra 0 ed 1, dove 1 indica un ottima precision e recall mentre 0 indica dei pessimi valori per queste due metriche.

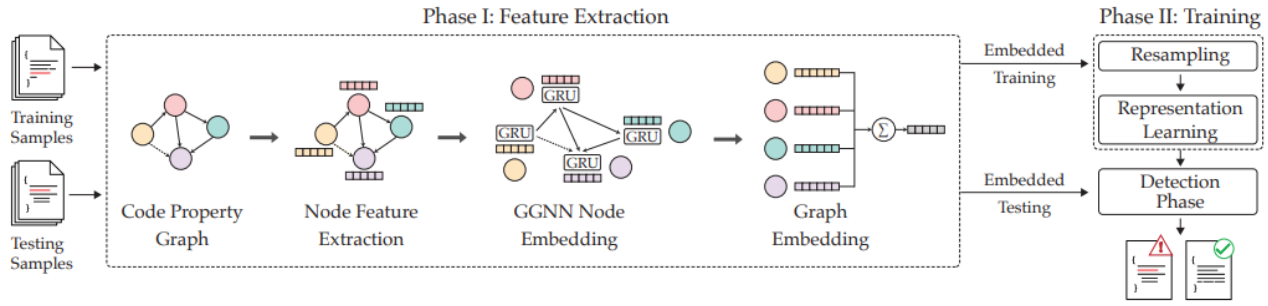

Figura 3.4: Fase di feature extraction e di addestramento di ReVeaL [11]

Una volta definito il modello, per comprendere la bontà delle sue prestazioni, abbiamo bisogno di testarlo tramite un set di esempi non presenti nel training set. Nella fase di valutazione ci sono 2 aspetti da considerare: tipo di formulazione del problema e relative metriche di valutazione utilizzate e il tipo di procedura di valutazione.
Nel primo caso la maggior parte degli approcci definiscono il problema come un task di classificazione dove dato un codice, lo si etichetta come vulnerabile o non. Ovviamente, questo implica un gran numero di esempi su cui addestrare il modello. Inoltre ciò che è fondamentale considerare durante la valutazione è il livello di granularità con cui lavora il machine learner ed in questo caso può essere a livello di funzione, a livello di pezzi di codice, ecc. Gli autori dell'articolo, come precedentemente detto, hanno definito un modello che lavora a livello di funzione e le metriche utilizzate per valutare il modello sono: Accuracy, Precision, Recall ed F1-score.
Nel secondo caso, dato che il modello di deep learning dipende fortemente dalla randomicità, per evitare di introdurre del bias, gli autori hanno eseguito per 30 volte l'intero processo sia per il modello da loro proposto e sia per i modelli esistenti che hanno analizzato. Ad ogni esecuzione, il dataset viene diviso randomicamente in training set, test set e validation set, assegnando rispettivamente l'80%, 20% ed un 10% prelevato dal training set. Effettuando iterativamente il processo, gli autori hanno riportano la media delle performance dei modelli e l'IQR. Per comprendere cos'è l'IQR bisogna definire cos'è un quartile: in statistica, quando si ha una distribuzione di una variabile (dove quest'ultima rappresenta un insieme di caratteristiche misurate su una o più unità statistiche, ovvero elementi di un insieme, come esito di un'indagine), i **quartili** sono quei valori che dividono i campioni in quattro parti di eguale quantità. Questi quattro quartili rappresentano degli indici di posizione che permettono di dare un'idea dell'ordine di grandezza dei valori che può assumere la variabile su una scala di numeri. Il primo quartile Q_1 è un valore tale che il 25% dei valori che può assumere la variabile, alla sua sinistra, sono minori o uguali a Q_1. Questo quartile viene

chiamato 25-esimo percentile ed è indicato con $P_{0.25}$. Il secondo quartile Q_2 rappresenta la mediana, ovvero quel valore centrale che divide la distribuzione in due parti uguali e viene chiamato 50-esimo percentile ($P_{0.50}$). Il terzo quartile Q_3, rappresenta quel valore tale che, il 75% dei valori che può assumere la variabile (si trovano alla sua sinistra) sono minori o uguali a Q_3 e viene chiamato 75-esimo percentile ed è indicato con $P_{0.75}$. L'**interquartile range** (IQR), quindi, rappresenta la differenza tra il terzo e il primo quartile, ovvero l'ampiezza della porzione della distribuzione che contiene la metà centrale dei valori osservati. L'IQR ci indica quanto i valori si discostano dalla mediana campionaria (Q_2). Una volta riportato i risultati dei modelli esistenti, essi verranno confrontati con i risultati di baseline (ovvero le performance comunicate dai rispettivi progetti) tramite un **statistical significance test** ed **effect size test**. [32] Statistical significance test permette di confrontare un insieme di esperimenti, verificando un'ipotesi che abbiamo formulato. Questa ipotesi viene chiamata **ipotesi nulla** e indicata con H_0. Tramite il test, possiamo verificare se quest'ipotesi viene confutata o verificata su un set di dati di un esperimento. Lo stesso test viene effettuato per accettare o meno un'altra ipotesi, chiamata **ipotesi alternativa** ed indicata con H_1 che verrà accettata se H_0 viene confutata. La statistical significance test, quindi, ci permette di verificare se i vari esperimenti sono realmente differenti oppure la differenza osservata è dovuta alle fluttuazioni del caso (cioè i campioni dei dati analizzati non sono significativi nella popolazione analizzata) e quindi gli esperimenti sono di fatto equivalenti. Per effettuare ciò, bisogna definire un **livello di significatività** (α) del test che rappresenta la probabilità di sbagliare, nel caso in cui arrivassimo alla conclusione di rigetto dell'ipotesi nulla. Nel nostro caso, quindi, H_0 rappresenta l'ipotesi che le distribuzioni delle performance ottenute sono uguali alle distribuzioni delle performance di baseline mentre H_1 rappresenta l'ipotesi che le performance sono diverse. L'effect size test, invece, misura l'effettiva differenza tra le distribuzioni (differenza tra le mediane).

Come precedentemente detto, gli autori dell'articolo accademico, hanno analizzato e testato i modelli esistenti per la predizione di vulnerabilità software. Essi hanno prima riportato i risultati di questi modelli ottenuti nelle rispettive ricerche, dopodichè hanno:

1. testato i modelli su dataset del mondo reale, confrontando i risultati ottenuti con quelli dichiarati;

2. poi hanno ri-addestrato i modelli esistenti su dati del mondo reale e confrontato le performance ottenute nuovamente con i risultati precedenti.

Nel primo caso gli autori hanno testato i modelli esistenti sul dataset ReVeal e FFMPeg+Qemu,

riscontrando un forte calo delle prestazioni rispetto a quelle dichiarate. Più nello specifico, si è riscontrato mediamente, un calo della F1-score di circa il 73%. Nel secondo caso, invece, gli autori hanno prima, riaddestrato i modelli esistenti su una porzione del dataset ReVeal utilizzando la restante parte dei dati come test set e poi li hanno riaddestrati e testati sul set di dati di FFMPeg+Qemu. In entrambi i casi, gli autori, hanno ripetuto il processo per 30 volte e ad ogni iterazione hanno utilizzato differenti training e test set. Anche in questo caso, si è notato un calo delle prestazioni dei modelli esistenti, ovvero la F1-score è diminuita mediamente del 54%. Questo ci consente di affermare che i modelli esistenti non riescono a generalizzare la predizione di vulnerabilità nel mondo reale. Come detto in precedenza, i problemi a cui sono soggetti questi modelli sono vari:

- **Dati duplicati**: le tecniche di pre-processing utilizzate dai modelli esistenti, come la tokenizzazione e lo slicing, introducono molte copie dei dati all'interno del dataset. Gli autori hanno applicato queste tecniche sia sul set di dati utilizzati da questi modelli sia sui due dataset contenenti istanze del mondo reale e confrontato il livello di dati duplicati. Si è osservato che, nei dataset dei rispettivi progetti si è raggiunto un livello di dati duplicati maggiore del 60% ed in particolar modo nei progetti in cui si utilizzano set di dati contenenti dati semi artificiali. Per quanto riguarda invece l'applicazione di queste tecniche di pre-processing sui dataset contenenti istanze del mondo reale, si sono ottenuti comunque dati duplicati ma con una percentuale nettamente inferiore rispetto ai set di dati dei rispettivi progetti. Questo problema comporta modelli di deep learning che imparano feature non rilevanti e che non riescono ad estrarre pattern dai dati;

- **Dati sbilanciati**: come precedentemente illustrato, il problema è di natura sbilanciato, questo comporta dataset sbilanciati, cioè avere più esempi di codice non vulnerabile rispetto a codice vulnerabile. Se viene addestrato un modello su un set di dati avente questo problema, ne scaturisce un machine learner che è più bravo a riconoscere la classe di maggioranza, ovvero codice non vulnerabile. In tutti i dataset dei modelli esistenti è stato riscontrato un grande tasso di sbilanciamento tra le due classi e questo spiega le pessime performance ottenute dai modelli su set di dati contenenti istanze del mondo reale;

