UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Rilevamento di Malware DGA Mediante Reti Transformer con Embedding Ibrido

DGA Malware Detection Using Transformer Networks with Hybrid Embedding

Candidato:
Massimo Mecarelli

Relatore:
Prof. Luca Spalazzi

Correlatore:
Prof. Alessandro Cucchiarelli

Anno Accademico 2022-2023

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Rilevamento di Malware DGA Mediante Reti Transformer con Embedding Ibrido

DGA Malware Detection Using Transformer Networks with Hybrid Embedding

Candidato:
Massimo Mecarelli

Relatore:
Prof. Luca Spalazzi

Correlatore:
Prof. Alessandro Cucchiarelli

Anno Accademico 2022-2023

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Alla mia famiglia

Abstract

A widespread attack mode nowadays is to exploit domain name generation algorithms, also called Domain Generation Algorithm (DGA), which provide URLs to allow connection to command and control servers (C&C) which gives an attacker the ability to control a machine.

Communication leads to the infected system becoming a bot and being part of remotely controlled botnets, which can be used for DDoS attacks (Distributed Denial of Service) or data theft.

These domain names are generated periodically in large numbers in order to bypass any blocks of more traditional systems that employ static blacklists, which are manually updated when new discoveries occur.

Their composition can be of different nature, often there are no words used in natural language, but rather confused sequences of characters and numbers, in some other cases, more advanced techniques are employed that exploit combinations of more words of a dictionary, making malicious domains quite indistinguishable from real domains.

This research aims to develop a neural network that has been trained on datasets from various DGA families and is capable of independently recognizing common patterns in domain names and detecting and classifying DGA names.

There are several deep learning architectures that can be used, in this case we want to rely on the Transformer one, in particular by using the encoder component, to which some changes are made.

This model uses an input embedding which is a hybrid of two types for each domain name, one based on characters and one based on bigrams, each then passed to two parallel networks that extract features on different scales, which are then concatenated to generate a classification output.

To evaluate its performance the k-fold cross validation has been used, which allows to divide the entire dataset into folds, that are cyclically used for validation, so that at the end accuracy, recall and F1 values are returned for each of them.

The model's unoptimal results for these different parameters are caused by an overfitting issue due to using a small dataset compared to the complexity of the neural network.

Sommario

Una modalità di attacco molto diffusa oggigiorno, consiste nello sfruttare algoritmi di generazione di nomi di dominio, anche chiamati DGA (Domain Generation Algorithm), che forniscono URL per permettere la connessione a dei server di comando e controllo (C&C), i quali danno la possibilità all'attaccante di controllare una macchina.

All'instaurarsi della comunicazione il sistema infetto diventa un bot, entrando a far parte di botnet controllate da remoto, al fine di essere utilizzate, ad esempio, per effettuare attacchi DDoS (Distributed Denial of Service) o per furti di dati.

Questi nomi di dominio vengono generati periodicamente in gran numero, al fine di aggirare eventuali blocchi di sistemi più tradizionali, che impiegano blacklist statiche, aggiornate manualmente ad ogni nuova scoperta.

La loro composizione può essere di diversa natura, spesso non sono presenti parole utilizzate nel linguaggio naturale, ma piuttosto sequenze di caratteri e numeri confuse.

In alcuni casi vengono impiegate anche tecniche più avanzate, che sfruttano combinazioni di più parole di un dizionario, rendendo i domini malevoli praticamente indistinguibili da domini legittimi.

L'obiettivo di questa ricerca è quello di implementare una rete neurale allenata su dataset di diverse famiglie di DGA, che sia in grado di riconoscere autonomamente pattern comuni nei nomi di dominio ed essere capace di rilevare nomi derivati da DGA e classificarli.

Esistono diverse architetture di deep learning che possono essere utilizzate, in questo caso ci si vuole basare su quella Transformer, in particolare sfruttando la componente di encoder, a cui vengono apportate alcune modifiche.

Questo modello possiede in ingresso un embedding che è un ibrido di due tipologie per ciascun nome di dominio, uno basato sui caratteri e uno basato sui bigrammi, ciascuno poi passato a due reti parallele che ne estraggono caratteristiche su scale differenti, le quali vengono infine concatenate per generare un output di classificazione.

Per valutarne le prestazioni viene utilizzata la tecnica di k-fold cross validation, che permette di suddividere l'intero dataset in folds, le quali vengono ciclicamente utilizzate per la validazione, cosicché alla fine vengano restituiti valori di accuratezza, precisione, recall ed F1 per ciascuna di esse.

I risultati ottenuti su questi diversi parametri non sono ottimali, a causa di un problema di overfitting, principalmente dovuto al fatto di aver utilizzato un dataset molto piccolo in relazione alla complessità della rete neurale.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Motivazioni

Un comune e sempre più diffuso tipo di cyber attacco, prevede dei virus opportunamente programmati per cercare di instaurare comunicazioni con determinati server di comando e controllo (C&C), una volta infettata una macchina, dando così vita a reti di bot.

Molti sistemi per il rilevamento di botnet usano delle blacklist contenenti i nomi che identificano dei C&C noti, così da bloccare il traffico di rete verso questi.

Questa modalità più tradizionale è un tipo di approccio statico, perché prevede una lista che viene aggiornata ogni volta che vengono scoperti nuovi domini malevoli, ed è quindi comparabile ai sistemi antivirus signature-based.

Per aggirare facilmente questo sistema di sicurezza, i botmasters, ovvero coloro che controllano le botnet, hanno iniziato ad impiegare i cosiddetti DGA (Domain Generation Algorithm), cioè algoritmi in grado di generare dinamicamente in poco tempo un gran numero di nomi di domini casuali, fornendone un sottoinsieme ai sistemi di C&C. [3].

In questo modo, un dominio adibito al comando e controllo, essendo creato dinamicamente e in maniera randomica, rende totalmente inefficaci i sistemi che si basano sull'utilizzo di liste statiche. Perciò questa adattabilità, permette al programma malevolo di persistere più a lungo e massimizzare il suo impatto.

Il funzionamento cardine dei malware DGA risiede, quindi, nella capacità di fornire autonomamente al sistema infetto dei nomi di dominio per instaurare comunicazioni con l'esterno.

Essi vengono generati algoritmicamente basandosi su dei parametri, come ad esempio la data corrente, resi così imprevedibili e difficili da determinare a priori.

La natura dinamica di questa categoria di software malevoli, necessita di approcci più avanzati per essere contrastata.

Oltre a metodologie più tradizionali che riguardano il controllo di anomalie nel traffico di rete, come ad esempio dei picchi improvvisi di risoluzioni di nomi di dominio, una delle sfide principali da affrontare per un'efficace rilevamento e mitigazione della minaccia, è rappresentata dall'identificazione di caratteristiche associabili a stringhe di nomi di dominio DGA, come lunghezza e anomalie linguistiche.

Vengono sempre più sfruttate tecniche di Machine Learning, le quali risultano parecchio efficaci nell'individuazione di pattern più impercettibili e complessi, e forniscono una risposta dinamica nel riconoscimento di nuovi domini di C&C, che possono sfuggire ai sistemi basati su regole statiche.

1.2 Obiettivo

L'obiettivo della tesi è quello di implementare e valutare l'efficacia di un modello di deep learning basato su reti Transformer, capace di apprendere caratteristiche che legano domini appartenenti alla medesima famiglia, per poi riuscire autonomamente a rilevarli e classificarli.

Questa architettura mette insieme due embedding per la medesima stringa, uno basato sui caratteri e uno sui bigrammi, così da combinare risultati ottenuti su due scale differenti.

Per riuscire in questo, è necessario reperire un dataset che sia strutturato in modo da contenere nomi di domini DGA divisi per classe, su cui si fanno valutazioni di performance generali e specifiche per ciascun gruppo.

1.3 Struttura della Tesi

La tesi è strutturata nel seguente modo:

Nel Capitolo 2 verranno approfonditi dei concetti riguardanti i malware DGA e le tipologie di domini malevoli più diffuse.

Nel Capitolo 3 verrà descritto il modello da implementare, partendo da una breve introduzione sull'architettura transformer, fino ad arrivare alla descrizione in dettaglio dei vari layer.

Nel Capitolo 4 verrà descritta l'effettiva implementazione con i vari strumenti di cui ci si è serviti, approfondendo l'aspetto riguardante il dataset.

Nel Capitolo 5 verranno mostrati i risultati sperimentali delle due versioni del modello.

Nel Capitolo 6 verranno discussi i risultati degli esperimenti.

Nel Capitolo 7 verranno tratte le conclusioni e le prospettive di sviluppo future.

Capitolo 2

Malware DGA

2.1 Definizione

Come citato nel Capitolo 1, il termine **malware DGA** fa riferimento ad una classe di software malevoli, che sfruttano algoritmi per generare periodicamente in modo dinamico degli URL, al fine di instaurare una comunicazione con dei server di comando e controllo.

I DGA generano un gran numero di nomi di dominio apparentemente casuali, di cui ne forniscono una parte ai C&C per brevi periodi di tempo, rendendo più complessa la prevenzione di attività malevoli per i sistemi di cyber security.

L'obiettivo dell'attaccante è quello di dar vita a botnets, utilizzate per compiere attività illegali.

2.2 Botnets

Le **Botnets** sono reti di macchine compromesse (bot) controllate da un'infrastruttura esterna.

Queste reti di bot vengono create infettando un gran numero di computer con dei malware, che trasformano il dispositivo in un nodo della rete gestita da remoto.

Coloro che controllano le botnet vengono chiamati "botmasters" e si servono di esse per compiere attività illecite come:

- *Attacchi DDoS* (Distributed Denial of Service), che consistono in più dispositivi che simultaneamente inviano una gran quantità di traffico in entrata ad un server o ad una rete, causandone l'intasamento e di conseguenza interrompendo l'erogazione di servizi ai client.
- *Spam e Phishing*
- *Mining di Criptovalute*, sfruttando la potenza computazionale delle macchine infette.
- *Attività di spionaggio e furti di dati sensibili* contenuti nei sistemi compromessi

2.3 Tipologie di Domini DGA

Le tipologie di domini DGA possono essere distinte in base alle modalità di generazione delle stringhe, in particolare da quale seme parte l'algoritmo pseudocasuale.

2.3.1 DGA a Seme Dinamico

La categoria degli algoritmi a seme dinamico, anche detti 'tempo dipendenti', è la categoria più frequente. La Tabella 2.1 riporta un elenco parziale di tali algoritmi, presenti nel dataset utilizzato per gli esperimenti.

Viene fornito allo pseudo-random generator DGA (PRNG) un seme per la generazione randomica deterministico, che quindi permette al malware di predire i domini che verranno creati se sincronizzato a priori con il DGA.

Spesso il seed ha una dipendenza temporale, cioè vengono utilizzate la data e l'ora dal momento in cui parte l'algoritmo.

DGA a Seme Dinamico	
Famiglia DGA	**Esempio di Dominio**
alureon	mzvbfkkoij.com
bedep	ennfhcevrcstf.com
ccleaner	ab4b2eb0651d.com
chinad	gcyz2qxdtd0xjdax.biz
corebot	3bw0m0sbkxab74u.ddns.net
cryptolocker	gaofnfucukok.ru
dyre	id1379fdac9a2ead4329efce48307d1ca9.cc
locky	kafcwlmvbhvpd.info
murofet v1	hhkbxkphtxhaqcepqglfobw.ru
murofet v2	mwnnvqrktkdzitqq.biz
murofet v3	pveumwpxa67oynqoyaxlslqk27e51o21lvdw.ru
necurs	sdwyqyimwqexqpkaib.ru
padcrypt	anedaokfkmfabokb.com
proslikefan	xesapy.se
qadars	oha7spufk56n.org
qakbot	mvqhrexfqcf.org
sisron	mtkwmti2mzea.info
symmi	gekuiwullu.ddns.net

Tabella 2.1: Dynamic Seed DGA

2.3.2 DGA a Seme Statico

Per la categoria degli algoritmi a seme statico, anche detti 'tempo indipendenti', viene meno la necessità di sincronizzare il malware con l'algoritmo, a discapito di un seme statico o fisso come input al PRNG, che risulta quindi più facile da predire.

Essendo indipendente dal tempo, si possono avere esecuzioni identiche in due istanti temporali differenti.

È una tipologia di DGA poco utilizzata per l'elevata prevedibilità.

Nella Tabella 2.2 viene riportato il sottoinsieme di algoritmi presente nel dataset utilizzato negli esperimenti.

DGA a Seme Statico	
Famiglia DGA	**Esempio di Dominio**
banjori	gowperionirkutskagl.com
dircrypt	ghoinmwwyduotlbq.com
fobber v1	ahtsvrcvjwwthtwhy.net
fobber v2	yszhthgzjp.com
kraken v1	trzgajn.dyndns.org
kraken v2	qehoshq.yi.org
pushdo	gifanheacugup.com
pykspa	jgioeypicsu.net
pykspa noise	vrtqxtig.net
ramdo	gqiwaeqeowgcisiu.org
ramnit	naephlgajgnotctkmqv.com
ranbyus v1	qghxhhakjdnitr.net
ranbyus v2	pmkbbwsdaripeksna.tw
shiotob	sscinmenf35.net
simda	qesewyg.info
tempedreve	icscvcsxum.net
tinba	okndllkgdjio.in
vawtrak v1	arktmfohqzk.top
vawtrak v2	isicgimli.com
vawtrak v3	eggaraldat.com
zeus newgoz	1y6e4fm4wf2701xxcdzj1cq99e3.org

Tabella 2.2: Dynamic Seed DGA

2.3.3 DGA Basati su un Vocabolario

Questa tipologia di DGA genera domini a partire da un vocabolario di una o più lingue, da cui si combinano delle parole o parti di esse, scelte in maniera pseudo-casuale, per creare una stringa. Nella Tabella 2.3 viene riportato un elenco parziale di tali algoritmi.

