UNIVERSITÀ DI PISA

Master in Cybersecurity

Project Work

Categorizzazione di malware Android tramite l'impiego di un'architettura basata su transformer

Studente

Giuseppe Floris

Tutor:

Andrea Saracino

Edizione 21/22

Indice

Elenco delle figure

Introduzione

I malware Android sono una minaccia crescente per la sicurezza dei dispositivi mobili.Dispositivi la cui corruzione diventa sempre più appetibile alla luce del fatto che il loro utilizzo è sempre più massivo e sempre più spesso parte fondamentale di alcuni sistemi di sicurezza. Pensiamo al suo utilizzo come mezzo principale per la MFA (Multi Factor Autorizzazion) che di fatto rende i nostri smartphone il punto ultimo per l'autenticazione per l'accesso a servizi fondamentali come le banche o ai più vari servizi pubblici tramite SPID. In quest'ottica si rende sempre più urgente sviluppare nuovi metodi che permettano di individuare e comprendere la natura dei malware riuscendo a superare le tecniche con le quali di volta in volta questi vengono occultati, basti pensare alla non rara presenza di malware anche sui canali di distribuzione ufficiale quali play store [1].

In questo project work si utilizzerà BERT (Rappresentazioni Bidirezionali da Trasformatori per Encoder) per l'individuazione e la categorizzazione dei malware attraverso l'analisi delle chiamate API Android. A permetterci l'estrazione delle chiamate alle API, tramite un opportuno filtro, sarà Androguard. Strumento che permette di estrarre un grafo dal quale carpire l'interazione delle applicazioni con il dispositivo, permettendo di identificare schemi o comportamenti sospetti. Analizzando tali comportamenti si andrà ad individuare e classificare la presenza e la tipologia di eventuali malware.

Individuazione (Detection) e classificazione avverranno tramite il fine tuning di un modello preadestato: BERT, questo approccio ci consentirà di rendere più dinamica l'individuazione dei malware rispetto ai tipici approcci basati sull'analisi delle firme, inoltre ci consentirà di sfruttare tutta la potenza di modelli come BERT nell'analisi del linguaggio e nella capacità di carpire dipendenze a lungo termine tra le diverse componenti del testo, che in questo caso rappresenteranno le chiamate alle API Android.

[1] https://github.com/DoctorWebLtd/malware-iocs/blob/master/Android.Spy.SpinOk/README.adoc

Capitolo 1

Machine learning

1.1 Tecniche per il machine learning

L'apprendimento automatico o Machine Learning (ML) è un campo dell'intelligenza artificiale che si dedica alla creazione di modelli e algoritmi capaci di permettere alle macchine di prendere decisioni ed eseguire task senza essere esplicitamente programmate per farlo. La costruzione di un modello inizia con la fase di training, durante la quale il modello apprende estraendo informazioni da un vasto set di dati noto come training set. Parallelamente, si svolge una fase di validazione del modello, che di solito avviene utilizzando una porzione separata del training set (attraverso metodi come la K-fold cross validation).

I risultati della validazione vengono poi utilizzati per valutare l'accuratezza del modello. Se l'accuratezza non è soddisfacente, si procede alla modifica dei parametri del modello stesso e si ripete la fase di training. Successivamente, si passa alla fase di testing, che ha lo scopo di verificare la capacità del modello di generalizzare su nuovi dati, assicurandosi che non soffra di overfitting.

L'overfitting si verifica quando un modello si adatta troppo bene ai dati di training, a scapito della sua capacità di interpretare nuovi dati, inclusi quelli del set di test. D'altro canto, un dataset di dimensioni ridotte o un addestramento inadeguato possono portare all'underfitting, ovvero all'incapacità del modello di stabilire una relazione tra i dati di input e di output. Per prevenire l'underfitting, è essenziale disporre di un set di dati di dimensioni adeguate e di un modello adeguatamente complesso per la natura del problema. Ad esempio, utilizzare modelli lineari per approssimare dati non lineari creerà un modello incapace di identificare relazioni tra input e output. Al contrario, un modello troppo

complesso rischia di apprendere anche il rumore presente nei dati di training, perdendo così la capacità di generalizzare bene su nuovi dati.

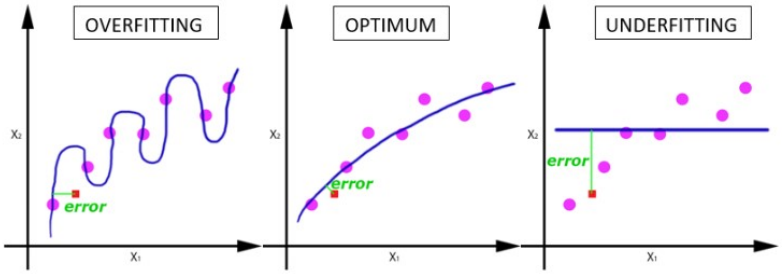


Figura 1.1: Rappresentazione di overfitting e underfitting dei dati

Per evitare i problemi sopracitati, oltre all'aumento della dimensione del data set (Fig.1.1), possiamo adottare diverse tecniche:

- Rimuovere feature (classi) non rilevanti, undersampling.
- Eseguire uno stop anticipato del processo di training quando i risultati della validation rispettano determinati parametri, questo permette che il training del modello si fermi prima di andare in overfitting, early stopping.
- regolarizzare i dati allo scopo di minimizzare i pesi nel modello, contribuendo così a stabilizzarlo e prevenire un eccessivo aumento delle sue dimensioni. È necessario aggiungere un termine di regolarizzazione alla funzione di errore, la cui minimizzazione è, ricordiamolo, l'obiettivo principale del processo di addestramento.

Figura 1.2: Andamento desiderato dell'errore sul modello

- Usare la K-fold cross validation, ovvero suddividere il training set in varie parti da utilizzare per validare il modello e quindi evitare che i dati di validazione combacino con quelli training set (Fig. 1.3).

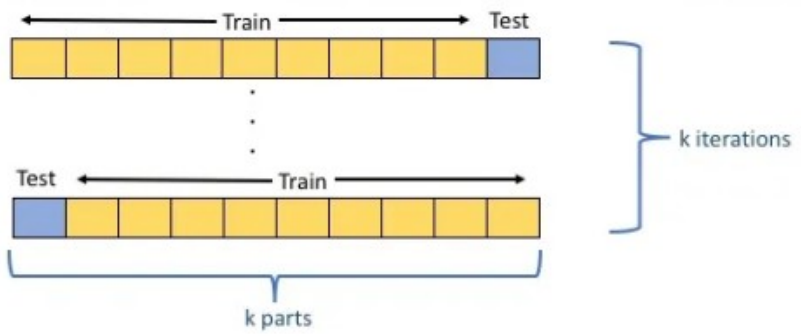


Figura 1.3: K-fold cross validation

1.1.1 Tipi apprendimento

Gli algoritmi da utilizzare per permettere il training di un modello possono essere classificati il tre tipologie:

- **Apprendimento supervisionato** In questa categoria abbiamo una serie di dati già etichettati, ovvero degli input a cui viene associato il rispettivo output, nel nostro caso avremo un'elaborazione del grafo dell'APK (Pacchetto installazione applicazioni Android.) con una label che indicherà la natura dell'APK stessa. L'obbiettivo è sempre quello di trovare la funzione che minimizzi l'errore tra modello e output. questa tipologia è utilizzata per risolvere problemi:
  - Classificazione: predire la classe di appartenenza dell'oggetto in input.
  - Regressione: trovare il valore numerico
- **Apprendimento non supervisionato** In questa categoria non abbiamo dei dati per i quali è già definita la classe di output corretta. Sarà l'algoritmo a trovare tutte le correlazioni tra i dati in modo autonomo. Questa tipologia è usata negli algoritmi di clustering dove per l'appunto si cercano similitudini tra diversi gruppi. Tale tecnica è più veloce dell'apprendimento supervisionato in quanto non è necessario avere dei dati già etichettati.
- **Apprendimento per rinforzo** In questa categoria non abbiamo l'esigenza di estrarre delle informazioni dai dati in input ma andiamo a interagire, in modo dinamico, con l'ambiente tramite un meccanismo di premi o punizioni a seconda del risultato ottenuto ad ogni azione. Questa tecnica trova impiego nei videogiochi e più in generale nei sistemi dedicati al problem solving

1.2 Reti Neurali

1.2.1 Cenni storici

Una rete neurale è un modello computazionale le cui componenti di base si ispirano al funzionamento dei neuroni umani, una rete quindi si pone come obiettivo quello di simulare alcuni aspetti del funzionamento del cervello umano. L'unità di base delle reti neurali è il neurone artificiale, ossia un modello matematico che imita il comportamento di un neurone biologico, ricevendo dei segnali in input, elaborandoli attraverso una funzione matematica e producendo un segnale in output.

