

# Università Politecnica delle Marche

---

Dipartimento di Ingegneria dell'Informazione

Facoltà di Ingegneria Informatica e dell'Automazione

*Corso di Advanced Cyber Security*



*Analisi e Classificazione di Domini Malevoli mediante Deep Learning*

**Professore:**

Prof. Spalazzi Luca

**Studenti:**

Bernovschi Denis  
Incicco Emanuele  
Pinciaroli Andrea  
Fratini Lorenzo  
Miscia Federico



# Indice

<b>1</b>	<b>Introduzione</b>	<b>9</b>
1.1	Tecnologie . . . . .	11
<b>2</b>	<b>Prima Parte</b>	<b>13</b>
2.1	Dataset GARR . . . . .	13
2.1.1	ETL . . . . .	14
2.1.2	Divisione in 1-grams, 2-grams, 3-grams . . . . .	15
2.2	FastText . . . . .	16
2.2.1	Introduzione . . . . .	16
2.2.2	Skipgram vs CBOW . . . . .	16
2.2.3	Caso di Studio . . . . .	16
<b>3</b>	<b>Seconda Parte</b>	<b>21</b>
3.1	Dataset UMUDGA . . . . .	21
3.2	Architettura Multi-Input . . . . .	21
3.3	Architettura Multi-Input con Embedding 1-gram Casuale . . . . .	22
3.4	Iperparametri . . . . .	22
3.5	Risultati . . . . .	23
3.6	Tempi Computazionali . . . . .	23
3.6.1	Tempi Addestramento FastText . . . . .	23
3.6.2	Tempi Addestramento Multi-Input Architecture . . . . .	23
3.6.3	Tempi Addestramento Multi-Input con Random Embedding . . . . .	23
3.7	Risultati Classificazione con Multi-Input . . . . .	24
3.7.1	Risultati Classificazione - Percentuale 0.017 . . . . .	24
3.7.2	Risultati Classificazione - Percentuale 0.034 . . . . .	25
3.7.3	Risultati Classificazione - Percentuale 0.068 . . . . .	26
3.7.4	Risultati Classificazione - Percentuale 0.135 . . . . .	27
3.8	Risultati Classificazione con Multi-Input e Random Embedding . . . . .	28
3.8.1	Risultati Classificazione - Percentuale 0.017 . . . . .	28
3.8.2	Risultati Classificazione - Percentuale 0.034 . . . . .	29
3.8.3	Risultati Classificazione - Percentuale 0.068 . . . . .	30
3.8.4	Risultati Classificazione - Percentuale 0.135 . . . . .	31
3.9	Confronto Risultati Architetture . . . . .	32
3.9.1	Risultati Classificazione . . . . .	32
<b>4</b>	<b>Conclusioni e Sviluppi Futuri</b>	<b>35</b>



# Elenco delle figure

1.1	Funzionamento di un DGA . . . . .	10
1.2	Linguaggio di Programmazione Python . . . . .	11
1.3	Google Colab . . . . .	11
2.1	Skipgram vs CBOW . . . . .	17
3.1	Architettura Multi-Input . . . . .	22
3.2	Architettura Multi-Input con Embedding Random . . . . .	22
3.3	CM 0.017 Multi Input . . . . .	25
3.4	CM 0.034 Multi Input . . . . .	26
3.5	CM 0.068 Multi Input . . . . .	27
3.6	CM 0.135 Multi Input . . . . .	28
3.7	CM 0.017 Multi Input con Random Embeddings . . . . .	29
3.8	CM 0.034 Multi Input con Random Embeddings . . . . .	30
3.9	CM 0.068 Multi Input con Random Embeddings . . . . .	31
3.10	CM 0.135 Multi Input con Random Embeddings . . . . .	32



# Listings

2.1	Script per ETL . . . . .	14
2.2	Script per la creazione dei n-gramss . . . . .	15
2.3	Implementazione . . . . .	16
2.4	Implementazione FastText . . . . .	17
2.5	Training del Modello . . . . .	18
2.6	Conversione da <i>Bin</i> a <i>Vec</i> . . . . .	18



# Capitolo 1

## Introduzione

Oggi giorno, gli attacchi informatici sono sempre più frequenti e con conseguenze sempre più importanti.

Volendo fare un'estrema sintesi, gli attacchi quali phishing, trojan e worm non fanno altro che tentare di recuperare informazioni sensibili, il cui utilizzo può essere svariato: da una semplice rivendita nel dark web (i dati sanitari sono i più richiesti e pagati) fino alla predisposizione di ulteriori azioni malevoli, come ad esempio attacchi DoS (Denial of Service). Attacchi del genere, tipicamente, utilizzano come mezzo di trasmissione la rete. Ne consegue che sempre più dispositivi utilizzati sia in ambito lavorativo che per scopi personali sono esposti al rischio di compromissione.

In questo elaborato l'attenzione è posta, nello specifico, sui malware che permettono di creare le *Botnet*, cioè reti di macchine infette (bot) che poi devono essere coordinate dall'hacker tramite un server di comando e controllo. Per contrastare un'infezione di questo tipo, spesso, si tenta di individuare ed offuscare il server di comando e controllo; ciò, però, non è così facile perché in risposta vengono adottate delle tecniche evasive che ne rendono difficile l'individuazione. Una di queste tecniche consiste nel cambiare il nome di dominio dopo un certo intervallo di tempo, attraverso un DGA (Domain Name Generation Algorithm).

Il cambio continuo di *domain name*, ad ogni modo, causa problemi anche al malware stesso che ha infettato una macchina: quando esso si attiva e cerca di contattare il server di comando e controllo, infatti, non sa esattamente a quale nome di dominio debba rivolgersi. A tal proposito, il malware, utilizzando anch'esso il medesimo DGA, genera tutta una serie di nomi di dominio e di volta in volta interroga il DNS<sup>1</sup> per vedere se c'è qualcuno registrato sotto il nome generato. Quando viene trovato un riscontro positivo, il DNS traduce il nome di dominio in un indirizzo IP.

---

<sup>1</sup>DNS ≡ Domain Name Server

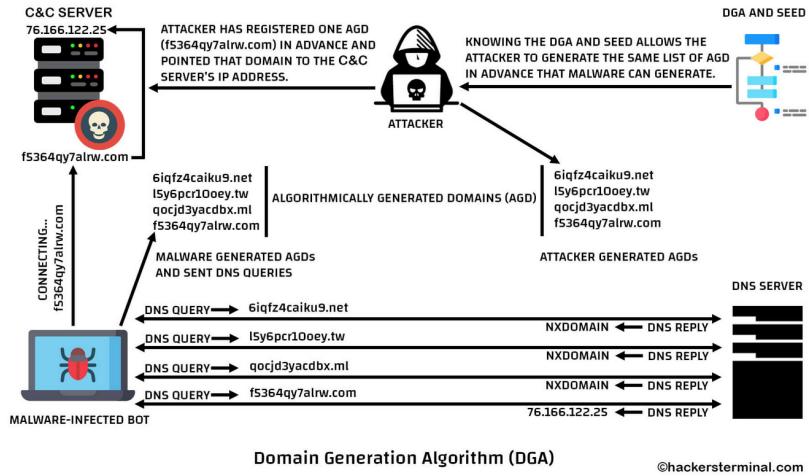


Figura 1.1: Funzionamento di un DGA

Per descrivere come funziona il sistema DNS di Internet, basti pensare ad una rubrica telefonica che gestisce la mappatura tra nomi e numeri. I server DNS traducono le richieste di nomi di dominio in indirizzi IP, controllando a quale server si conterà un utente finale nel momento in cui egli digiterà un certo nome di dominio nel proprio browser Web. A questo punto, dunque, appare chiaro che se il dominio digitato è in realtà malevolo si è vittima di un attacco. Andare a scovare questi domini, tentando di rintracciarne la forma o le caratteristiche peculiari, dunque, diventa fondamentale nel contrastare questa tipologia di attacco.