In questo caso i nomi di dominio posseggono elementi del linguaggio naturale, risultando quindi meno sospetti e più difficili da rilevare.

DGA basati su un vocabolario	
Famiglia DGA	**Esempio di Dominio**
gozi gpl	transactionits.ru
gozi luther	vivenveniefulmeodemeamc.com
gozi nasa	opportunityandfi.com
gozi rfc4343	actuthistherecursive.com
matsnu	channelassociate.com
nymaim	scenarios-mature.ec
pizd	eitherwindow.net
rovnix	whosehassothepowerdespotism.biz
suppobox 1	wintercourse.net
suppobox 2	callmarry.net
suppobox 3	constancesheridan.net

Tabella 2.3: Vocabulary-based DGA

2.4 Tecniche per il Rilevamento Automatico di DGA

Le soluzioni attuali per il rilevamento di domini DGA includono: blacklist statiche, reverse engineering, machine learning e deep learning [4].

Il grande limite delle blacklist, risiede nel fatto di avere un aggiornamento delle liste estremamente più lento rispetto alla velocità di comparsa di nuovi nomi malevoli.

Il reverse engineering è un metodo che necessita di diversi elementi di esempio dei malware, che però non sono semplici da reperire [4]. Questo strumento non serve a bloccare i malware, ma è utile per ricostruire la loro struttura software così da comprenderne le funzionalità, lo scopo e in modo tale da far generare all'algoritmo nomi malevoli. Questi possono essere utilizzati per alimentare blacklist con ulteriori nomi che non sono ancora apparsi, ma anche per addestrare algoritmi di machine learning. Infatti uno dei limiti del machine learning risiede proprio nel reperire nomi per l'addestramento.

Per questi motivi, le tecniche di Machine Learning, in particolare di Deep Learning, stanno diventando sempre più popolari negli ultimi anni.

2.4.1 Shallow Learning

Le tecniche di Shallow Learning, sono tecniche di Machine Learning che utilizzano algoritmi con un piccolo numero di layers nel modello, per effettuare ad esempio valutazioni sulla legittimità dei domini, in base a dei parametri prestabiliti riguardanti le caratteristiche delle stringhe.

Alcuni esempi sono: regressione lineare, regressione logistica, alberi decisionali e SVG (support vector machines).

Questi algoritmi lavorano bene nel rilevamento di pattern più semplici, e soprattutto necessitano di un insieme di dati di addestramento ridotto.

2.4.2 Deep Learning

Il Deep Learning è un sottoinsieme del Machine Learning, che utilizza per l'apprendimento delle reti neurali organizzate su diversi strati, ciascuno che elabora l'informazione per quello successivo; da qui il termine 'deep'.

Questi modelli sono in grado di individuare pattern molto complessi e non hanno bisogno di stabilire a priori le particolari caratteristiche da valutare, a differenza del Shallow Leaning, ma sono loro stessi ad estrarle autonomamente.

Per rilevare proprietà più specifiche e in modo più accurato, necessitano di dataset di grandi dimensioni, e ciò diventa il loro principale limite.

Le tecniche di Deep Learning sono impiegate in task come il riconoscimento di immagini e nel NLP (Natural Language Processing).

2.4.3 Tipologie di Apprendimento

Le modalità di apprendimento, applicate alle tecniche sopracitate, si possono dividere in due categorie principali, supervisionato e non supervisionato.

L'apprendimento supervisionato prevede l'allenamento di un modello su un dataset etichettato, quindi dove abbiamo ciascun input accoppiato con l'output corrispondente.

Questo viene applicato ad esempio, nella classificazione binaria tra domini DGA e benevoli e in quella multipla tra le famiglie di DGA.

Il limite principale risiede nel reperimento dei dati, che sono più complessi da elaborare e aggiornare, poiché includono tutta una fase di 'labeling'.

L'apprendimento non supervisionato invece, non richiede un dataset con etichettatura, sarà infatti il modello stesso a farla, attraverso la creazione di cluster, ovvero gruppi di elementi in cui sono state rilevate caratteristiche comuni.

Il modello Transformer Ibrido Modificato, proposto nella pubblicazione [2] e di seguito sviluppato, appartiene alla categoria di Deep Learning con apprendimento supervisionato.

Capitolo 3

Descrizione del Modello

I modelli più diffusi, adibiti al rilevamento di nomi DGA, sono quelli basati su CNN (Convolutional Neural Network) e CNN-LSTM (Convolutional Neaural Network con Long Short-Term Memory) con embedding a livello di caratteri, i quali risultano molto efficaci nella classificazione binaria per la distinzione tra DGA e benevoli [2].

Altre tecniche presenti in letteratura sono ad esempio, combinazioni di reti Bi-Directional Long Short-Term Memory (BiLSTM), Attention and Convolutional Neural Network (CNN) che superano alcuni limiti di reti più tradizionali come le Recurrent Neural Networks (RNN) [4].

Esiste, invece, un margine di miglioramento maggiore per la fase di classificazione multipla delle famiglie di botnet, che risulta più complessa, in particolare per il fatto di avere spesso uno sbilanciamento della quantità di campioni tra le classi nei datasets [2].

Il modello descritto si basa sulla combinazione di un embedding ibrido e la struttura standard dell'architettura Transformer [1], a cui viene sostituito il blocco di decoder e modificato l'encoder.

3.1 Architettura Transformer

L'architettura Transformer, Figura 3.1, è stata introdotta per la prima volta nel 2017 nella pubblicazione [1]. Ha rivoluzionato il campo del processamento del linguaggio naturale (NLP) e ha dimostrato prestazioni molto elevate in differenti task come nel Computer Vision (CV).

Il modello è caratterizzato principalmente dal meccanismo della "self-attention", che permette di pesare l'importanza delle diverse parti della sequenza in input per fare predizioni.

Questo permette di tener conto di relazioni e dipendenze che ci sono tra le diverse parole.

I Transformer utilizzano una struttura Encoder-Decoder: essi sono tipicamente utilizzati in compiti dove una sequenza in input è trasformata in una certa sequenza in output (ad esempio una traduzione).

Il modello è quindi composto da un Encoder per processare un input e fornire a sua volta un ulteriore input a un Decoder per generare l'output finale.

Una delle peculiarità che rende questo modello molto performante, è la sua capacità di parallelizzare i calcoli, che quindi permette di sfruttare le caratteristiche intrinseche alle GPU di parallelizzazione, impiegate normalmente nel rendering delle matrici di pixel degli schermi.

La computazione sulle parole in input può essere eseguita allo stesso momento, ad esempio, si possono calcolare gli Word Embeddings su diversi processori allo stesso momento e aggiungere il positional encoding allo stesso istante, così come per il calcolo delle Queries, Keys, Values e i valori di Self Attention.

La valutazione delle sequenze di parole può essere riassunta in questo modo:

- *Word Embedding*, codifica delle parole in numeri
- *Positional Encoding*, per tenere traccia dell'ordine delle parole, aggiungendo al loro valore numerico la corrispettiva codifica della posizione in cui si trovano
- *Self-Attention,* codifica delle relazioni tra le parole dentro le sequenze
- *Encoder-Decoder Attention*, per valutare le relazioni tra le sequenze di input e output

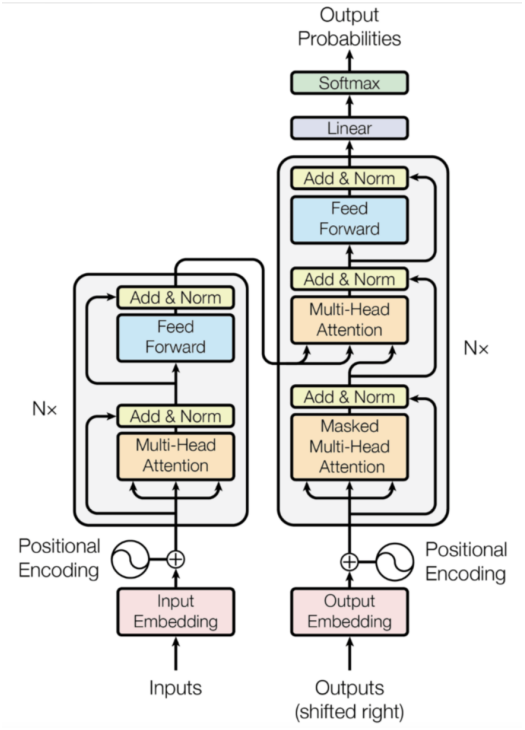


Figura 3.1: Architettura Transformer. Illustrazione tratta da [1]

3.2 Codifiche del Testo

Per poter fornire un input alla rete neurale, bisogna codificare le parole in valori numerici attraverso il Word Embedding.

In input le word vengono quindi associate a un token e poi moltiplicate per un peso.

Uno dei word embedding più comunemente utilizzato nei task di NLP (natural language processing), è il modello pre-trained **word2vec** [2], ma in questo caso non trova una corretta applicazione, in quanto i nomi di dominio sono spesso composti da una serie confusa di caratteri, e quindi viene meno l'utilità di avere un vettore con valori che indicano la quantità di informazione semantica su una certa dimensione [2].

Sono state create due varianti del modello, che differiscono nella tokenizzazione del testo in valori numerici nella fase di preparazione dei dati, che precede il training, e nella quale vengono presi i dati dal dataset, puliti e manipolati.

I nomi di dominio vengono dapprima resi tutti della stessa lunghezza, andando quindi a pareggiare quella della stringa più lunga, aggiungendo a tutti i nomi un pad di zeri come header.

Negli esperimenti, la stringa più lunga misura 46 caratteri, quindi ad esempio il dominio **google.it**, possedendo un totale di 8 caratteri (escluso il punto), subisce la concatenazione con una stringa da 38 zeri nel seguente modo: **00000000000000000000000000000000000000googleit**.

3.2.1 Vettori One-Hot

Successivamente alla suddivisione dei nomi di dominio in sequenze di bigrammi e caratteri, vengono creati i due rispettivi vocabolari di dimensioni L1=1334 e L2=34.

In seguito viene eseguita la codifica one-hot, che consiste nella creazione di vettori di lunghezza pari alla rispettiva dimensione di dizionario, quindi L1 per i bigrammi e L2 per i caratteri, che possiedono tutti zeri ad eccezione di un 1, che corrisponde alla posizione all'interno del vocabolario dell'elemento che si sta codificando.

Esempio di sequenza di caratteri: ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', 'n', 'o', 'n', 'e', 'w', 'h', 'e', 'r', 'e', 'n', 'e', 't']

Codifica: [array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), ⋯ , array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])]

Esempio di sequenza di bigrammi: ['00', '0n', 'no', 'on', 'ne', 'ew', 'wh', 'he', 'er', 're', 'en', 'ne', 'et']

Codifica: [array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), ⋯ , array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.])]

3.2.2 Codifica con Indici di Vocabolario

Come per la precedente, si creano i due vocabolari, e ciascun elemento viene codificato con il numero di indice utilizzato per identificarlo nel dizionario.

Di conseguenza, per ogni word, ovvero un carattere o un bigramma, avremo in output uno scalare intero che lo rappresenta.

Esempio di sequenza di caratteri: ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', 't', 'r', 'a', 'i', 'l', 'e', 'r', '-', 'd', 'e', 'f', 'e', 'n', 's', 'e', 'c', 'o', 'm']

Codifica: [32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 4, 29, 0, 36, 15, 7, 29, 27, 13, 7, 25, 7, 20, 23, 7, 33, 21, 3]

Esempio di sequenza di bigrammi: ['00', '0t', 'tr', 'ra', 'ai', 'il', 'le', 'er', 'r-', '-d', 'de', 'ef', 'fe', 'en', 'ns', 'se', 'ec', 'co', 'om']

Codifica: [637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 637, 122, 415, 1111, 309, 138, 318, 8, 725, 629, 486, 1281, 194, 909, 1052, 616, 74, 754, 307]

3.3 Embedding Ibrido

Il modello proposto prende il nome di Transformer Modificato con embedding ibrido perchè combina due tipologie di embedding:

- Word embedding a livello di caratteri, dove la stringa di dominio viene suddivisa dapprima in una sequenza di caratteri, e ciascuno di questi viene succesivamente codificato e trattato come una parola in input al layer di embedding.
- Word embedding a livello di bigrammi, in cui la stringa di dominio viene suddivisa in una sequenza di bigrammi, codificati e valutati anch'essi come singole parole in input.

3.4 Struttura del Modello

Di seguito vengono descritti i vari moduli che compongono l'architettura del modello, illustrata nella Figura 3.2.

