Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Adversarial Machine Learning per il Rilevamento di Botnet

Relatore:
Prof. Michele Colajanni

Candidato:
Alessandro Aleotti

Correlatore:
Ing. Mirco Marchetti

Anno Accademico 2017/2018

Indice

Todo list

Capitolo 1

Introduzione

In questo capitolo si propongono degli esempi per gli oggetti utilizzati più di frequente in latex: la Sezione 1.1 descrive come scrivere citazioni, la Sezione 1.2 propone degli esempi di oggetti float, la Sezione 1.3 descrive come compilare questo documento.

1.1 Citazioni

Inserisco qualche citazione per mostrare la bibliografia. Per gli articoli accademici è quasi sempre possibile reperire i blocchi da inserire nel file bib da scholar, come ad esempio. Scholar in questo caso è una risorsa/sito online e per questo. Precediamo le citazione da uno spazio indivisibile tramite il carattere ~.

1.2 Oggetti float

Nella Sezione 1.2.1 si propone un esempio di figura float, mentre nella Sezione 1.2.2 si propone un esempio di tabella float.

1.2.1 Figure

La Figura 1.1 è un esempio di figura float.

Figura 1.1: Esempio di figura float in latex.

1.2.2 Tabelle

La Tabella 1.1 è un esempio di tabella.

allineamento centrale	allineamento a sinistra	allineamento a destra
centrale	sinistra	destra

Tabella 1.1: Esempio di tabella float in latex.

1.3 Compilazione

Di seguito il codice da utilizzare per generare il pdf:

```
$ pdflatex main.tex
$ bibtex main.aux
$ pdflatex main.tex
$ pdflatex main.tex
```

Capitolo 2

Stato dell'arte

2.1 Machine Learning

2.2 Reti neurali

2.2.1 Percettrone Multistrato

2.2.2 Adversarial Networks

The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution p_g over data $\boldsymbol{x}$, we define a prior on input noise variables $p_{\boldsymbol{z}}(\boldsymbol{z})$, then represent a mapping to data space as $G(\boldsymbol{z}; \theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters θ_g. We also define a second multilayer perceptron $D(\boldsymbol{x}; \theta_d)$ that outputs a single scalar. $D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1 - D(G(\boldsymbol{z})))$:

In other words, D and G play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]. \tag{2.1}$$

In the next section, we present a theoretical analysis of adversarial nets, essentially showing that the training criterion allows one to recover the data generating distribution as G and D are given enough capacity, i.e., in the non-parametric limit. See Figure 2.1 for a less formal, more pedagogical explanation of the approach. In practice, we must implement the game using an iterative, numerical approach. Optimizing D to completion in the inner loop of training is computationally prohibitive, and on finite datasets would result in overfitting. Instead, we alternate between k steps of optimizing D and one step of optimizing G. This results in D being maintained near its optimal solution, so long as G changes slowly enough. This strategy is analogous to the way that SML/PCD [**? ?**] training maintains samples from a Markov chain from one learning step to the next in order to avoid burning in a Markov chain as part of the inner loop of learning. The procedure is formally presented in Algorithm 1.

In practice, equation 2.1 may not provide sufficient gradient for G to learn well. Early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data. In this case, $\log(1 - D(G(\boldsymbol{z})))$ saturates. Rather than training G to minimize $\log(1 - D(G(\boldsymbol{z})))$ we can train G to maximize $\log D(G(\boldsymbol{z}))$. This objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning.

2.2.2.1 Autoencoder

2.3 Adversarial Learning

??? come?

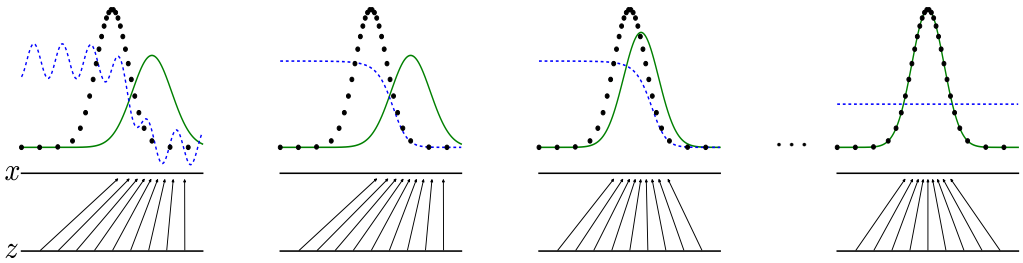

Figura 2.1: Generative adversarial nets are trained by simultaneously updating the **d**iscriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the **g**enerative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}$. (c) After an update to G, gradient of D has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

• Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

• Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.

• Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

• Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

• Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Capitolo 3

Progetto

In questo capitolo si propone il progetto realizzato per raggiungere gli obiettivi preposti: la realizzazione di un classificatore basato sull'algoritmo di machine learning *Random Forest* in grado di distinguere domini generati algoritmicamente da domini reali. Si è successivamente progettato un classificatore più raffinato basato su rete neurale in grado di superare le mancanze de precedente classificatore. A partire da tale sistema, si è passati alla progettazione di un sistema di *adversarial learning* in grado di rafforzare il classificatore neurale: una *Generative Adversarial Network* (abbr. *GAN*) in grado di creare domini sintetici realistici da fornire successivamente al classificatore in fase di training (Figura 3.1). Tale GAN è stata progettata a partire da un Autoencoder, il cui compito è mimare la distribuzione di un dataset fornito in input (Figura 3.2).

parlare di funzioni di loss, di ottimizzatori

3.1 Classificatore Random Forest

La prima fase di questo studio è stata quella di implementare un classificatore in grado di separare efficacemente domini *DGA* da domini reali basandosi unicamente sulle caratteristiche linguistiche dei domini: infatti, ad un esame preliminare, i domini *DGA* presentano caratteristiche ben differenti da semplici frasi o parole che solitamente compongono i domini reali.

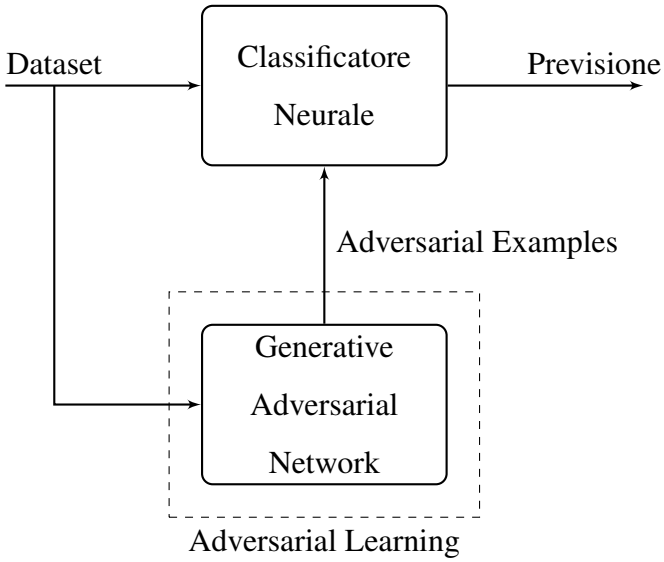

Figura 3.1: Diagramma generale di progetto.

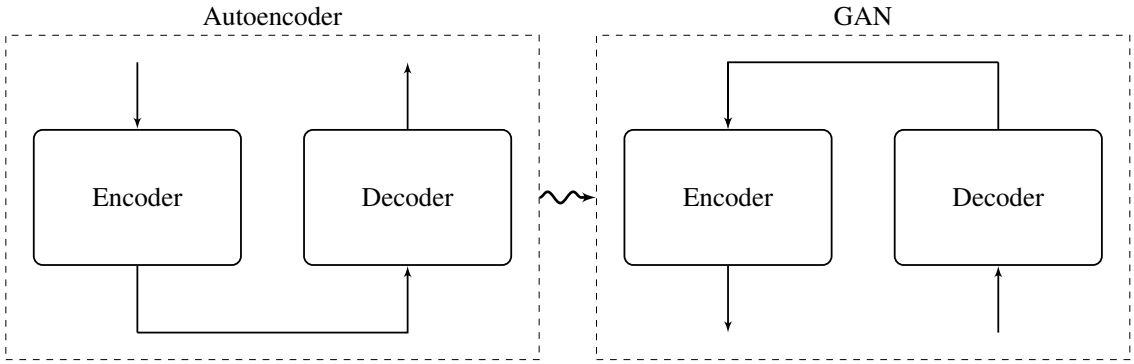

Figura 3.2: Schema generale della progettazione da Autoencoder a Gan.