Il primo modello di neurone artificiale fu presentato da Walter Pitts e Warren McCulloch nel 1943 [1]. La sua struttura semplice permetteva l'elaborazione di soli dati binari. Nel 1959, lo psicologo Frank Rosenblatt [2] propose il Perceptron, un modello più complesso rispetto al suo predecessore, con uno strato di input e uno di output. Questa struttura più complessa gli permetteva di riconoscere e classificare forme.

Nel 1974 venne introdotto il Multi-Layer Perceptron (MLP), una rete neurale che includeva strati nascosti, cioè strati intermedi di neuroni situati tra quelli di input e di output. Questa struttura permetteva di catturare un maggior numero di caratteristiche dai dati in input.

Il punto di svolta si ebbe nel 1986 con l'introduzione dell'algoritmo di backpropagation dell'errore (sviluppato da Rumelhart, Hinton e Williams) [3]. Questo algoritmo consente alla rete neurale di migliorare le sue prestazioni attraverso il confronto tra l'output prodotto e quello desiderato, e la contestuale correzione dei pesi.

1.2.2 Funzionamento neurone

L'input di un neurone è costituito dai segnali provenienti da altri neuroni [X1,...,Xn]. Ogni segnale è associato a un peso, che è un parametro appreso durante l'addestramento della rete e che riflette l'importanza di quel particolare input.

L'unità di calcolo del neurone calcola la somma pesata degli input, alla quale viene applicata una funzione di attivazione. Questa funzione trasforma la somma pesata e produce l'output del neurone, che può variare a seconda della funzione utilizzata.

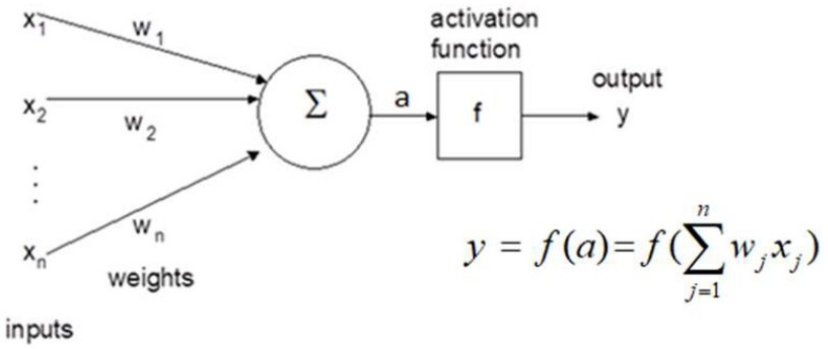


Figura 1.4: Modello neurone

Le reti neurali attuali sono composte da più strati: uno strato di input, uno o più strati nascosti e uno strato di output. Quando una rete ha più di uno strato nascosto, si parla di deep learning.

1.2.3 Deep Learning

Il deep learning impiega reti neurali profonde, solitamente caratterizzate da due o più strati nascosti. Gli strati più vicini all'input tendono a estrarre informazioni di basso livello, come bordi, angoli e texture in caso di immagini. Al contrario, gli strati più vicini all'output elaborano informazioni di alto livello, individuando dettagli e pattern più complessi.

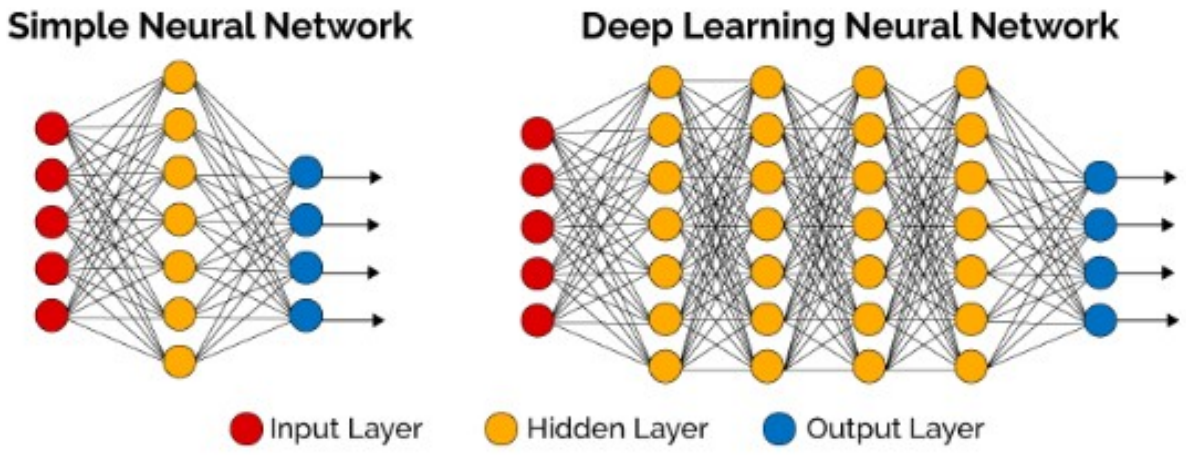


Figura 1.5: Confronto tra una Rete Neurale Semplice e una Rete Neurale di Deep Learning

A differenza del machine learning tradizionale, il deep learning può gestire dati meno strutturati e non richiede necessariamente una selezione manuale delle caratteristiche. Tuttavia, richiede grandi quantità di dati e risorse computazionali per evitare l'overfitting,

in particolare in scenari con modelli molto complessi. Questa flessibilità rende il deep learning particolarmente adatto a problemi che coinvolgono dati non strutturati o semi-strutturati, come il riconoscimento di immagini, video, voce e il trattamento del linguaggio naturale (NLP).

Capitolo 2

Natural Language Processing

2.1 Cosa si intende per NLP?

Il Natural Language Processing (NLP) è uno dei rami dell'intelligenza artificiale che ha come obiettivo quello di costruire sistemi capaci di interpretare, comprendere, ma anche generare uno o più linguaggi per interagire con gli esseri umani. In questo ambito si collocano i traduttori automatici più evoluti e le sempre più diffuse IA generative, come ChatGPT e Google Bard.

L'utilizzo degli strumenti di NLP non si limita, come evidenziato in precedenza, solamente alla generazione di testo. Infatti, possiamo utilizzarli anche per classificare input testuali di varia origine. Nel seguito del project work, verrà illustrato come estrarre informazioni da un'APK Android e rielaborarle in forma testuale.

Questo testo potrà, poi essere classificato utilizzando strumenti di elaborazione del linguaggio naturale.

2.2 Encoder Decoder

Encoder e Decoder sono due componenti, generalmente delle reti neurali, fondamentali in molte architetture di machine learning e offrono ottime prestazioni nell'ambito del NLP. L'encoder prende in input dei dati, ad esempio delle sequenze di parole, e li trasforma in rappresentazioni vettoriali. Questa rappresentazione è solitamente di dimensioni ridotte rispetto all'input e contiene le informazioni considerate più rilevanti.

Il Decoder, invece, opera un processo inverso: prende in input la rappresentazione vettoriale proveniente dall'output dell'encoder ed elabora questi dati con l'obiettivo di ricreare i dati originali o una loro trasformazione.

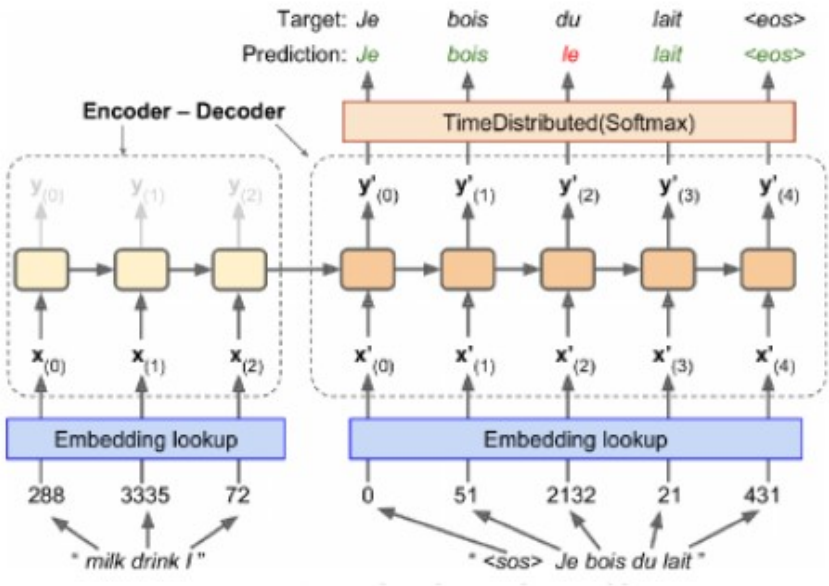


Figura 2.1: Esempio di architettura encoder-decoder utilizzata per la traduzione di una frase

Due delle principali sfide di questa architettura riguardano la difficoltà nell'elaborare sequenze di input di grandi dimensioni e nella gestione delle dipendenze a lungo termine tra i dati, aspetto fondamentale in operazioni come la traduzione di testi o nell'elaborazione di grandi volumi di dati. Una soluzione a questi problemi è stata introdotta con il meccanismo dell'attenzione, che vedremo tra poco.