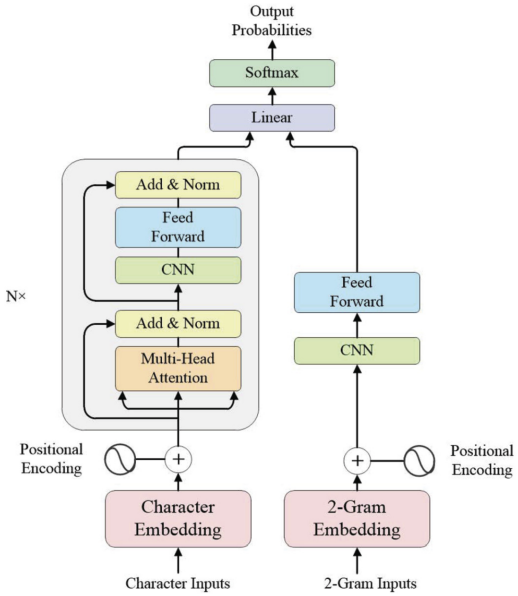


Figura 3.2: Architettura Hybrid-Modified Transformer. Illustrazione tratta da [2]

3.4.1 Embedding Layer

Abbiamo inizialmente delle sequenze di caratteri e bigrammi tokenizzati, che passano per un layer di embedding per ciascuna delle due tipologie.

La dimensione scelta è n=128 (128 embedding per ogni word). Negli esperimenti la lunghezza massima delle sequenze di caratteri è pari a $L_1 = 46$, mentre per i bigrammi è $L_2 = 45$, considerando che in ogni dominio è stato escluso il punto.

L'embedding consiste nel moltiplicare ciascun input per dei pesi, i quali sono i parametri che vengono ottimizzati nel processo di training:

$E_1 = W_1' X_{L1}$

$E_2 = W_2' X_{L2}$

3.4.2 Positional Encoding

Il Transformer ha bisogno di fare uso del valore posizionale di ciascuna parola, perciò questo layer rappresenta un'addizione dei termini definiti nelle Equazioni (3.1) e (3.2) e non necessita di essere ricalcolato e ottimizzato ad ogni iterazione del training, in questo modo ad ogni sequenza vengono sommati sempre gli stessi valori.

Viene utilizzato lo stesso positional encoding di [1]:

$$PE_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{\text{d\_model}}}}\right) \tag{3.1}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{\text{d\_model}}}}\right) \tag{3.2}$$

pos=posizione dell'elemento.
d_model=dimensione del modello (128).

Le codifiche di posizione hanno dimensione pari a d_model=128 e sono sommate separatamente con diversi valori per bigrammi e caratteri [2].

3.4.3 Encoder Modificato

Come per l'Encoder del Transformer originale, esso è composto da una serie di blocchi ripetuti, in questo caso N=3.

Multiheaded Self-Attention

La Self Attention calcola le relazioni che ci sono tra le parole nella sequenza di input.

Viene eseguita una trasformazione lineare all'input, che viene mappato in tre vettori: query e key di dimensione d_k, e values di dimensione d_v.

Sfruttando la parallelizzazione, si costruiscono le matrici Q, K, V e si calcola la self-attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.3}$$

Viene utilizzato un self-attention multiplo per imparare diverse caratteristiche, sfruttando head multiple che vengono poi concatenate [2]:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{3.4}$$

Dove

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3.5}$$

Sono state scelte h=8 head, e la dimensione dell'output è 128 con la stessa lunghezza della sequenza di input [2].

Layer di Convoluzione nel blocco

Rispetto alla struttura originale, viene aggiunto un layer di convoluzione monodimensionale, che prende in input l'output della self-attention.

Viene creato un vettore per ogni elemento in posizione j:

$$w_j = [e_j, e_{j+1}, \ldots, e_{j+k-1}] \tag{3.6}$$

Ognuno genera una mappa di caratteristiche c_j:

$$c_j = f((w_j \odot m) + b) \tag{3.7}$$

Dove $\odot$ è la moltiplicazione per elemento dei due vettori, b è il bias e f è la funzione di attivazione ReLU.

Infine, queste vengono concatenate per riga per ottenere la feature map del nome di dominio C, le quali vengono poi concatenate per colonne [2].

La dimensione delle window è k=3, vengono usati 256 kernel e lo stesso padding, per mantenere invariata la lunghezza della sequenza.

Feed-Forward Network

Consiste in una trasformazione lineare con la funzione di attivazione ReLU: $FFN(x) = f(Wx + b)$ Con un input di dimensione 256 e un output di d_model=128.

Effettua la rimodellazione delle feature maps generate dalla CNN, lungo la stessa dimensione del d_model. Serve ad imparare le caratteristiche principali fornite dai layer precedenti.

3.4.4 Rete dei Bigrammi

CNN e FFN che funzionano come quelle nel blocco del Transformer, ma vengono utilizzati 64 kernels per la convoluzione [2].

3.4.5 Layer di Concatenazione

Vengono concatenate le caratteristiche estratte dalla sequenza a livello di caratteri e da quella a livello di bigrammi.

3.4.6 Dense Layer

Nell'ultima fase viene effettuata la classificazione. Questo layer è composto da un numero di nodi pari al numero delle classi (labels) delle famiglie di domini e da neuroni che applicano la funzione:

$$\hat{y} = f(W_d \cdot \text{feature} + b_d) \tag{3.8}$$

Dove f è la funzione di attivazione non lineare adibita alla classificazione, che per problemi di classificazione multi-classe è rappresentata tipicamente dalla softmax:

$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{con } j = 1, 2, \ldots, L \tag{3.9}$$

Capitolo 4

Implementazione

4.1 Dataset UMUDGA

Per gli esperimenti è stato utilizzato il dataset UMUDGA [5, 6], creato da un gruppo di ricercatori dell'università di Murcia e disponibile gratuitamente.

Esso contiene domini etichettati manualmente su 51 classi, di cui una 'legit' contenente nomi benevoli.

Le restanti sono una selezione di 50 famiglie DGA appartenenti alle diverse categorie descritte nel Capitolo 2.3, con alcuni esempi di queste nelle Tabelle 2.1, 2.2, 2.3.

Il dataset è reso disponibile in tre formati: arff, txt e csv.

Ciascuno viene fornito con varianti che differiscono per numero di elementi contenuti, con il txt che arriva fino a 50000 samples per classe.

Per l'esperimento è stato scelto il formato csv, il quale è composto da una tabella con 131 colonne, di cui la prima rappresenta ciascun nome di dominio e l'ultima il nome della famiglia di appartenenza.

In particolare è stato selezionato il taglio da 1000, cioè in cui ciascuna delle 51 classi possiede circa 1000 elementi, con il dataset nel suo complesso che ne contiene 50898.

Le dimensioni dei vocabolari sono pari a 1344 per i bigrammi e 37 per i caratteri.

Nella fase di estrazione dei dati, le stringhe dei nomi di dominio vengono duplicate, in modo da avere per ciascuna due input differenti, uno successivamente suddiviso in monogrammi e l'altro che verrà suddiviso in bigrammi, ignorando il punto in entrambi.

Facendo un esempio con **google.it**:

- Bigrammi : ['**go**' '**oo**' '**og**' '**gl**' '**le**' '**ei**' '**it**']
- Caratteri : ['**g**' '**o**' '**o**' '**g**' '**l**' '**e**' '**i**' '**t**']

L'80% del dataset è utilizzato per il training, mentre il restante 20% per la validazione.

Per sfruttare tutti i domini in entrambe le fasi, ci si serve della k-fold cross validation 4.3, che è una tecnica per valutare le prestazioni del modello, grazie alla quale tutti i domini vengono utilizzati ciclicamente nell'allenamento e nella validazione.

4.2 Packages

Per l'implementazione sono stati utilizzati alcuni moduli dei package di PyTorch[1], Pandas [7] e Sklearn [8, 9].

Pytorch è un framework di programmazione per il deep learning basato su Python, che supporta anche l'accelerazione GPU per il training dei modelli, ed è servito per l'implementazione dell'intero modello e per il suo allenamento e validazione.

Pandas è stato sfruttato nella fase di estrazione, pulizia e manipolazione dei dati dai vari file csv.

Sklearn fornisce il modulo KFold[2] per applicare la kfold cross validation e le librerie per il calcolo di metriche per le valutazioni delle performance, rappresentate graficamente sfruttando il modulo pyplot del package Matplotlib [10].

4.3 K-Fold Cross Validation

La Kfold cross validation è una tecnica utilizzata per valutare le performance di un modello. Essa prevede la suddivisione dell'intero dataset in k sottoinsiemi della stessa dimensione, di cui k-1 utilizzati per l'allenamento e il rimanente per la validazione.

Tutto questo processo viene ripetuto per k volte, così ogni fold viene usata esattamente una volta come validation data. Al termine di un'iterazione vengono azzerati i pesi e la successiva riparte da zero, con una differente distribuzione dei dati per l'allenamento e la conseguente validazione. La stima delle performance generali del modello è rappresentata dalla media delle k iterazioni.

In questo modo si sfrutta tutto il dataset per entrambe le fasi, evitando problemi di suddivisione statica in dati di training e validation.

Kfold cv è una tecnica molto diffusa per la validazione e il tuning degli iperparametri dei modelli di machine learning, utile per stimare le loro prestazioni su nuovi set di dati.

4.4 Layer di Dropout

I layer di Dropout rappresentano una tecnica di regolarizzazione applicata alle reti neurali per prevenirne l'overfitting. Questo problema avviene quando un modello si adatta troppo a un set di dati, catturandone il rumore e variazioni casuali, e conseguenza di ciò è una performance elevata nei dati di training e scarsa su dati nuovi, non visti in precedenza. Quindi l'overfitting induce il modello ad essere troppo complesso e inefficace nella generalizzazione di ciò che impara.

Perciò durante il training, il dropout imposta casualmente una parte degli input ad una funzione a zero, in modo da evitare che la rete faccia troppo affidamento

[1] `https://github.com/pytorch/pytorch`

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html`

su input specifici e venga costretta ad imparare una rappresentazione più robusta e generalizzata dei dati.

Il dropout può essere applicato a più tipologie di layer, con uno specifico valore, solitamente tra 0.2 a 0.5, che rappresenta la probabilità di impostare un neurone a zero.

Nella loro prima citazione [11], i dropout, con probabilità p=0.5, vengono utilizzati su ciascun fully connected layer (dense layer) prima dell'output. Questa è la configurazione più diffusa, anche se recentemente [12] sono state proposte ulteriori soluzioni che prevedono la loro applicazione nelle CNN, dopo la funzione di attivazione di ciascun layer di convoluzione, con p=0.1 o p=0.2.

Nella struttura originale dei Transformers [1], vengono utilizzati dei dropout con p=0.1 anche nelle somme di embeddings con i positional encodings.

In questo modello sono presenti con p=0.2 dopo la funzione di attivazione ReLU nelle Feed Forward Neural Networks (FFN), e con p=0.1 dopo il Multihead Self-Attention dell'encoder.

In due varianti del modello è presente la **Batch Normalization**, che è sempre una tecnica di regolarizzazione, ma a differenza del dropout, non elimina un certo ratio predefinito di input, ma va a normalizzare ciascuna unità di batch con una sua deviazione standard e media.

4.5 Iperparametri per il Training e la Validazione

La funzione di loss utilizzata è la Cross Entropy Loss, molto diffusa nei problemi di classificazione multipla.

L'allenamento permette quindi di ottimizzare la seguente funzione [2]:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{\text{class}} y_i^{(i)}\hat{y}_j^{(i)} \tag{4.1}$$

Per ottimizzare la funzione obiettivo si usa il package torch.optim[3], da cui si è scelto l'algoritmo di ottimizzazione Adam[4]. Nella Tabella 4.1 sono riportati i vari iperparametri scelti per gli esperimenti.

Iperparametri dell'algoritmo di ottimizzazione	learning rate = 0.001 $\beta_1 = 0.9$ $\beta_2 = 0.999$ $\varepsilon = 10^{-8}$
Numero di folds	5
Iperparametri del training loop	n° batch = 30 epoche = 6

Tabella 4.1: Iperparametri di Allenamento e Validazione

[3] `https://pytorch.org/docs/stable/optim.html`
[4] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

Capitolo 5

Risultati Sperimentali

Di seguito sono riportati i risultati ottenuti dalle due differenti versioni del modello descritto nei capitoli precedenti.

Inizialmente sono presenti i dati di allenamento attraverso ciascuna epoca, in particolare sono mostrati i valori di perdita (loss) ogni 300 iterazioni del training loop.

Vengono poi riportati i grafici che rappresentano l'andamento dell'*accuratezza* (5.1), della *precisione* (5.2), della *recall* (5.3) ed il *punteggio F1* (5.4), per ogni epoca di training e per ciascuna fold.

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

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

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

$$F1\ Score = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

TP = True positives, è il numero di elementi per cui la famiglia predetta coincide con quella reale.
TN = True negatives, è il numero di elementi valutati correttamente come non appartenenti ad una classe.
FN = False negatives, è il numero di elementi predetti incorrettamente, cioè nella classificazione multipla, significa che il modello ha associato ad un dominio una classe diversa da quella reale.
FP = False positives, è il numero degli elementi che sono stati predetti per una determinata classe in maniera errata.
Nº Predizioni Corrette = TP + TN
Predizioni Totali = Numero Totale di Samples = TP + TN + FP + FN

Gli stessi parametri vengono utilizzati anche per valutare le prestazioni sulle singole famiglie di domini.

5.1 Modello con Codifica di Vettori One-Hot

5.1.1 Output di Esecuzione

Di seguito sono riportati i dati forniti durante l'esecuzione dell'esperimento del modello che utilizza la codifica con vettori one-hot. Vengono mostrati i parametri di Accuracy, Precision, Recall e F1 per valutare le performance, ed il valore d'errore ogni 300 step del training loop.

In quest'unico esperimento viene fatta una regolarizzazione attraverso layer di Dropout.