Il quadro è ulteriormente complicato dal fatto che un DGA può generare diverse centinaia di domini, rendendo di fatto l'analisi un'operazione onerosa e assai complessa per l'uomo, dato che bisogna verificare la legittimità di ogni nome di dominio sulla base delle richieste che lo interessano.

I DGA sviluppati odiernamente, inoltre, sfruttano un meccanismo che combina lettere o parole in modo pseudo-casuale, fino ad arrivare alla creazione di un dizionario, in modo da complicare ulteriormente il lavoro di analisi: sulla base delle parole contenute all'interno del dizionario, infatti, vengono creati nomi di dominio dall'aspetto molto simile a quelli reali e talvolta facilmente scambiabili per legittimi. Da ciò si intuisce come l'evoluzione dei sistemi di detection da una parte e il progresso continuo di questi algoritmi, dall'altra, siano in una rincorsa reciproca e continua.

L'elaborato qui presente, dunque, ha l'obiettivo di testare una tipologia di architettura basata su Deep Learning che permetta di classificare i domini malevoli e non.

## 1.1 Tecnologie

Per lo sviluppo del progetto e l'analisi degli approcci di apprendimento automatico utili alla questione, è stato utilizzato il tool Google Colab, il quale mette a disposizione un ambiente di sviluppo condiviso per il linguaggio Python con all'interno disponibili già diverse librerie di interesse. Altri aspetti a favore di Google Colab sono legati alle computational resources e alla possibilità di storage sharing, semplificando il lavoro del team.



Figura 1.2: Linguaggio di Programmazione Python



Figura 1.3: Google Colab



# Capitolo 2

## Prima Parte

### 2.1 Dataset GARR

Per quanto concerne i datasets utilizzati, nella prima parte del lavoro si è fatto uso dei file di log forniti dalla rete GARR. Dopo un'attenta analisi e una fase di ETL<sup>1</sup>, si è attuata anche una pulizia dei duplicati e dei nomi di dominio non conformi alle specifiche assegnate. Il dataset ottenuto in questa sede, ad ogni modo, non risulta etichettato, ossia non distingue tra nomi di dominio benevoli o malevoli.

**Selezione Log** Per la selezione e la creazione del dataset, sono stati selezionati i logs relativi a sei giorni distinti di ciascuno degli ultimi 12 mesi disponibili. Successivamente, è stato considerato solo il 10% dei records presenti all'interno di ogni file di log scaricato.

#### Precisazioni

1. Per quanto riguarda le ore selezionate, esse rappresentano l'intero intervallo giornaliero, ovvero dalle 00 : 00 alle 23 : 00;
2. Per la scelta dei giorni, si è proceduto in modo ciclico, selezionandoli tutti dal lunedì alla domenica, estremi inclusi;
3. Relativamente alle mensilità, ripercorrendo a ritroso, l'intervallo considerato va da Novembre 2021 fino a Febbraio 2021, dopodiché da Settembre 2020 ad Agosto 2020. La ragione di tale scelta è un'assenza di dati nell'arco temporale che va da Febbraio 2021 a Settembre 2020 all'interno del repository del GARR.
4. Di seguito un resoconto del numero di logs selezionati per ciascun giorno della settimana:

Giorno	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì	Sabato	Domenica
Nº Logs	9	10	10	10	12	12	9

Tabella 2.1: N Logs considerati per ciascuna giorno della settimana

---

<sup>1</sup>ETL ≡ Extraction, Transform e Loading

### 2.1.1 ETL

Nella seguente sezione è possibile visionare lo script utilizzato per la fase di ETL. Come già anticipato, oltre alla fase di selezione dei dati di log da considerare, è stata realizzata anche la fase di pulizia di questi ultimi. Come riportato nel listato seguente, sono state effettuate diverse tipologie di operazioni di pulizia e di verifica della correttezza dei dati a disposizione. In estrema sintesi, è possibile individuare:

- Una clausola per la lunghezza massima del nome di dominio  $LEN\_THRESHOLD = 100$ ;
- Una clausola per la lunghezza minima del nome di dominio:  $len(DN\_name) < 3$ . Sono stati esclusi, cioè, i nomi di dominio più corti di 3 caratteri, così facendo elidiamo i possibili problemi nel implementazione della divisione in n-grams;
- Implementazione di una REGEX, in modo da considerare solo i nomi di dominio che non contengono all'interno indirizzi IP;
- Infine, la selezione del 10% dei records, come da specifiche richieste.

```

1 import os
2 import csv
3 import re
4 from random import sample
5 import math
6
7 # PARAMS
8 PATH = os.path.join(path_progettoADV, '_log_files')
9 OUT_FILENAME = 'out_temp_file.csv'
10 LEN_THRESHOLD = 100
11 # get the list of files
12 files_list = os.listdir(PATH)
13 # open output file
14 out_file = open(os.path.join(path_progettoADV, OUT_FILENAME), 'w', newline=',')
15 writer = csv.writer(out_file)
16 # domain names list
17 DNS_list = []
18 DNS_list_samples = []
19
20 # for each file
21 for filename in files_list:
22     # skip not '.log' files
23     if '.log' not in filename:
24         continue
25
26     # open file
27     file = open(os.path.join(PATH, filename), 'r')
28
29     # for each line
30     for i, line in enumerate(file):
31         # get domain name
32         DN_name = line.split(';')[5]
33
34         # clear final dot in the domain name
35         if DN_name[-1] == '.':
36             DN_name = DN_name[0:-1]
37         # skip domain names with length greater or lower than threshold
38         if len(DN_name) > LEN_THRESHOLD or len(DN_name) < 3:
39             continue
40         # skip domain names with addresses

```

```

41     if re.match(r'.*\d*\.\d*\.\d*', DN_name):
42         continue
43     else:
44         DNs_list.append(DN_name)
45
46     # random choice 10% of DNs
47     DNs_list_sample = sample(DNs_list, math.floor(len(DNs_list)/100))
48     DNs_list_samples += DNs_list_sample
49
50 # get unique domain names
51 set_of_DNs = set(DNs_list_samples)
52 print(f'Estratti {len(set_of_DNs)} nomi di dominio.')
53 # write domain names and domain names without dots
54 writer.writerows([value, f'{value}'.replace('.', '')] for value in set_of_DNs)

```

Listing 2.1: Script per ETL

A seguito delle operazioni di ETL, sono risultati 4548258 nomi di dominio a partire da una cifra iniziale di circa 28 milioni. I record che non hanno rispettato i vincoli sulla lunghezza (maggiore di 100 o minore di 3) sono stati 816473. I record scartati tramite l'utilizzo di espressioni regolari sono stati 9359778. La quota rimanente è stata filtrata grazie alla rimozione dei duplicati.

### 2.1.2 Divisione in 1-grams, 2-grams, 3-grams

Una volta terminata la fase di ETL, si è passati ad implementare uno script per la creazione di 1-grams, 2-grams e 3-grams, cioè la suddivisione di ciascun nome di dominio in sotto-unità costituite rispettivamente da una, due e tre lettere. Nel listato seguente è visibile tale implementazione.