2.3 Meccanismo attenzione

Il meccanismo dell'attenzione permette di focalizzarsi su alcune parti dei dati forniti in input anziché sull'intero blocco di informazioni. Questo è possibile attraverso l'attribuzione di pesi ai dati in input, determinando la loro importanza in relazione all'output da produrre.

Dal punto di vista matematico, possiamo descrivere il meccanismo dell'attenzione considerando:

- Q: matrice delle query di dimensione: numero query X dimensione chiavi.

- K: matrice delle chiavi di dimensione: numero chiavi X dimensione chiavi.

- V: matrice dei valori di dimensione: numero chiavi X dimensione valori

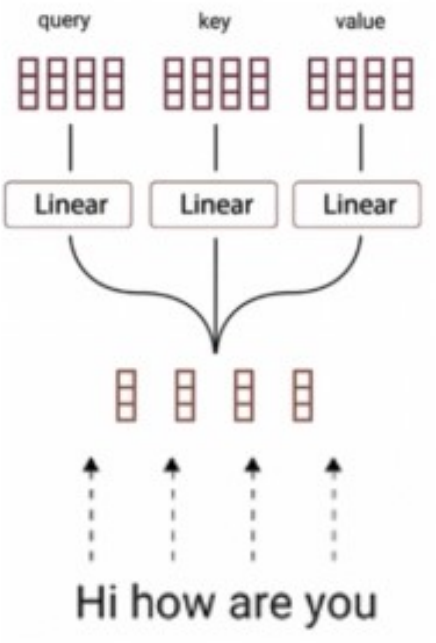


Figura 2.2: Rappresentazione grafica del processo di calcolo dell'attenzione

Dove per "dimensione" si intende la grandezza dei vettori utilizzati per la rappresentazione di chiavi e valori. Le matrici Q, K e V rappresentano rispettivamente:

- Q (Query) rappresenta cosa si cerca di enfatizzare.

- K (Keys) aiuta a determinare dove concentrare l'attenzione.

- V (Values) determina quali informazioni includere nell'output dell'attenzione.

Una volta create le matrici a partire dall'input, si procede prima con il prodotto tra le matrici Q e K (la sua trasposta) che ci fornisce una matrice degli scores ovvero un'indicazione su quelle che sono le parole che hanno maggior peso, quindi importanza. Dopo una normalizzazione della matrice degli scores si procede con il prodotto tra questa e la matrice V ottenendo così gli output desiderati (Z in Fig. 2.3).

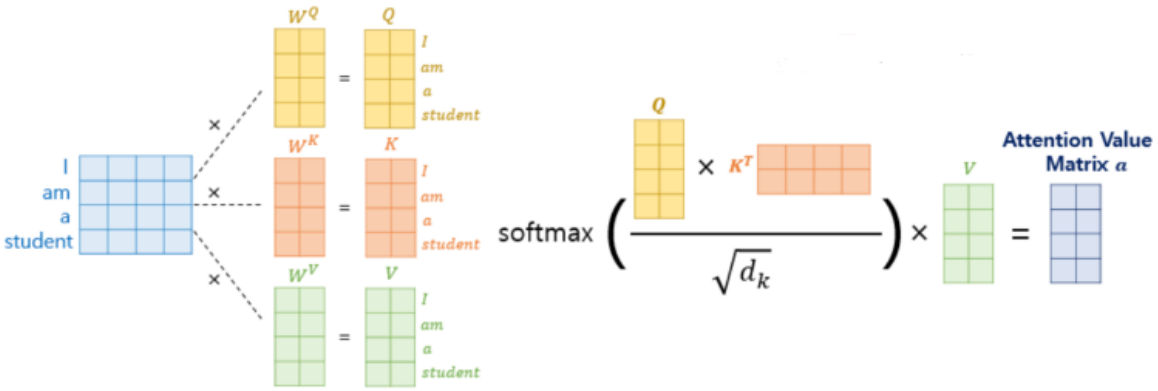


Figura 2.3: encoder decoder

2.4 Modelli Transformer

2.4.1 Transformer

Il transformer è modello di deep learning basato sull'attenzione introdotto per la prima volta nel 2017 da Vaswani et al. nella loro pubblicazione "Attention Is All You Need" [4] e rappresentano il motore dell'IA generativa, grazie alla loro capacità di catturare in modo efficace le dipendenze a lungo raggio tra le varie parole. Il transformer è adatto a processare testi e ha rapidamente guadagnato popolarità nell'elaborazione del linguaggio naturale (NLP), dove ha dimostrato di risolvere in modo più efficiente e migliore rispetto ai suoi predecessori in problemi come: la traduzione automatica, rilevamento di entità e sintesi di testo. I transformer sono costituiti da due componenti principali: un encoder e un decoder (Fig. 2.4). L'encoder è responsabile della codifica dell'input di una sequenza, mentre il decoder è responsabile della decodifica dell'output di una sequenza. L'encoder è responsabile della trasformazione dell'input in uno stato nascosto che può essere utilizzato dal decoder per generare l'output. L'encoder è costituito da una serie di Multi-Head Attention layer (Fig 2.5 a) e moduli di feed forward.

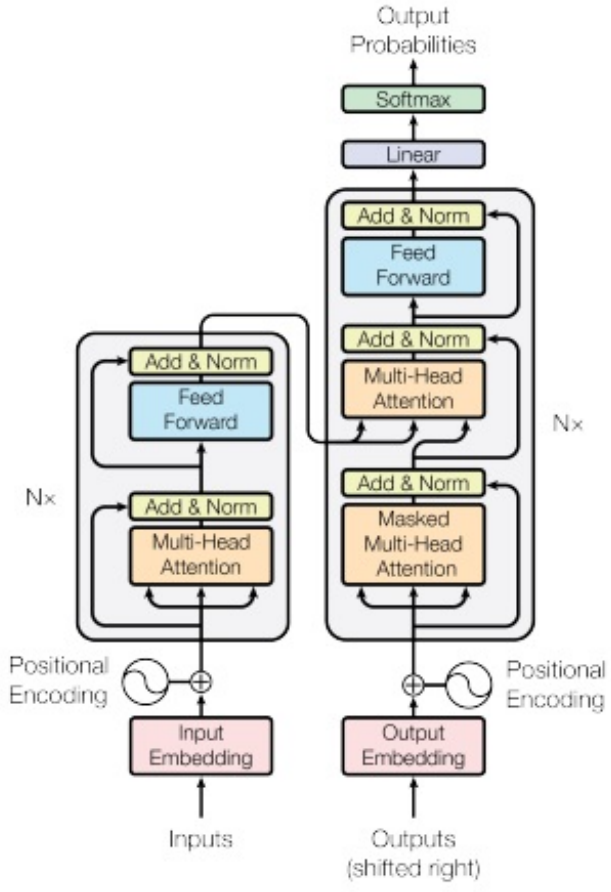


Figura 2.4: Transformer - modello architetturale

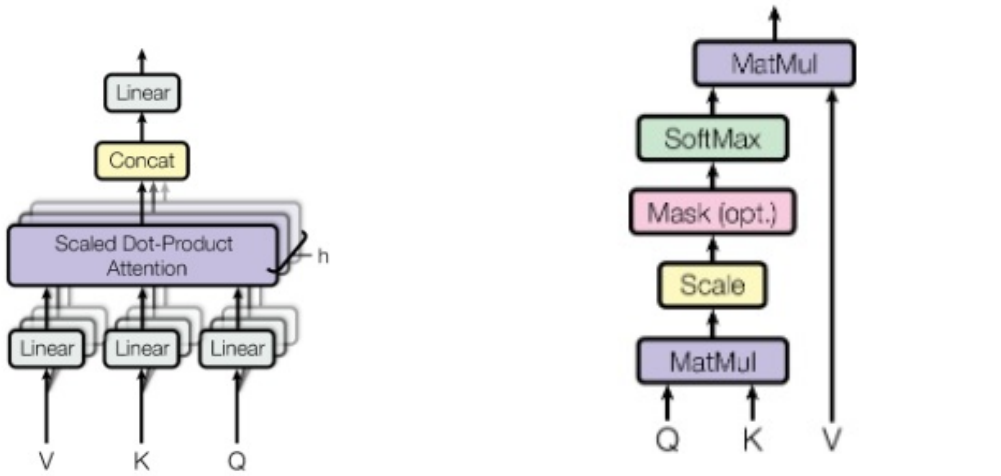
Il blocco di Multi-Head Attention layer consentono all'encoder di imparare le relazioni tra le diverse posizioni (parola) all'interno della sequenza di input (frase) dando un peso maggiore alle parole più rilevanti attraverso il meccanismo dell'attenzione.