- **Imparare feature non rilevanti**: per scegliere un buon modello di deep learning, bisogna comprendere quali caratteristiche dei dati la rete utilizza per effettuare la predizione. Nel nostro caso, un buon modello dovrebbe dare un peso maggiore alle feature relate

alla vulnerabilità. Per comprendere quali feature utilizza un modello esistente, bisogna individuare la **feature importance** assegnata ad un codice su cui effettuare la predizione. Per gli approcci esistenti basati su token, si utilizza **Lemna**. Questa tecnica assegna ad ogni token in input al modello un valore w_i^t che rappresenta il contributo fornito da questo token per la predizione. Ovviamente, un valore alto indica una forte influenza sulla predizione ed un valore basso indica una scarsa influenza. Per i modelli basati su grafi, non si può utilizzare Lemna, così gli autori, per definire la feature importance considerano le attivazioni dei vari nodi della rete neurale (ovvero il valore prodotto dal nodo dopo aver applicato la funzione di attivazione). Più è grande il valore di attivazione più quel nodo influisce sulla predizione. Per visualizzare la feature importance, gli autori utilizzano una mappa di colori (**heatmap**) in modo da evidenziare sia i frammenti di codice importanti che quelli meno rilevanti. Tramite questa mappa si è evidenziato che i modelli basati su token non considerano la semantica del codice, mentre i modelli basati sui grafi, grazie al CPG riescono ad imparare le dipendenze semantiche tra i vari nodi del grafo e sfruttare ciò per effettuare la predizione;

- **Mancata separazione delle classi**: come precedentemente illustrato, vari modelli trasformano il codice in vettori delle feature (feature vector) che saranno poi utilizzati per addestrare la rete. Uno dei fattori che influenza le performance del modello durante la predizione è la separazione delle due classi che può assumere la variabile target in quanto, una maggiore separazione delle categorie permette alla rete di distinguere meglio le due tipologie di istanze. Per valutare la separazione delle classi nei dataset dei modelli esistenti, utilizziamo **t-SNE plot** che è una tecnica di riduzione della dimensionalità molto utile per questo task. Essa permette di visualizzare come sarà il dataset con grandi dimensioni nel feature space. Per misurare la separazione delle classi, utilizziamo la **centroid distance** che indica la distanza tra due centroidi. Un **centroide** è la miglior rappresentazione di un determinato cluster, quindi nel nostro caso avremo due centroidi che sono rispettivamente la miglior rappresentazione di una delle due categorie che può assumere la variabile target. Il centroide per un dato insieme viene calcolato in questo modo:

$$m_i = \frac{1}{n_i} \cdot \sum_{x \in D_i} x \quad \text{con} \quad i = 1, ..., k$$

dove n_i rappresenta il numero di istanze appartenenti alla classe D_i, k è il numero di classi (nel nostro caso k=2) ed m_i è il centroide della classe D_i. La metrica utilizzata per

definire la distanza tra centroidi è la **distanza euclidea** che è così definita:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

dove p_1 e p_2 sono i due punti tra cui vogliamo calcolare la distanza euclidea, mentre x_i e y_i rappresentano le coordinate dei due punti. Un'evidente separazione delle classi nello spazio t-SNE ed un alto valore per la distanza euclidea, indicano una buona separazione delle classi nel dataset. In tutti i modelli esistenti analizzati, si è evidenziato che questi hanno una considerevole sovrapposizione dei vettori delle due classi nel feature space e di conseguenza anche una pessima distanza euclidea. Quest'analisi della separazione delle classi per i vari progetti è riportata in Figura 3.5.

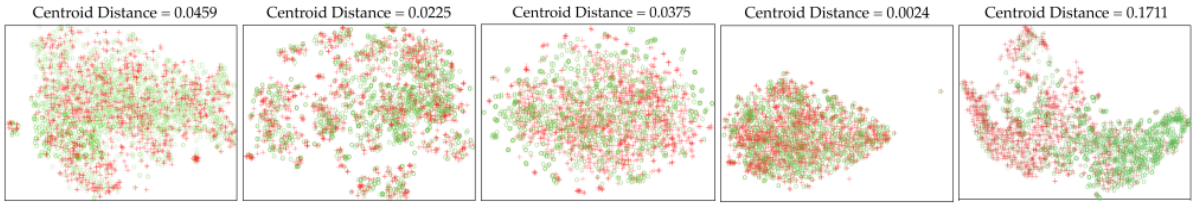

Figura 3.5: t-SNE plot dei vari modelli dove il simbolo + rappresenta le istanze vulnerabili mentre ○ rappresenta le istanze non vulnerabili

Il progetto ReVeal proposto da Chakraborty e gli altri risolve tutti questi problemi che affliggono i modelli esistenti, infatti:

1. Viene risolto il problema della duplicazione dei dati andando a realizzare un'unica feature per ogni istanza. Per fare ciò il codice viene trasformato nel corrispettivo CPG, poi viene applicato il GGNN su ogni nodo del grafo ed infine si applica la funzione di aggregazione. In questo modo non si avrà mai un vettore delle feature uguale per due istanze diverse a meno che non siano uguali;

2. Viene risolto il problema del dataset sbilanciato andando ad applicare la tecnica SMOTE, in questo modo il modello sarà indipendente dalla distribuzione delle classi;

3. Viene risolto l'apprendimento delle informazioni, ovvero grazie al CPG è possibile rappresentare sia dipendenze sintattiche che semantiche e grazie al GGNN ogni nodo rappresenterà anche le informazioni dei nodi adiacenti e la struttura del grafo;

4. Viene risolta la separazione delle classi tramite un representation learner (Multi Layer Perceptron) che impara a trasformare lo spazio delle feature in un altro spazio in cui le due categorie sono correttamente separate

Confrontando le performance del progetto ReVeal con gli altri approcci esistenti sul dataset ReVeal e FFMPeg+Qemu, si evince che ha migliori prestazioni in tutte le metriche di valutazione. Per evidenziare l'importanza delle varie tecniche adoperate nel progetto ReVeal, per risolvere i problemi precedentemente elencati, gli autori hanno realizzato varie alternative del modello, ovvero:

1. una variante in cui non viene utilizzato il GGNN e quindi sui vertici iniziali del CPG viene applicata direttamente la funzione di aggregazione. Confrontando le metriche di valutazione di questa variante con il progetto originale si evince che l'utilizzo del GGNN migliora in modo considerevole la F1-score sia sul dataset ReVeal che sul dataset FFMPeg+Qemu;

2. una variante in cui non viene applicata la tecnica SMOTE sul training set e confrontando i risultati con il progetto originale, applicare l'undersampling permette di ottenere performance migliori. Nel dettaglio, applicando la SMOTE sul dataset FFMPeg+Qemu si ha un miglioramento solo del 3% sulla F1-score rispetto alla variante, mentre per il dataset ReVeal si ha un miglioramento del 22%. C'è da notate che la variante senza SMOTE ha una precision migliore ma al costo di una bassa recall dovuta ai dati sbilanciati;

3. tre varianti in cui viene sostituito il representation learning utilizzando rispettivamente **Random Forest**, **SVM** e **MLP pre addestrato**. In entrambi i dataset, il progetto originale ottiene comunque prestazioni migliori rispetto alle tre varianti;

3.2 Limitazioni del progetto ReVeal e le loro implicazioni in pratica

Dopo un'attenta analisi del progetto ReVeal nella Sezione 3.1, si evince dal pre-processing dei dati e dalla configurazione dei parametri della rete neurale che:

1. Utilizzano come funzione di attivazione nel representation learning la funzione ReLU. Questa funzione è soggetta al problema chiamato **dying ReLU**, ovvero durante la fase di training alcuni neuroni della rete potrebbero "morire", cioè producono in output solo valori pari a zero. Diremo che, un neurone muore quando i suoi pesi delle connessioni in ingresso vengono modificati a tal punto che la somma pesata dei suoi input è negativa per ogni istanza del training set. Quando ciò accade, l'output emesso dal neurone sarà pari a zero e il Gradient Descent non potrà fare nulla per questi neuroni in quanto il gradiente della ReLU è zero. Inoltre, un ulteriore fattore che agisce sulla disattivazione

dei neuroni è l'algoritmo **Dropout** che viene applicato subito dopo la funzione ReLU. In quest'algoritmo, ad ogni neurone viene assegnata una probabilità p chiamata **dropout rate** che rappresenta la probabilità di disattivare momentaneamente il neurone per un determinato step di forward. Queste probabilità sono indipendenti tra loro ad ogni esecuzione di forward e l'algoritmo viene applicato soltanto in fase di training [5].