Fold 1/5

epoch 1 / 6, step 300/1358, loss = 107.6193
epoch 1 / 6, step 600/1358, loss = 12.1391
epoch 1 / 6, step 900/1358, loss = 9.1245
epoch 1 / 6, step 1200/1358, loss = 2.3958
Epoch 1/6:
Accuracy: 0.4764
Precision: 0.7067
Recall: 0.4764
F1 Score: 0.4680

---

epoch 2 / 6, step 300/1358, loss = 5.9589
epoch 2 / 6, step 600/1358, loss = 1.9789
epoch 2 / 6, step 900/1358, loss = 5.9375
epoch 2 / 6, step 1200/1358, loss = 4.3359
Epoch 2/6:
Accuracy: 0.6158
Precision: 0.7086
Recall: 0.6158
F1 Score: 0.5961

---

epoch 3 / 6, step 300/1358, loss = 1.3397
epoch 3 / 6, step 600/1358, loss = 2.7442
epoch 3 / 6, step 900/1358, loss = 2.1872
epoch 3 / 6, step 1200/1358, loss = 1.0630
Epoch 3/6:
Accuracy: 0.7070
Precision: 0.7277
Recall: 0.7070
F1 Score: 0.6999

---

epoch 4 / 6, step 300/1358, loss = 0.0002
epoch 4 / 6, step 600/1358, loss = 0.8470
epoch 4 / 6, step 900/1358, loss = 1.9966
epoch 4 / 6, step 1200/1358, loss = 0.0000
Epoch 4/6:
Accuracy: 0.7328
Precision: 0.7475
Recall: 0.7328
F1 Score: 0.7274

---

epoch 5 / 6, step 300/1358, loss = 0.0000
epoch 5 / 6, step 600/1358, loss = 0.0000
epoch 5 / 6, step 900/1358, loss = 3.7515
epoch 5 / 6, step 1200/1358, loss = 1.4077
Epoch 5/6:
Accuracy: 0.6915
Precision: 0.7240
Recall: 0.6915
F1 Score: 0.6872

---

epoch 6 / 6, step 300/1358, loss = 0.0016
epoch 6 / 6, step 600/1358, loss = 0.5112
epoch 6 / 6, step 900/1358, loss = 0.0562
epoch 6 / 6, step 1200/1358, loss = 2.9933
Epoch 6/6:
Accuracy: 0.7250
Precision: 0.7282
Recall: 0.7250
F1 Score: 0.7154

Fold 2/5

epoch 1 / 6, step 300/1358, loss = 84.1236
epoch 1 / 6, step 600/1358, loss = 17.9234
epoch 1 / 6, step 900/1358, loss = 10.4318
epoch 1 / 6, step 1200/1358, loss = 6.7129
Epoch 1/6:
Accuracy: 0.5087
Precision: 0.7321
Recall: 0.5087
F1 Score: 0.5240

---

epoch 2 / 6, step 300/1358, loss = 2.2194
epoch 2 / 6, step 600/1358, loss = 4.8000
epoch 2 / 6, step 900/1358, loss = 7.4280
epoch 2 / 6, step 1200/1358, loss = 4.3728
Epoch 2/6:
Accuracy: 0.5907

Precision: 0.7255
Recall: 0.5907
F1 Score: 0.5963

---

epoch 3 / 6, step 300/1358, loss = 0.2184
epoch 3 / 6, step 600/1358, loss = 2.0934
epoch 3 / 6, step 900/1358, loss = 2.3666
epoch 3 / 6, step 1200/1358, loss = 2.1633
Epoch 3/6:
Accuracy: 0.7041
Precision: 0.7356
Recall: 0.7041
F1 Score: 0.7032

---

epoch 4 / 6, step 300/1358, loss = 0.0019
epoch 4 / 6, step 600/1358, loss = 0.0007
epoch 4 / 6, step 900/1358, loss = 3.1393
epoch 4 / 6, step 1200/1358, loss = 0.7778
Epoch 4/6:
Accuracy: 0.7106
Precision: 0.7414
Recall: 0.7106
F1 Score: 0.7096

---

epoch 5 / 6, step 300/1358, loss = 0.2058
epoch 5 / 6, step 600/1358, loss = 0.0000
epoch 5 / 6, step 900/1358, loss = 2.3545
epoch 5 / 6, step 1200/1358, loss = 5.1331
Epoch 5/6:
Accuracy: 0.7093
Precision: 0.7352
Recall: 0.7093
F1 Score: 0.7087

---

epoch 6 / 6, step 300/1358, loss = 1.0949
epoch 6 / 6, step 600/1358, loss = 8.5256
epoch 6 / 6, step 900/1358, loss = 7.3803
epoch 6 / 6, step 1200/1358, loss = 0.0000
Epoch 6/6:
Accuracy: 0.7166
Precision: 0.7381
Recall: 0.7166
F1 Score: 0.7181

Fold 3/5

epoch 1 / 6, step 300/1358, loss = 122.8906
epoch 1 / 6, step 600/1358, loss = 6.3820
epoch 1 / 6, step 900/1358, loss = 6.1097
epoch 1 / 6, step 1200/1358, loss = 6.8664
Epoch 1/6:
Accuracy: 0.5223
Precision: 0.7362
Recall: 0.5223
F1 Score: 0.5339

---

epoch 2 / 6, step 300/1358, loss = 5.5073
epoch 2 / 6, step 600/1358, loss = 5.3330
epoch 2 / 6, step 900/1358, loss = 5.6341
epoch 2 / 6, step 1200/1358, loss = 2.6632
Epoch 2/6:
Accuracy: 0.6681
Precision: 0.7230
Recall: 0.6681
F1 Score: 0.6581

---

epoch 3 / 6, step 300/1358, loss = 0.3129
epoch 3 / 6, step 600/1358, loss = 0.7587
epoch 3 / 6, step 900/1358, loss = 1.5883
epoch 3 / 6, step 1200/1358, loss = 1.0300
Epoch 3/6:
Accuracy: 0.7037
Precision: 0.7409
Recall: 0.7037
F1 Score: 0.6993

---

epoch 4 / 6, step 300/1358, loss = 1.0913
epoch 4 / 6, step 600/1358, loss = 1.3389
epoch 4 / 6, step 900/1358, loss = 3.8972
epoch 4 / 6, step 1200/1358, loss = 2.1147
Epoch 4/6:
Accuracy: 0.7325
Precision: 0.7319
Recall: 0.7325
F1 Score: 0.7223

---

epoch 5 / 6, step 300/1358, loss = 0.2206
epoch 5 / 6, step 600/1358, loss = 1.8980
epoch 5 / 6, step 900/1358, loss = 1.4212
epoch 5 / 6, step 1200/1358, loss = 0.2847
Epoch 5/6:
Accuracy: 0.7251
Precision: 0.7312
Recall: 0.7251
F1 Score: 0.7204

---

epoch 6 / 6, step 300/1358, loss = 0.0000
epoch 6 / 6, step 600/1358, loss = 0.0000
epoch 6 / 6, step 900/1358, loss = 2.4191
epoch 6 / 6, step 1200/1358, loss = 5.1989
Epoch 6/6:
Accuracy: 0.7229
Precision: 0.7231
Recall: 0.7229
F1 Score: 0.7165

Fold 4/5

epoch 1 / 6, step 300/1358, loss = 143.2125
epoch 1 / 6, step 600/1358, loss = 9.0069
epoch 1 / 6, step 900/1358, loss = 6.5375
epoch 1 / 6, step 1200/1358, loss = 11.5256
Epoch 1/6:
Accuracy: 0.5516
Precision: 0.6877
Recall: 0.5516
F1 Score: 0.5178

---

epoch 2 / 6, step 300/1358, loss = 4.5554
epoch 2 / 6, step 600/1358, loss = 3.5623
epoch 2 / 6, step 900/1358, loss = 5.8108
epoch 2 / 6, step 1200/1358, loss = 3.8212
Epoch 2/6:
Accuracy: 0.5931
Precision: 0.7216
Recall: 0.5931
F1 Score: 0.5664

---

epoch 3 / 6, step 300/1358, loss = 0.0906
epoch 3 / 6, step 600/1358, loss = 2.6199
epoch 3 / 6, step 900/1358, loss = 6.8945
epoch 3 / 6, step 1200/1358, loss = 4.8782
Epoch 3/6:
Accuracy: 0.6540
Precision: 0.7228
Recall: 0.6540
F1 Score: 0.6508

---

epoch 4 / 6, step 300/1358, loss = 3.2866
epoch 4 / 6, step 600/1358, loss = 0.1259
epoch 4 / 6, step 900/1358, loss = 0.2934
epoch 4 / 6, step 1200/1358, loss = 0.7603
Epoch 4/6:
Accuracy: 0.6974
Precision: 0.7332
Recall: 0.6974
F1 Score: 0.6991

---

epoch 5 / 6, step 300/1358, loss = 0.0000
epoch 5 / 6, step 600/1358, loss = 0.0000
epoch 5 / 6, step 900/1358, loss = 0.0000
epoch 5 / 6, step 1200/1358, loss = 0.1360
Epoch 5/6:
Accuracy: 0.7163
Precision: 0.7376
Recall: 0.7163
F1 Score: 0.7080

---

epoch 6 / 6, step 300/1358, loss = 4.1611
epoch 6 / 6, step 600/1358, loss = 0.7679
epoch 6 / 6, step 900/1358, loss = 0.0000
epoch 6 / 6, step 1200/1358, loss = 8.2095
Epoch 6/6:
Accuracy: 0.7280
Precision: 0.7358
Recall: 0.7280
F1 Score: 0.7224

Fold 5/5

epoch 1 / 6, step 300/1358, loss = 143.9898
epoch 1 / 6, step 600/1358, loss = 14.1627
epoch 1 / 6, step 900/1358, loss = 6.4447
epoch 1 / 6, step 1200/1358, loss = 6.3456
Epoch 1/6:
Accuracy: 0.3974
Precision: 0.6974
Recall: 0.3974
F1 Score: 0.4086

---

epoch 2 / 6, step 300/1358, loss = 5.2779
epoch 2 / 6, step 600/1358, loss = 2.5862
epoch 2 / 6, step 900/1358, loss = 4.7120
epoch 2 / 6, step 1200/1358, loss = 5.8614
Epoch 2/6:
Accuracy: 0.6121
Precision: 0.6783
Recall: 0.6121
F1 Score: 0.5868

---

epoch 3 / 6, step 300/1358, loss = 0.3673
epoch 3 / 6, step 600/1358, loss = 0.1175
epoch 3 / 6, step 900/1358, loss = 1.3994
epoch 3 / 6, step 1200/1358, loss = 2.4861
Epoch 3/6:
Accuracy: 0.6467
Precision: 0.7360
Recall: 0.6467
F1 Score: 0.6552

---

epoch 4 / 6, step 300/1358, loss = 0.3096
epoch 4 / 6, step 600/1358, loss = 0.4836
epoch 4 / 6, step 900/1358, loss = 0.0253
epoch 4 / 6, step 1200/1358, loss = 3.8482
Epoch 4/6:
Accuracy: 0.7136
Precision: 0.7354
Recall: 0.7136
F1 Score: 0.7095

---

epoch 5 / 6, step 300/1358, loss = 0.0727
epoch 5 / 6, step 600/1358, loss = 1.0014
epoch 5 / 6, step 900/1358, loss = 0.4448
epoch 5 / 6, step 1200/1358, loss = 13.1587
Epoch 5/6:
Accuracy: 0.7297
Precision: 0.7399
Recall: 0.7297
F1 Score: 0.7245

---

epoch 6 / 6, step 300/1358, loss = 1.3959
epoch 6 / 6, step 600/1358, loss = 2.4462
epoch 6 / 6, step 900/1358, loss = 0.0000
epoch 6 / 6, step 1200/1358, loss = 2.8738
Epoch 6/6:
Accuracy: 0.7148
Precision: 0.7460
Recall: 0.7148
F1 Score: 0.7195

5.1.2 Grafici di Validazione

Nei seguenti grafici è possibile visualizzare l'andamento delle curve di Accuracy (Figura 5.1), Precision (Figura 5.2), Recall (Figura 5.3) e F1 (Figura 5.4). Nelle ascisse sono riportate le epoche di allenamento, mentre nelle ordinate si indicano i valori del parametro che si sta analizzando, con la stessa scala per ciascun grafico. Si possono notare degli andamenti molto simili, fatta eccezione per la Precision, che rimane pressoché invariata sin dalla prima epoca.