Inizialmente vengono applicate una serie di trasformazioni lineari a Q, K e V, proiettandoli in differenti sottospazi, ognuno dei quali con un focus su determinate caratteristiche della parola dove Q, K e V rappresentano rispettivamente: Q (Query) rappresenta cosa si cerca di enfatizzare, K (Keys) aiuta a decidere dove concentrare l'attenzione, e V (Values) determina quali informazioni includere nell'output dell'attenzione.

Il layer Scaled Dot-Product Attention (Fig 2.5 b) implementa la fase di prodotto scalare (MatMul) tra Q(Query) e K(Keys), prodotto che poi viene scalato (Mask) per evitare valori troppo elevati e infine viene fatta una normalizzazione (Softmax) dei valori. Il risultato ottenuto viene moltiplicato (MatMul) per V (Values) che ci indica le informazioni

che vogliamo includere nell'output e quindi considerare.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) * V \tag{2.1}$$



(a) Multi-Head Attention (b) Scaled Dot-Product Attention

Figura 2.5: Componenti base Transformer

2.4.2 BERT

BERT (Bidirectional Encoder Representations from Transformers) è un modello di apprendimento automatico di NLP, Self Supervised, pre addestrato introdotto da Google nel 2019 [5]. L'architettura di BERT è basata sui transformer e implementa una codifica bidirezionale, questo permette al modello di apprendere relazioni di una parola sia con quelle che la precedono che con quelle che la succedono, a differenza di altri modelli come OpenAI GPT che è unidirezionale, ciò consente di apprendere relazioni più complesse. In BERT ci sono due fasi, una di pre-training e una di fine-tuning, nella prima fase il modello viene addestrato su dati non etichettati mentre nella seconda c'è un addestramento su dati etichettati specifici per ogni downstream, ovvero ogni task desiderato.

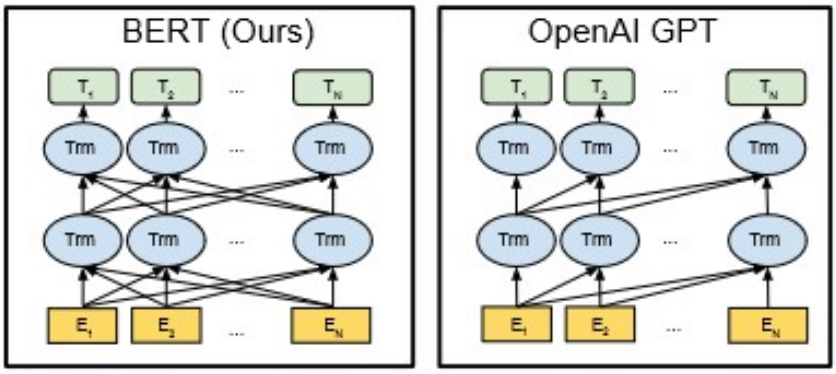


Figura 2.6: architettura BERT e Openai-GPT

Il pre-addestramento di BERT avviene tramite due task: Masked Language Modeling (MLM) e Next Sentence Prediction (NSP). Masked Language Modeling (MLM) con questa tecnica si va a sostituire in modo randomico il 15% delle parole con il token [MASK], l'obbiettivo è quello di addestrare il modello a prevedere le parole mancanti a partire dal contesto della frase. Poiché la fase di fine-tuning non possiede i token [MASK] e al fine di evitare che il modello si specializzi nel riconoscere i Token non sempre si procede alla sostituzione con il Token [MASK], in particolare del 15% delle parole selezionate l'i-esimo Token viene sostituito:

- Nell'80% dei casi viene lasciato il simbolo [MASK].
- Nel 10% dei casi un token selezionato casualmente.
- Nel 10% dei casi, il token stesso, lasciando quindi tutto invariato.

Ad esempio nella frase: "il mio cane è peloso" si avrà nell'80% dei casi "il mio cane è [MASK]" nel 10% "il mio cane è rosso" e nel restante 10% "il mio cane è peloso". Nel secondo Task, che utilizza la Next Sentence Prediction (NSP), si addestra il modello a individuare le relazioni tra le frasi. Per tutte le istanza del training set vengono create delle coppie, nel 50% dei casi l'istanza più quella successiva nell'altro 50% l'istanza più una casuale. L'obbiettivo è quello di addestrare il modello a comprendere se la seconda istanza di ogni coppia è quella che segue correttamente la prima o meno.

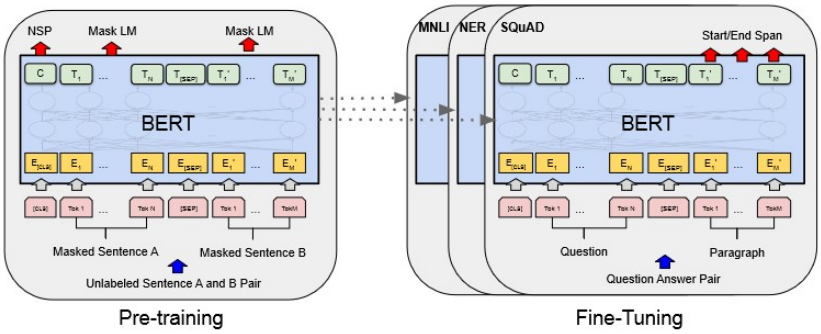


Figura 2.7: Pre-training e Fine-tunning

Infine una fase Fine-tuning viene utilizzata per migliorare le prestazioni del modello pre-addestrato su un nuovo set di dati, operazione che eseguiremo in seguito sia per la Detection dei malware sia per la loro classificazione.

Capitolo 3

Strumenti per Data Preprocessing

3.1 MADAM

MADAM (Multi-Level Anomaly Detector for Android Malware)[6] è concepito per svolgere funzioni di intrusion detection e identificazione di malware, con l'obiettivo di rilevare tali minacce e prevenire i loro effetti bloccandoli non appena vengono identificati. Si tratta di una soluzione host-based, ovvero è installata direttamente sul dispositivo che intende proteggere.

A differenza di altri sistemi che si basano su un approccio blacklist, come Virus Total, questo sistema adotta una strategia whitelist. In pratica, definisce una serie di comportamenti conosciuti e li classifica come benigni, considerando potenzialmente malevoli tutti quelli che si discostano da questa categorizzazione. Questo approccio si rivela particolarmente efficace nell'individuazione di attacchi zero-day, ossia attacchi sconosciuti e di nuova concezione, dei quali non si conosce né la firma né il comportamento tipico. Tuttavia, l'adozione di questa strategia comporta anche un rischio più elevato di falsi positivi.

Per mitigare il problema dei falsi positivi, il sistema integra meccanismi di machine learning, i quali contribuiscono a perfezionare l'analisi e a ridurre il margine di errore. L'analisi condotta è multilivello e si articola in diversi aspetti:

- **Metadata**: In questa fase, il sistema esamina informazioni come i dettagli dello sviluppatore, le recensioni presenti su Play Store, i permessi richiesti nel manifest dell'applicazione e altre caratteristiche statiche.

- **System Calls**: Vengono analizzate le operazioni di read e write, mettendo insieme i dati per esaminare i comportamenti e il numero di invocazioni di ciascuna system call in determinati intervalli di tempo.

- **User Activity**: Questo livello valuta l'interazione dell'utente con il dispositivo, identificando eventuali anomalie, come un'elevata frequenza di system calls in momenti in cui il dispositivo non è in uso.

- **Critical API & SMS**: Il sistema presta particolare attenzione alle API critiche, con un focus specifico sulle funzionalità relative agli SMS.