Si è scelto di utilizzare Random Forest in quanto ritenuto il più adatto al caso in esame. L'algoritmo è stato inoltre messo a confronto con *Support Vector Machine* e *Naive-Bayes*.

All'interno del classificatore *Random Forest* [1], ogni albero dell'insieme è costruito a partire da un campione estratto con sostituzione dal *training set*. In aggiunta, al momento della divisione del nodo durante la costruzione di un albero, la divisione scelta non è più la migliore soluzione tra tutte le *features*. Al suo posto, la divisone che viene scelta è la migliore divisione all'interno di un *subset* casuale tra tutte le *features*. Come risultato di questa casualità, il *bias* della foresta di solito aumenta leggermente (rispetto al *bias* di un singolo albero non casuale) ma, a causa della media, la sua varianza diminuisce, di solito compensando l'aumento di *bias*, quindi dando un modello generale migliore.

3.1.1 Dataset

I *dataset* di *training* e *testing* sono stati ricavati due fonti differenti: per quel che riguarda i domini reali si è fatto riferimento alla classifica dei domini più visitati al mondo fornita da *Alexa Internet Inc.* [2] , per un totale di 1 milione di siti realmente esistenti; mentre grazie al repository fornito da [3] è stato possibile ottenere un *dataset* esaustivo di esempi *DGA* da diverse famiglie di *malware* tra i quali ransomware come *cryptolocker* e *cryptowall*, trojans bancari come *hesperbot*, e information stealers come *ramnit*. Le tecniche DGA tradizionali variano in complessità da semplici approcci che estraggono caratteri casualmente a quelli che cercano di mimare la distribuzione di lettere o parole trovate nei domini reali. Il DGA di *ramnit*, ad esempio, crea nomi di dominio usando una combinazione di moltiplicazioni, divisioni e resti a partire da un seme randomico. Agli antpodi, *suppobox* crea domini concatenando due parole scelte in maniera pseudo-casuale da un piccolo dizionario Inglese. In tabella 3.1 vengono mostrati alcuni esempi di domini generati algoritmicamente a seconda delle diverse famiglie di malware. La maggior parte dei DGA opera a livello di singoli caratteri, mentre altre tipologie comuni come beebone hanno una struttura rigida, che produce domini come ns1.backdates13.biz e ns1.backdates0.biz. Il DGA symmi produce domini vagamente pronunciabili tra i quali “hakeshoubar.ddns.net” estraendo una vocale o una consonante casualmente per ogni indice pari e successivamente estraendo l’opposto all’indice successivo oltre che ad aggiungere un dominio di primo e secondo livello al termine della stringe come .ddns.net La distribuzione dei singoli caratteri (unigrammi) per 4 famiglie di DGA e Alexa sono mostrate di seguito. La distribuzione di cryptolocker e ramnit sono entrambe uniformi all’interno dello stesso range. Si tratta di una caratteristica attesa in quanto entrambi sono generati tramite una serie di moltiplicazioni, divisioni e resti basati su di un singolo seme. Suppobox, d’altro canto presenta caratteristiche interessanti in quanto genera unigrammi simili per distribuzione ad Alexa.

corebot	ep16g6gjwfixyhs8gfy.ddns.net
	ev5texifc43nebil3pk.ddns.net
	gf7bm4163fmjkje.ddns.net
cryptolocker	agryjvdaabkyt.ru
	pwitjnqgjfaqm.org
	dhhubfepcdgfv.co.uk
dircrypt	hedhryendqlss.com
	lgnggnlufbtyjpnvct.com
	tzrbdmhoumoy.com
kraken v2	fwulvdmdytm.com
	gybuisybe.cc
	gyinkvye.net
lockyv2	btlwubflhfllshn.info
	cpgcjsysfwuwa.click
	jlbroeji.biz
pykspa	gqjgflhop.net
	gqumcwaa.org
	jpivjh.net
qakbot	fgfifkyfut.info
	flzuzsaekkipatbtet.biz
	owpbsjekk.com
ramdo	kugmywaaiymaegiq.org
	ocywskaagmmqscoc.org
	uomywsaaqggiwouo.org
ramnit	byqdmekgd.com
	dpmdbwwcmpk.com
	gkkcoufektvhiqr.com
simda	gatyfusyfi.com
	lyvyxoryco.com
	puvyxilomo.com

Tabella 3.1: Esempi di domini generati algoritmicamente da Malware.

A partire da tale *dataset* combinato si è proceduto alla creazione di un classificatore binario che fosse in grado di distinguere domini reali da domini generati algoritmicamente. Il passo seguente stato creare una serie di *features* che fossero in grado di descrivere le caratteristiche linguistiche dei domini presi in esame.

Per raggiungere tale obiettivo si è fatto riferimento a ricerche già esistenti: [5] [6] [7] [8]. Di seguito viene illustrato l'insieme di tali *features*:

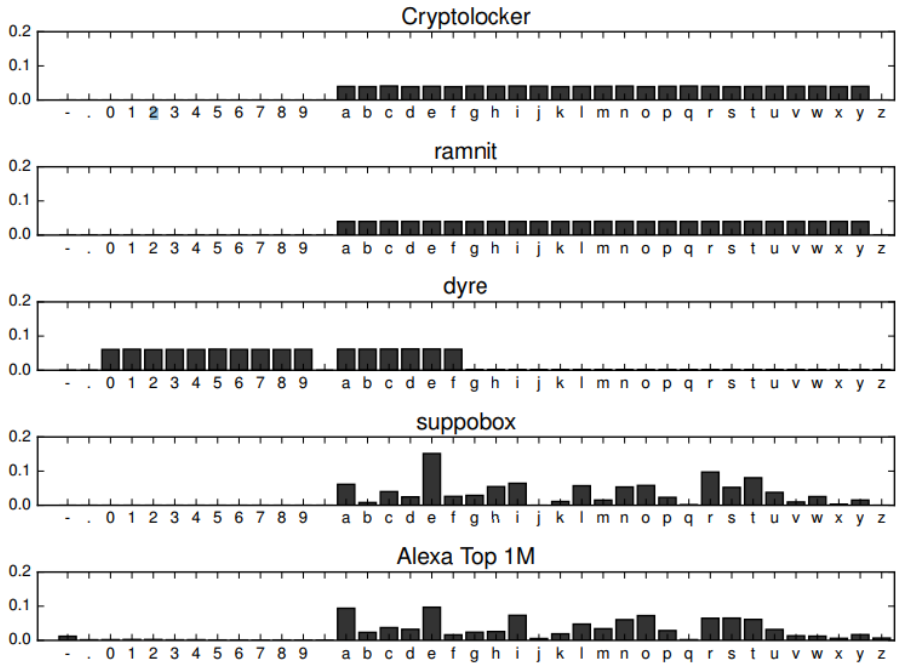

Figura 3.3: Distribuzione dei singoli caratteri in *cryptolocker, ramnit, dyre*, *suppobox* (dictionary-based DGA) ed il primo milione di domini più visitati secondo Alexa [2]. *fonte:* [4].

3.1.2 Features

- **Rapporto tra caratteri significativi**. Modella il rapporto dei caratteri della stringa p che formano una parola significativa all'interno del dizionario Inglese. Un valore basso indica la presenza di algoritmi automatici. In dettaglio, si divide p in n sotto-parole significative w_i di almeno 3 caratteri: $|wi| \geq 3$ cercando di lasciare fuori meno caratteri possibili:

$$R(d) = R(p) = \frac{max(\sum_{i=1}^{n} |wi|)}{|p|}$$

  Se $p =$ facebook, $R(p) = \frac{(|\text{face}|+|\text{book}|)}{8} = 1$ allora il dominio è composto completamente da parole significative, mentre $p =$ pub03str, $R(p) = \frac{|\text{pub}|}{8} = 0.375$.