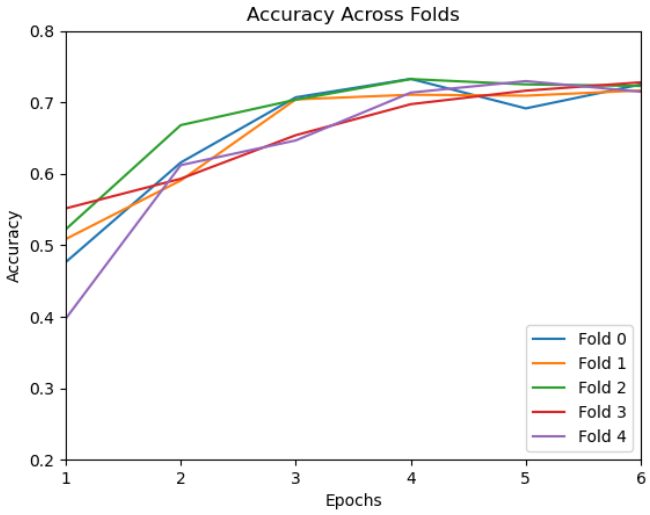


Figura 5.1: Accuracy One-Hot Encoding

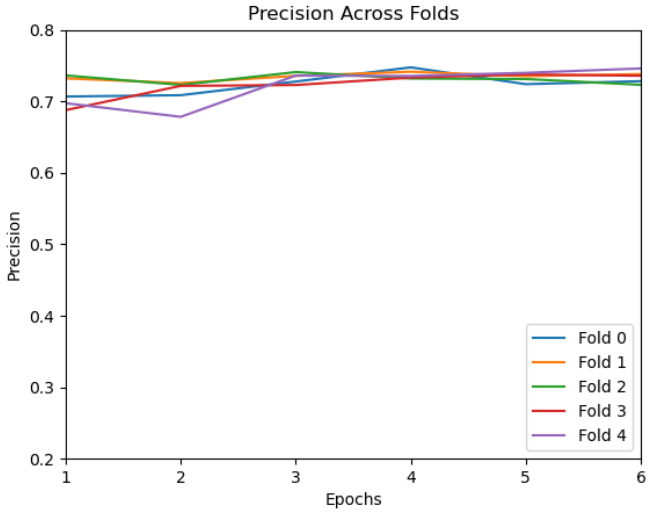


Figura 5.2: Precision One-Hot Encoding

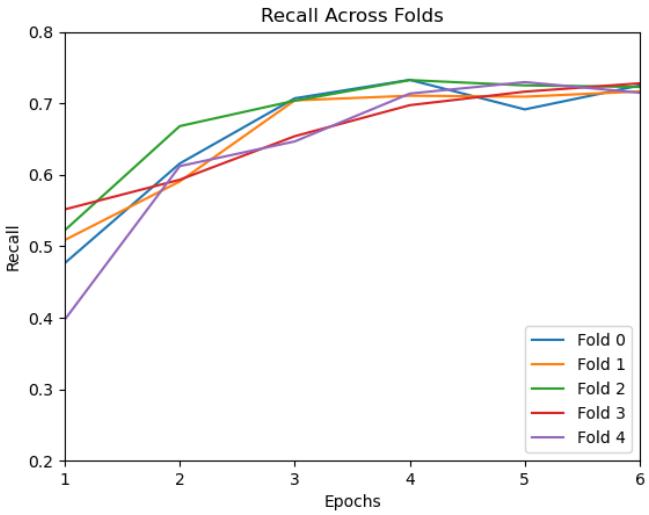


Figura 5.3: Recall One-Hot Encoding

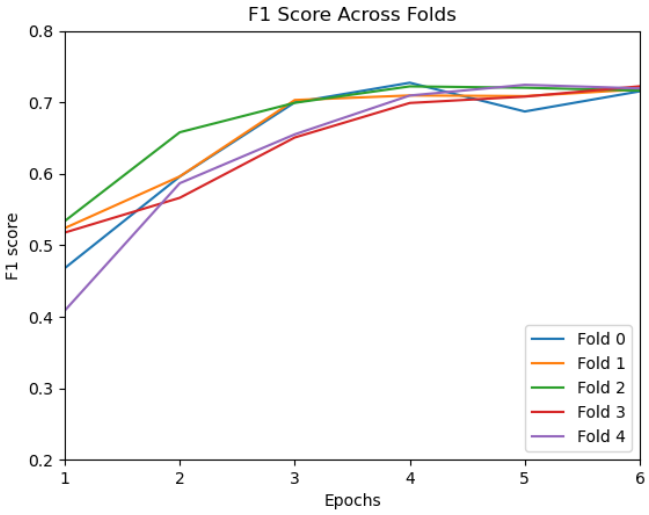


Figura 5.4: F1 Score One-Hot Encoding

5.1.3 Risultati di Classificazione Multiclasse

Nella Tabella 5.1 vengono riportate le performance di Precision, Recall ed F1 score che il modello ha registrato su ciascuna famiglia di DGA, con indicato anche il numero totale di elementi presenti in ognuna di esse.

Domain Family	**Precision**	**Recall**	**F1-Score**	**Support**
rovnix	0.74	0.73	0.73	1000
qadars	0.77	0.90	0.83	1000
gozi_luther	0.79	0.62	0.70	1000
gozi_rfc4343	0.59	0.53	0.55	995
qakbot	0.30	0.27	0.28	1000
cryptolocker	0.35	0.38	0.36	1000
zeus-newgoz	0.91	0.86	0.88	1000
chinad	0.79	0.66	0.72	1000
suppobox_3	0.99	0.99	0.99	1000
fobber_v1	0.81	0.96	0.88	1000
legit	0.70	0.63	0.66	1000
vawtrak_v2	0.92	0.97	0.94	1000
matsnu	0.78	0.80	0.79	996
bedep	0.37	0.31	0.34	1000
simda	0.96	0.95	0.95	1000
symmi	0.91	0.98	0.94	1000
fobber_v2	0.36	0.46	0.40	1000
corebot	0.99	0.90	0.94	1000
necurs	0.57	0.64	0.60	1000
murofet_v1	0.96	0.96	0.96	1000
ranbyus_v2	0.69	0.73	0.71	1000
nymaim	0.78	0.79	0.78	1000
tinba	0.55	0.76	0.64	1000
dyre	1.00	0.98	0.99	1000
vawtrak_v1	0.98	0.98	0.98	1000
pushdo	0.95	0.88	0.91	1000
kraken_v1	0.79	0.75	0.77	1000
gozi_gpl	0.83	0.85	0.84	1000
pykspa_noise	0.26	0.28	0.27	1000
alureon	0.40	0.29	0.34	1000
murofet_v3	0.98	1.00	0.99	1000
suppobox_1	0.91	0.97	0.94	962
proslikefan	0.52	0.52	0.52	1000
ccleaner	0.99	1.00	1.00	1000
kraken_v2	0.57	0.53	0.55	1000
locky	0.59	0.45	0.51	1000
ranbyus_v1	0.70	0.77	0.73	1000
pykspa	0.27	0.21	0.24	1000
ramnit	0.17	0.24	0.20	1000
dircrypt	0.21	0.20	0.20	1000
ramdo	0.93	1.00	0.96	1000

sisron	1.00	1.00	1.00	1000
gozi_nasa	0.55	0.63	0.59	991
tempedreve	0.60	0.53	0.57	1000
shiotob	0.53	0.33	0.41	1000
pizd	0.95	0.91	0.93	956
padcrypt	0.97	0.96	0.96	1000
suppobox_2	0.98	0.99	0.99	998
banjori	1.00	1.00	1.00	1000
murofet_v2	0.81	0.86	0.84	1000
vawtrak_v3	0.87	0.90	0.89	1000
Macro Average	0.72	0.72	0.72	50898
Weighted Average	0.72	0.72	0.72	50898

Tabella 5.1: Classification Report per la codifica One-Hot

5.2 Modello a Codifica in Indici di Vocabolario

5.2.1 Regolarizzazione con Layer di Dropout

Output di Esecuzione

Di seguito sono riportati i dati forniti durante l'esecuzione dell'esperimento del modello che utilizza la codifica con indici di vocabolario. Vengono mostrati i parametri di Accuracy, Precision, Recall e F1 per valutare le performance, ed il valore d'errore ogni 300 step del training loop.

La regolarizzazione del modello avviene sfruttando layer di Dropout.

```
Fold 1/5

epoch 1 / 6, step 300/1358, loss = 1.0291
epoch 1 / 6, step 600/1358, loss = 1.5417
epoch 1 / 6, step 900/1358, loss = 0.7735
epoch 1 / 6, step 1200/1358, loss = 1.0567
Epoch 1/6:
Accuracy: 0.7638
Precision: 0.7616
Recall: 0.7638
F1 Score: 0.7546

---

epoch 2 / 6, step 300/1358, loss = 0.5696
epoch 2 / 6, step 600/1358, loss = 0.4813
epoch 2 / 6, step 900/1358, loss = 0.8984
epoch 2 / 6, step 1200/1358, loss = 0.8890
Epoch 2/6:
Accuracy: 0.7960
Precision: 0.7995
Recall: 0.7960
F1 Score: 0.7891

---

epoch 3 / 6, step 300/1358, loss = 0.6545
epoch 3 / 6, step 600/1358, loss = 0.9390
epoch 3 / 6, step 900/1358, loss = 0.6177
epoch 3 / 6, step 1200/1358, loss = 0.6108
Epoch 3/6:
Accuracy: 0.8057
Precision: 0.8162
Recall: 0.8057
F1 Score: 0.8014

---

epoch 4 / 6, step 300/1358, loss = 0.5426
epoch 4 / 6, step 600/1358, loss = 0.3816
epoch 4 / 6, step 900/1358, loss = 0.3470
epoch 4 / 6, step 1200/1358, loss = 0.6445
Epoch 4/6:
Accuracy: 0.8087
Precision: 0.8128
Recall: 0.8087
F1 Score: 0.8026

---

epoch 5 / 6, step 300/1358, loss = 0.7627
epoch 5 / 6, step 600/1358, loss = 0.2701
epoch 5 / 6, step 900/1358, loss = 0.5171
epoch 5 / 6, step 1200/1358, loss = 0.4733
Epoch 5/6:
Accuracy: 0.8078
Precision: 0.8130
Recall: 0.8078
F1 Score: 0.8028

---

epoch 6 / 6, step 300/1358, loss = 0.2153
epoch 6 / 6, step 600/1358, loss = 0.2010
epoch 6 / 6, step 900/1358, loss = 0.7421
epoch 6 / 6, step 1200/1358, loss = 0.4346
Epoch 6/6:
Accuracy: 0.7952
Precision: 0.8088
Recall: 0.7952
F1 Score: 0.7953

---

Fold 2/5

epoch 1 / 6, step 300/1358, loss = 1.1340
epoch 1 / 6, step 600/1358, loss = 0.6688
epoch 1 / 6, step 900/1358, loss = 0.7340
epoch 1 / 6, step 1200/1358, loss = 0.8216
Epoch 1/6:
Accuracy: 0.7732
Precision: 0.7766
Recall: 0.7732
F1 Score: 0.7565

---

epoch 2 / 6, step 300/1358, loss = 0.6858
epoch 2 / 6, step 600/1358, loss = 1.0095
epoch 2 / 6, step 900/1358, loss = 0.9784
epoch 2 / 6, step 1200/1358, loss = 0.3445
Epoch 2/6:
Accuracy: 0.7925
Precision: 0.7983
Recall: 0.7925
F1 Score: 0.7820

---

epoch 3 / 6, step 300/1358, loss = 0.6743
epoch 3 / 6, step 600/1358, loss = 0.5840
epoch 3 / 6, step 900/1358, loss = 0.2483
epoch 3 / 6, step 1200/1358, loss = 0.4968
Epoch 3/6:
Accuracy: 0.8084
```

Precision: 0.8166
Recall: 0.8084
F1 Score: 0.8024

---

epoch 4 / 6, step 300/1358, loss = 0.1656
epoch 4 / 6, step 600/1358, loss = 0.2520
epoch 4 / 6, step 900/1358, loss = 0.2251
epoch 4 / 6, step 1200/1358, loss = 0.6621
Epoch 4/6:
Accuracy: 0.7989
Precision: 0.8020
Recall: 0.7989
F1 Score: 0.7919

---

epoch 5 / 6, step 300/1358, loss = 0.7353
epoch 5 / 6, step 600/1358, loss = 0.3480
epoch 5 / 6, step 900/1358, loss = 0.4766
epoch 5 / 6, step 1200/1358, loss = 0.4602
Epoch 5/6:
Accuracy: 0.8098
Precision: 0.8128
Recall: 0.8098
F1 Score: 0.8058

---

epoch 6 / 6, step 300/1358, loss = 0.4396
epoch 6 / 6, step 600/1358, loss = 0.5442
epoch 6 / 6, step 900/1358, loss = 0.2224
epoch 6 / 6, step 1200/1358, loss = 0.5591
Epoch 6/6:
Accuracy: 0.8072
Precision: 0.8119
Recall: 0.8072
F1 Score: 0.8051

Fold 3/5

epoch 1 / 6, step 300/1358, loss = 1.1576
epoch 1 / 6, step 600/1358, loss = 0.8506
epoch 1 / 6, step 900/1358, loss = 0.7802
epoch 1 / 6, step 1200/1358, loss = 0.9144
Epoch 1/6:
Accuracy: 0.7488
Precision: 0.7886
Recall: 0.7488
F1 Score: 0.7351

---

epoch 2 / 6, step 300/1358, loss = 0.7740
epoch 2 / 6, step 600/1358, loss = 0.6905
epoch 2 / 6, step 900/1358, loss = 0.5143
epoch 2 / 6, step 1200/1358, loss = 0.7511
Epoch 2/6:
Accuracy: 0.7869
Precision: 0.7946
Recall: 0.7869
F1 Score: 0.7752

---

epoch 3 / 6, step 300/1358, loss = 0.9887
epoch 3 / 6, step 600/1358, loss = 0.2705
epoch 3 / 6, step 900/1358, loss = 0.5143
epoch 3 / 6, step 1200/1358, loss = 1.0015
Epoch 3/6:
Accuracy: 0.7923
Precision: 0.7981
Recall: 0.7923
F1 Score: 0.7897

---

epoch 4 / 6, step 300/1358, loss = 0.5013
epoch 4 / 6, step 600/1358, loss = 0.3351
epoch 4 / 6, step 900/1358, loss = 0.5391
epoch 4 / 6, step 1200/1358, loss = 0.5010
Epoch 4/6:
Accuracy: 0.7966
Precision: 0.8027
Recall: 0.7966
F1 Score: 0.7936