La peculiarità all'interno di questo script è l'utilizzo della libreria **nltk**: essa infatti mette a disposizione la funzione *ngrams*, la quale permette di estrapolare gli n-grams in maniera agevole, specificando direttamente il parametro *n* desiderato.

```

1 import csv
2 from nltk import ngrams
3
4
5 # PARAMS
6 IN_FILENAME = 'out_temp_file.csv'
7 OUT_FILENAME = 'out_final_file.csv'
8
9 # open input and output files
10 with open(os.path.join(path_progettoADV, IN_FILENAME), 'r') as in_file, open(os.
    path.join(path_progettoADV, OUT_FILENAME), 'w', newline='') as out_file:
11     reader = csv.reader(in_file)
12     writer = csv.writer(out_file)
13     for row in reader:
14         # get domain name without dots
15         domain_name = row[1]
16
17         # extract 1-grams from the domain name
18         n = 1
19         one_grams = ngrams(domain_name, n)
20         one_gram = [','.join(el) for el in one_grams]
21
22         # extract 2-grams from the domain name
23         n = 2
24         two_grams = ngrams(domain_name, n)
25         two_gram = [','.join(el) for el in two_grams]
26

```

```

27     # extract 3-grams from the domain name
28     n = 3
29     three_grams = ngrams(domain_name, n)
30     three_gram = [','.join(el) for el in three_grams]
31
32     # write extracted info to the output file
33     writer.writerow([row[0], row[1], ', '.join(one_gram), ', '.join(two_gram),
34                      ', '.join(three_gram)])

```

Listing 2.2: Script per la creazione dei n-gramss

## 2.2 FastText

### 2.2.1 Introduzione

FastText è una libreria open-source, sviluppata dal laboratorio AI Research di Facebook (FAIR), che consente agli utenti di apprendere a partire da rappresentazioni di testo. FastText, più in dettaglio, permette l'apprendimento delle incorporazioni di parole e la classificazione dei testi. Il modello offerto da FastText fa affidamento su un algoritmo di apprendimento non supervisionato o supervisionato al fine di ottenere rappresentazioni vettoriali delle parole. Nel moderno Machine Learning e nel Natural Language processing, infatti, ha trovato riscontro positivo l'idea di rappresentare le parole tramite vettori. Tali vettori catturano informazioni nascoste di un linguaggio, come analogie sintattiche o analogie semantiche. FastText *viene anche utilizzato per migliorare le prestazioni dei classificatori di testo*[1].

### 2.2.2 Skipgram vs CBOW

FastText fornisce due modelli per il calcolo delle rappresentazioni di parole: *skipgram* e *cbow* ("continuous-bag-of-words").

Il modello *skipgram* impara a prevedere una parola target grazie ad una parola vicina. D'altra parte, il modello *cbow* predice la parola d'interesse in base al suo contesto. Il contesto è rappresentato tramite un insieme di parole contenute in una finestra di dimensione fissa intorno alla parola target. È possibile mostrare questa differenza con un esempio: data la frase "*I am selling these fine leather jackets*" e la parola target "*fine*", un modello *skipgram* cerca di predirla usando una parola casuale vicina, come "*selling*" o "*these*". Il modello *cbow* invece prende tutte le parole in una finestra circostante, come {*selling*, *these*, *leather*, *jackets*}, e utilizza la somma dei loro vettori per prevedere la parola target. La figura seguente riassume questa differenza[2].

### Implementazione

Una possibile implementazione di FastText potrebbe essere la seguente. Come si può notare, oltre a passare i dati su cui addestrarsi, è possibile settare il parametro relativo a CBOW o SkipGram.

```

1 import fasttext
2 model = fasttext.train_unsupervised("data/fil9", "cbow")

```

Listing 2.3: Implementazione

### 2.2.3 Caso di Studio

Nel lavoro svolto, l'utilizzo della FastText ha permesso di attuare un pre-addestramento da sfruttare successivamente negli strati di embeddings dell'architettura che verrà descritta nella

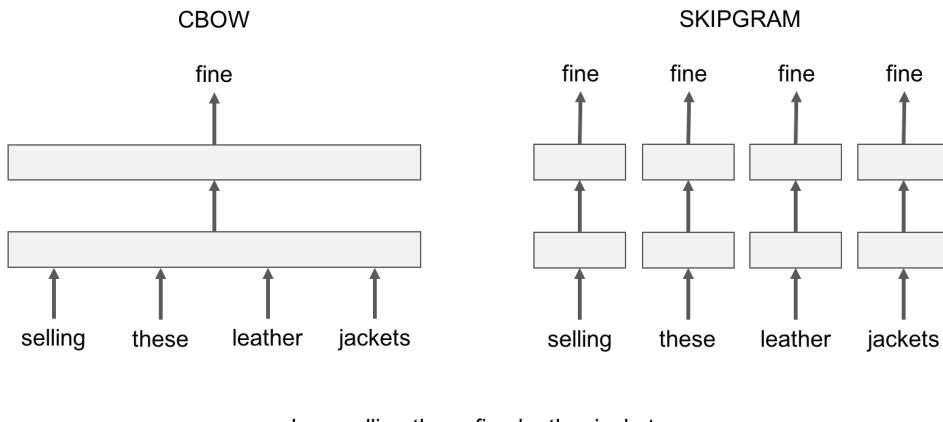


Figura 2.1: Skipgram vs CBOW

seconda parte della relazione. L’addestramento della FastText, inoltre, è stato eseguito per ogni tipologia di dato a disposizione, ovvero per *1-grams*, *2-grams* e *3-grams*. Nello specifico, si è realizzato un addestramento non supervisionato, utilizzando la modalità *skipgram* messa a disposizione da FastText. Il risultato sono dei vettori pesati per ciascun elemento di *1-grams*, *2-grams* e *3-grams*, da caricare poi come pesi nella seconda architettura addestrata nella parte successiva del progetto.

## Implementazione

Nella seguente sezione si discute l’implementazione della FastText per il progetto in esame. Per una maggiore comprensione e chiarezza, di seguito è riportata la road-map seguita in questa parte del lavoro.

1. Implementazione dell’algoritmo per effettuare il training, con la lettura dei file *n-grams* e la dichiarazione del modello utilizzato;
2. Training del modello;
3. Trasformazione del file con i pesi della FastText da un formato *.bin* ad uno *.vec*.

Nella parte sottostante sono descritti nel dettaglio tutti gli steps. Lo script relativo al primo passaggio è riportato nel seguente listato.

```

1 #installazione pacchetto
2 !pip install fasttext -q
3
4 import fasttext
5 import pandas as pd
6 import os
7
8 #lettura file
9 df = pd.read_csv('out_final_file.csv', header=None)
10
11 #split per n-gram
12 _label_ = [',', '1-gram', '2-gram', '3-gram']
13 j = 1;
14 for i in range (2, 5):
15     columns_ = df[i]
16     print(columns_.values)

```

```

17 print('len di {} : {}'.format(_label_[j], len(columns_.values)))
18 with open('out_final_file_{}.txt'.format(j), 'w') as f:
19     for line in columns_.values:
20         f.write(str(line).strip('\n'))
21         f.write('\n')
22     j = j + 1

```

Listing 2.4: Implementazione FastText

Come si nota nel listato precedente, il file da leggere è **out\_final\_file.csv** (output dei tasks precedenti, relativi alla fase di ETL); di tale csv, vengono considerate solo le colonne dalla seconda alla quarta, ovvero quelle contenenti gli *n-grams* (rispettivamente 1-gram nella colonna 2, 2-gram nella colonna 3, 3-gram nella colonna 4).

Come definito nella road map, successivamente si passa all’allenamento dei modelli (tre modelli, rispettivamente per 1-grams, 2-grams e 3-grams). Per fare ciò, è sufficiente una semplice invocazione del metodo *train\_unsupervised* dell’oggetto *fasttext*, passando come parametro in primo luogo il file creato precedentemente per la tipologia di n-grams in questione. I restanti parametri sono relativi alla scelta tra il modello *skipgram* o *cbow* ed infine la dimensione di *embedding* che, da specifiche progettuali, è impostata a 128. Di seguito è riportato il listato del training del modello con input 1-gram.

```

1 model_1 = fasttext.train_unsupervised(
2     'out_final_file_1.txt',
3     "skipgram",
4     dim=128)
5 model_1.words
6 model_1.save_model("out_final_file_1.bin")

```

Listing 2.5: Training del Modello

Terminata la fase di training dei tre modelli, i file ottenuti in output, contenenti i pesi generati come risultato dell’addestramento, sono stati convertiti dal formato *.bin* a *.vec*. Per fare ciò, è stato utilizzato lo script fornito da FastText<sup>2</sup>. Per una maggior fruizione dell’elaborato, tale script è riportato nel listato successivo.

```

1 #!/usr/bin/env python
2
3 # Copyright (c) 2017-present, Facebook, Inc.
4 # All rights reserved.
5 # This source code is licensed under the MIT license found in the
6 # LICENSE file in the root directory of this source tree.
7
8 from __future__ import absolute_import
9 from __future__ import division
10 from __future__ import print_function
11 from __future__ import unicode_literals
12 from __future__ import division, absolute_import, print_function
13
14 from fasttext import load_model
15 import argparse
16 import errno
17
18 if __name__ == "__main__":
19     parser = argparse.ArgumentParser(
20         description=("Print fasttext .vec file to stdout from .bin file"))
21     )
22     parser.add_argument(
23         "model",