- **Punteggio di normalità degli n-grammi**: Questa classe di *features* modella la pronunciabilità di un nome di dominio rispetto la lingua Inglese. Più la combinazione di fonemi del dominio è presente all'interno del Dizionario Inglese più tale dominio è pronunciabile. Domini con un basso numero di tali combinazioni sono probabilmente generati algoritmicamente. Il calcolo avviene estraendo lo n-gramma di p di lunghezza

$n \in \{1, 2, 3, 4, 5\}$ e contando il numero di occorrenze di tale n-gramma all'interno del Dizionario Inglese. Tali *features* sono quindi parametriche rispetto ad n:

$$S_n(d) = S_n(p) \ = \frac{\sum_{\text{n-gramma t in p}} count(t)}{|p| - n + 1}$$

dove $count(t)$ sono le occorrenze dello n-gramma nel dizionario. Ad esempio $S_2(facebook) = fa_{109} + ac_{343} + ce_{438} + eb_{29} + bo_{118} + oo_{114} + ok_{45} = 170.8$

- **Rapporto tra caratteri numerici** Questa *feature* rappresenta il rapporto tra i caratteri numerici presenti all'interno del nome di dominio rispetto la lunghezza totale della parola. Molte famiglie di *malware* utilizzano *DGA* che generano domini tramite una distribuzione uniforme di caratteri alfabetici minuscoli e numeri, questo porta a domini generati algoritmicamente che presentano una maggior presenza di numeri al loro interno rispetto ai domini reali.

- **Rapporto tra vocali e consonanti** Questa *feature* modella il rapporto tra vocali e consonanti all'interno del nome di dominio.

- **Lunghezza del nome di dominio** Questa *feature* calcola la lunghezza del dominio. Molte famiglie di *malware* utilizzano *DGA* che generano domini di lunghezza costante, generalmente molto lunghi rispetto ai domini reali.

L'implementazione di tali *features* ha permesso di ottenere un *dataset* in grado di modellare le caratteristiche linguistiche dei nomi di dominio mostrati al capitolo **??**. Da tale spunto è partita la fase iniziale di *testing*

3.1.3 Output

L'obiettivo di tale classificatore è quello di riuscire a separare in maniera efficace i domini reali da quelli generati algoritmicamente. Durante la fase di sperimentazione il classificatore si è rivelato efficace rispetto la maggior parte delle famiglie di *DGA*; tuttavia il caso particolare della famiglia *suppobox* [9] ha messo in particolare difficoltà il classificatore in quanto tale algoritmo genera domini in maniera pseudo-casuale, concatenando due parole a partire da un *subset* di 384 parole provenienti dal dizionario inglese. Tale caratteristica fa si

che le *features* linguistiche estratte da questa famiglia di *malware* siano molto simili a quelle presenti nei domini reali. In tabella 3.2 sono mostrati alcuni esempi di domini generati da Suppobox.

Suppobox
increaseinside.net
wouldinstead.net
rememberinstead.net
wouldexplain.net
rememberexplain.net
wouldbright.net
rememberbright.net
wouldinside.net
rememberinside.net

Tabella 3.2: Esempio di domini generati da Suppobox.

A partire da questo risultato si scelto di procedere con la progettazione di un classificatore neurale in grado di superare tale problematica.

3.2 Classificatore Neurale

Questo classificatore neurale nasce con l'intento di superare le difficoltà incontrate dal precedente classificatore basato su *Random Forest*, utilizzando le caratteristiche delle reti neurali, in grado di estrarre *features* a partire dai dati grezzi. Si è scelto di partire dall'architettura di tipo *Multilayer Perceptron* con l'obiettivo di ottenere risultati migliori rispetto al caso mostrato nella sezione precedente.

I passi del progetto sono stati la codificazione dei domini in valori numerici, l'individuazione di una architettura ottimale per classificare i dati in esame ed un'ultima fase di *tuning* degli iperparametri della rete neurale.

3.2.1 Dataset

A partire dal *dataset* creato per il precedente caso, si è deciso di convertire direttamente i nomi di dominio alfanumerici in vettori numerici, mappati secondo il dizionario di tutti i caratteri ammessi [10] (lettere minuscole a-z, numeri 0-9, tratto d'unione "-"). L'obiettivo è quello di fornire al classificatore neurale in questione una rappresentazione il più possibile aderente ai dati reali, senza l'ausilio di *features* ingegnerizzate a priori, lasciando così la libertà alla rete neurale di estrarre le caratteristiche più appropriate per la distinzione dei domini. Come scelta progettuale si è deciso di limitare la dimensione dei domini a 15 caratteri per ognuno, in modo da ottenere un *dataset* di dimensioni fissate e sopperire alle differenti lunghezze di ogni dominio tramite un semplice *padding* di zeri in testa ad ogni stringa codificata.

Assieme ai dati codificati è stato generato un vettore di *target* nel quale viene indicato da 0 o da 1 se il dominio in esame è di tipo reale o generato algoritmicamente. L'obiettivo quindi è di attuare un classificatore binario in grado di prevedere correttamente a quale categoria appartiene un dominio esaminato

3.2.2 Architettura Classificatore Neurale

L'architettura scelta in prima fase è stata quella del *Multilayer Perceptron (abbr. MLP)*, una tipologia di rete neurale *feedforward* tipicamente formata da almeno tre livelli di nodi. Ad esclusione del livello di *input* i livelli del MLP utilizzano funzioni di attivazione non lineari che permettono di eseguire distinzioni tra dati non linearmente separabili. Considerando una rete formata da m neuroni, se si considera d come numero di input, si avrà il seguente output

$$y_j = y\left(\sum_{i=0}^{d} w_{ji}x_i\right)$$

nel quale x_i sono gli input e w_{ji} sono i pesi di ogni input combinati con ogni output.

Nel caso in esame è stata utilizzata per i livelli interni la funzione di attivazione *Rectifier Linear Unit* (ReLU) [11] definita dalla funzione sottostante e mostrato in figura 3.4

$$f(x) = x^+ = max(0, x)$$

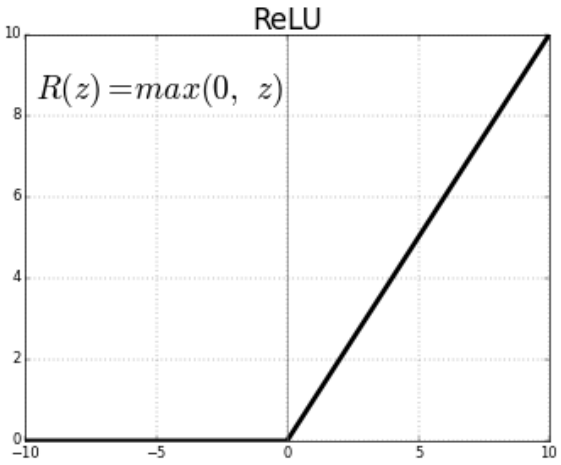

Figura 3.4: *fonte:* [12].

dove x rappresenta l'*input* del neurone. I vantaggi di tale funzione sono una migliorata *performance* rispetto ad altre funzioni similari come *tanh* e *sigmoid* per quel che riguarda la convergenza della discesa stocastica del gradiente.

Per quel che riguarda la funzione di attivazione del livello di *output* si è scelta la funzione *sigmoidea*, definita dalla formula

$$P(t) = \frac{1}{1 + e^{-t}}$$

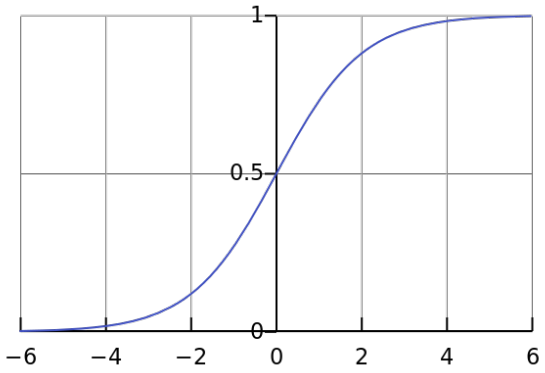

Figura 3.5: *fonte:* [13]

La struttura finale del *MLP* in esame è stata raggiunta dopo una serie di test sperimentali in cui si sono messi a confronto tre modelli differenti di per numero di neuroni all'interno degli *hidden layer*:

- un modello ridotto composto da un layer di input con un numero di neuroni pari alla dimensione delle stringhe codificate, un layer intermedio di dimensione dimezzata rispetto al precedente ed il layer finale di uscita di dimensione 1 per attuare la classificazione binaria, oggetto di studio.