---

epoch 5 / 6, step 300/1358, loss = 0.4296
epoch 5 / 6, step 600/1358, loss = 0.3587
epoch 5 / 6, step 900/1358, loss = 0.6629
epoch 5 / 6, step 1200/1358, loss = 0.6947
Epoch 5/6:
Accuracy: 0.8023
Precision: 0.8078
Recall: 0.8023
F1 Score: 0.7954

---

epoch 6 / 6, step 300/1358, loss = 0.3780
epoch 6 / 6, step 600/1358, loss = 0.3201
epoch 6 / 6, step 900/1358, loss = 0.5098
epoch 6 / 6, step 1200/1358, loss = 0.5371
Epoch 6/6:
Accuracy: 0.8033
Precision: 0.8066
Recall: 0.8033
F1 Score: 0.7997

Fold 4/5

epoch 1 / 6, step 300/1358, loss = 1.1127
epoch 1 / 6, step 600/1358, loss = 1.2496
epoch 1 / 6, step 900/1358, loss = 0.9581
epoch 1 / 6, step 1200/1358, loss = 1.2636
Epoch 1/6:
Accuracy: 0.7776
Precision: 0.7903
Recall: 0.7776
F1 Score: 0.7700

---

epoch 2 / 6, step 300/1358, loss = 1.0829
epoch 2 / 6, step 600/1358, loss = 0.5345
epoch 2 / 6, step 900/1358, loss = 1.2021
epoch 2 / 6, step 1200/1358, loss = 0.6078
Epoch 2/6:
Accuracy: 0.7979
Precision: 0.8031
Recall: 0.7979
F1 Score: 0.7852

---

epoch 3 / 6, step 300/1358, loss = 1.0251
epoch 3 / 6, step 600/1358, loss = 0.2280
epoch 3 / 6, step 900/1358, loss = 0.5116
epoch 3 / 6, step 1200/1358, loss = 0.3711
Epoch 3/6:
Accuracy: 0.8009
Precision: 0.8161
Recall: 0.8009
F1 Score: 0.7956

---

epoch 4 / 6, step 300/1358, loss = 0.8419
epoch 4 / 6, step 600/1358, loss = 0.7741
epoch 4 / 6, step 900/1358, loss = 0.6802
epoch 4 / 6, step 1200/1358, loss = 0.2833
Epoch 4/6:
Accuracy: 0.8158
Precision: 0.8241
Recall: 0.8158
F1 Score: 0.8112

---

epoch 5 / 6, step 300/1358, loss = 0.2523
epoch 5 / 6, step 600/1358, loss = 0.3239
epoch 5 / 6, step 900/1358, loss = 0.8572
epoch 5 / 6, step 1200/1358, loss = 0.2112
Epoch 5/6:
Accuracy: 0.8184
Precision: 0.8237
Recall: 0.8184
F1 Score: 0.8120

---

epoch 6 / 6, step 300/1358, loss = 0.2034
epoch 6 / 6, step 600/1358, loss = 0.4206
epoch 6 / 6, step 900/1358, loss = 0.2489
epoch 6 / 6, step 1200/1358, loss = 0.3127
Epoch 6/6:
Accuracy: 0.8208
Precision: 0.8314
Recall: 0.8208
F1 Score: 0.8168

Fold 5/5

epoch 1 / 6, step 300/1358, loss = 1.2142
epoch 1 / 6, step 600/1358, loss = 1.2098
epoch 1 / 6, step 900/1358, loss = 1.0318
epoch 1 / 6, step 1200/1358, loss = 0.7513
Epoch 1/6:
Accuracy: 0.7752
Precision: 0.8017
Recall: 0.7752
F1 Score: 0.7637

---

epoch 2 / 6, step 300/1358, loss = 0.6484
epoch 2 / 6, step 600/1358, loss = 0.7430
epoch 2 / 6, step 900/1358, loss = 0.4216
epoch 2 / 6, step 1200/1358, loss = 0.4822
Epoch 2/6:
Accuracy: 0.7993
Precision: 0.8070
Recall: 0.7993
F1 Score: 0.7940

---

epoch 3 / 6, step 300/1358, loss = 0.3817
epoch 3 / 6, step 600/1358, loss = 0.6204
epoch 3 / 6, step 900/1358, loss = 0.3339
epoch 3 / 6, step 1200/1358, loss = 0.5134
Epoch 3/6:
Accuracy: 0.7953
Precision: 0.8122
Recall: 0.7953
F1 Score: 0.7850

---

epoch 4 / 6, step 300/1358, loss = 0.4488
epoch 4 / 6, step 600/1358, loss = 0.7671
epoch 4 / 6, step 900/1358, loss = 0.7149
epoch 4 / 6, step 1200/1358, loss = 0.5418
Epoch 4/6:
Accuracy: 0.8012
Precision: 0.8082
Recall: 0.8012
F1 Score: 0.7914

---

epoch 5 / 6, step 300/1358, loss = 0.2215
epoch 5 / 6, step 600/1358, loss = 0.4730
epoch 5 / 6, step 900/1358, loss = 0.3005
epoch 5 / 6, step 1200/1358, loss = 0.3017
Epoch 5/6:
Accuracy: 0.8131
Precision: 0.8171
Recall: 0.8131
F1 Score: 0.8072

---

epoch 6 / 6, step 300/1358, loss = 0.5650
epoch 6 / 6, step 600/1358, loss = 0.2939
epoch 6 / 6, step 900/1358, loss = 0.5209
epoch 6 / 6, step 1200/1358, loss = 0.1188
Epoch 6/6:
Accuracy: 0.8109
Precision: 0.8175
Recall: 0.8109
F1 Score: 0.8075

Grafici di Validazione

Nelle seguenti figura sono rappresentati i grafici dell'andamento delle curve di Accuracy (Figura 5.5), Precision (Figura 5.6), Recall (Figura 5.7) e F1 (Figura 5.8). Nelle ascisse sono riportate le epoche di allenamento, mentre nelle ordinate si indicano i valori del parametro che si sta analizzando, con la stessa scala per ciascun grafico.

Si possono notare degli andamenti molto simili, fatta eccezione per la Precision, che si discosta leggermente, in particolare per quanto riguarda la prima epoca della fold 2.

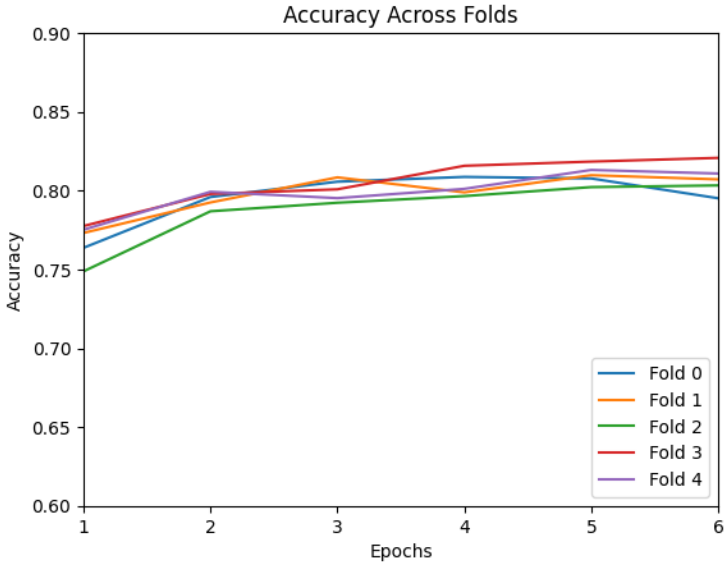


Figura 5.5: Accuracy Vocabulary Encoding

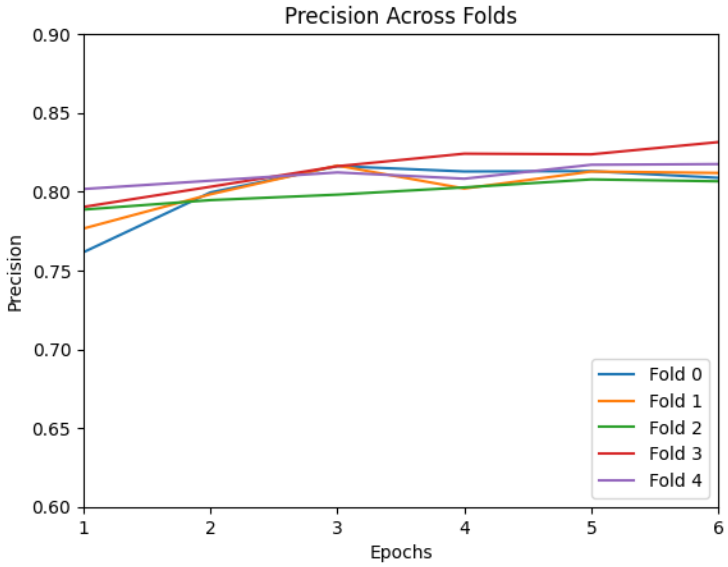


Figura 5.6: Precision Vocabulary Encoding

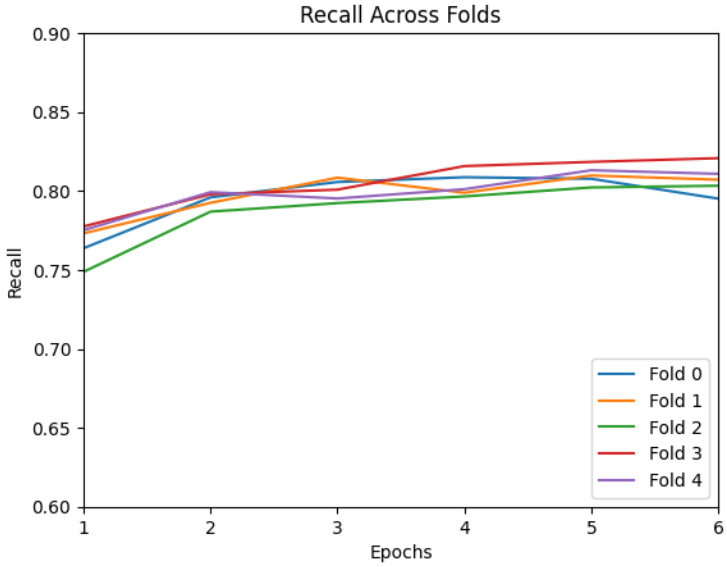


Figura 5.7: Recall Vocabulary Encoding

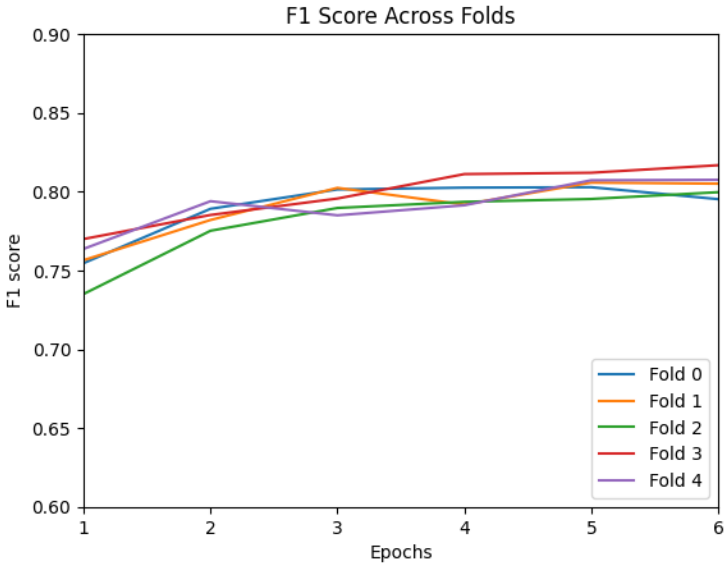


Figura 5.8: F1 Score Vocabulary Encoding

Risultati di Classificazione Multiclasse

Nella Tabella 5.2 vengono riportate le performance di Precision, Recall ed F1 score che il modello ha registrato su ciascuna famiglia di DGA, con indicato anche il numero totale di elementi presenti in ognuna di esse.