```

<sup>2</sup>URL [https://github.com/facebookresearch/fastText/blob/main/python/doc/examples/bin\\_to\\_vec.py](https://github.com/facebookresearch/fastText/blob/main/python/doc/examples/bin_to_vec.py)

```

24         help="Model to use",
25     )
26 args = parser.parse_args()
27
28 f = load_model(args.model)
29 words = f.get_words()
30 print(str(len(words)) + " " + str(f.get_dimension()))
31 for w in words:
32     v = f.get_word_vector(w)
33     vstr = ""
34     for vi in v:
35         vstr += " " + str(vi)
36     try:
37         print(w + vstr)
38     except IOError as e:
39         if e.errno == errno.EPIPE:
40             pass

```

Listing 2.6: Conversione da *Bin* a *Vec*

Lo script appena presentato permette di realizzare la conversione di formato desiderata per mezzo del comando:

```
1 python bin_to_vec.py nome_modello_fastText.bin > nome_file_vec.vec
```

Si è deciso di convertire il file ".bin" in un file ".vec" in quanto nel repository che ospita l'architettura utilizzata nella seconda fase del progetto si fa uso di un file con estensione ".vec". In questo modo, dunque, si evitano modifiche ulteriori al codice presente nel repository.



# Capitolo 3

## Seconda Parte

Nella seconda parte dell'elaborato vengono descritti i passi per la realizzazione e l'addestramento dell'architettura di classificazione assegnata, ovvero la Multi-Input. L'addestramento di tale architettura avverrà sfruttando i pesi ottenuti dal precedente addestramento della FastText (rispettivamente su 1-grams, 2-grams e 3-grams) i quali andranno ad alimentare lo strato di embedding da agganciare alla LSTM<sup>1</sup>. L'LSTM, infine, deve essere a sua volta addestrata come classificatore: in ingresso, infatti, riceverà un dataset etichettato.

### 3.1 Dataset UMUDGA

Per la seconda parte del lavoro si è utilizzato il dataset UMUDGA [3], differente rispetto a quello creato per mezzo dei file di log della rete GAR.R.

Tale dataset è composto da 50 classi di DGA, oltre alla classe dei nomi di dominio legittimi; in totale, quindi, si hanno 51 classi.

Il dataset, inoltre, risulta bilanciato in quanto presenta lo stesso numero di nomi di dominio appartenenti a ciascuna delle 51 classi.

Al fine di utilizzare il dataset UMUDGA, l'addestramento dell'architettura è avvenuto mediante il file "*umudga\_1k.csv*" presente nel repository e considerando quattro differenti percentuali di training.

Il file "*umudga\_1k.csv*" è formato dai seguenti campi: label binaria (dga o legit), label multiclass (relativa ai singoli DGA), nome di dominio senza punti, nome di dominio originario, 1-grams, 2-grams, 3-grams e words ottenuti dal nome di dominio senza punti.

### 3.2 Architettura Multi-Input

Come architettura da addestrare è stata utilizzata la Multi-Input. Essa è caratterizzata dal prendere in input tipologie differenti di dato.

Come si può osservare dalla figura 3.1, tale architettura è caratterizzata dall'avere 4 rami di input, il primo prende sequenze di words ed è caratterizzato da uno strato di embedding basato su ELMo, gli altri 3 prendono in input sequenze di 1-grams, 2-grams e 3-grams e sono caratterizzati da strati di embeddings basati su FastText.

---

<sup>1</sup>LSTM = Long short-term memory, si tratta di una rete neurale

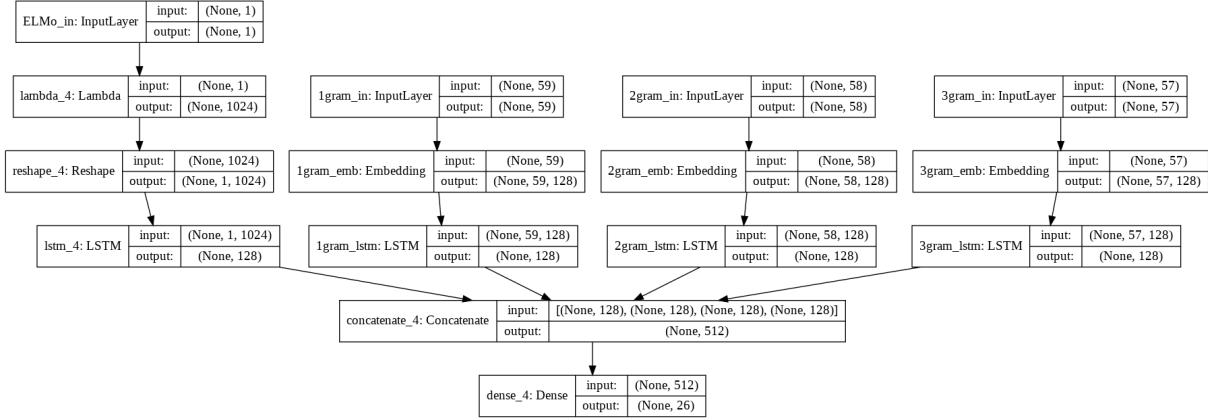


Figura 3.1: Architettura Multi-Input

### 3.3 Architettura Multi-Input con Embedding 1-gram Casuale

Come test aggiuntivo, si è deciso di addestrare una configurazione aggiuntiva dell'architettura Multi-Input. Questa ulteriore architettura, a differenza della precedente, possiede un ramo in più caratterizzato dal prendere in input sequenze di 1-grams con uno strato di embedding randomico. Avendo addestrato ambo le tipologie di Multi-Input, è stato possibile eseguire un confronto dei risultati per le diverse percentuali di training set utilizzate.

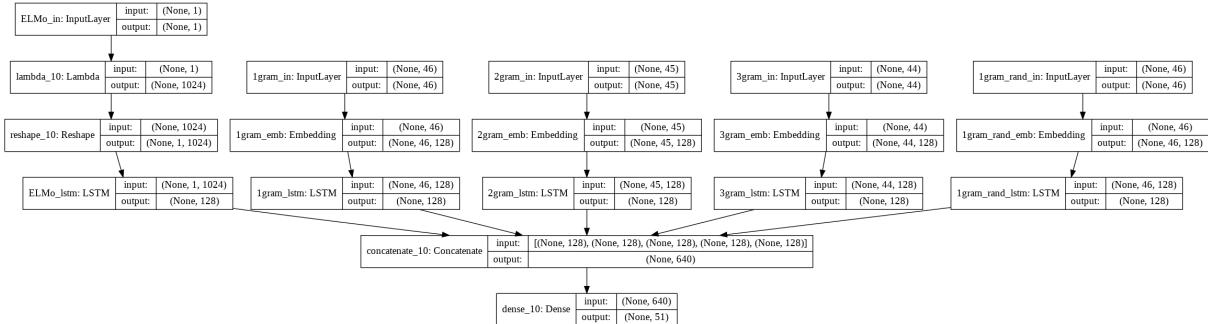


Figura 3.2: Architettura Multi-Input con Embedding Random

Nella figura 3.2 è visibile l'architettura del secondo modello addestrato in questo lavoro.

### 3.4 Iperparametri

Negli esperimenti effettuati sono stati utilizzati diversi iper-parametri, alcuni fissati secondo le specifiche del progetto, altri lasciando i valori di default.

I parametri utilizzati sono riportati di seguito.

- **Numero di epoch**: settato al valore di default, ovvero 20;
- **Batch size**: stato settato al valore di default, ovvero 128;
- **Dimensioni fastText-based embeddings**: settato secondo specifiche progettuali a 128.

## 3.5 Risultati

Nelle sezioni successive sono riportati i risultati, sia in termini computazionali che in termini di metriche ordinarie, quali ad esempio accuracy, precision, etc. per ambo le architetture descritte nelle sezioni (Sez. 3.2, Sez.3.3), divisi in base alla percentuale di training set considerato.