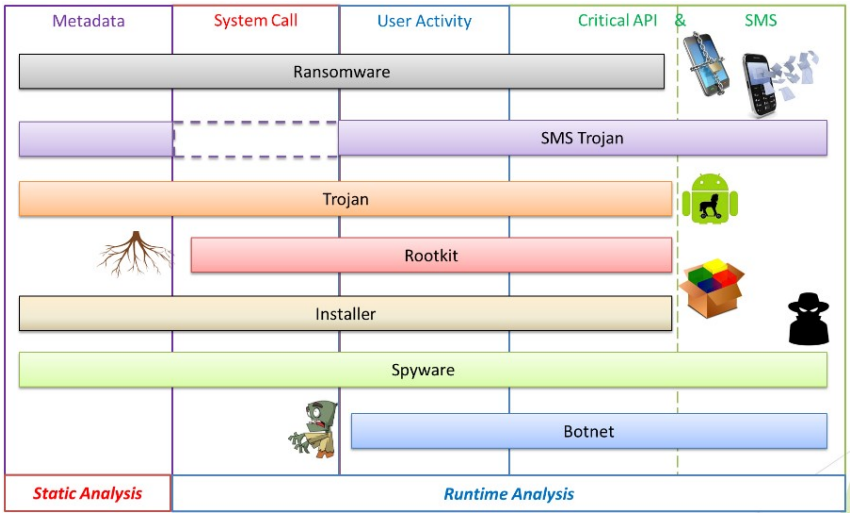


Figura 3.1: Madam analisi multi livello

L'analisi si sviluppa su due livelli distinti per ottenere una visione completa su quanto avviene nel dispositivo. A un livello globale, viene eseguito un monitoraggio complessivo delle chiamate e delle interazioni che avvengono su tutto il dispositivo, per avere una panoramica generale delle attività in corso. Successivamente, l'analisi si sposta su un livello locale, dove l'attenzione si concentra sulle singole applicazioni. In questa fase, vengono esaminati aspetti più specifici come la frequenza con cui vengono inviati i messaggi, i processi in esecuzione per ogni pacchetto di software e del manifest. Questo approccio garantisce un'analisi approfondita e dettagliata, permettendo di capire meglio la natura delle applicazioni analizzate.

A questo si aggiunge anche un'analisi statica, preinstallazione APP, con il fine di assegnare un valore di fiducia (trust value) alla stessa. Se tale valore supera una determinata

soglia, l'applicazione viene considerata sicura; in caso contrario, viene generato un avviso all'utente.

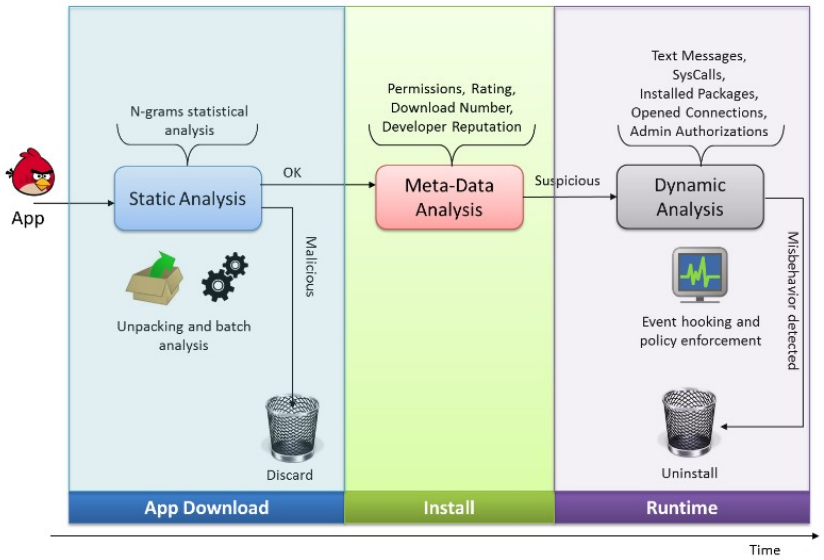


Figura 3.2: Madam Work Flow

Il sistema implementa un monitoraggio su un'ampia varietà di metriche per garantire la sicurezza del dispositivo. Queste includono la sorveglianza su 13 diversi tipi di system calls, il controllo delle chiamate ad API, la verifica degli SMS inviati e ricevuti, il monitoraggio dei processi attivi, il tracciamento delle installazioni di nuovi APP e la valutazione dell'attività dell'utente, determinando se esso sia attivo o meno (tramite il parametro isActive); Questo permette, ad esempio, di capire se vi sono attività in background non compatibili con l'assenza dell'utente, come l'invio di SMS.

In sintesi, il sistema si configura come una soluzione avanzata e strutturata su più livelli per la protezione dei dispositivi.

Nel contesto specifico del project work, MADAM è stato impiegato per stabilire una associazione tra le famiglie di malware, indicate all'interno del dataset Drebin, e le rispettive categorie di appartenenza dei malware. Passo fondamentamele per andare a classificare in modo più preciso i malware presenti nel dataset in uso.

3.2 Androguard

Androguard [1] è un potente strumento open-source utilizzato per analizzare e ispezionare applicazioni Android. Androguard offre una serie di funzionalità chiave per esaminare in

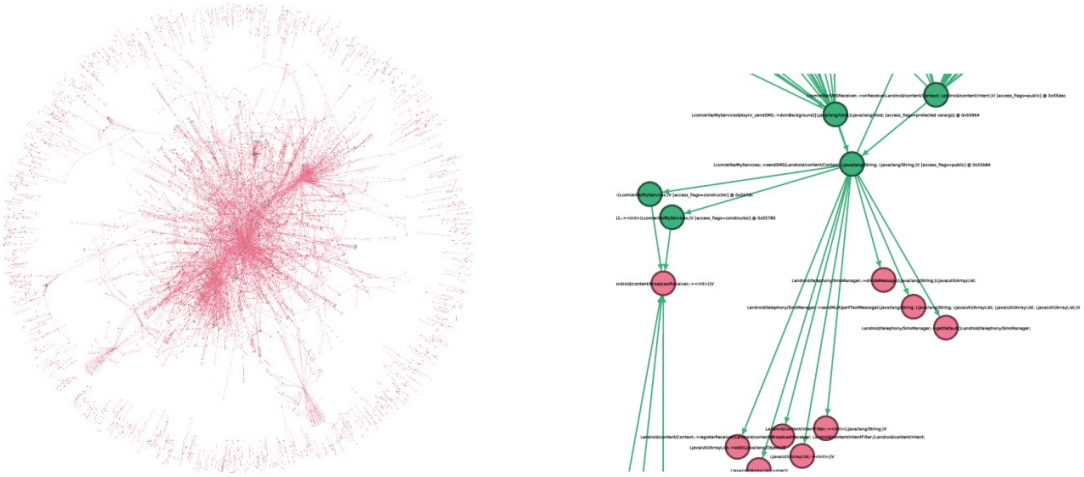
[1] https://androguard.readthedocs.io/en/latest/tools/androcg.html?highlight=graph

dettaglio le applicazioni Android. Tra queste, l'estrazione del codice sorgente dall'APK (Android Package), l'analisi statica e dinamica delle applicazioni, la decompilazione delle risorse, la ricerca di vulnerabilità e molto altro.

In particolare, Androguard è noto per la sua capacità di decompilare il bytecode dell'applicazione in un linguaggio più leggibile, rendendo più agevole la comprensione del funzionamento delle applicazioni. Di particolare interesse per il nostro caso sono le funzioni:

- **AnalyzeAPK**
- **Get_Call_Graph**

AnalyzeAPK è una funzione di Androguard che ci permette di analizzare globalmente un file APK, fornendo informazioni su codice sorgente, risorse, permessi richiesti, chiamate API e i vari componenti dell'applicazione. AnalyzeAPK restituisce informazioni tramite una serie di oggetti che rappresentano diversi componenti dell'APP, fornendo così una base solida per ulteriori analisi.



(a) Grafo androguard APK non filtrato (b) Porzione grafo androguard APK filtrato

Figura 3.3: Esempi di grafi ottenuti tramite Androguard

A partire dalle informazioni recuperate tramite AnalyzeAPK possiamo, tramite Get Call Graph, costruire un grafo di tutte le relazioni tra classi e metodi all'interno dell'applicazione, essenziale per comprendere il flusso di esecuzione dell'applicazione. Inoltre vi è la possibilità di ottenere un grafo filtrato in modo da avere una rappresentazione ridotta, permettendo così di porre il focus sulle componenti di maggior interesse, nel nostro caso le chiamate alle API Android.

Capitolo 4

Progetto

4.1 obiettivo

L'obiettivo è quello di creare un sistema in grado di classificare un'applicazione Android, al fine di definirne la natura, benevola o malevola, e in quest'ultimo caso, determinarne l'appartenenza a una specifica categoria. Nei paragrafi successivi verrà illustrato come raggiungere questo obiettivo, in particolare come trasformare l'applicazione (APK) in una forma che possa essere elaborata da un modello di machine learning. Questo, a sua volta, ci permetterà di identificare ed eventualmente classificare il malware incontrato.