Domain Family	**Precision**	**Recall**	**F1-Score**	**Support**
tinba	0.63	0.83	0.72	1000
gozi_rfc4343	0.70	0.76	0.73	995
gozi_gpl	0.91	0.84	0.87	1000
kraken_v2	0.61	0.57	0.59	1000
gozi_luther	0.86	0.84	0.85	1000
sisron	0.99	1.00	1.00	1000
ranbyus_v1	0.81	0.84	0.82	1000
alureon	0.41	0.37	0.39	1000
nymaim	0.92	0.84	0.88	1000
murofet_v3	1.00	1.00	1.00	1000
matsnu	0.94	0.91	0.92	996
qakbot	0.68	0.50	0.58	1000
vawtrak_v1	0.99	1.00	1.00	1000
shiotob	0.93	0.80	0.86	1000
kraken_v1	0.78	0.77	0.77	1000
fobber_v2	0.41	0.59	0.49	1000
murofet_v2	0.84	0.93	0.88	1000
dyre	1.00	1.00	1.00	1000
dircrypt	0.40	0.35	0.37	1000
simda	0.95	0.99	0.97	1000
necurs	0.89	0.75	0.81	1000
corebot	1.00	0.99	1.00	1000
padcrypt	0.97	0.99	0.98	1000
banjori	1.00	1.00	1.00	1000
zeus-newgoz	1.00	1.00	1.00	1000
necurs	0.89	0.75	0.81	1000
corebot	1.00	0.99	1.00	1000
padcrypt	0.97	0.99	0.98	1000
banjori	1.00	1.00	1.00	1000
zeus-newgoz	1.00	1.00	1.00	1000
cryptolocker	0.51	0.49	0.50	1000
pykspa	0.32	0.25	0.28	1000
suppobox_2	0.97	0.99	0.98	998
ramdo	0.98	1.00	0.99	1000
proslikefan	0.67	0.62	0.64	1000
tempedreve	0.62	0.78	0.69	1000
chinad	0.99	0.97	0.98	1000
ranbyus_v2	0.82	0.79	0.80	1000
pushdo	0.93	0.96	0.95	1000
pykspa_noise	0.33	0.34	0.34	1000
rovnix	0.77	0.82	0.79	1000

locky	0.72	0.52	0.60	1000
gozi_nasa	0.72	0.75	0.73	991
bedep	0.60	0.64	0.62	1000
ramnit	0.36	0.40	0.38	1000
ccleaner	1.00	1.00	1.00	1000
vawtrak_v3	0.90	0.99	0.94	1000
legit	0.81	0.67	0.73	1000
symmi	0.99	1.00	1.00	1000
suppobox_1	0.96	0.96	0.96	962
vawtrak_v2	0.98	0.99	0.99	1000
qadars	0.90	0.92	0.91	1000
fobber_v1	0.87	0.97	0.92	1000
pizd	0.92	0.97	0.95	956
murofet_v1	0.98	0.98	0.98	1000
suppobox_3	0.98	0.99	0.99	1000
Macro Average	0.81	0.81	0.81	50898
Weighted Average	0.81	0.81	0.81	50898

Tabella 5.2: Classification Report per la Codifica di Vocabolario

La Figura 5.9 rappresenta la matrice di confusione totale per ognuna delle 51 famiglie, numerate nello stesso ordine con cui si presentano nella Tabella 5.2.

I valori che seguono la diagonale principale della matrice, rappresentano il numero di elementi per cui è stata predetta la famiglia corretta, maggiore è la quantità, più scuro è il colore della cella.

Il risultato ottimale è rappresentato dalle sole celle della diagonale principale a possedere un valore maggiore di zero, ovvero che per ciascuna classe si vada a pareggiare il numero dei samples predetti correttamente con il loro totale, indicato dalla colonna 'Support' nella Tabella 5.2. Ad esempio, il risultato ideale per la famiglia numero 51 (suppobox_3), corrisponde ad avere tutti zeri lungo la riga 51 e la colonna 51, ad eccezione della cella nella loro intersezione, che ha un valore pari a 1000.

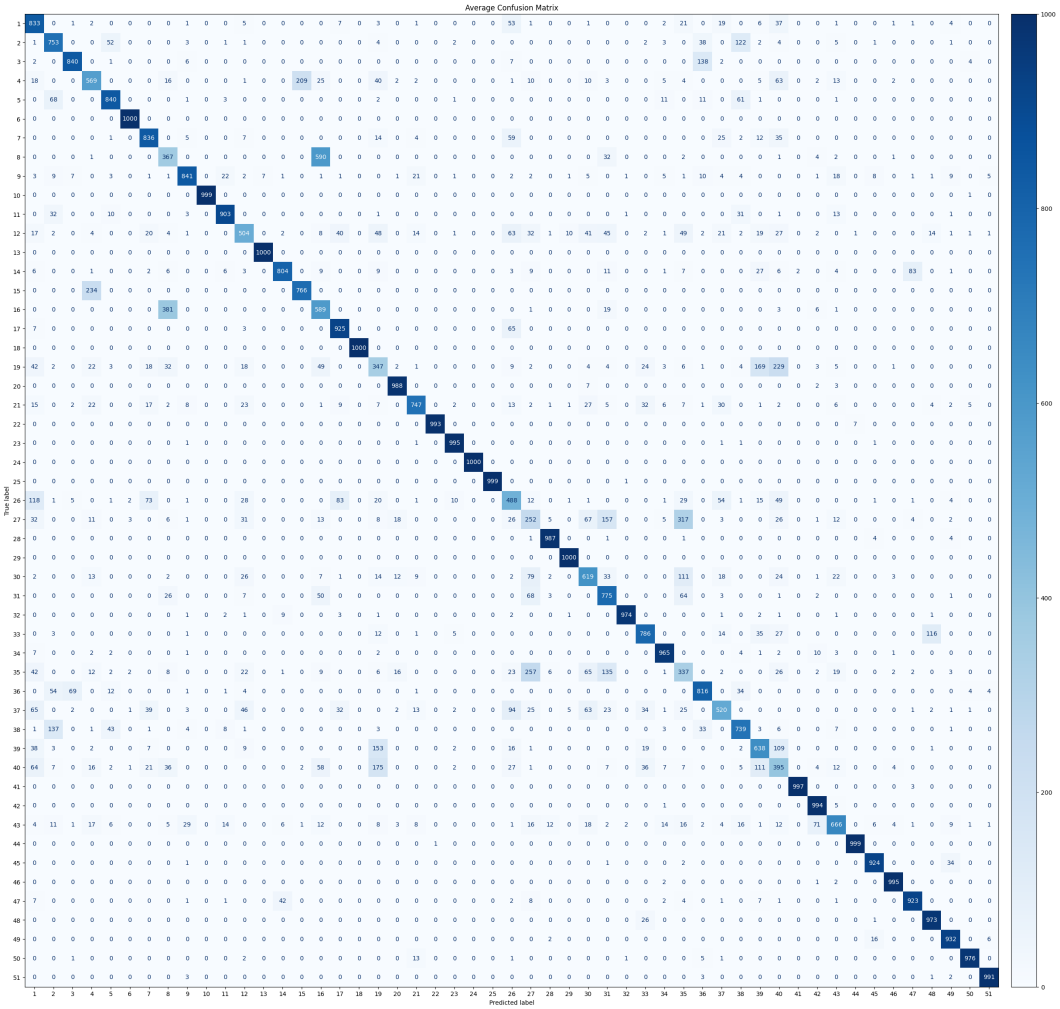


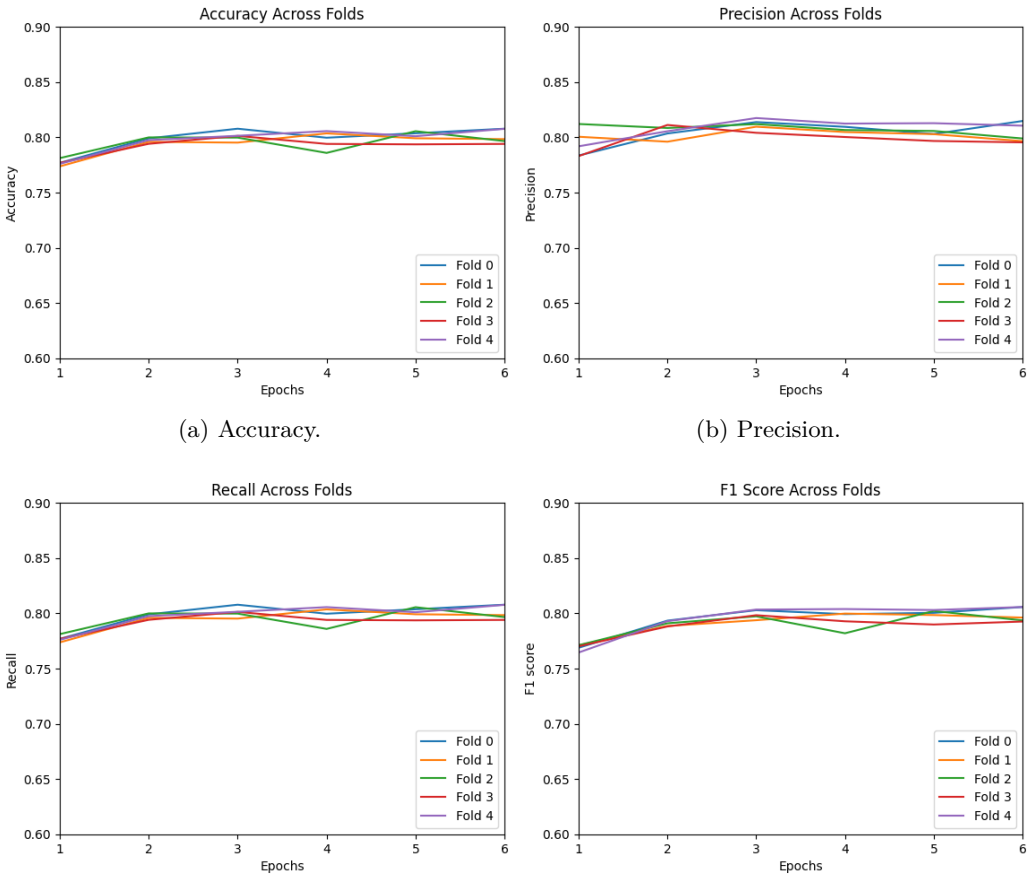
Figura 5.9: Confusion Matrix Vocabulary Encoding

5.2.2 Regolarizzazione con Batch Normalization e lr=0.0005

Per il modello con codifica ad indici di vocabolario, sono stati effettuati ulteriori esperimenti facendo il tuning del learning rate e utilizzando una differente regolarizzazione. Si sono voluti fare ulteriori test per valutare il comportamento del modello con differenti configurazioni degli iperparametri, oltre a quella proposta nel paper [2], utilizzata per gli esperimenti precedenti e descritta nel Capitolo 4.

In questo caso viene utilizzata la Batch Normalization sugli output delle CNN e viene dimezzato il learning rate rispetto a prima, portandolo a 0.0005.

Nella Figura 5.10 sono riportati i grafici degli andamenti delle curve di Accuracy (5.10a), Precision (5.10b), Recall (5.10c) e F1 (5.10d). Nelle ascisse sono presenti le epoche di allenamento, mentre nelle ordinate si indicano i valori del parametro che si sta analizzando, con la stessa scala per ciascun grafico.



(a) Accuracy.

(b) Precision.

(c) Recall.

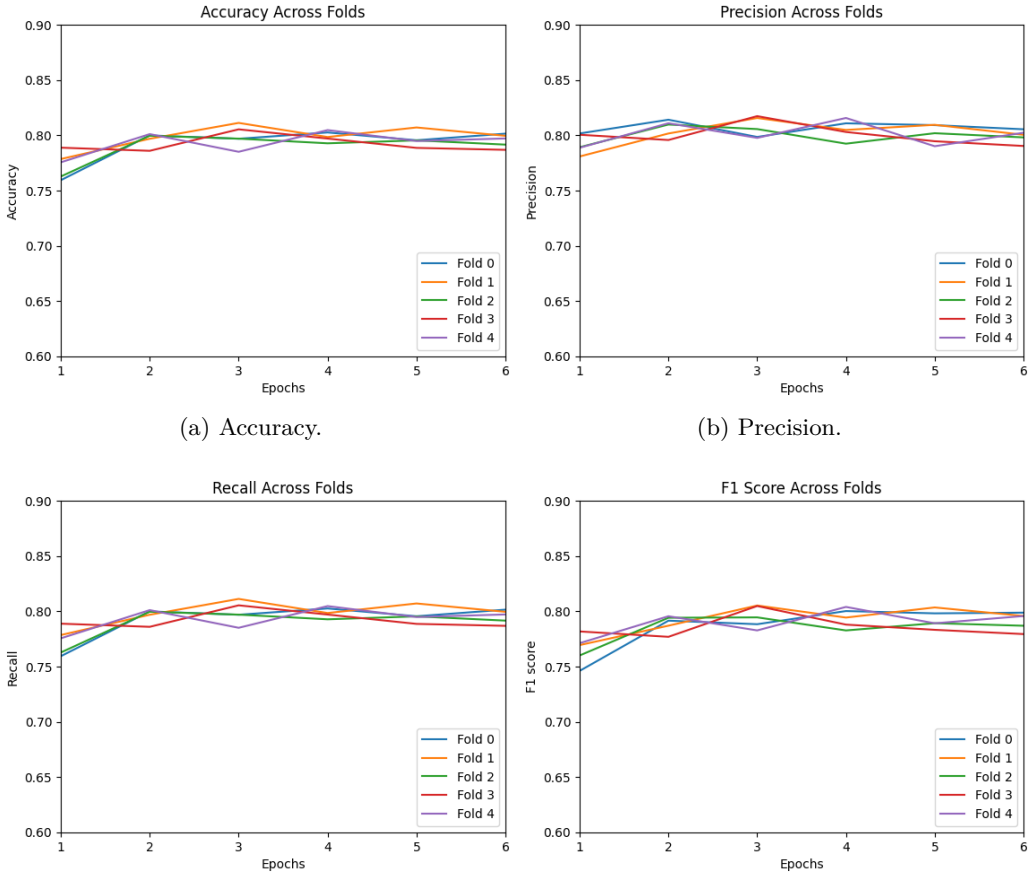
(d) F1-score.

Figura 5.10: Risultati per il modello con Batch Norm e lr=0.0005

5.2.3 Regolarizzazione con Batch Normalization e lr=0.001

In questo esperimento viene utilizzata la Batch Normalization degli output delle CNN e si mantiene invariato il learning rate (= 0.001) rispetto alla configurazione originale.

Nella Figura 5.11 sono riportati i grafici degli andamenti delle curve di Accuracy (5.11a), Precision (5.11b), Recall (5.11c) e F1 (5.11d). Nelle ascisse sono presenti le epoche di allenamento, mentre nelle ordinate si indicano i valori del parametro che si sta analizzando, con la stessa scala per ciascun grafico.



(a) Accuracy.

(b) Precision.

(c) Recall.

(d) F1-score.