- un modello allargato composto da un layer di input con un numero di neuroni pari alla dimensione delle stringhe codificate, due layer intermedi di dimensioni moltiplicate di diversi ordini rispetto al layer iniziale ed un layer finale di dimensione 1.

- un modello intermedio composto da un layer di input con un numero di neuroni pari alla dimensione delle stringhe codificate, un layer intermedio di dimensione 128, un layer di dimensione minore a 64 ed un layer finale di dimensione 1. (Figura 3.6)

I tre modelli messi a confronto hanno mostrato risultati simili, tuttavia il modello intermedio si è dimostrato più performante, con un costo computazionale irrisorio rispetto al modello allargato, pertanto è stato scelto come riferimento per gli studi successivi.

3.2.3 Output

L'intento della rete neurale proposta è quello di classificare autonomamente domini reali da domini generati algoritmicamente, con l'obiettivo di superare le fragilità del classificatore precedente (3.1) ed avere una linea di confronto affidabile per lo *step* di lavoro successivo: l'introduzione di un sistema di *adversarial learning* che possa rafforzare tale classificatore.

3.3 Realizzazione Adversarial Learning

Ricerche precedenti hanno dimostrato che molti modelli di machine learning, incluse le reti neurali, sono vulnerabili agli *adversarial examples* [14], [15]. In particolare la ricerca proposta in [15] introduce il metodo del *fast gradient sign* per scoprire *adversarial examples* perturbando un campione noto x con una piccola quantità $\Delta x = \in sign(\nabla_x J(\theta, x, y))$ dove θ rappresenta i parametri del modello e J il costo necessario a classificare x come y. Separatamente [16] propone l'uso di *Generative Adversarial Network (abbr. GAN*) come *framework* in grado di generare campioni artificiali provenienti dalla stessa distribuzione del

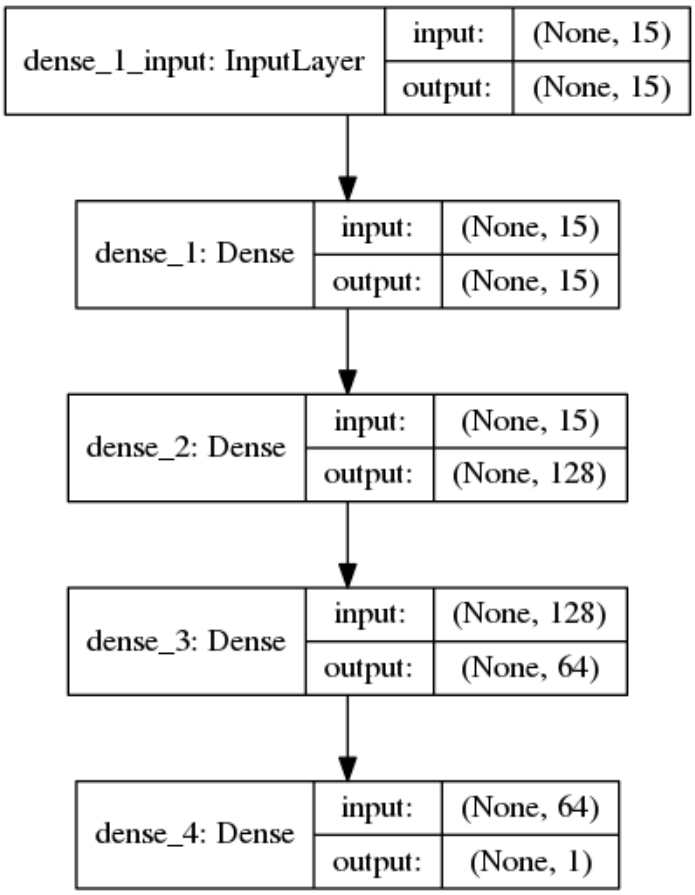

Figura 3.6: Grafico del modello intermedio. Escluso il layer di input iniziale, si notino gli *hidden layer* dense2 e dense3 di dimensioni rispettivamente 128 e 64.

training set. Le *GAN* incorporano due modelli: un generatore ed un discriminatore i quali competono in una serie di turni antagonisti. All'interno del contesto del lavoro presentato in questo elaborato, il generatore impara a creare nuovi domini artificiali mentre il discriminatore impara a distinguere tali domini artificiali da quelli reali. L'intento di tale lavoro è usare la *GAN* per produrre domini artificiali realistici e di conseguenza incrementare la precisione del classificatore presentato nella sezione precedente attraverso l'*adversarial training*. I presupposti progettuali di questo elaborato sono ispirati alla ricerca presentata in [4].

3.4 Autoencoder

Il punto di partenza per il lavoro di progettazione di una *GAN* è stato l'implementazione di un *Autoencoder* funzionante. Un *Autoencoder* è un modello di rete neurale non supervisionata

con lo scopo di riprodurre il proprio input passando attraverso una rappresentazione codificata, generalmente a dimensione inferiore [17] [18]. Si supponga di avere un set di training $\left\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\right\}$ dove $x^{(i)} \in \mathbb{R}^n$. L'obiettivo di un autoencoder generico è $y^{(i)} = x^{(i)}$ cercando di imparare una funzione che approssima x $h_{W,b}(x) \approx x$. Un *autoencoder* tipicamente consiste in due macro-componenti:

- funzione **Encoder** $h = f(x)$ la quale trasforma l'input una rappresentazione codificata (generalmente a dimensione minore)
- funzione **Decoder** $r = g(h)$ in grado di ricostruire l'input a partire dalla rappresentazione codificata.

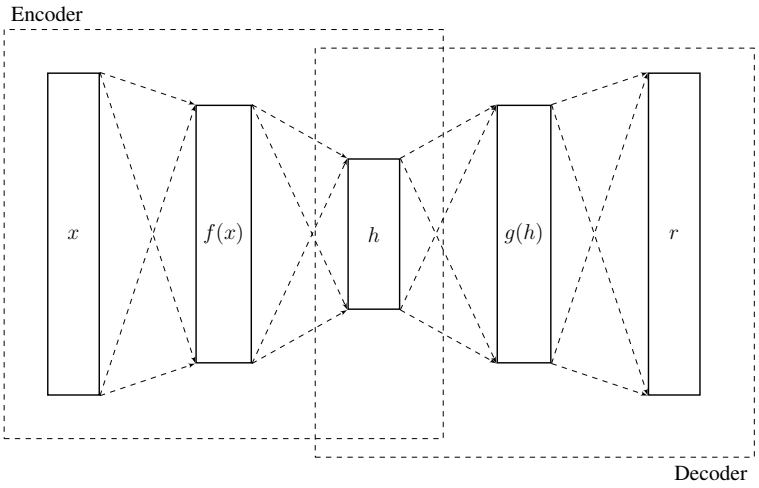

Figura 3.7: Struttura generica di un autoencoder, il quale mappa l'input x in un output r attraverso una rappresentazione codificata h.

Tuttavia il reale obiettivo di un *autoencoder* non è quello di imparare perfettamente a riprodurre l'input fornito (in quanto sarebbe un'operazione priva di utilità), bensì vengono introdotti vincoli che ne limitano la capacità di riproduzione ad una sola approssimazione dei dati di ingresso. Grazie a tali vincoli il modello è obbligato a dare priorità agli aspetti fondamentali dell'input, imparandone le proprietà principali. L'obiettivo di tale implementazione nel contesto di questo elaborato è poter cogliere le caratteristiche fondamentali che compongono i domini reali, per poterli riprodurre al meglio all'interno della *GAN* e generare domini simili a quelli reali a partire da rumore casuale.

3.4.1 Dataset Autoencoder

Il *dataset* utilizzato per il training di tale *autoencoder* è lo stesso mostrato nella sezione 3.2.1, in cui i domini sono mappati in vettori numerici, secondo il dizionario di caratteri ammissibili per i domini. Durante la fase di implementazione si è reso necessario un ulteriore *step* di *preprocessing*: i domini codificati in sequenze di valori interi sono stati ulteriormente codificati tramite il *one hot encoding* [19] in modo da formare un tensore 2D per ogni dominio, in cui ogni riga è formata da sequenze di bit a 0 tranne il carattere nella posizione indicata dal dizionario, il quale è indicato ad 1. I domini così codificati vengono trattati come una sequenza temporale, in cui ogni *step* è caratterizzato da un vettore nel quale è indicato a 1 quale carattere del dizionario $\mathbb{V}$ vi è rappresentato.