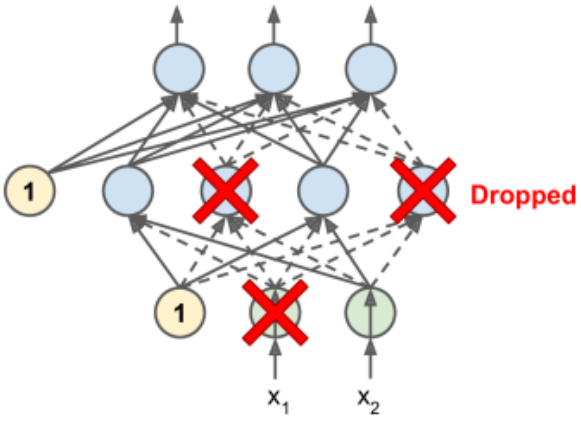

Figura 3.6: Esempio di Dropout [5]

2. Durante l'addestramento, il parametro `max_patience` è impostato ad un valore molto basso. Esso permette di terminare prima l'addestramento del modello, ovvero se durante le varie epoche, un determinato settaggio della rete ottiene prestazioni migliori sul validation set rispetto ai parametri del modello attuale per un numero di volte pari a max patience, l'addestramento terminerà e la rete neurale verrà configurata come il modello migliore. Se questo parametro è molto basso, potrebbe capitare che le iterazioni successive definirebbero una rete neurale migliore che non verrebbe considerata perchè l'addestramento terminerebbe prima;

3. Analizzando l'addestramento del modello, esso ottiene una loss function per ogni epoca molto elevata. Quest'ultima è pari alla somma delle loss function ottenute dal modello sui vari mini-batch del training set. Se la rete neurale ha una loss function considerevole comporta metriche di valutazioni pessime;

4. Utilizzano una dimensione dei mini-batch per addestrare il modello pari a 128, ovvero il numero di istanze appartenenti a un mini-batch corrisponde a 128. Incrementando questo parametro, ad ogni iterazione si forniscono maggiori esempi al modello e quindi si può ulteriormente migliorare la rete;

5. Nell'analizzare i risultati ottenuti dal modello gli autori danno un peso maggiore alla F1-score rispetto all'accuracy e la matrice di confusione. Quest'ultima ci permette di ottenere informazioni inerenti al numero di predizioni corrette ed errate effettuate dalla rete neurale.

3.3 Studio empirico degli hyper-parameter dei modelli di Deep Learning

Per poter fronteggiare le limitazioni definite nella sezione precedente, in questa tesi si effettua uno studio empirico sull'impatto che possono avere le varie configurazioni degli hyper-parameter sulle performance del modello, andando ad apportare alcune modifiche alla pipeline di ReVeal realizzando le seguenti varianti:

1. Una variante in cui si sostituisce la funzione di attivazione ReLu con Leaky ReLu in quanto quest'ultima consente di risolvere le limitazioni di ReLU ed è anche più performante. Questa funzione è così definita:

$$LeakyReLU_{\alpha}(x) = max(\alpha \cdot x, x)$$

Il parametro α indica l'inclinazione della funzione quando x è negativo. Grazie a questa funzione i neuroni non si disattivano mai [5].

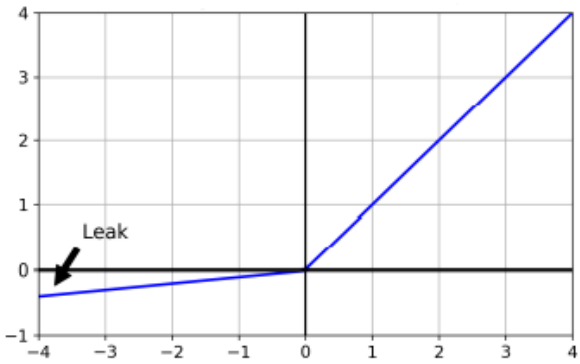

Figura 3.7: Leaky ReLU [5]

2. Una variante in cui viene incrementato il valore del parametro `max_patience` in modo da eseguire un maggior numero di epoche e di conseguenza questo comporta un aumento della probabilità di individuare una configurazione del modello migliore. Nello specifico il valore è stato modificato da 5 a 15;

3. Una variante in cui si aumenta il numero di istanze appartenenti al mini-batch, ovvero da 128 a 200 istanze in modo da fornire più esempi alla volta alla rete neurale;

4. Una variante in cui si uniscono le modifiche apportate dal punto 1 al punto 3;

L'obiettivo di queste modifiche è quello di migliorare le metriche di valutazione e minimizzare la loss function del modello. C'è da considerare, però che la loss function e l'accuracy sono due elementi inversamente proporzionali; infatti, si può notare che l'accuracy aumenta al decrementare della funzione di perdita e viceversa anche se questo non è sempre vero. Dato che non si può annullare la loss function ed ottenere il 100% di accuracy, bisogna definire un trade-off che andrà sicuramente ad impattare sulle prestazioni della rete neurale. Perchè non si può massimizzare il più possibile l'accuracy e allo stesso tempo minimizzare la loss function? Non si può annullare la loss function perché questo andrebbe a generare un modello che non effettua nessun errore sul training set, ciò può significare o che il modello è perfetto (il che è impossibile) o che è fortemente dipendente dai dati di training, ovvero abbiamo un alto overfitting. Allo stesso tempo non si può avere un'accuracy significativamente elevata in quanto è pur sempre un'indicazione che deve essere interpretata in base al problema in analisi, infatti, se il dataset è sbilanciato il modello predice correttamente le istanze della classe di maggioranza ma non quelle della classe di minoranza e questo comporta una buona accuracy ma priva di significato. In generale:

- Avere una bassa accuracy e un alto valore per la loss function significa che la rete neurale ha effettuato molti errori su molti dati;

- Avere come accuracy e loss function valori bassi significa che il modello ha effettuato pochi errori su molti dati;

- Avere una buona accuracy e bassa loss function significa che il modello ha effettuato pochi errori su pochi dati (caso ottimale).

Riporto nelle successive sottosezioni le modifiche apportate alla pipeline ReVeal per realizzare le varie varianti.

3.3.1 Variante: Leaky ReLU

Per poter applicare la Leaky ReLU sono state apportate le modifiche al file `models.py`. Di seguito viene riportato il codice originale e il codice modificato: Codice originale:

```
import numpy as np
import torch
from sklearn.metrics import accuracy_score as acc, precision_score as pr, recall_score as rc, f1_score as f1
from torch import nn
from torch.optim import Adam

from tsne import plot_embedding

import sys

class MetricLearningModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, dropout_p=0.2, aplha=0.5, lambda1=0.5, lambda2=0.001, num_layers=1):
        super(MetricLearningModel, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.internal_dim = int(hidden_dim / 2)
        self.dropout_p = dropout_p
        self.alpha = aplha
        self.layer1 = nn.Sequential(
            nn.Linear(in_features=self.input_dim, out_features=self.hidden_dim, bias=True),
            nn.ReLU(),
            nn.Dropout(p=self.dropout_p)
        )
        self.feature = nn.ModuleList([nn.Sequential(
            nn.Linear(in_features=self.hidden_dim, out_features=self.internal_dim, bias=True),
            nn.ReLU(),
            nn.Dropout(p=self.dropout_p),
            nn.Linear(in_features=self.internal_dim, out_features=self.hidden_dim, bias=True),
            nn.ReLU(),
            nn.Dropout(p=self.dropout_p),
        ) for _ in range(num_layers)])

        self.classifier = nn.Sequential(
            nn.Linear(in_features=self.hidden_dim, out_features=2),
            nn.LogSoftmax(dim=-1)
        )
        self.lambda1 = lambda1
        self.lambda2 = lambda2
        self.loss_function = nn.NLLLoss(reduction='none')
        # print(self.alpha, self.lambda1, self.lambda2, sep='\t', end='\t')

        ...
```

Codice modificato:

```
class MetricLearningModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, dropout_p=0.2,
    ↪  aplha=0.5, lambda1=0.5, lambda2=0.001, num_layers=1):
        super(MetricLearningModel, self).__init__()

        self.input_dim = input_dim #pari al numero di features
        ↪  dell'istanza
        self.hidden_dim = hidden_dim #256
        self.internal_dim = int(hidden_dim / 2)
        self.dropout_p = dropout_p
        self.alpha = aplha #0.5 gamma in PROJECTION LOSS

        #Definiamo il 1 layer della rete
        #Modifica --> nn.LeakyReLU(negative_slope=0.01,
        ↪  inplace=False)

        self.layer1 = nn.Sequential(
            nn.Linear(in_features=self.input_dim,
            ↪  out_features=self.hidden_dim, bias=True),
            nn.LeakyReLU(negative_slope=0.01, inplace=False),
            nn.Dropout(p=self.dropout_p)
        )

        #Definiamo il 2 layer della rete

        self.feature = nn.ModuleList([nn.Sequential(
            nn.Linear(in_features=self.hidden_dim,
            ↪  out_features=self.internal_dim, bias=True),
            nn.LeakyReLU(negative_slope=0.01, inplace=False),
            nn.Dropout(p=self.dropout_p),
            nn.Linear(in_features=self.internal_dim,
            ↪  out_features=self.hidden_dim, bias=True),
            nn.LeakyReLU(negative_slope=0.01, inplace=False),
            nn.Dropout(p=self.dropout_p),
        ) for _ in range(num_layers)])


        self.classifier = nn.Sequential(
            nn.Linear(in_features=self.hidden_dim, out_features=2),
            nn.LogSoftmax(dim=-1)
        )
        self.lambda1 = lambda1 #0.5
        self.lambda2 = lambda2 #0.001

        self.loss_function = nn.NLLLoss(reduction='none')

        ...
```

3.3.2 Variante: max_patience

Per poter incrementare la `max_patience` sono state apportate le modifiche al file `api_test.py`. Di seguito viene riportato il codice originale e il codice modificato:
Codice originale:

```
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/../../graph_features/'
        elif dataset == 'devign':
```

```
                ds = '../../../../devign/../../graph_features/'
            else:
                raise ValueError('Imvalid 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 _ 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
```