4.2 Preprocessamento dataset

4.2.1 Dataset

Le applicazioni (APK) necessarie per addestrare il modello sono state recuperate da due differenti dataset, uno per le APK malevole e uno per quelle benevole, in particolare:

- **Maldroid** per APK Benevole [1]
- **Drebin** per APK Malevole [2]

Sul dataset delle APK Benevole non c'è stato alcun bisogno di classificare le applicazioni in quanto il dataset è ovviamente omogeneo. Mentre per quanto riguarda il dataset

[1] https://www.unb.ca/cic/datasets/maldroid-2020.html
[2] https://www.sec.cs.tu-bs.de/ danarp/drebin/

ricavato da Drebin abbiamo associato ciascuna APK a una categoria [3] di malware, facendo riferimento all'associazione tra famiglia [4] e classe fatta precedentemente da MADAM [7].

Abbiamo quindi messo in relazione ogni APK a una famiglia di malware tramite MADAM e quindi associato a ogni APK alla corrispettiva categoria, ovvero una tra le seguenti categorie di malware:

- SMS Trojan
- Spyware
- Rookit
- SMS Trojan + Spyware
- Installer
- SMS Trojan +Installer'
- SMS Trojan + Botnet
- Ransomware
- Trojan
- Botnet

Oltre il 91% dei malware contenuti nel dataset fanno riferimento alle prime 3 classi: SMS Trojan, Spyware e Rootkit. Pertanto, è stato effettuato un undersampling mantenendo solamente le 3 classi principali, questo al fine di evitare che il modello si specializzasse in classi rappresentate in modo marginale nel Dataset di partenza.

4.2.2 Creazione ed elaborazione dei dataset

Per passare da un file binario, qual è l'APK, a una rappresentazione dell'informazione sotto forma di sequenza, utile al training di un modello come BERT, è stato utilizzato in prima istanza il tool Androguard e successivamente uno script Python.

[3] Tipo di comportamento malevolo o obiettivo del malware.cosa fa il malware?
[4] Malware che condividono caratteristiche comuni. da dove proviene il malware?

Con Androguard è stato creato un grafo delle chiamate, opportunamente filtrato, questo sia per avere una dimensione adeguata del grafo che per acquisire solo informazioni di interesse per l'individuazione e classificazione dei malware, ovvero le chiamate alle API di sistema android. Metodo per l'estrazione del grafo:

```
def get_api_call_graph(apk_file):
# Carica il file APK in Androguard
a, d, dx = AnalyzeAPK(apk_file)
# Crea il grafo di chiamate API utilizzando Androguard
api_call_graph = dx.get_call_graph("Lnet/maxicom|Lcom/software/|Ljava/
    security|Ljava/rmi|Ljava/io|Landroid/service/|Landroid/hardware|
    Landroid/util/|Landroid/os/|Landroid/content/|Landroid/media/|
    Landroid/net/|Landroid/app/|Landroid/nfc/|Landroid/location/|Landroid
    /accounts/|Landroid/bluetooth/|Landroid/provider/|Landroid/nfc/|
    Landroid/telecom|Landroid/telephony/|Ljava/net/|Ljavax/crypto/")
nodes = api_call_graph.number_of_nodes()
return api_call_graph, nodes
```

Il risultato è un grafo nel seguente formato:

```
{<analysis.ExternalMethod Landroid/content/BroadcastReceiver;-><init>()V
    >: {}, <analysis.ExternalMethod Landroid/app/Activity;-><init>()V>:
    {},
<analysis.ExternalMethod Landroid/app/Activity;->onKeyDown(I Landroid/
    view/KeyEvent;)Z>: {},
<analysis.ExternalMethod Landroid/app/Activity;->onStart()V>: {}}
```

Una volta ottenuti i grafi delle chiamate API di tutti gli APK, è necessario trasformarli in delle sequenze. Questo processo è fondamentale per rendere i dati compatibili con modelli di linguaggio basati su trasformer, come BERT. La trasformazione in sequenze permette di mantenere la struttura e le relazioni tra le chiamate API, preservando la topologia originale del grafo. I passi cruciali di questa trasformazione avvengono attraverso due metodi, prima si collezionano tutti i nodi del grafo in ordine topologico (_follow_topology) e in seguito si costruisce la sequenza (_build_sequence) estraendo le informazioni sulle API dalla lista popolata precedentemente.

```
def _follow_topology(self, api_call_graph, queue, node_order):
while queue:
# Estrai il primo nodo dalla coda
node = queue.pop(0)
# Aggiungi il nodo alla lista degli ordini
node_order.append(node)
# Aggiungi i successori del nodo alla coda
for successor in api_call_graph.successors(node):
queue.append(successor)

def _build_sequence(self, node_order):
api_call_string = ""
for node in node_order:
node = str(node).split(';->', 1)[1].split('(')[0]
api_call_string += str(node) + " "
return api_call_string
```

Il risultato è una sequenza di questo tipo:

- getString edit getBoolean putString commit putBoolean <init> onCreate println <init> parse getNetworkOperator read read close skip.

Si evidenzia che la grandezza del grafo e della sequenza qui riportati è notevolmente inferiore rispetto alle dimensioni medie riscontrate nelle APK analizzate.

le sequenze ottenute per ogni dataset sono:

- Benevole: 4036
- Malware: 2854 - 2608 dopo undersampling per selezione classi

4.2.3 Undersampling delle classi

Come evidenziato in precedenza il dataset di APK benevoli è omogeneo al suo interno, ovvero tutti gli elementi sono della stessa classe, mentre il dataset dei malware possiede 10 classi (categorie), così distrubuite:

Appare quindi evidente come oltre il 90% dei malware facciano riferimento alle 3 categorie: 'Sms Trojan', 'Spyware' e 'Rootkit', pertanto si effettuerà un undersampling e si selezioneranno solamente le 3 classi più rappresentative del dataset.

Tipo di Malware	Conteggio
Sms Trojan	1086
Spyware	826
Rootkit	696
SMS Trojan + Spyware	100
SMS Trojan + Botnet	17
Installer	77
SMS Trojan + Installer	27
Ransomware	12
Trojan	7
Botnet	6

Tabella 4.1: Distribuzione categorie di malware

4.2.4 Augmentation

Per bilanciare i dataset son state utilizzate delle tecniche di augmentation, sia per bilanciare classi malevole e benevole (Malware Detection) sia per le sole classi malevole (Malware Classification). Per la parte di detection, visto l'alto numero di APK benevole a disposizione rispetto a quelle malevole, mi sono concentrato sulla creazione di istanze sintetiche delle categorie malware minoritarie: Spyware e Rootkit. Mentre per la classificazione dei malware ho creato delle istanze sintetiche per tutti e tre le categorie. Di seguito una porzione di codice con la quale vado a creare istanze sintetiche:

```
def generate_new_sequence(seq1, seq2):
apis_seq1 = seq1.split()
apis_seq2 = seq2.split()

if len(apis_seq1) < 2 or len(apis_seq2) < 2:
return "Both sequences should have at least two APIs."

num_apis_seq1 = random.randint(2, len(apis_seq1))
cut_apis_seq1 = random.sample(apis_seq1, num_apis_seq1)
new_sequence = cut_apis_seq1 + apis_seq2
new_sequence_str = ' '.join(new_sequence)

return new_sequence_str

modified_sequence = generate_new_sequence(random_sequence.iloc[0]['
    sequences'], random_sequence.iloc[1]['sequences'])
```

```
modified_sequence = modified_sequence.replace("<init>", "")
```

Il processo descritto dalla funzione generate_new_sequence consiste nel generare una nuova sequenza di chiamate API partendo da due sequenze, selezionate in modo randomico dal dataset iniziale e tra quelle delle classe prescelta.

Prima si verifica che entrambe le sequenze siano di lunghezza maggiore a 2 (ovvero che contenga almeno due chiamate API) e poi la funzione seleziona un sottoinsieme casuale di API dalla prima sequenza e le combina con le API della seconda. Inoltre, si effettua l'eliminazione della stringa <init> dalle nuove sequenze.