Questo ulteriore passaggio è diventato necessario durante l'implementazione della *GAN*, in modo da poter utilizzare il tensore di output del *decoder* come ingresso per l'*encoder*.

3.4.2 Architettura Autoencoder

L'architettura dell'*encoder* in esame è ispirato al lavoro mostrato in [20] mentre il *decoder* è approssimativamente una immagine speculare dell'*encoder*.

Al domini codificati come indicato nella sezione precedente, vengono applicati dei filtri convoluzionali con l'obiettivo di catturare n-grammi significativi all'interno dei domini reali. Il layer successivo di concatenazione assembla l'output dei diversi filtri in un tensore di dimensione ridotta rispetto all'input iniziale e lo passa ad una LSTM la quale accumula stato lungo la sequenza di caratteri e ritorna in uscita il dominio codificato in forma di vettore mono-dimensionale.

Il *decoder* è lascamente l'inverso del processo di codifica: il dominio codificato dato in input viene ripetuto un numero di volte equivalente alla lunghezza massima di nome di dominio decisa a priori e passato ad una LSTM. La sequenza di emissioni da parte del layer LSTM viene fornita agli stessi filtri convoluzionali presenti all'interno dell'*encoder*. Questo risulta in un vettore $\mathbb{V}$-dimensionale per ogni elemento della sequenza che compone il dominio. Lo step finale consiste di un dense layer con distribuzione temporale che agisce come regressore multinomiale. A causa dell'attivazione *softmax* attuata sul *dense layer*, l'output

del decoder rappresenta una distribuzione multinomiale dei caratteri di $\mathbb{V}$ per ogni step temporale, la quale può essere campionata per produrre un nuovo nome di dominio contenente le caratteristiche principali dei nomi di dominio usati in input.

In figura 3.8 è mostrata la struttura di massima dell'autoencoder. Di seguito vengono illustrati in dettaglio le principali componenti che compongono l'*autoencoder*.

3.5 Encoder

La composizione interna dell'*encoder* è formata da una Rete Convoluzionale accoppiata ad una LSTM. A differenza del lavoro proposto in [4], si è voluto mantenere la composizione dell'autoencoder il più semplice possibile, in quanto la trasformazione in *GAN* ed il suo *tuning* in fase di *training* è notoriamente difficoltoso in presenza di molti parametri. In Figura 3.9 viene mostrata la struttura semplificata dell'encoder.

3.5.1 Rete Convoluzionale

Una Rete Convoluzionale è un modello di rete neurale usato generalmente per classificazione di immagini, in alternativa ai layer densamente connessi. Il vantaggio nell'uso delle reti convoluzionali è la capacità di quest'ultime di memorizzare pattern locali all'interno dello spazio di input mentre layer densamente connessi sono in grado di riconoscere solo pattern globali. Nel caso in esame si sono applicati due filtri convoluzionali in parallelo, con l'obiettivo di cogliere pattern locali all'interno dei domini, ovvero n-grammi significativi da poter replicare. Generalmente i filtri convoluzionali lavorano su di un tensore 3-dimensionale, chiamato *feature map*, avente due assi spaziali ("altezza" e "larghezza") ed un asse di profondità; nel caso di riconoscimento di immagini tali assi corrispondono alle dimensioni dell'immagine di input ed al numero dei canali colore. Nel caso in esame si sono trattati di input tridimensionali ad altezza 1, larghezza equivalente alla dimensione massima dei domini nel dataset e canale di dimensione 1. L'operazione di convoluzione estrae frammenti dalla *feature map* di input ed applica una trasformazione a tutti i frammenti, generando una *output feature map* la quale è ancora in forma di tensore 3D, di dimensione ridotta, contenente sull'asse della profondità i valori dei *filtri*. I filtri codificano determinati aspetti caratteristici dell'input, analizzando

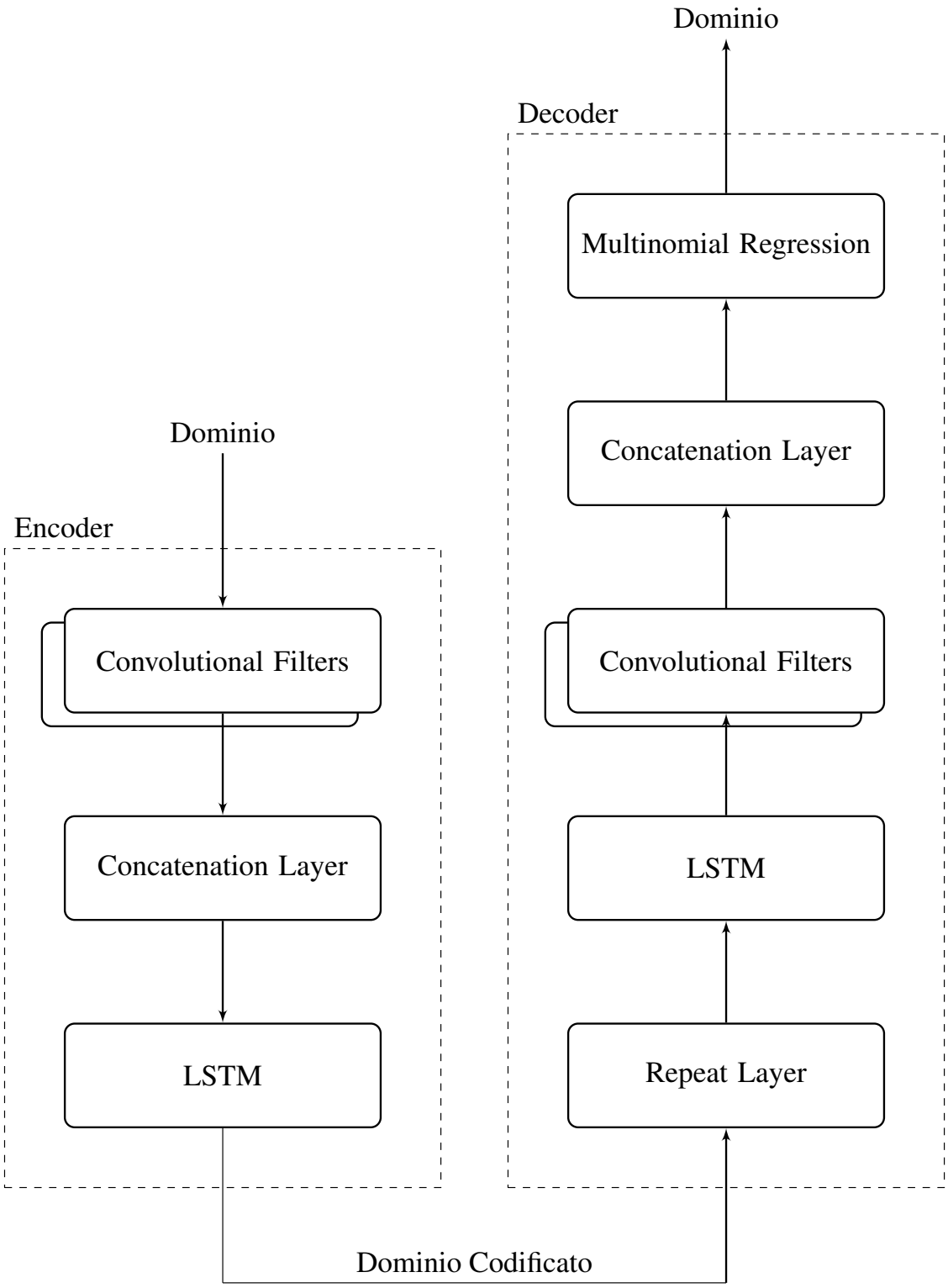

Figura 3.8: Struttura dell'*autoencoder* in esame. L'input in ingresso dato dai domini viene codificato attraverso l'*encoder* e dato in ingresso al *decoder* che ne genera una approssimazione.