## 3.6 Tempi Computazionali

In questa sezione sono riportati i tempi computazionali impiegati per gli addestramenti della FastText e dell’architettura Multi-Input.

Gli addestramenti della FastText e dell’architettura Multi-Input sono stati eseguiti su Colab. In particolare, si è sfruttata un’istanza Pro di Colab, il che ha permesso di avere a disposizione GPU più performanti e, di conseguenza, ridurre i tempi di addestramento rispetto ad un’istanza gratuita.

### 3.6.1 Tempi Addestramento FastText

In questa sezione si riportano i tempi di addestramento impiegati dalla FastText per eseguire l’addestramento non supervisionato dei dati estratti dal GARR. I tempi sono divisi in base alla tipologia di suddivisione con cui si è addestrata la FastText.

Tabella 3.1: Tempi Addestramento fastText

Tipologia Dato	Durata
1-grams	~10 secondi
2-grams	~5 minuti
3-grams	~20 minuti

### 3.6.2 Tempi Addestramento Multi-Input Architecture

In questa sezione si riportano i tempi di addestramento impiegati dall’architettura Multi-Input per eseguire l’addestramento supervisionato utilizzando il dataset UMUDGA. I tempi vengono divisi in base alla percentuale di training utilizzata.

Tabella 3.2: Tempi Addestramento Multi-Input UMUDGA

Percentuale Train	Durata
0.017	~22 minuti
0.034	~26 minuti
0.068	~33 minuti
0.135	~50 minuti

### 3.6.3 Tempi Addestramento Multi-Input con Random Embedding

In questa sezione si riportano i tempi di addestramento impiegati dall’architettura **Multi-Input con Random Embedding** per eseguire l’addestramento supervisionato utilizzando il dataset UMUDGA. I tempi vengono divisi in base alla percentuale di training utilizzata.

Tabella 3.3: Tempi Addestramento Multi-Input con Random Embedding - UMUDGA

<b>Percentuale Train</b>	<b>Durata</b>
0.017	~26 minuti
0.034	~29 minuti
0.068	~35 minuti
0.135	~49 minuti

## 3.7 Risultati Classificazione con Multi-Input

In questa sezione vengono riportati i risultati della classificazione ottenuti tramite l'architettura descritta nella sezione (Sez.3.2), suddividendoli per ciascun intervallo di dataset considerato.

### 3.7.1 Risultati Classificazione - Percentuale 0.017

Tabella 3.4: Risultati Metriche di Classificazione - 0.017

<b>Metrica</b>	<b>Risultato</b>
<b>Accuracy</b>	0.594858
<b>F1-Score</b>	0.578561
<b>Precision</b>	0.609030
<b>Recall</b>	0.594858

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.3)

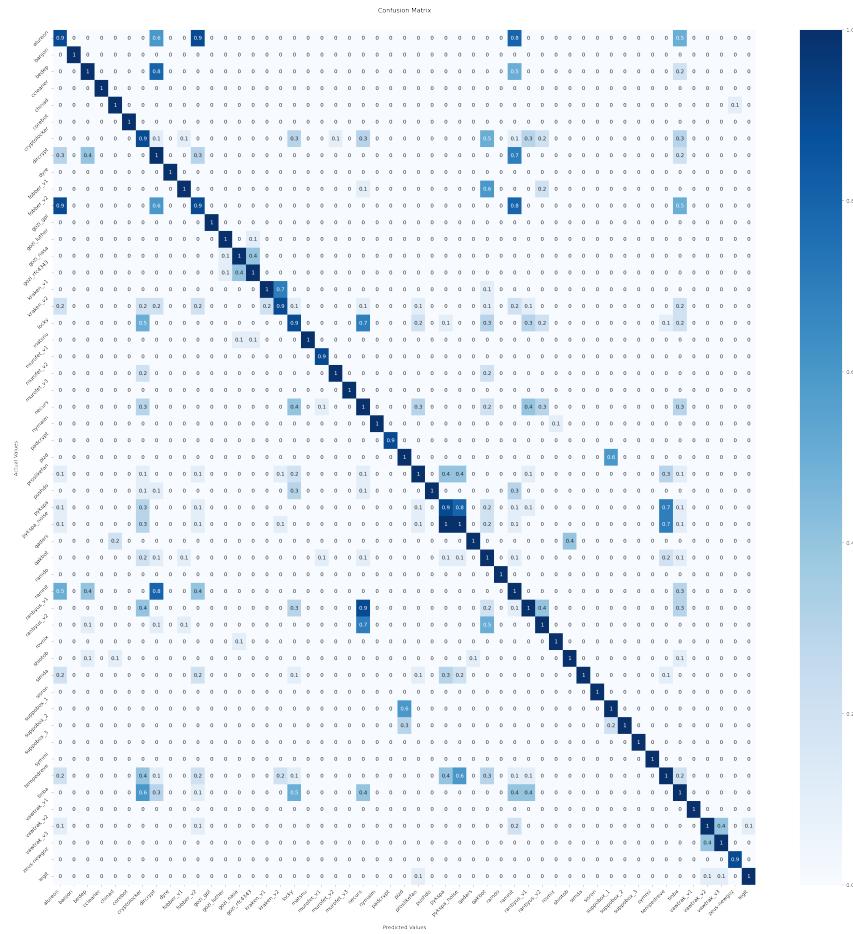


Figura 3.3: CM 0.017 Multi Input

### 3.7.2 Risultati Classificazione - Percentuale 0.034

Tabella 3.5: Risultati Metriche di Classificazione - 0.034

Metrica	Risultato
<b>Accuracy</b>	0.658689
<b>F1-Score</b>	0.648831
<b>Precision</b>	0.680713
<b>Recall</b>	0.658690

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.4)

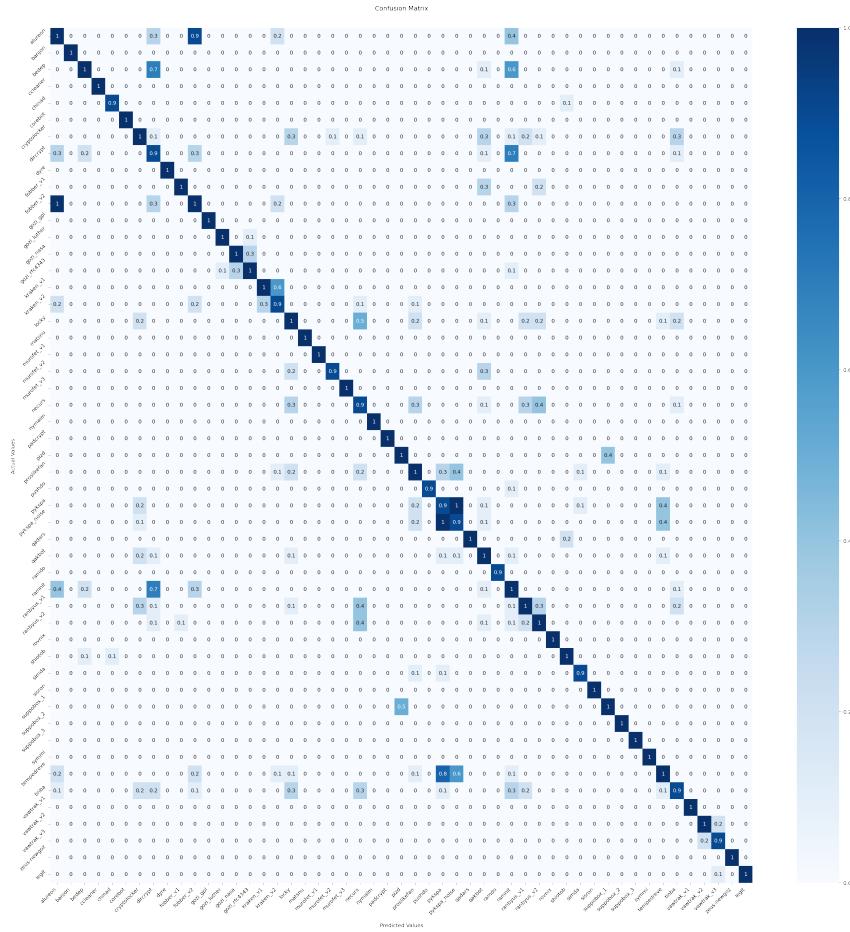


Figura 3.4: CM 0.034 Multi Input

### 3.7.3 Risultati Classificazione - Percentuale 0.068

Tabella 3.6: Risultati Metriche di Classificazione - 0.068

Metrica	Risultato
<b>Accuracy</b>	0.719612
<b>F1-Score</b>	0.708704
<b>Precision</b>	0.741019
<b>Recall</b>	0.719612