4.2.5 Creazione dataset per esperimenti

- **Detection malware**: aumentiamo le istanze delle classi Spyware e Rootkit per portarle allo stesso valore di quelle dell'Sms Trojan e infine aggiungiamo le classi benevole, ottenendo 3258 istanze e aggiungendone altrettante di natura benevola, per un totale di 6516 istanze

- **Classification malware**: aumentiamo le istanze delle classi Spyware e Rootkit per portarle allo stesso valore di quelle dell'Sms Trojan, per un totale di 4.758 istanze equamente suddivise tra le 3 classi

4.3 Sviluppo e Addestramento del Modello

4.3.1 Creazione modelli

Il modello scelto per l'addestramento è costituito da: un livello di prepocessing dell'input (bert_preprocess [5]), uno strato con BERT preaddestrato (bert_encoder [6]), una rete neurale fully connect e un layer di output; vediamo nel dettaglio le varie componenti:

- **Bert preprocess**: Questo layer è responsabile del preprocessing dell'input, che viene trasformato e preparato per essere processato dal livello successivo. Questa fase di preprocessing include la tokenizzazione del testo e altre trasformazioni necessarie per adeguare i dati all'input atteso dal modello BERT.
- **Bert encoder**: In questo layer l'input testuale preprocessato viene elaborato e trasformato in vettori, che catturano i contesti e le relazioni semantiche tra le parole. BERT processa il testo in maniera bidirezionale.
- **Rete neruale fully connect**:
  - **Pooling Max** per riduzione dimensionalità dell'output del modello BERT
  - **DropOut** di 0,1 ovvero 10% di probabilità che un neruone venga temporaneamente escluso dalla rete, così da ridurne la complessità e l'eventuale overfitting.
  - **Hidden Layers** da 1024, 2048, 4096 per la Detection e un ulteriore livello di 4086 neuroni per la Classificazione.
  - **Layer di output**
    - Detection 1 neurone con funzione di attivazione sigmoide
    - Classification 3 neuroni con funzione di attivazione softmax

[5] https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/
[6] https://tfhub.dev/tensorflow/lambert_en_uncased_L-24_H-1024_A-16/2

```
preprocessed_text = bert_preprocess(text_input)
outputs = bert_encoder(preprocessed_text)

# Neural network layers
l = tf.keras.layers.Dropout(0.1, name="dropout")(outputs['pooled_output'
    ])
l = tf.keras.layers.Dense(1024, activation='relu', name='inter_layer')(l
    )
l = tf.keras.layers.Dense(2048, activation='relu', name='
    intermediate_layer')(l)
l = tf.keras.layers.Dense(4096, activation='relu', name='ultimate_layer'
    )(l)

#solo classificazione
l = tf.keras.layers.Dense(4096, activation='relu', name='last_layer')(l)

#Output detection
l = tf.keras.layers.Dense(1, activation='sigmoid', name="output")(l)
#Output classificazione
l = tf.keras.layers.Dense(N, activation='softmax', name="output")(l)
```

4.3.2 training

Per il training del modello, procediamo con la creazione del training set e del validation test applicando per due volte la funzione "train_test_split" in modo da calcolare prima il training set e poi il validation set, a partire dal dataset bilanciato con le tecniche di augmentation esposte in precedenza; il parametro stratify fa in modo che vi sia bilanciamento nella distribuzione delle varie classi.

```
X\_train, X\_test, y\_train, y\_test = train\_test\_split(df\_Train[\'
    sequences\'],df\_Train[\'label\'], stratify=df\_Train[\'label\'])
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_train, y\
    _train, stratify=y\_train)
```

L'algoritmo utilizzato per l'aggiornamento dei pesi è Adam e le funzioni di perdita sono rispettivamente per detection e classificazione: la Binary Cross Entropy e la Categorical Cross Entropy.

Il modello viene addestrato per un numero di epoche elevato, 100, in modo che la conclusione venga sempre detterminata dall'early stopping che blocca il training se per 5 epoche non vi è un miglioramento del valore della funzione di perdita (val_loss). l'intervallo dell'aggiornamento dei pesi (batch) è stato lasciato invariato, e per le librerie utilizzate è pari a 32.

4.3.3 testing

Il test set è calcolato a partire dal dataset non bilanciato, in modo tale che questa fase non sia influenzata dalle tecniche di augmentation; ciò aiuta anche a testare la capacità del modello di generalizzare e di classificare elementi quanto più vicini a come sono nella realtà.

Per il test sui dati abbiamo utilizzato le metriche di precision, recall, F1-score e support per ottenere una visione completa delle prestazioni del modello su ogni classe. Per la classificazione dei malware sono state calcolata anche la la matrice di confusione e la curva ROC.

Capitolo 5

Analisi dei risultati

5.1 Detection

Classe	Precisione	Recall	F1-Score	Supporto
0.0 (Benevole)	0.98	0.99	0.99	815
1.0 (Malevole)	0.99	0.98	0.99	814

Tabella 5.1: Metriche di prestazione del modello

Valori complessivi	Loss Value	Accuracy	Precision	Recall
	0.0843	0.976	0.974	0.974

Tabella 5.2: Metriche di prestazione del modello

La performance del modello sul set di dati testato è stata alta (Tab. 5.1 e 5.2), come si può vedere dalla tabella 5.1 per la classe 0 (APK benevole) abbiamo una precisione pari al 98% e un richiamo (recall) del 99% viceversa per la classe 1 (APK malevole) abbiamo una precisione del 99% e una recall del 98%.

Le metriche presentate indicano che il modello ha ottenuto un'ottima prestazione sia nell'individuare classi "Benevole" che classi "Malevole". L'accuratezza generale del modello è elevata, avvicinandosi al 99%, e anche le altre metriche (Precisione, Recall e F1-Score) sono molto vicine al valore ottimale di 1.0. Questo suggerisce che il modello ha un basso tasso di falsi positivi e falsi negativi e riesce a classificare correttamente la maggior parte delle istanze.

Nella Figura 5.1, osserviamo l'andamento delle curve di errore per il training e il validation set durante le varie epoche. L'andamento indica convergenza tra le due curve, ovvero che si sta evitando l'overfitting. Il loss sia in fase di training e validazione, che in quella di test (0.0843) è basso e su valori che si discostano di poco tra training, validazione e test. Questo indica che il modello ha una buona capacità di predizione e potrebbe avere un'ottima performance anche su dati che non ha mai visto prima.

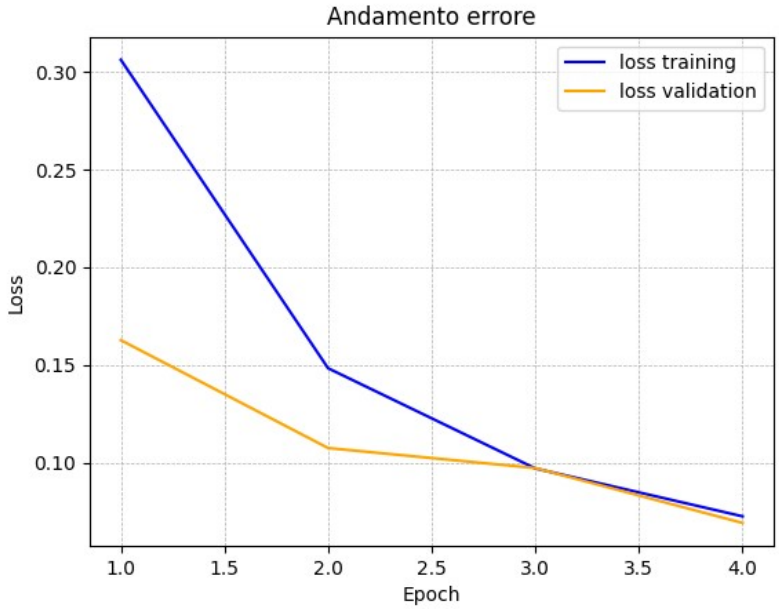


Figura 5.1: Errore su training e validazione - Detection

5.2 Classificazione

Classe	Precisione	Recall	F1-Score	Supporto
0.0 (Sms Trojan)	0.951	0.934	0.942	272
1.0 (Spyware)	0.937	0.932	0.934	206
2.0 (Rootkit)	0.917	0.948	0.932	174

Tabella 5.3: Metriche di prestazione del modello

Valori complessivi	Loss Value	Accuracy	Precision	Recall
	0.163	0.966	0.968	0.929

Tabella 5.4: Metriche di prestazione del modello

Anche nella classificazione le performace sono elevate (Tab. 5.3 e 5.4), anche se leggermente inferiori rispetto alla detection. Come si nota dalla tabella 5.3 Per la categoria degli Sms Trojan, la precisione è di pari al 95,1% e il recall leggermente inferiore pari a 93,4%. Ciò significa che il modello è molto preciso nel riconoscere correttamente gli Sms Trojan, e contemporaneamente è in grado di identificare il 93.% dei veri positivi per questa categoria presenti nel dataset di test.

Per quanto riguarda la categoria degli Spyware, la precisione è pari al 93,7% e il recall di 93,2%. Questo indica che c'è un leggero aumenti numero di falsi positivi rispetto alla categoria degli Sms Trojan, a causa di una minore precisione, d'altro canto il valore di recall ci indica che il modello è efficace nel riconoscere e recuperare la maggior parte dei veri esempi di Spyware presenti nel dataset di test.