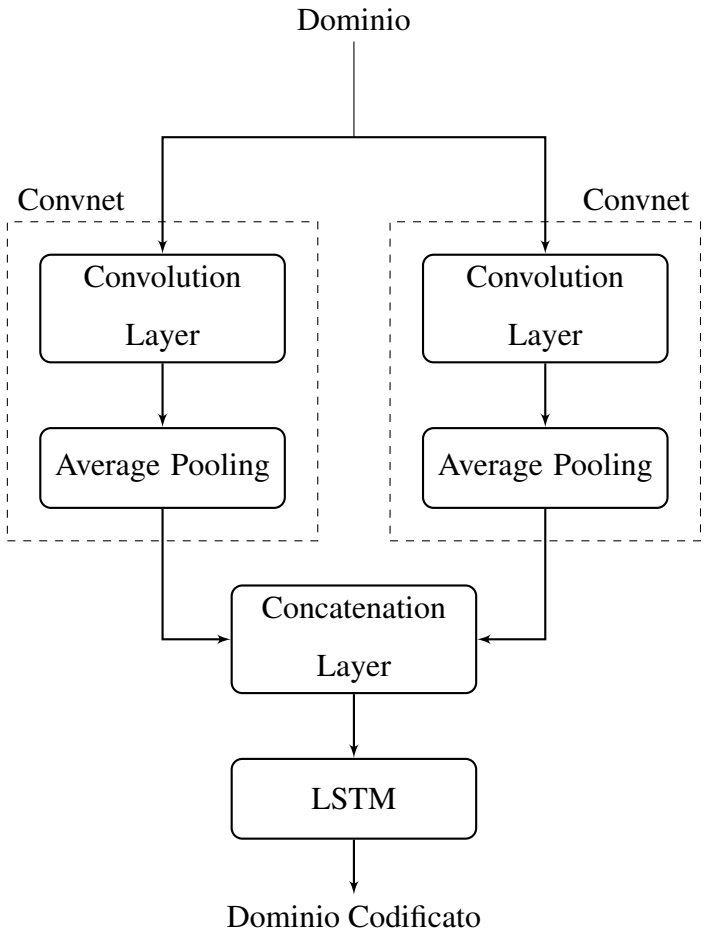

Figura 3.9: Struttura dell'*encoder*.

l'input in una "finestra" di dimensione fissata che scorre lungo la sequenza; ad ogni passo il filtro estrae il sotto-tensore 3-dimensionale per trasformarlo(tramite un prodotto tensore con una matrice di pesi (chiamato *convolutional kernel*) in un vettore 1D di dimensione fissata. L'insieme di vettori vengono riassemblati in un tensore 3D con altezza e larghezza identiche alle precedenti e l'inseme di features come terzo asse. In figura 3.10 è possibile vedere un diagramma del funzionamento di una rete convoluzionale.

3.5.2 Pooling

Parte integrante di una rete convoluzionale è il livello di *pooling*, in cui le features più importanti del livello precedente vengono ridotte in un tensore di dimensione inferiore, secondo il valor medio all'interno della *feature map*. La decisione di utilizzare average pooling anziche

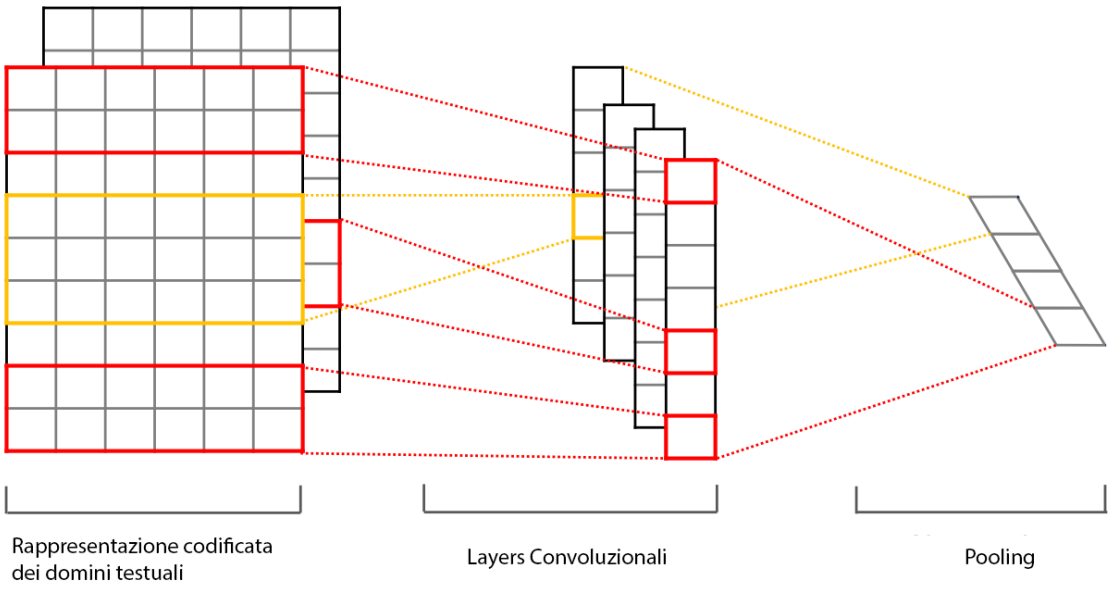

Figura 3.10: Funzionamento di una Convolutional Neural Network, *fonte* [21].

max pooling è stata presa a causa dell'elevata instabilità intrinseca alle GAN, per le quali max pooling è un fattore contribuente.

3.5.3 Concatenazione

Il layer aggiuntivo di *concatenazione* funge permete di formare un unico vettore monodimensionale in grado di fornire le caratteristiche principali dei domini analizzati.

3.5.4 Long Short-term Memory

Seconda fase della sottorete Encoder è la presenza di una Long Short-term Memory Network *(abbr LSTM)* [22]. Si tratta di un modello di *Recurrent Neral Network (abbr. RNN)* particolare, in grado di apprendere dipendenze a lungo termine all'interno di una sequenza temporale che nasce con l'intento di superare le principale problematiche delle RNN semplici, le quali non sono in grado di gestire dipendenze a lungo termine all'interno di una sequenza temporale. Le celle LSTM consistono di uno stato che può essere letto, scritto e resettato attraverso una serie di *gates*. Siano dati W e U *layer* di una cella LSTM corrispondenti a matrici di pesi per l'input x ed emissione h, mentre b vettore di bias. Lo stato c di una cella LSTM ha

connessioni periodiche che permettono ad ogni cella di mantenere stato attraverso gli step temporali:

$$c_t = f \cdot c_{t-1} + i_t \cdot g_t$$

dove $\cdot$ denota moltiplicazione tra elementi. Gli stati possono essere aggiornati in maniera additiva tramite

$$g_t = tanh(W^g x_t + U^g h_{t-1} + b^g)$$

attraverso i gate di inpu i, in grado di moltiplicare l'aggiornamento di stato di un numero che varia da 0 ad 1. Alla stessa maniera il *forget gate* f modula la connessione *self-recurrent* tra ogni cella di un numero compreso tra 0 ed 1. In tal maniera è possibile ignorare l'input e mantenere lo stato, oppure sovrascrivere lo stato corrente o resettarlo a 0. L'output gate o modula il contributo fornito dallo stato di ogni cella come

$$h_t = o_t \cdot tanh(c_t),$$

il quale è propagato agli input gate dei livelli successivi. In particolare i gate di input, forget e output sono definite da una funzione dell'input x_t e dall'emissione del layer LSTM precedente h_t all'istante t come:

$$\begin{aligned} i_t &= \sigma\left(W^i x_t + U^i h_{t-1} + b^i\right) \\ f_t &= \sigma\left(W^f x_t + U^f h_{t-1} + b^f\right) \\ o_t &= \sigma\left(W^o x_t + U^o h_{t-1} + b^o\right). \end{aligned}$$

Il design delle celle LSTM con gate moltiplicativi permette ad una rete neurale di immagazzinare ed accedere allo stato attraverso lunghe sequenze, mitigando le problematiche presenti all'interno delle *RNN* semplici. Nel contesto di questo progetto, lo spazio degli stati è inteso a catturare le combinazioni di tokens (n-grammi) che sono importanti per modellare nomi di dominio realistici. In figura 3.11 è indicata la struttura di una cella LSTM.