Codice modificato:

```
        ...
        for _ in range(30):
            ...

            print("Definiamo il modello (api_test)...", file=sys.stderr,
            ↪  flush=True)
            print(args.baseline)
            if args.baseline:
                print("Svm...", file=sys.stderr, flush=True)
                model = SVMLearningAPI(True, args.baseline_balance,
                ↪  model_type=args.baseline_model) #definiamo il modello
                ↪  SVM
            else:
                print("Representation learning...", file=sys.stderr,
                ↪  flush=True)
                #Modifica
                #max_patience=5 --> io: 15
                model = RepresentationLearningModel(
                    lambda1=args.lambda1, lambda2=args.lambda2,
                    ↪  batch_size=128, print=True, max_patience=15,
                    ↪  balance=True,
                    num_layers=args.num_layers
                )
        ...
```

3.3.3 Variante: dimensione mini-batch

Per poter incrementare il numero di istanze appartenenti a un mini-batch sono state apportate le modifiche al file `api_test.py`. Di seguito viene riportato il codice modificato:

```
            ...
                for _ in range(30):
                    ...
                    print("Definiamo il modello (api_test)...", file=sys.stderr,
                    ↪ flush=True)
                    print(args.baseline)
                    if args.baseline:
                        print("Svm...", file=sys.stderr, flush=True)
                        model = SVMLearningAPI(True, args.baseline_balance,
                        ↪ model_type=args.baseline_model) #definiamo il
                        ↪ modello SVM
                    else:
                        print("Representation learning...", file=sys.stderr,
                        ↪ flush=True)
                        #Modifica
                        #batch_size= 128 --> io:200
                        model = RepresentationLearningModel(
                            lambda1=args.lambda1, lambda2=args.lambda2,
                            ↪ batch_size=200, print=True, max_patience=5,
                            ↪ balance=True,
                            num_layers=args.num_layers
                        )
            ...
```

3.3.4 Unione delle modifiche

Per poter unire tutte le modifiche illustrate nelle precedenti sottosezioni sono stati apportate le modifiche agli stessi file. Di seguito viene riportato il codice modificato:

`api_test.py`:

```
            ...
                for _ in range(30):
                    ...
                    print("Definiamo il modello (api_test)...", file=sys.stderr,
                    ↪ flush=True)
                    print(args.baseline)
                    if args.baseline:
                        print("Svm...", file=sys.stderr, flush=True)
                        model = SVMLearningAPI(True, args.baseline_balance,
                        ↪ model_type=args.baseline_model) #definiamo il
                        ↪ modello SVM
                    else:
                        print("Representation learning...", file=sys.stderr,
                        ↪ flush=True)
                        #Modifica
                        #batch_size= 128 --> io:200
                        #max_patience = 5 --> io:15
```

```
                        model = RepresentationLearningModel(
                            lambda1=args.lambda1, lambda2=args.lambda2,
                            ↪ batch_size=200, print=True, max_patience=15,
                            ↪ balance=True,
                            num_layers=args.num_layers
                        )
            ...
```

models.py:

```
            class MetricLearningModel(nn.Module):
                def __init__(self, input_dim, hidden_dim, dropout_p=0.2,
                ↪ aplha=0.5, lambda1=0.5, lambda2=0.001, num_layers=1):
                    super(MetricLearningModel, self).__init__()

                    self.input_dim = input_dim #pari al numero di features
                    ↪ dell'istanza
                    self.hidden_dim = hidden_dim #256
                    self.internal_dim = int(hidden_dim / 2)
                    self.dropout_p = dropout_p
                    self.alpha = aplha #0.5 gamma in PROJECTION LOSS

                    #Definiamo il 1 layer della rete
                    #Modifica --> nn.LeakyReLU(negative_slope=0.01,
                    ↪ inplace=False)

                    self.layer1 = nn.Sequential(
                        nn.Linear(in_features=self.input_dim,
                        ↪ out_features=self.hidden_dim, bias=True),
                        nn.LeakyReLU(negative_slope=0.01, inplace=False),
                        nn.Dropout(p=self.dropout_p)
                    )

                    #Definiamo il 2 layer della rete

                    self.feature = nn.ModuleList([nn.Sequential(
                        nn.Linear(in_features=self.hidden_dim,
                        ↪ out_features=self.internal_dim, bias=True),
                        nn.LeakyReLU(negative_slope=0.01, inplace=False),
                        nn.Dropout(p=self.dropout_p),
                        nn.Linear(in_features=self.internal_dim,
                        ↪ out_features=self.hidden_dim, bias=True),
                        nn.LeakyReLU(negative_slope=0.01, inplace=False),
                        nn.Dropout(p=self.dropout_p),
                    ) for _ in range(num_layers)])


                    self.classifier = nn.Sequential(
                        nn.Linear(in_features=self.hidden_dim, out_features=2),
                        nn.LogSoftmax(dim=-1)
```

```
35                      )
36                      self.lambda1 = lambda1 #0.5
37                      self.lambda2 = lambda2 #0.001
38
39                      self.loss_function = nn.NLLLoss(reduction='none')
40
41                      ...
```

CAPITOLO 4

Risultati

In questo capitolo vengono riportati i risultati ottenuti dalle varianti applicate al progetto ReVeal definite nel Capitolo 3 e si confrontano i risultati.

4.1 Risultati del progetto ReVeal e varianti

4.1.1 Risultati progetto ReVeal

In questa tesi è stato replicato il progetto ReVeal e come dichiarato dagli autori, si è eseguito l'addestramento ed il test del modello per 30 volte, in modo da evitare di introdurre bias dovuto alla casualità. Ad ogni iterazione è stata testata la rete neurale migliore, memorizzandone gli esiti e la loss function ottenuta durante l'addestramento. Effettuando la media delle metriche di valutazione si ottiene:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.3197	84.434	1.872	83.5398	85.4115
Precision	33.4891	33.353	1.922	32.6891	34.6108
Recall	69.9423	70.709	9.708	65.7068	75.4144
F1-score	45.1331	45.148	2.033	44.1251	46.1578

Tabella 4.1: IQR metriche di valutazione ReVeal

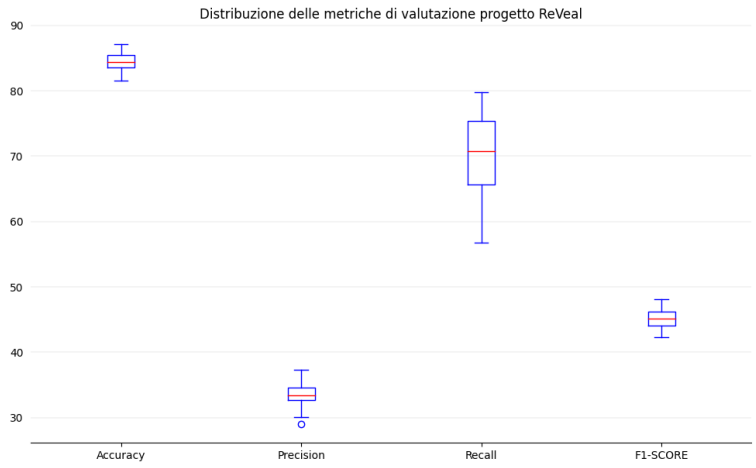

Figura 4.1: Plot metriche di valutazione ReVeal

Il miglior modello in base alla F1-score sulle 30 iterazioni viene riportato in seguito nella Figura 4.15. Dagli esiti sopra riportati si evince che il modello ha ottenuto mediamente una precision molto bassa e questo implica che la rete neurale ha classificato erroneamente molte istanze come "vulnerabili" quando in realtà sono istanze appartenenti alla classe "non vulnerabile". Per quanto riguarda la loss function, si sono ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Loss function	10357.8294	9508.103	4489.419	7772.8038	12262.2230

Tabella 4.2: IQR loss function ReVeal

In media, la funzione di perdita del modello è molto alta nonostante una buona accuracy, infatti, questo può spiegare l'alto tasso di falsi positivi e negativi ottenuti. In questo caso, l'accuracy non è molto rilevante in quanto dai risultati si evince che il modello è più propenso a riconoscere le istanze "non vulnerabili" rispetto a quelle "vulnerabili".