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.5)

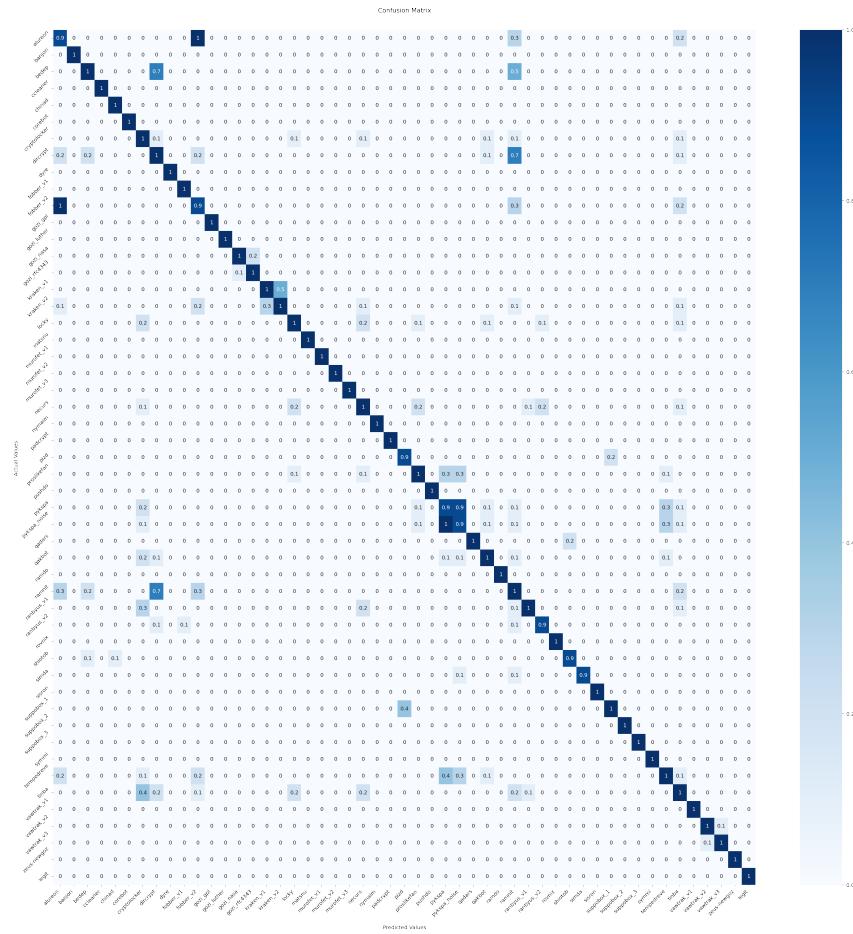


Figura 3.5: CM 0.068 Multi Input

### 3.7.4 Risultati Classificazione - Percentuale 0.135

Tabella 3.7: Risultati Metriche di Classificazione - 0.135

Metrica	Risultato
<b>Accuracy</b>	0.779504
<b>F1-Score</b>	0.773470
<b>Precision</b>	0.784416
<b>Recall</b>	0.779504

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.6)

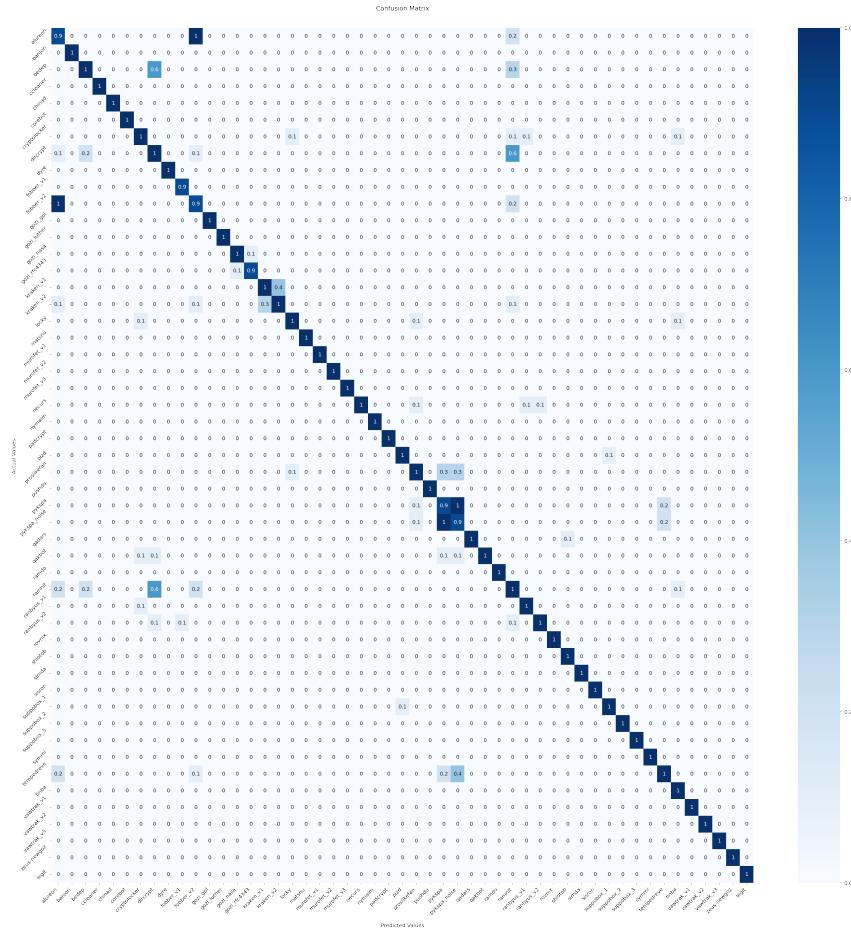


Figura 3.6: CM 0.135 Multi Input

### 3.8 Risultati Classificazione con Multi-Input e Random Embedding

In questa sezione sono riportati i risultati della classificazione con Random Embedding, dividen-doli per ciascun intervallo di dataset considerato.

#### 3.8.1 Risultati Classificazione - Percentuale 0.017

Tabella 3.8: Risultati Metriche di Classificazione - 0.017

Metrica	Risultato
<b>Accuracy</b>	0.613173
<b>F1-Score</b>	0.601629
<b>Precision</b>	0.631673
<b>Recall</b>	0.613173

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.7)