Figura 5.11: Risultati per il modello con Batch Norm e lr=0.001

Capitolo 6

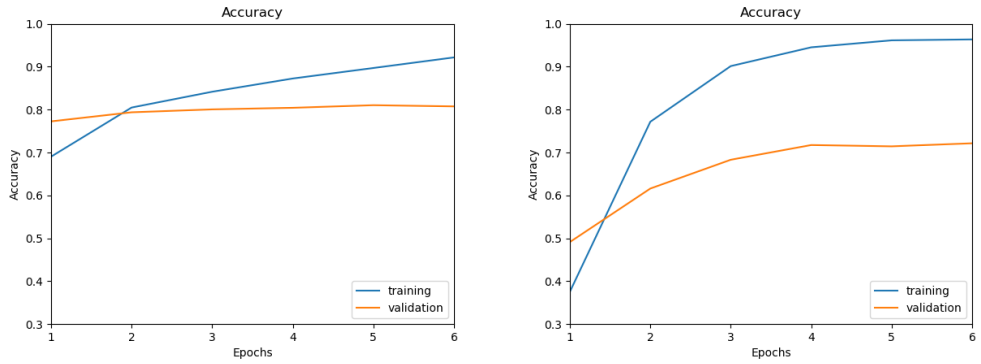
Discussione dei Risultati

Dai risultati di validazione di entrambi i modelli, si può notare che, i parametri scelti per valutare le performance, non raggiungono valori elevati.

Le curve di accuratezza, recall, precisione ed f1, infatti, seguono tutte un pattern simile di miglioramento, che però, raggiunge un plateau dopo poche epoche intorno all'80%.

Confrontando le prestazioni ottenute in fase di training con quelle ottenute in fase di validazione, si nota come questi vadano a divergere velocemente.

Se si prende come esempio l'accuratezza durante l'allenamento, essa presenta un andamento monotono crescente, diversamente da quella riferita alla validazione, che tende ad appiattirsi velocemente. Ciò significa che il modello sta imparando, ma non è in grado di generalizzare quando gli vengono forniti nuovi elementi su cui fare predizioni, diversi da quelli su cui si è addestrato.



(a) Codifica con Indici.

(b) Codifica One-Hot.

Figura 6.1: Confronto tra Training e Validation Accuracy Media

La relazione mostrata nella Figura 6.1, rappresenta l'indizio principale del fatto che il modello soffre di Overfitting, problema molto comune nei modelli di Deep Learning. Ovvero, si ha l'accuratezza di training che migliora ad ogni epoca, e quella di validazione che ad un certo punto permane invariata.

In generale, le cause possono essere differenti, ad esempio: dei dati poco puliti che contengono del 'rumore', ovvero contenuti irrilevanti di disturbo; la mancanza di

funzioni di regolarizzazione, utili ad evitare che il modello diventi troppo complesso; un'architettura troppo complessa; oppure una scarsità di elementi nel dataset, che è il limite riscontrato in questo caso, in particolare se rapportato alla complessità del modello.

È stato anche testato l'uso di regolarizzazioni per ridurre il numero di funzioni, come dei layer di Dropout (vedi Paragrafo 4.4) con differenti probabilità di scarto (cioè una parte degli input che vengono scartati casualmente), inseriti dopo le funzioni ReLU nelle FFN, e la Batch Normalization degli output delle CNN nelle due varianti della codifica ad indici di vocabolario (Paragrafi 5.2.3, 5.2.2).

In entrambi i casi i risultati non hanno sortito alcun effetto evidente, con le curve di accuratezza, precisione, recall ed f1 che risultano leggermente più appiattite tra le varie epoche per la tecnica di normalizzazione.

Per quanto concerne le prestazioni del modello sulle singole famiglie di domini, queste potrebbero essere state influenzate dalla tipologia di k-fold utilizzata, ovvero la funzione `KFold()` con parametro 'Shuffle' impostato a True.

Nonostante sia stata eseguita una suddivisione casuale del dataset in 5 fold, così da cercare di rendere quest'ultime potenzialmente più bilanciate, permane comunque la probabilità che ciò non accada per certe classi, alcune delle quali, ad esempio, potrebbero non trovarsi alcun elemento nella validazione per una certa iterazione.

La distribuzione delle famiglie nelle fold, essendo casuale, varia ad ogni singola esecuzione e di questo ne risentono le prestazioni generali su ognuna di esse, che sono differenti ad ogni nuova run. Per questo motivo non vengono tenute in considerazione le prestazioni specifiche nel confronto tra le varianti dei modelli.

6.0.1 Codifica One-Hot

Questa variante, proposta in [2], risulta di complessa esecuzione in quanto il modello raggiunge dimensioni molto elevate, per questo motivo ci si è limitati a solo 6 epoche di allenamento.

In input al modello, infatti, si hanno word parecchio estese, perché ciascun carattere e bigramma, valutato come se fosse una singola parola, è codificato con un vettore di lunghezza pari alla dimensione di vocabolario, rispettivamente 37 e 1344.

Quindi, partendo dal fatto che le stringhe di dominio hanno tutte una grandezza pari a 46 caratteri e 45 bigrammi, ci troviamo a gestire in fase di training, frasi di 46 parole, ciascuna di lunghezza pari a 37 elementi, e frasi di 45 parole, ciascuna di lunghezza pari a 1344 elementi.

Valutando l'andamento dei valori di perdita raggiunti durante l'allenamento, Figura 6.2a, questi tendono ad azzerarsi, anche se risultano spesso instabili con dei picchi improvvisi molto alti. Definiti anche come 'errore', essi quantificano la differenza tra l'output predetto e i valori dei target effettivi durante l'allenamento, rappresentando quindi un problema di minimizzazione.

Questa evoluzione può essere legata al fatto di avere un learning rate troppo elevato dopo una certa epoca, ma essendo l'esecuzione molto lenta e dispendiosa, non si sono riusciti ad eseguire sufficienti test.

Nell'esperimento del Paragrafo 5.2.2 della codifica con indici, nonostante il dimezzamento del learning rate, le prestazioni per le prime 6 epoche sono rimaste invariate, ma potrebbero risultare differenti per un numero di epoche maggiore.

A tal proposito, in futuro si potranno sfruttare delle tecniche utili proprio all'ottimizzazione degli iperparametri come la **nested cross validation**, essenziale nel fornire una stima realistica delle capacità di generalizzazione di un modello.

Questo metodo utilizza due loop annidati, uno esterno per la valutazione del modello ed uno interno per il tuning degli iperparametri. In questa maniera si assicura di stimare le performance per ciascuna variante in maniera accurata, selezionando la migliore tra queste.

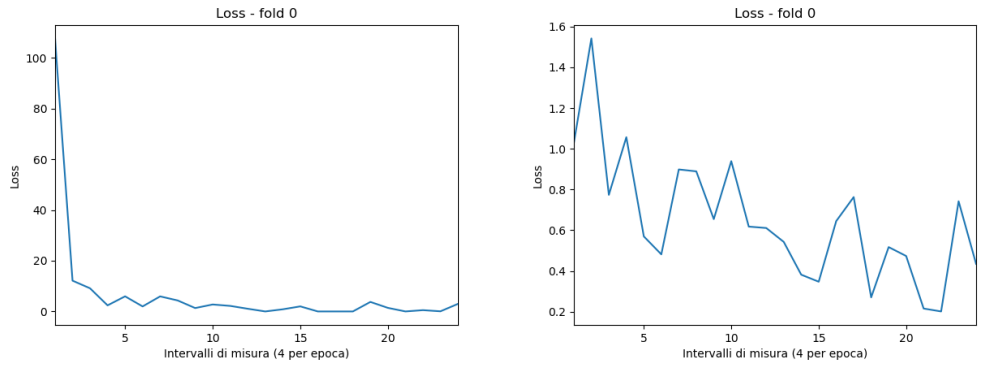
6.0.2 Codifica con Indici di Vocabolario

Questa variante del modello è stata proposta, in quanto risulta più leggera nel consumo di memoria di diversi ordini di grandezza rispetto alla precedente e quindi di maggiore semplicità, utile per testare più esecuzioni. Ciò è dovuto dal fatto che si effettuano calcoli su word codificate con un singolo intero.

Si può notare che, probabilmente per le dimensioni più ridotte, le prestazioni raggiunte, a parità di iperparametri di training, sono migliori rispetto al One-Hot encoding, anche se è più evidente la loro saturazione.

Anche nei grafici, per tutti gli indici ed in ciascuna fold, le curve partono da valori più alti già dalla prima epoca di allenamento.

Se si vanno a guardare i dati forniti durante l'esecuzione (Figura 6.2b), si nota che l'andamento dell'errore risulta più stabile nella sua discesa, con variazioni più contenute rispetto al modello precedente, anche se in quest'ultimo raggiunge misure inferiori.



(a) Codifica One-Hot.

(b) Codifica con Indici.

Figura 6.2: Andamento dell'errore nella fold 0 (scale differenti sull'asse delle ordinate)

Capitolo 7

Conclusioni

Il modello proposto rientra nella categoria del Deep Learning supervisionato e di conseguenza, come detto nel Capitolo 6 , necessita di dataset etichettati di grandi dimensioni per essere in grado di imparare pattern più complessi, essendo una rete che possiede un gran numero di layer.

Non disponendo di risorse computazionali sufficienti, i test sono stati eseguiti su una versione ridotta del dataset. I risultati quindi danno un'idea delle prestazioni del modello, ma sono limitati e non mostrano l'effettivo potenziale.

Inoltre, come citato nel Capitolo 6, la conseguenza di ciò è rappresentata da un problema di Overfitting, ovvero di un modello che non possiede dati a sufficienza per generalizzare i pattern che sta imparando.

Perciò in questo caso, le basse performance sono molto probabilmente dovute alla ridotta dimensione del dataset scelto, che però possiede un quantitativo uniforme di domini per ciascuna famiglia.

Per fare un esempio a scopo di confronto, nell'articolo [2] il dataset è composto da più di 2 milioni di samples, di cui la metà sono domini benevoli.

Negli ulteriori test, eseguiti sul modello con codifica con indici di vocabolario, è stato abbassato il learning rate ed è stata modificata la modalità di regolarizzazione, ma gli andamenti delle curve sono rimasti invariati.

Quindi anche con parametri differenti, si sono ottenuti gli stessi indizi che fanno ipotizzare un overfitting.

Un problema che invece si può presentare con dataset di dimensioni nettamente maggiori, riguarda lo sbilanciamento degli elementi a disposizione, che influisce sulle prestazioni generali.

In letteratura vengono proposti determinati modelli, come quello fornito nella pubblicazione [13], specificatamente indicati per mitigare questa difficoltà.

Per quanto riguarda i risultati ottenuti sulle singole classi, per azzerare totalmente il rischio dello sbilanciamento dei loro elementi tra le fold, che comunque si cerca di mitigare con il mescolamento dei dati, vedi Capitolo 6, si può andare a sostituire la funzione di convalida incrociata con una `Stratified KFold`.

A differenza della KFold standard, la versione stratificata prevede che i campioni vengano prelevati in modo da preservare la stessa proporzione che si trova nella popolazione originale. Se ad esempio in un dataset il 50% dei nomi di dominio sono

benevoli, il 30% sono generati dal DGA 1 e il 20% dal DGA 2, in ogni fold dobbiamo trovare il 50% di benevoli, il 30% di DGA 1 e il 20% di DGA 2.

7.0.1 Sviluppi Futuri

Ulteriori test su questo modello, potranno essere fatti sfruttando dataset più grandi e di più recente pubblicazione, considerando che gli elenchi dei domini noti vengono aggiornati continuamente, per cui possono anche presentarsi nuovi pattern su cui addestrare la rete.

Sviluppi futuri possono riguardare il fine-tuning di questa architettura con iperparametri ottimali, ottenuti adottando una nested cross validation (vedi Paragrafo 6.0.1). Inoltre si potrebbe procedere con l'aggiunta o la sostituzione di alcuni moduli creando così ibridi di tecnologie diverse. Così come fatto in questo stesso modello rispetto alla struttura Transformer, modificata con l'aggiunta di CNN, adattate per ricevere in input stringhe di testo, trattate come vettori monodimensionali.

Bibliografia

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Ling Ding, Peng Du, Haiwei Hou, Jian Zhang, Di Jin, and Shifei Ding. Botnet dga domain name classification using transformer network with hybrid embedding. *Big Data Research*, 33:100395, 2023.

[3] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From {Throw-Away} traffic to bots: Detecting the rise of {DGA-Based} malware. In *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[4] Xinjie Sun and Zhifang Liu. Domain generation algorithms detection with feature extraction and domain center construction. *Plos one*, 18(1):e0279866, 2023.

[5] Mattia Zago, Manuel Gil Pérez, and Gregorio Martínez Pérez. Umudga - university of murcia domain generation algorithm dataset, v1. `https://data.mendeley.com/datasets/y8ph45msv8/1`, 2020.

[6] Mattia Zago, Manuel Gil Pérez, and Gregorio Martínez Pérez. Umudga: A dataset for profiling algorithmically generated domain names in botnet detection. *Data in Brief*, 30:105400, 2020.

[7] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences

from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[10] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[11] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[12] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In *Computer Vision–ACCV 2016: 13th Asian Conference on Computer Vision, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part II 13*, pages 189–204. Springer, 2017.

[13] Duc Tran, Hieu Mac, Van Tong, Hai Anh Tran, and Linh Giang Nguyen. A lstm based framework for handling multiclass imbalance in dga botnet detection. *Neurocomputing*, 275:2401–2413, 2018.