4.1.2 Risultati variante Leaky ReLU

Come citato nella Sezione 3.3, è stata modifica la funzione di attivazione, ovvero da ReLU a Leaky ReLU per i motivi spiegati nella Sezione 3.2. Effettuando la media delle metriche di valutazione si ottiene:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.2930	84.544	1.431	83.6361	85.0674
Precision	32.8641	32.895	2.378	32.0072	34.3855
Recall	68.3163	67.816	9.277	64.3585	73.6359
F1-score	44.2171	44.086	2.849	43.0029	45.8522

Tabella 4.3: IQR metriche di valutazione variante Leaky ReLU

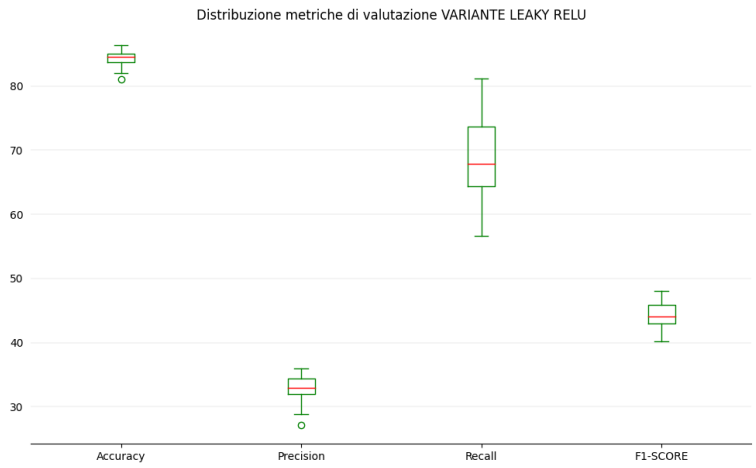

Figura 4.2: Plot metriche di valutazione variante Leaky ReLU

Il miglior modello in base alla F1-score sulle 30 iterazioni ha ottenuto la seguente matrice di confusione:

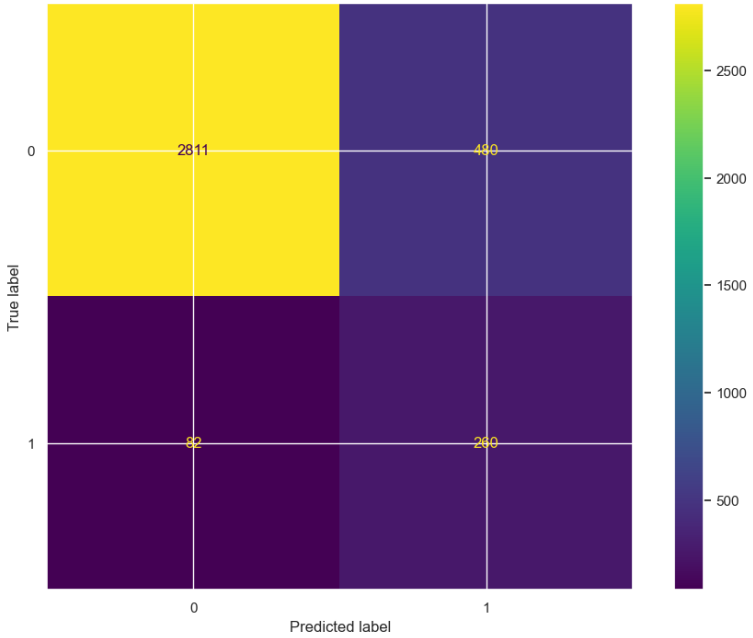

Figura 4.3: Matrice confusione miglior modello della variante Leaky ReLU

Per quanto riguarda la loss function, si sono ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Loss function	9984.4703	9370.073	3113.273	8479.4566	11592.7292

Tabella 4.4: IQR loss function Leaky ReLU

4.1.3 Risultati variante Max Patience

Come suddetto, nella Sezione 3.3 è stato modificato il parametro `max_patience`, ovvero da 5 a 15 per i motivi spiegati nella Sezione 3.2. Effettuando la media delle metriche di valutazione si ottiene:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.6564	84.985	1.06	84.3449	85.4046
Precision	33.3721	33.1517	2.45	31.8437	34.2938
Recall	67.2204	67.432	6.211	63.1430	69.3540
F1-score	44.4806	43.925	2.999	42.7243	45.7237

Tabella 4.5: IQR metriche di valutazione variante max patience

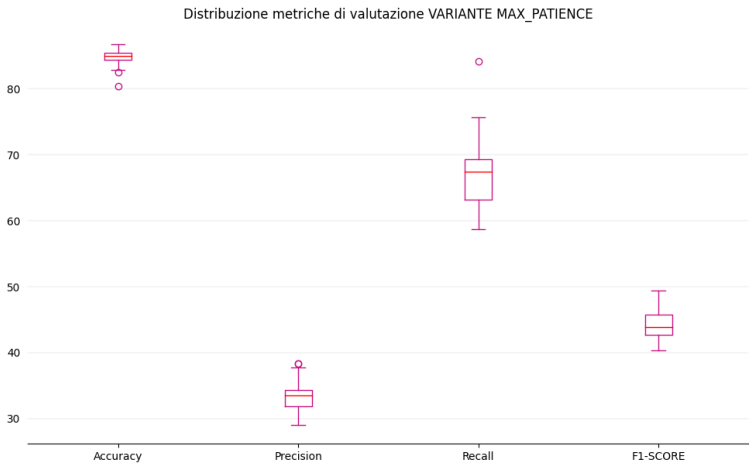

Figura 4.4: Plot metriche di valutazione variante max patience

Il miglior modello in base alla F1-score sulle 30 iterazioni ha ottenuto la seguente matrice di confusione:

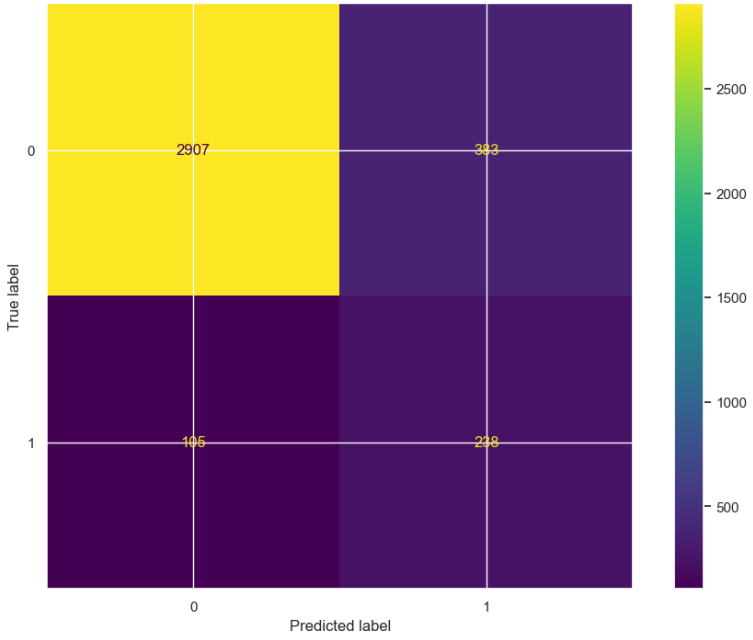

Figura 4.5: Matrice confusione miglior modello della variante max patience

Per quanto riguarda la loss function, si sono ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Loss function	9808.1569	9451.013	2689.35	7941.3721	10630.7224

Tabella 4.6: IQR loss function max patience

4.1.4 Risultati variante Mini-Batch

Come già detto nella Sezione 3.3, è stata modifica la dimensione dei mini-batch del training set, ovvero da 128 a 200 per i motivi spiegati nella Sezione 3.2. Effettuando la media delle metriche di valutazione si ottiene:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.3398	84.421	3.0	82.9824	85.9826
Precision	33.4052	33.377	2.951	32.0814	35.0320
Recall	68.9923	68.796	12.521	63.4451	75.9666
F1-score	44.7565	44.83	2.492	43.3009	45.7926

Tabella 4.7: IQR metriche di valutazione variante mini-batch

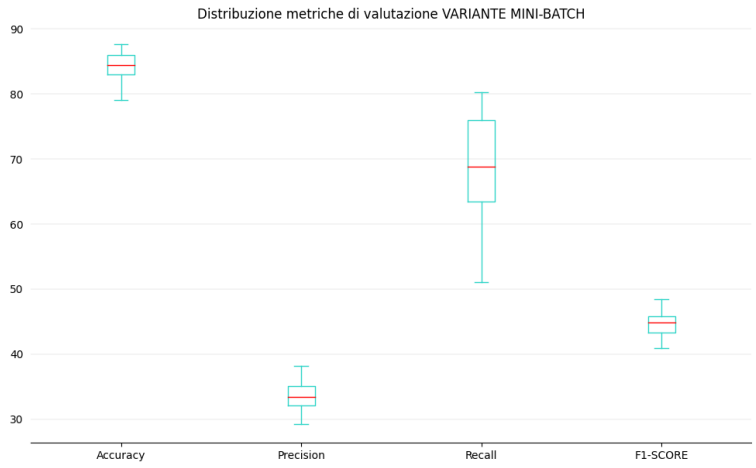

Figura 4.6: Plot metriche di valutazione variante mini-batch

Il miglior modello in base alla F1-score sulle 30 iterazioni ha ottenuto la seguente matrice di confusione:

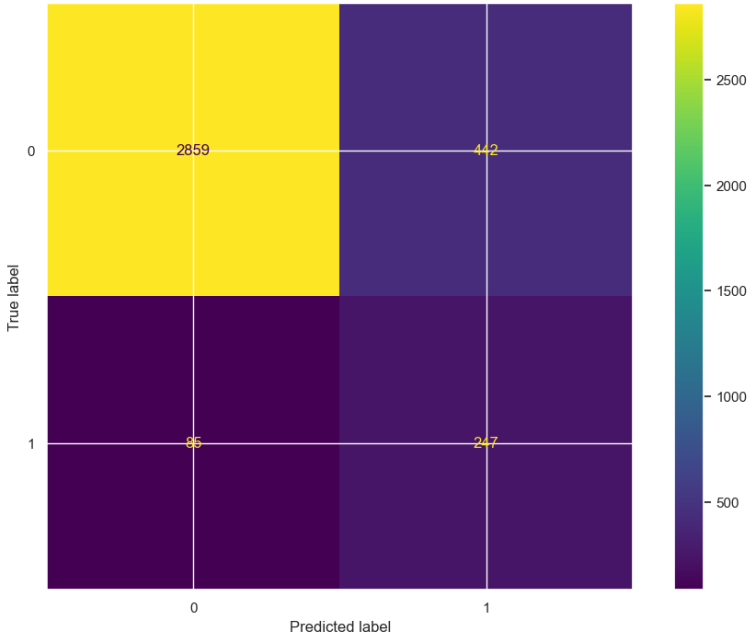

Figura 4.7: Matrice confusione miglior modello della variante mini-bacth

Per quanto riguarda la loss function, si sono ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Loss function	10936.7260	10216.789	7802.813	7633.0010	15435.8138

Tabella 4.8: IQR loss function mini-batch

Anche se i valori ottenuti sono considerevoli, esso è un risultato atteso in quanto fornendo maggiori esempi in contemporanea alla rete neurale, aumenta la probabilità che quest'ultima possa sbagliare in fase di predizione. Nonostante ciò, questo contribuisce a migliorare ulteriormente i parametri del modello durante l'addestramento anche se comporta comunque un alto tasso di falsi positivi.