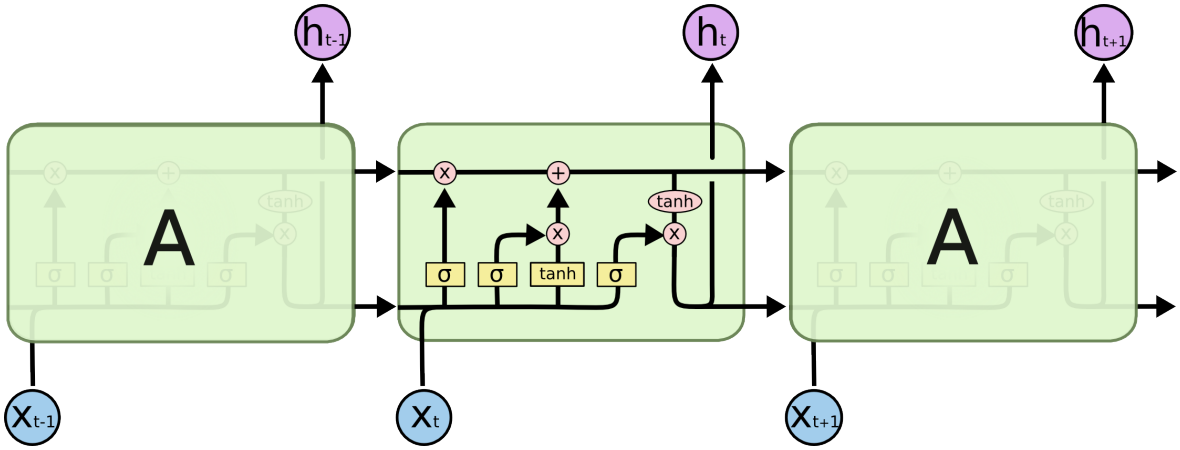

Figura 3.11: Struttura di una cella LSTM. *fonte:* [23].

3.5.5 Output Encoder

L'output in uscita da questo encoder è in forma di un tensore monodimensionale, di dimensione fissata, contenente le caratteristiche principali che compongono un nome di dominio.

3.6 Decoder

La struttura del decoder è lascamente una copia speculare dell'encoder. L'obiettivo principale è riuscire a ricampionare un dominio partendo da un tensore a domensione inferiore rispetto alla codifica fornita in input all'encoder.

La struttura di massima presenta come primo layer un *repeat vector* che ripete il tensore di input per un numero di volte pari alla lunghezza massima di dominio, fissata in fase di codifica del dataset, formando così un tensore 3-dimensionale, dello stessa dimensione dei domini provenienti dal dataset. La sequenza così formata viene data in input ad un layer LSTM avente le identiche caratteristiche del layer LSTM presente all'interno dell'encoder. Successivamente la sequenza di emissioni in output dalla LSTM viene fornita ad due reti convoluzionali parallele identiche al quelle inserite all'interno dell'encoder. il risultato di tale operazione è un vettore di dimensione pari a dizionario di caratteri ammissibili per i domini, per ogni step della sequenza di caratteri che compone un dominio. In Figura 3.12 viene mostrato lo schema di massima del *decoder*.

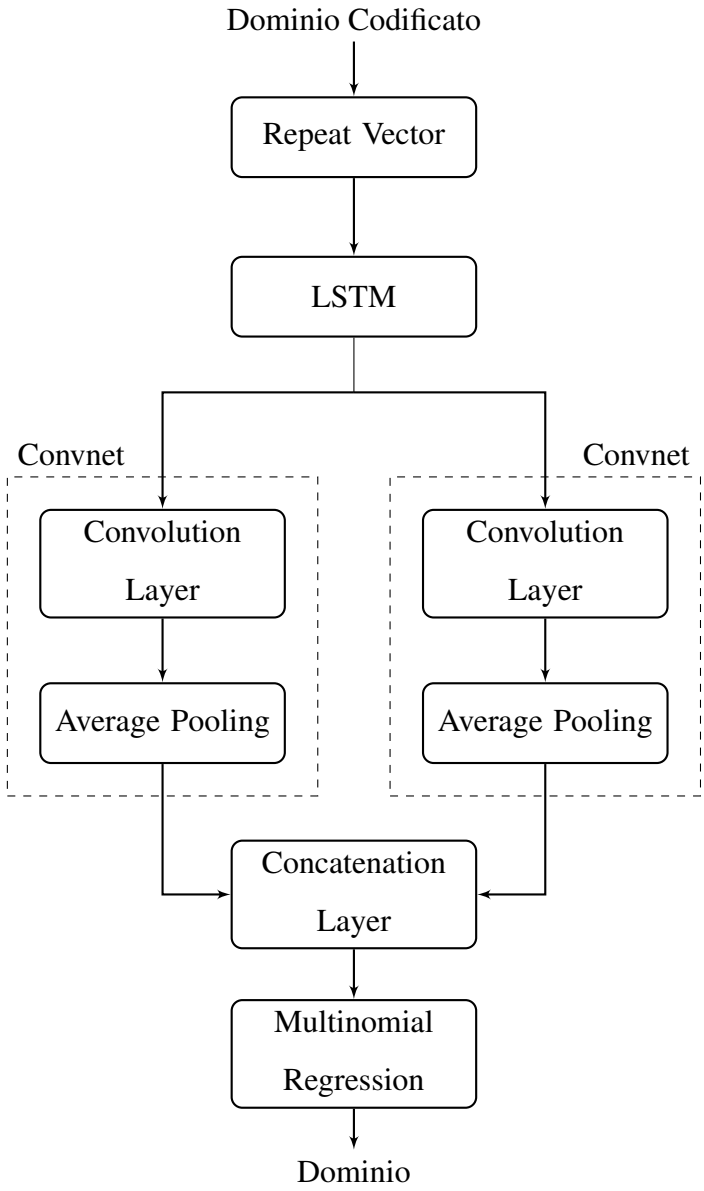

Figura 3.12: Struttura del *decoder*.

3.6.1 Regressione Multinomiale

Lo step finale del decoder è un livello *dense* di tipo *time-distributed* che agisce come *regressore multinomiale*, in grado di eseguire classificazione in diverse classi (in questo caso una classe per ogni carattere possibile nel dizionario). Il regressore multinomiale utilizza un tipo di attivazione *softmax*, definito dalla funzione:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad \text{per } j = 1, \ldots, K.$$

dove k è la dimensione del vettore di input $\mathbf{z}$ mentre in uscita si ottiene un vettore k-dimensionale $\sigma(\mathbf{z})$ di valori compresi in un intervallo $(0, 1)$. Grazie a tale attivazione sull'ultimo layer, l'output del decoder rappresenta una distribuzione multinomiale rispetto ai caratteri di ammissibili, per ogni step temporale che compone. Tale distribuzione può essere campionata in modo da ottenere di nuovo un vettore numerico di interi contenente i caratteri codificati all'interno del vocabolario di caratteri ammissibili per i domini.

Per il campionamento è stato usata una funzione *softmax* utilizzata nel campo dell'apprendimento per rinforzo [24], definita come:

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)},$$

Dove il valore di *"azione"* $q_t(a)$ corrisponde alla "ricompensa" ottenuta per l'azione a e τ è definito come parametro di *"temperatura"*. Per alte *"temperature"* ($\tau \to \infty$), tutte le azioni hanno la stessa probabilità e più bassa la temperatura, più la "ricompensa" influisce sulla probabilità. Per basse tempeature ($\tau \to 0^+$), la probabilità di *azione* con la ricompensa più alta tende ad 1.

Tramite questa funzione è stato possibile mettere a a punto in fase sperimentale il campionamento dei domini dalla lo rappresentazione in forma di tensore alla rappresentazione testuale.

3.7 Generative Adversarial Network

La *Generative Adversarial Network (abbr. GAN)* èun modello di rete neurale formata da due reti che competono l'una contro l'altra: un *generatore* che cerca di creare dati sintetici basati sulla vera distribuzione, con l'aggiunta di rumore in input ed un *discriminatore*, che riceve un campione e deve predire se appartiene al set reale o sintetico. Questo processo continua in forma di competizione tra le due reti, nella quale il discriminatore apprende in maniera sempre più accurata a distinguere i campioni ed il generatore apprende la costruzione di dati sintetici sempre più realistici. Notevole è la struttura della GAN, la quale rende particolarmente difficoltosa la fase di *training*: a differenza di reti neurali più semplici, non è possibile fissare un minimo da ottimizzare. Normalmente si attua una di discesa del gradiente, la quale

fornisce l'ottimo per la fase di training, mentre nel caso di GAN, ogni passo effettuato durante la discesa del gradiente causa un cambiamento nella conformazione superficie di discesa. La composizione della GAN fa si che sia necessario invece trovare un equilibrio tra le due forze in gioco, impersonate da discriminatore e generatore. In figura 3.13 si può vedere una struttura di massima di una GAN.