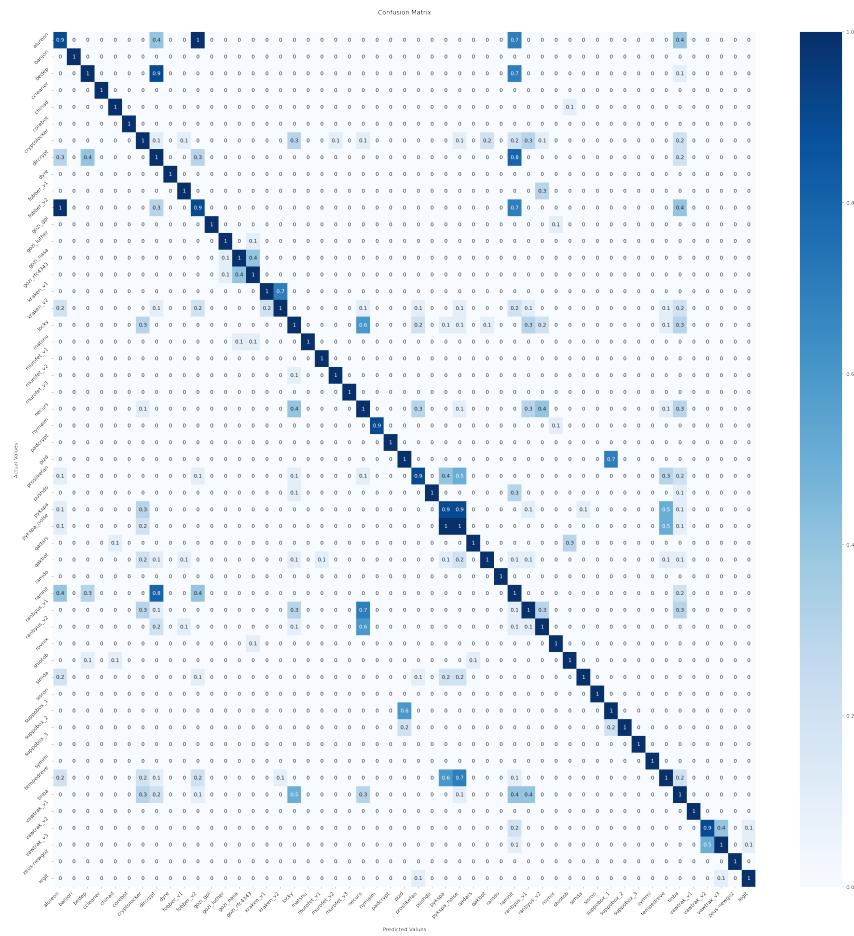


Figura 3.7: CM 0.017 Multi Input con Random Embeddings

### 3.8.2 Risultati Classificazione - Percentuale 0.034

Tabella 3.9: Risultati Metriche di Classificazione - 0.034.

Metrica	Risultato
Accuracy	0.671993
F1-Score	0.661358
Precision	0.685101
Recall	0.671993

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.8)

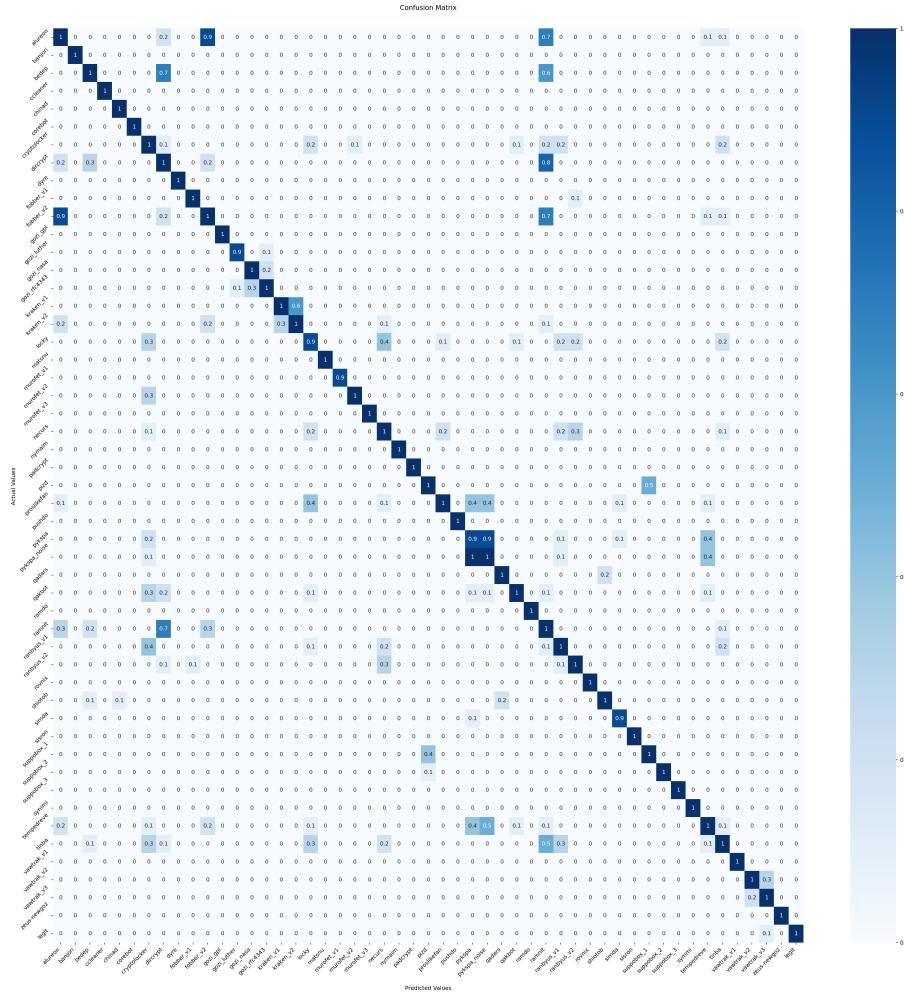


Figura 3.8: CM 0.034 Multi Input con Random Embeddings

### 3.8.3 Risultati Classificazione - Percentuale 0.068

Tabella 3.10: Risultati Metriche di Classificazione - 0.068

Metrica	Risultato
<b>Accuracy</b>	0.697278
<b>F1-Score</b>	0.692756
<b>Precision</b>	0.748620
<b>Recall</b>	0.697278

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.9)

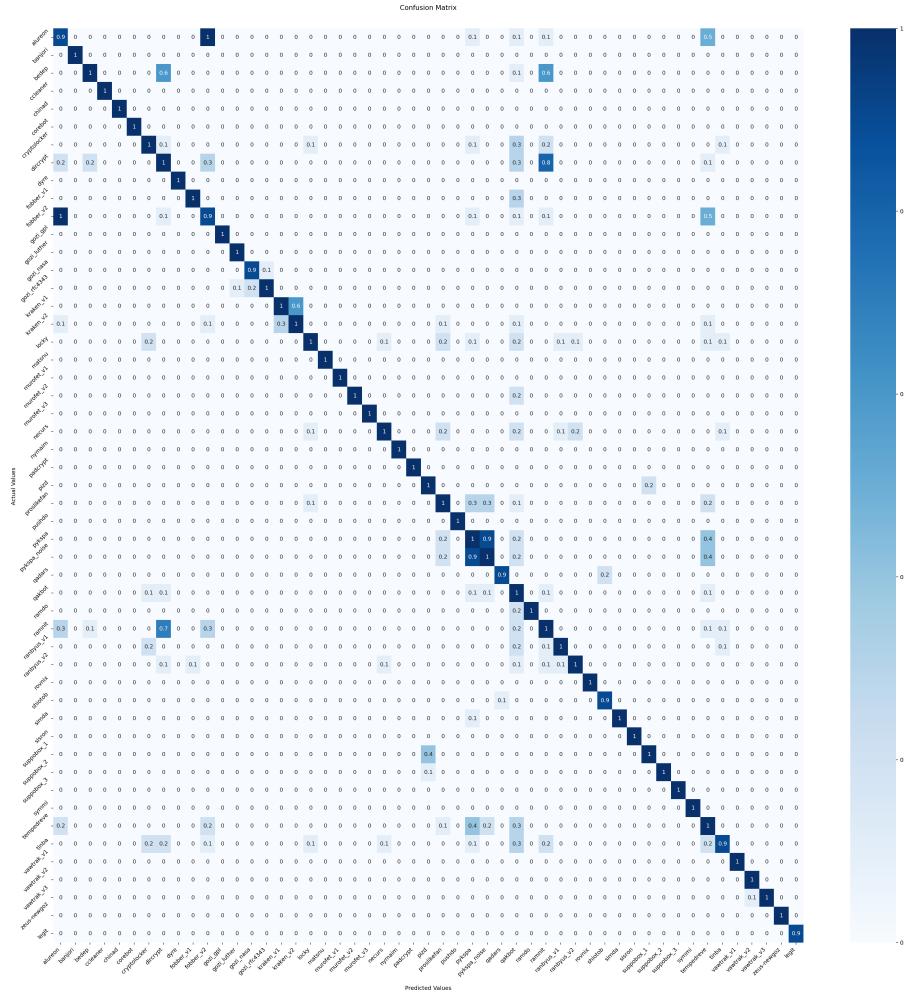


Figura 3.9: CM 0.068 Multi Input con Random Embeddings

### 3.8.4 Risultati Classificazione - Percentuale 0.135

Tabella 3.11: Risultati Metriche di Classificazione - 0.135

Metrica	Risultato
<b>Accuracy</b>	0.782414
<b>F1-Score</b>	0.780383
<b>Precision</b>	0.788942
<b>Recall</b>	0.782414