4.1.5 Risultati variante 4

In questa variante sono state applicate in contemporanea tutte le modifiche illustrate precedentemente. Effettuando la media delle metriche di valutazione si ottiene:

	Media	Mediana	IQR	25° percentile	75° percentile
Accuracy	84.3536	84.393	2.023	83.2920	85.3152
Precision	33.2060	33.502	2.118	32.1781	34.2959
Recall	69.3069	70.787	7.932	65.7185	73.6506

F1-score	44.7482	44.799	2.486	43.5145	46.0002

Tabella 4.9: IQR metriche di valutazione variante 4

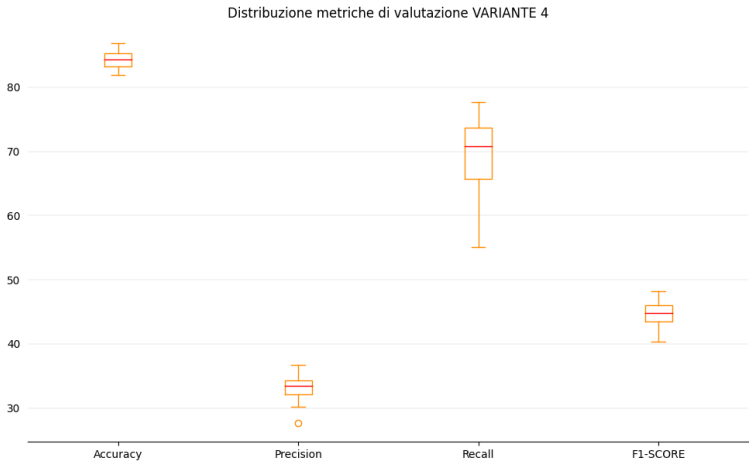

Figura 4.8: Plot metriche di valutazione variante 4

Il miglior modello in base alla F1-score sulle 30 iterazioni viene riportato in seguito nella Figura 4.14. Per quanto riguarda la loss function si sono ottenuti i seguenti risultati:

	Media	Mediana	IQR	25° percentile	75° percentile
Loss function	10466.7276	10596.583	3975.449	8411.9468	12387.3961

Tabella 4.10: IQR loss function variante4

4.2 Confronto risultati progetto ReVeal e varianti

Dopo un'analisi dei risultati ottenuti dalle singole varianti applicate alla pipeline ReVeal, confrontandoli tra loro sulla base della distribuzione sulle varie metriche si è ottenuto:

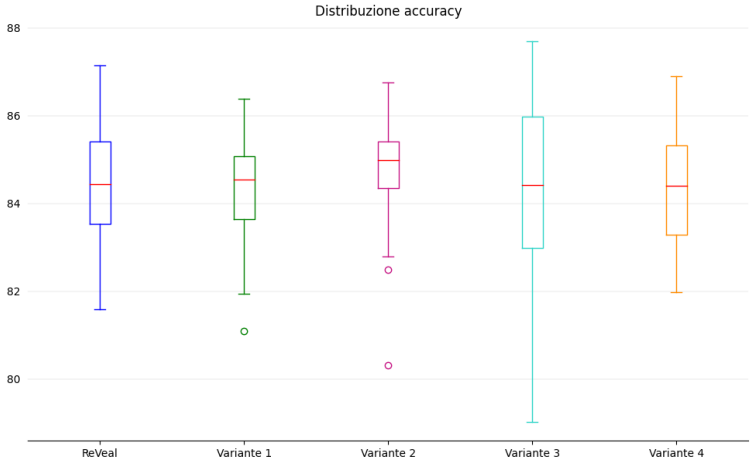

Figura 4.9: Confronto varianti sull'accuracy

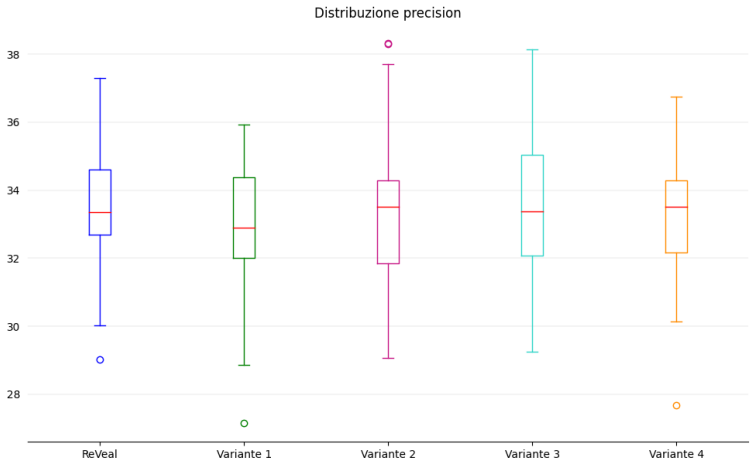

Figura 4.10: Confronto varianti sulla precision

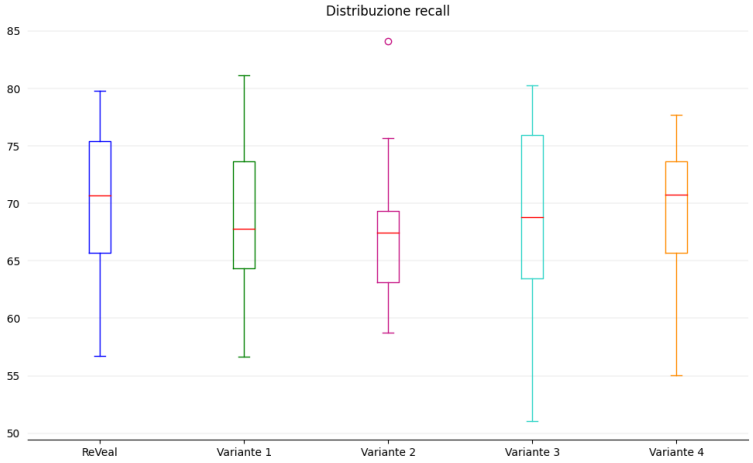

Figura 4.11: Confronto varianti sulla recall

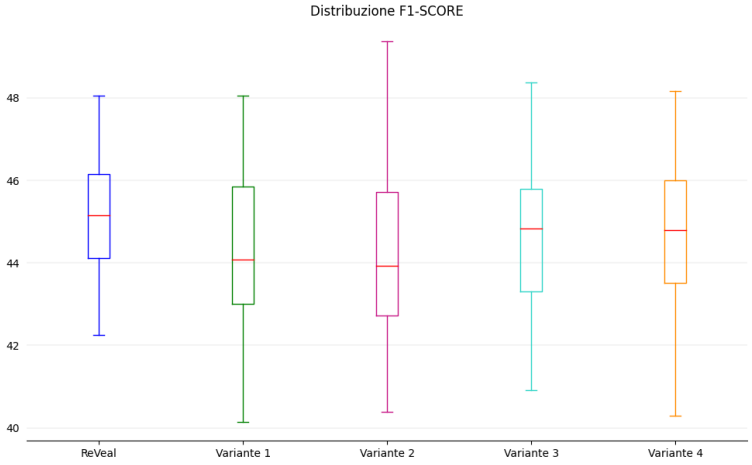

Figura 4.12: Confronto varianti sulla F1-SCORE

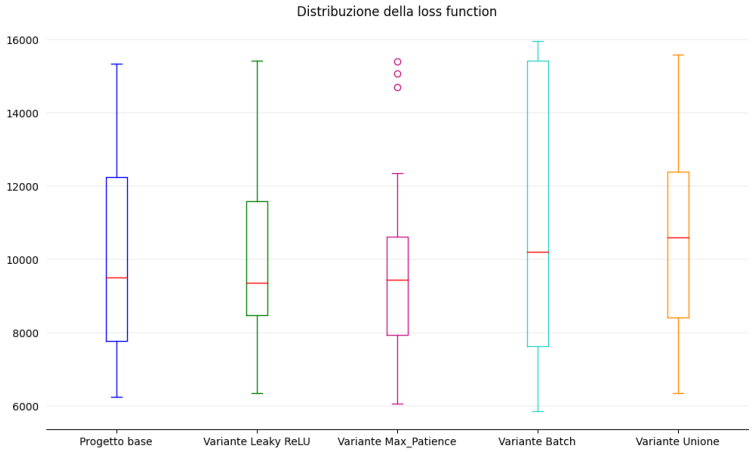

Figura 4.13: Confronto varianti sulla loss function

Dal plot delle distribuzioni delle metriche ottenute dalle varianti si evince che:

- La variante `max_patience` ottiene la miglior mediana per l'accuracy e la precision, confermando l'ipotesi che l'aumento del parametro in questione consente di incrementare la probabilità di individuare una configurazione della rete neurale migliore;
- La variante Leaky ReLU ottiene la miglior mediana per la loss function, evidenziando la proprietà della funzione di attivazione nel non disattivare i neuroni. Questo, infatti, consente di poter applicare la tecnica di Gradient Descent su più neuroni, in quanto essi non vengono disattivati e potrebbero incidere sulla predizione;
- Unendo le varianti nella variante 4, si ottengo sia vantaggi che svantaggi, infatti essa ottiene la peggior mediana per la loss function ma con un IQR inferiore rispetto alla variante dei mini-batch. Come vantaggi, invece, ottiene rispettivamente una mediana per

la precision e per la F1-score lievemente inferiore rispetto alla variante `max_patience` ed il progetto ReVeal. Inoltre, la variante 4, ottiene complessivamente una buona mediana per la recall.