Infine, per la categoria Rootkit la precisione è pari al 91,7%, la più bassa per le 3 classi, e il recall di 94,8% questo ci suggerisce suggerisce che, nonostante una leggera diminuzione nella precisione rispetto alle altre due categorie, esiste comunque una buona capacità notevole del modello di fare previsioni accurate. Mentre, l'alto valore di recall sottolinea come il modello sia efficace nel classificare correttamente la stragrande maggioranza degli elementi di questa categoria.

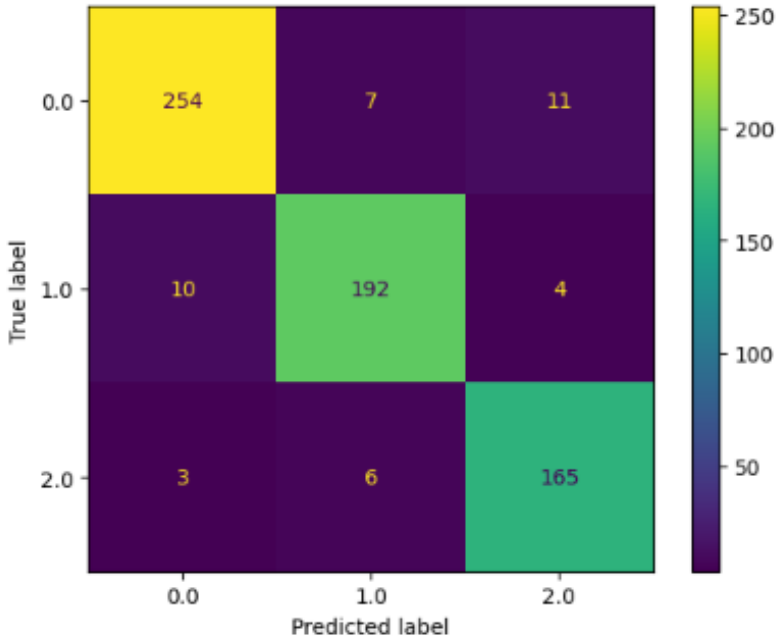


Figura 5.2: Matrice di confusione

Come possiamo osservare, la matrice di Confusione (Fig 5.2) mostra chiaramente che il

modello possiede una notevole capacità di classificazione. La maggior parte delle previsioni si concentra lungo la diagonale principale, indicando che la maggior parte dei malware viene correttamente classificata. Si può inoltre notare un basso numero di falsi positivi e falsi negativi, ulteriori indicatori dell'affidabilità del modello nelle sue previsioni.

La curva ROC (Fig 5.3) rappresenta la capacità di un classificatore di distinguere tra due classi, misurando la sensibilità (Recall o TPR) e la specificità (FPR), che rappresenta la percentuale di esempi negativi erroneamente classificati come positivi. Una curva ROC diagonale rappresenterebbe un classificatore casuale, mentre un classificatore perfetto è rappresentato da una retta che segue l'asse delle ordinate e poi, formando un angolo retto, prosegue parallela all'asse delle ascisse, con un'area sottesa dalla curva pari a 1. Nel nostro caso, ci avviciniamo a questo ideale e le aree sotto le curve per i 3 tipi di malware hanno valori elevati, intorno a 0.99.

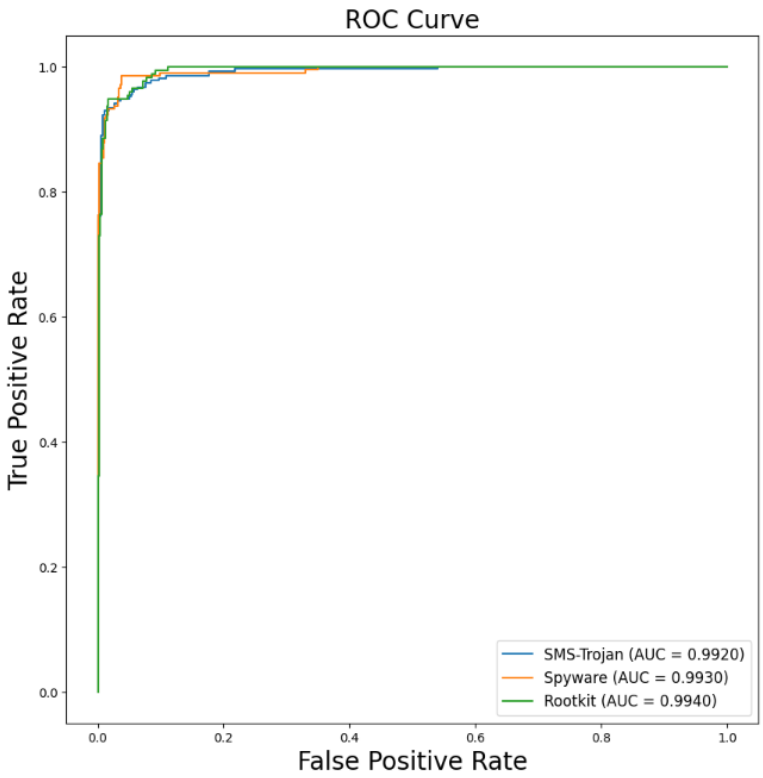


Figura 5.3: ROC curve

Nella Figura 5.4, osserviamo l'andamento delle curve di errore per il training e il validation set durante le varie epoche. L'andamento indica convergenza tra le due curve, ovvero che si sta evitando l'overfitting. In questo caso abbiamo delle discrepanze maggiori sul valore di loss rispetto alla detection, possiamo infatti notare dal grafico (Fig 5.4) una minore convergenza della perdita di errore tra training e validazione, la distanza rimane

comunque ridotta: 0.356 per il training e 0.33 per la validazione. Notiamo invece come sul set test tale valore si assesti su valori inferiori 0.163, indicazione della buona capacità del modello di generalizzare.

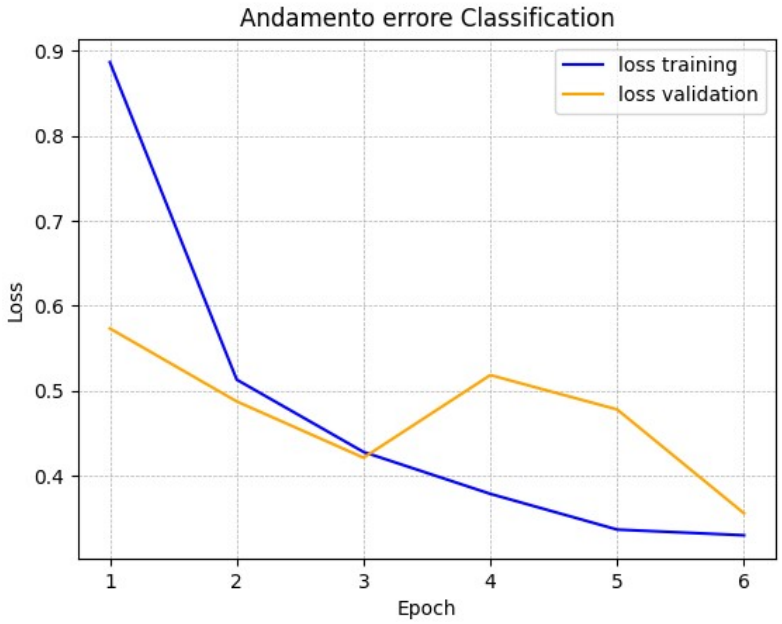


Figura 5.4: Errore su training e validazione - Classificazione

Capitolo 6

Conclusione

Lo scopo del project work è valutare, attraverso tecniche di apprendimento automatico, la natura di applicazioni Android esaminando i legami tra le numerose chiamate alle API di Android. In questo contesto, un modello come BERT e l'utilizzo di Androguard (con la successiva elaborazione del suo output) hanno permesso, da un lato, l'estrazione di informazioni sulle chiamate alle API Android in un formato facilmente trasformabile in forma testuale e, dall'altro, l'impiego di un modello per sua natura adatto a catturare relazioni tra parole distanti in un testo e, quindi nel caso specifico, relazioni tra API. Alla luce dei risultati, benché l'addestramento sia avvenuto su dataset ridotti, l'idea di utilizzare queste tecniche si è rivelata promettente. Per approfondire ulteriormente questo approccio e sfruttare appieno la potenza di modelli come BERT o dei suoi successori, diventa fondamentale accedere a un maggior numero di dati. Con una quantità di dati più significativa, potremo spingerci, nella fase di training dei modelli, ad aggiornare i pesi presenti all'interno del modello preaddestrato o, quantomeno, i pesi degli strati più esterni della rete.

Bibliografia

[1] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[2] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 1958.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1:318–362, 1986.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[6] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: A multi-level anomaly detector for android malware.

[7] Daniele Sgandurra Andrea Saracino. Madam: Effective and efficient behavior-based android malware detection and prevention.