**Matrice di Confusione** Nel seguente paragrafo è riportata la matrice di confusione, così da avere un overview migliore sui risultati della classificazione (Fig.3.10)

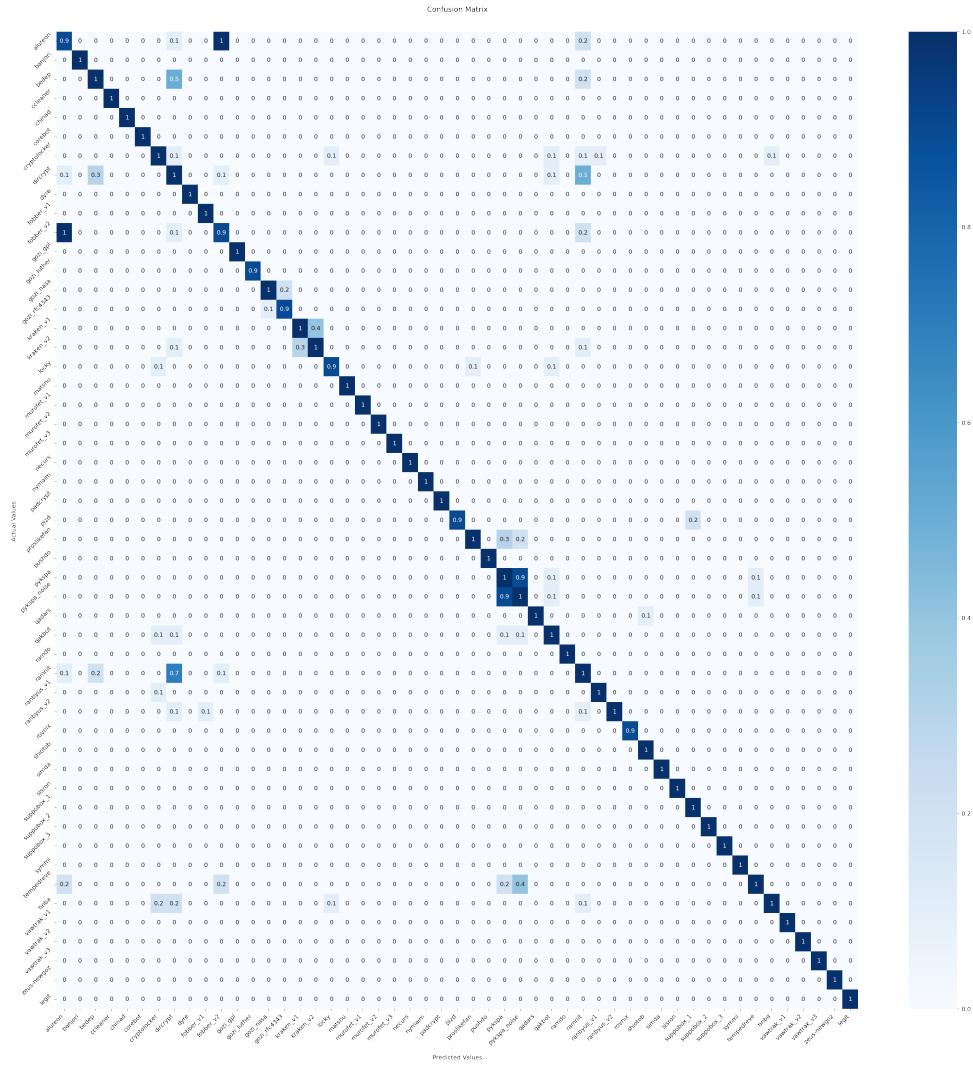


Figura 3.10: CM 0.135 Multi Input con Random Embeddings

## 3.9 Confronto Risultati Architetture

### 3.9.1 Risultati Classificazione

Tabella 3.12: Risultati Metriche di Classificazione

Architettura	Percentuale Train	Accuracy	F1-Score	Precision	Recall
Multi-Input	0.017	0.594858	0.578561	0.609030	0.594858
Multi-Input con Random Emb.	0.017	<b>0.613173</b>	<b>0.601629</b>	<b>0.631673</b>	<b>0.613173</b>
Multi-Input	0.034	0.658689	0.648831	0.680713	0.658690
Multi-Input con Random Emb.	0.034	<b>0.671993</b>	<b>0.661358</b>	<b>0.685101</b>	<b>0.671993</b>
Multi-Input	0.068	<b>0.719612</b>	<b>0.708704</b>	0.741019	<b>0.719612</b>
Multi-Input con Random Emb.	0.068	0.697278	0.692756	<b>0.748620</b>	0.697278
Multi-Input	0.135	0.779504	0.773470	0.784416	0.779504
Multi-Input con Random Emb.	0.135	<b>0.782414</b>	<b>0.780383</b>	<b>0.788942</b>	<b>0.782414</b>

Nella tabella 3.12 si possono confrontare i valori delle metriche ottenuti con le due differenti architetture addestrate precedentemente.

Ponendo particolare attenzione sulle metriche di accuracy e f1-score, si può osservare come, per i tagli più piccoli (0.017 e 0.034), si ottengono risultati migliori utilizzando l'architettura che racchiude anche lo strato di embedding randomico. Per le altre percentuali si ottengono invece valori poco differenti.



## Capitolo 4

# Conclusioni e Sviluppi Futuri

In conclusione, con il lavoro realizzato è possibile addestrare e testare differenti architetture al fine di effettuare una classificazione multi-classe per vari nomi di dominio appartenenti al dataset UMUDGA.

Dal confronto fra le diverse architetture utilizzate è possibile osservare che l'architettura Multi-Input che include lo strato di embedding casuale riesce ad ottenere risultati migliori soprattutto per percentuali di training più piccole, quindi nei casi in cui si hanno a disposizione pochi nomi di dominio (si pensi a quelle situazioni in cui un nuovo DGA è stato individuato). Il vantaggio di tale architettura tende a diminuire, confrontato con i valori dell'architettura Multi-Input senza strato di embedding casuale, al crescere della percentuale di train e quindi del numero di nomi di dominio.

Si può concludere, quindi, che l'obiettivo prefissato di classificare i nomi di dominio presenti nel dataset sfruttando un pre-addestramento della FastText, è stato raggiunto.

Possibili sviluppi futuri potrebbero riguardare l'allenamento di nuove architetture, i cui risultati possono essere confrontati con quelli presenti in questo lavoro. Altri possibili sviluppi potrebbero riguardare la prima fase compiuta in questo lavoro, ovvero l'addestramento della FastText da sfruttare poi negli strati di embedding. Si potrebbe pensare di ampliare il dataset utilizzato o effettuare un'operazione di pulizia dei nomi di dominio estratti dal GARR più profonda, ad esempio tenendo soltanto i nomi di dominio di primo livello. Così facendo, si potrebbero confrontare i risultati ottenuti con quelli presentati in questo elaborato, così da osservare i cambiamenti e la dipendenza dei risultati della seconda fase con il dataset utilizzato nella prima fase.



# Bibliografia

- [1] FastText. Word representations, May 2022. URL <https://fasttext.cc/docs/en/unsupervised-tutorial.html>.
- [2] FastText. Cbow vs skipgram, May 2022. URL <https://fasttext.cc/docs/en/unsupervised-tutorial.html#advanced-readers-skipgram-versus-cbow>.
- [3] Mattia Zago, Manuel Gil Pérez, and Gregorio Martínez Pérez. Umudga: A dataset for profiling dga-based botnet. *Computers & Security*, 92:101719, 2020. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2020.101719>. URL <https://www.sciencedirect.com/science/article/pii/S0167404820300067>.