Infine, confrontando la matrice di confusione del modello migliore della variante 4 con quello del progetto ReVeal (scelti sulla base della F1-score) si evince, così come riportato dalle figure di seguito che, il primo ha un tasso di true negative più elevato ed un tasso di false positive più basso rispetto al secondo, mentre ReVeal ha un tasso più basso di false negative ed un tasso più alto di true positive.

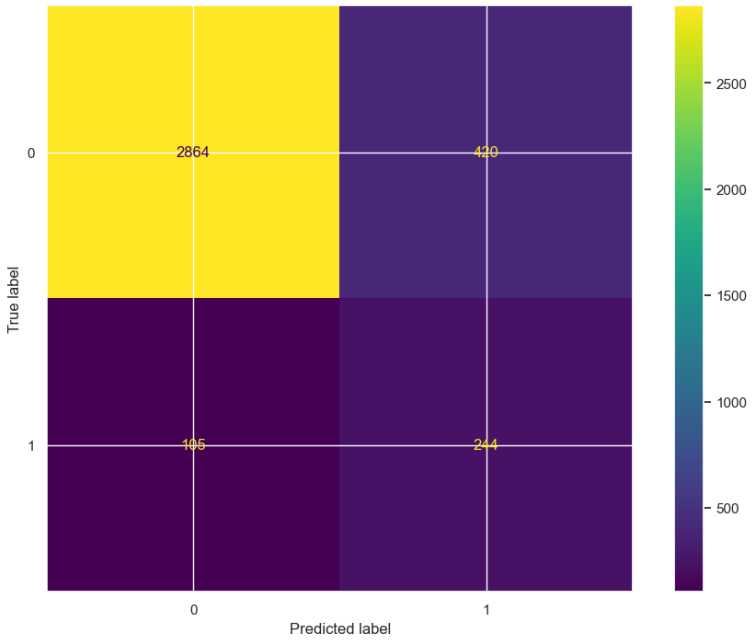

Figura 4.14: Matrice confusione miglior modello della variante 4

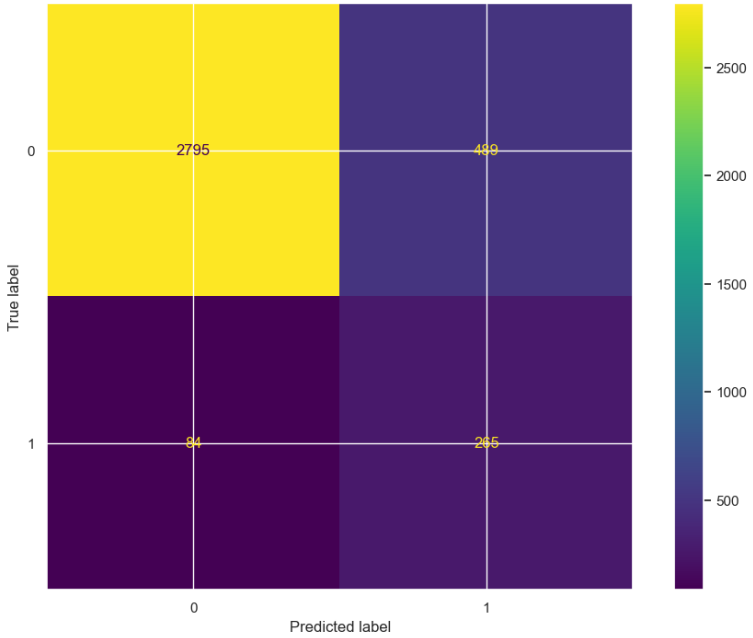

Figura 4.15: Matrice confusione miglior modello di ReVeal

Riassunto risultati ottenuti

In generale le varianti proposte hanno ottenuto lievi miglioramenti sulle metriche di valutazione rispetto al progetto ReVeal. In particolare, la variante `max_patience` ha ottenuto il miglio punteggio per accuracy e precision, la variante Leaky ReLU ha ottenuto il miglior punteggio per la loss function e la variante 4 ha ottenuto il miglior punteggio per la recall. Per quanto riguarda il tasso di falsi positivi, tutte le varianti e il progetto ReVeal hanno superato la soglia dei 400 mentre per il tasso di true negative tutti i modelli hanno ottenuto ottimi risultati.

CAPITOLO 5

Conclusioni

In questo capitolo si analizzano i risultati ottenuti dalle varie varianti da un punto di vista pratico ed infine si propongono diversi spunti che potrebbero essere utilizzati in futuro per migliorare i modelli nella predizione delle vulnerabilità software.

5.1 Riflessione sui risultati ottenuti

Dagli esiti ottenuti dalle varianti e dai confronti effettuati nelle Sezioni 4.1 e 4.2 si evincono vari problemi, infatti, il progetto ReVeal, così come le varianti proposte, hanno un alto tasso di falsi positivi e falsi negativi. Nel primo caso, si supera la soglia di 400, mentre nel secondo caso i valori si aggirano tra gli 80 e i 100. Questo è dovuto:

- in parte al problema che è di natura sbilanciato, infatti si può notare che tutti i modelli riconoscono maggiormente la classe di maggioranza ("non vulnerabile") ottenendo così un ottimo valore di true negative e di accuracy. Proprio per questo, non bisogna affidarsi ciecamente a quest'ultima metrica in fase di valutazione;
- in parte da un dataset bilanciato tramite la tecnica SMOTE che realizza istanze vulnerabili sintetiche addestrando quindi la rete su dati fittizi;

Da questi esiti, si deduce che i vari modelli analizzati, non apprendono correttamente le feature che caratterizzano le istanze vulnerabili, ottenendo così pessimi risultati per precision, false positive e false negative. C'è da notare che, grazie ad alcune modifiche, è stato possibile migliorare lievemente alcune metriche come la precision nella variante `max_patience` ed un forte decremento della loss function nella variante Leaky ReLU. Proprio per questi motivi, si è deciso di unire le modifiche nella variante 4, ottenendo un lieve incremento della mediana della recall rispetto a ReVeal, una mediana per la precision prossima a quella della variante `max_patience` ma ottenendo la più alta mediana per la loss function ereditata dalla variante in cui si modifica la dimensione dei mini-batch. In conclusione, è possibile affermare che la modifica degli hyper-parameter di un modello, impattano significativamente in fase di predizione, quindi bisogna approfondire la configurazione di questi parametri in studi futuri.

5.2 Sviluppi futuri

Al fine di migliorare ulteriormente i risultati ottenuti, bisogna ampliare questo studio. In particolare, si propone di prestare maggiore attenzione sul pre-processing dei dati, concentrandosi ad esempio sulla configurazione del modello Word2Vec che consente di codificare il testo in un vettore embeddings. Nel progetto ReVeal, questa tecnica viene utilizzata configurando molti hyper-parameter con valori di default, quindi, una modifica più accurata di quest'ultimi potrebbe incidere significativamente sulle performance del modello. Un'ulteriore tecnica che potrebbe essere utilizzata per la codifica del codice è Code2Vec che rispetto a Word2Vec è più orientato al linguaggio di programmazione. Quest'ultimo prende in input il codice sorgente, la corrispettiva etichetta ed il parser del linguaggio in cui è scritto il programma e fornisce in output un **code vector**, ovvero un vettore che rappresenta il codice. Grazie a questa tecnica, il modello potrà riconoscere vulnerabilità in programmi scritti in diversi linguaggi di programmazione in quanto viene utilizzato il parser [33]. Nel progetto ReVeal, invece, il modello viene addestrato su codice C e questo consente di avere una rete neurale in grado di elaborare e riconoscere vulnerabilità solo su istanze scritte in C. In generale, la tecnica di Code2Vec ha il seguente funzionamento [33]:

- Definisce l'Abstract Syntax Tree del codice di input;
- Naviga l'AST in modo da definire i vari path del grafo;
- La composizione di ogni path e i valori delle foglie che esso connette, vengono rappresentate da una tupla chiamata **path-context**;
- Ogni componente del path-contex verrà trasformato in una rappresentazione vettoriale;
- Questi tre vettori verranno concatenati in modo da realizzare un unico vettore (**context vector**) che rappresenta l'intero path-context;
- Il modello tramite l'hidden layer comprime il context vector realizzando un altro vettore, ovvero il **combined context vector**;
- Il modello assegnerà ad ogni path-context un peso che indicherà l'importanza di quel percorso e questo determinerà anche la larghezza del cammino. Questi pesi sono definiti in un vettore chiamato **attention vector** ed i valori che esso contiene verranno appresi dalla rete neurale durante l'addestramento;

- Infine, il modello andrà ad aggregare tutte le rappresentazioni vettoriali dei path-context in un unico vettore che rappresenterà l'intero codice dato in input. Così come l'attention vector, quest'ultima rappresentazione vettoriale verrà man mano migliorata durante l'addestramento.

Per poter migliorare ulteriormente i risultati in questo caso di studio, i futuri lavori, dovranno incrementare la quantità di dati di addestramento, in particolar modo inserendo nel dataset codici di diversi linguaggi di programmazione e provenienti dal mondo reale in modo tale che il modello possa riconoscere vulnerabilità in diversi linguaggi ed apprendere feature di vulnerabilità reali. Per effettuare ciò, si richiede alle aziende tech di rendere il loro codice open-source in modo tale da poterlo utilizzare per l'addestramento della rete neurale ed inoltre questo consente all'azienda di ricevere proposte di miglioramento del prodotto software grazie a una community esperta nel settore.

Ringraziamenti

Ringrazio il mio relatore Fabio Palomba che mi ha guidato nella fase più importante del mio percorso accademico e mi ha fatto appassionare al fantastico mondo dell'intelligenza artificiale. Vorrei ringraziare i miei genitori e mia sorella che mi hanno sempre supportato e incoraggiato in questo percorso ed hanno sempre creduto in me. Ringrazio tutti i miei familiari, in particolar modo ringrazio i miei nonni: Giuseppe Trezza e Lucia Padovano Sorrentino che hanno contribuito a rendere me l'uomo che sono oggi. Ringrazio i miei compagni di avventura, Alfonso Califano (Alfonso JJ) e Simone Della Porta (personal trainer) con i quali ho trascorso questi bellissimi sei anni fatti di divertimento, studio matto fino a tarda sera e stupidaggini. Ringrazio il magnifico team RAAF-GAMING, ovvero Antonio De Lucia, Antonio Maddaloni e Francesco Peluso che mi hanno accompagnato in questi tre anni fatti di risate e di imprecazioni contro il codice (come da veri programmatori) ma realizzando sempre le nostre idee con entusiasmo ed orgoglio. Ringrazio i miei amici che ho incontrato lungo il percorso, ovvero Luigi Emanuele Sica (PP) e Francesco Ciccone che sono sempre stati disponibili e mi hanno contagiato positivamente con la loro curiosità. Ringrazio Gerardo Di Pascale, Francesco Vece (Bandana man) e Manuel Di Matteo che, anche se ho conosciuto al termine di questo percorso mi hanno sempre aiutato e strappato un sorriso. Ringrazio gli amici di una vita, ovvero Vincenzo Di Serio (esperto di bisturi) e Ferdinando Cuomo che ho sempre considerato come fratelli con i quali son cresciuto e fatto le peggiori stupidaggini. Ringrazio tutti riservando una dedica particolare a me stesso per il lavoro svolto e l'ambito traguardo raggiunto. Infine, ho riservato i ringraziamenti più speciali alla mia metà, Desirée Sissi Russo che è sempre stata al mio fianco, mi ha supportato e sopportato in qualsiasi momento illuminando anche i miei momenti bui. Grazie per aver condiviso con me, con tanta pazienza questo ambito progetto. Grazie per l'amore che sei.

Bibliografia

[1] "Report attacchi informatici clusit." Disponibile al link: `https://clusit.it/pubblicazioni/`. (Citato alle pagine iv e 2)

[2] "Second annual cost of cyber crime study – benchmark study of u.s. companies." Disponibile al link: `https://securityaffairs.co/wordpress/4631/cyber-crime/analysis-of-cybercrime-and-its-impact-on-private-and-military-sectors.html`. (Citato alle pagine iv e 3)

[3] P. K. Shamal, K. Rahamathulla, and A. Akbar, "A study on software vulnerability prediction model," in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 703–706, 2017. (Citato alle pagine iv, 15, 17, 18, 19 e 20)

[4] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, (New York, NY, USA), Association for Computing Machinery, 2010. (Citato alle pagine iv e 18)

[5] A. Geron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. (Citato alle pagine iv, v, 15, 21, 22, 23, 24, 25, 26, 27, 28, 34, 56 e 57)

[6] W. Farsal, S. Anter, and M. Ramdani, "Deep learning: An overview," in *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*, SITA'18, (New

York, NY, USA), Association for Computing Machinery, 2018. (Citato alle pagine iv, 21 e 23)

[7] "Operazioni effettuate all'interno di un layer." Disponibile al link: `https://www.youtube.com/watch?v=aircAruvnKk&t=7s`. (Citato alle pagine iv e 25)

[8] L. Gerling and K. Schmid, "Variability-aware semantic slicing using code property graphs," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, (New York, NY, USA), p. 65–71, Association for Computing Machinery, 2019. (Citato alle pagine iv, 36 e 37)

[9] A. A. Mohammed and V. Umaashankar, "Effectiveness of hierarchical softmax in large scale classification tasks," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 1090–1094, 2018. (Citato alle pagine iv, 42 e 43)

[10] "Latent space." Disponibile al link: `https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d`. (Citato alle pagine iv e 47)

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. (Citato alle pagine iv, 20, 31, 32, 38, 45, 47 e 50)

[12] "Costi annuali dovuti alla criminalità informatica." Disponibile al link: `https://www.europarl.europa.eu/news/it/headlines/society/20211008STO14521/perche-la-sicurezza-informatica-e-importante-per-l-ue`. (Citato a pagina 2)

[13] "Hardware vulnerability." Disponibile al link: `https://www.techtarget.com/whatis/definition/hardware-vulnerability`. (Citato a pagina 6)

[14] 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, 2012. (Citato alle pagine 6 e 13)

[15] T. Feglar and J. K. Levy, "Protecting cyber critical infrastructure (cci): Integrating information security risk analysis and environmental vulnerability analysis," in *2004 IEEE International Engineering Management Conference (IEEE Cat. No. 04CH37574)*, vol. 2, pp. 888–892, IEEE, 2004. (Citato a pagina 6)

[16] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: Attacking arm pointer authentication with speculative execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), Association for Computing Machinery, 2022. (Citato a pagina 7)

[17] "Poodle protocol vulnerability." Disponibile al link: `https://www.cisa.gov/uscert/ncas/alerts/TA14-290A`. (Citato a pagina 8)

[18] B. Möller, T. Duong, and K. Kotowicz, "This poodle bites: exploiting the ssl 3.0 fallback," *Security Advisory*, vol. 21, pp. 34–58, 2014. (Citato a pagina 8)

[19] "Heatwave forced google and oracle to shut down computers." Disponibile al link: `https://www.bbc.com/news/technology-62202125`. (Citato a pagina 8)

[20] H. Orman, "The morris worm: a fifteen-year perspective," *IEEE Security Privacy*, vol. 1, no. 5, pp. 35–43, 2003. (Citato a pagina 9)

[21] "Amsi windows." Disponibile al link: `https://docs.microsoft.com/it-it/windows/win32/amsi/antimalware-scan-interface-portal`. (Citato a pagina 10)

[22] "Local privilege escalation vulnerability fixed in eset products for windows." Disponibile al link: `https://support.eset.com/en/ca8223-local-privilege-escalation-vulnerability-fixed-in-eset-products-for-windows`. (Citato a pagina 10)

[23] L. Zhang, X. Deng, and Y. Wang, "Shellshock bash vulnerability modeling analysis based on petri net," in *2021 International Conference on Networking and Network Applications (NaNA)*, pp. 242–247, 2021. (Citato a pagina 10)

[24] S. Kuraku and D. Kalla, "Emotet malware—a banking credentials stealer," *Iosr J. Comput. Eng*, vol. 22, pp. 31–41, 2020. (Citato a pagina 11)

[25] G. Larsen, E. Fong, D. A. Wheeler, and R. S. Moorthy, "State-of-the-art resources (soar) for software vulnerability detection, test, and evaluation," tech. rep., INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA, 2014. (Citato a pagina 11)

[26] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 771–781, 2012. (Citato a pagina 12)

[27] 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, 2006. (Citato a pagina 13)

[28] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, MetriSec '12, (New York, NY, USA), p. 7–10, Association for Computing Machinery, 2012. (Citato a pagina 20)

[29] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 23–33, 2014. (Citato a pagina 20)

[30] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, (Valletta, Malta), pp. 45–50, ELRA, May 2010. `http://is.muni.cz/publication/884893/en`. (Citato a pagina 38)

[31] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015. (Citato a pagina 44)

[32] M. D. Smucker, J. Allan, and B. Carterette, "A comparison of statistical significance tests for information retrieval evaluation," in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, (New York, NY, USA), p. 623–632, Association for Computing Machinery, 2007. (Citato a pagina 51)

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, pp. 40:1–40:29, Jan. 2019. (Citato a pagina 81)