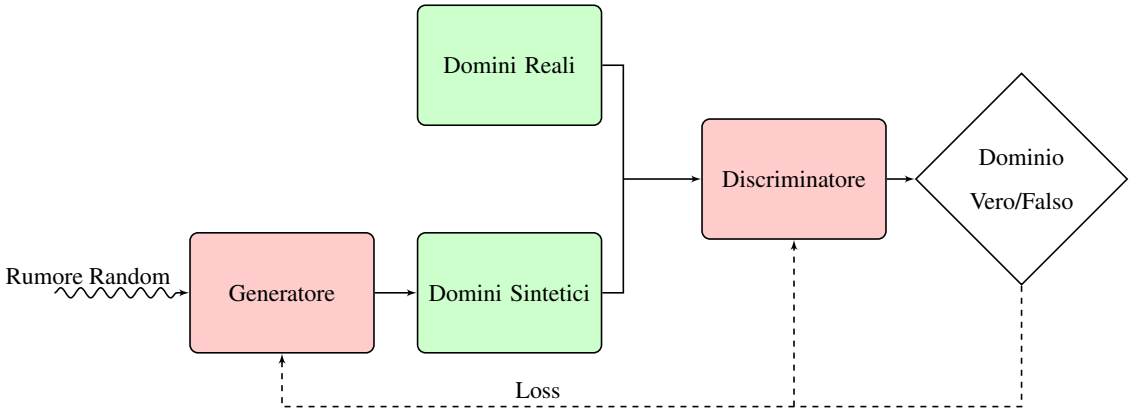

Figura 3.13: Struttura di una *Generative Adversarial Network*.

La struttura dell'autoencoder descritto a partire dalla sezione 3.4 permette di produrre adeguatamente nomi di dominio che assomigliano a domini reali, ma in realtà sono generati pseudo-casualmente campionando le distribuzioni multinomiali in output all'autoencoder. Tuttavia l'uso di un autoencoder richiede la presenza di un ampio dataset di domini come input. Attraverso poche modifiche è stato possibile trasformare l'autoencoder in una *GAN* che accetta in input un seme randomico ed emette nomi di dominio che appaiono simili ai domini reali.

3.7.1 Architettura GAN

Come mostrato in figura 3.14, la struttura della GAN in esame è composta fondamentalmente dalle componenti dell'autoencoder, estese in modo da trasformare l'encoder in un discriminatore che possa classificare domini sintetici da domini reali, ed il decoder in un generatore che a partire da uno spazio latente possa generare domini codificati.

Rispetto al lavoro presentato in [4], si è deciso di implementare una architettura più semplificata. Come mostrato in [25] le GAN sono tipicamente difficoltose da allenare, in quanto l'uso di funzioni di costo porta ad un difficile *tuning* per ottenere un equilibrio di Nash tra le

due parti in gioco. La decisione finale è stata presa dopo una serie di test sperimentali, in cui si sono messe a confronto architetture similari comprendenti diverse conformazioni di layers atte ad rendere stabile la fase di training, tra le quali Batch Normalization, come mostrato in [26], l'utilizzo di Leaky Relu [27] come layer di attivazione all'interno della sottorete Convoluzionale e Dropout come strumento per evitare il fenomeno di *overfitting* [28]. L'uso di Leaky ReLU come funzione di attivazione è giustificata dalla riduzione della sparsità del gradiente, la quale generalmente causa instabilità.

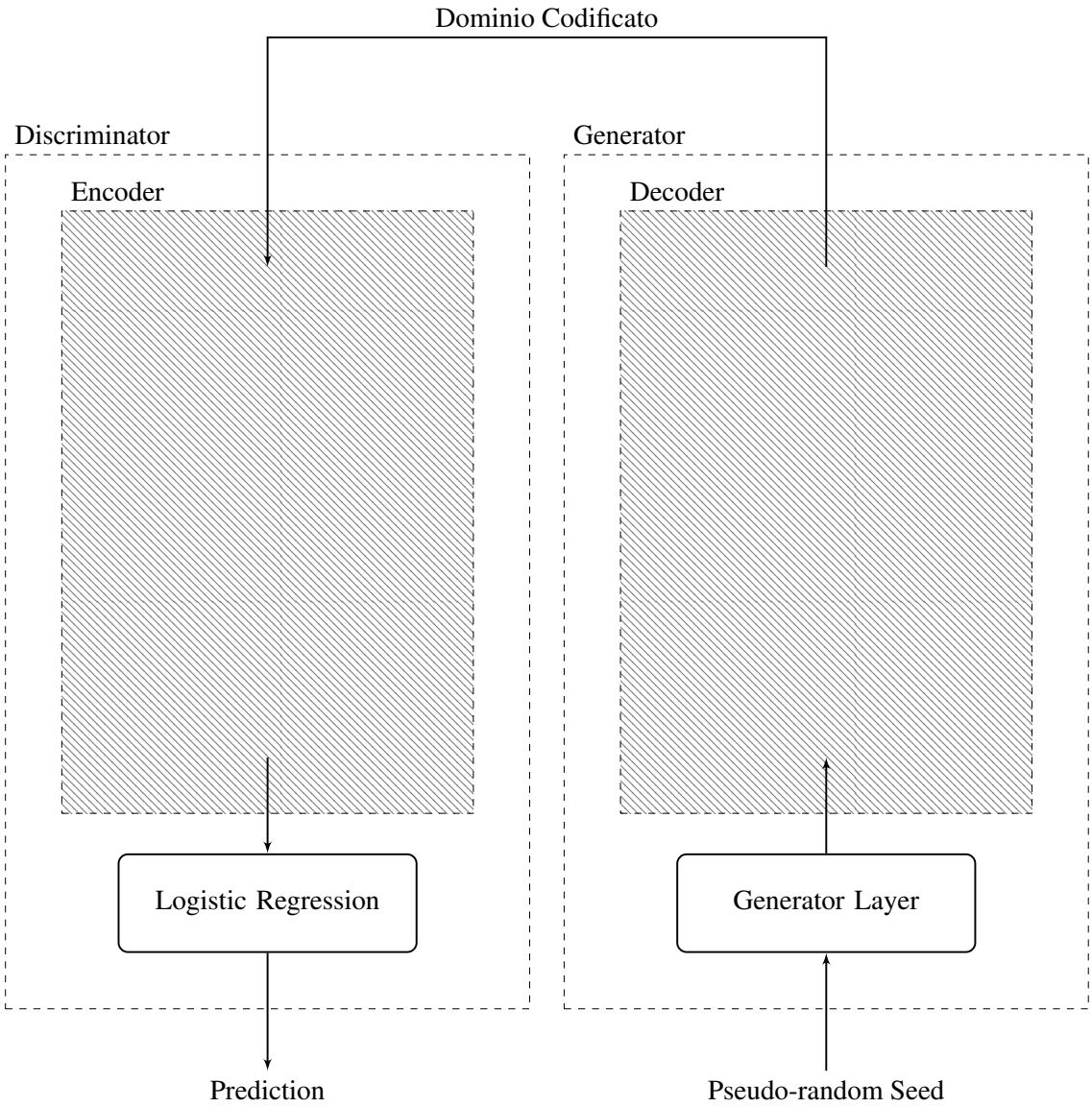

Figura 3.14: Struttura della GAN evoluta dell'autoencoder.

3.7.1.1 Batch Normalization

L'utilizzo di Batch Normalization è stato introdotto per contrastare la nota difficoltà nel training delle GAN. Generalmente il training delle reti neurali è reso difficoltoso dal fatto che la distribuzione dell'input di ogni layer cambia durante il training così come cambiano i parametri dei precedenti layer. Questo fenomeno rallenta il training, richiedendo *learning rates* più bassi ed una attenta scelta degli iperparametri iniziali. Tale fenomeno viene definito *covariate shift* da [26]. Tramite l'uso di *batch normalization* è possibile fare fronte a tali problematiche, le quali vengono amplificate grandemente dalla struttura avversaria delle due reti Generatore e Discriminatore che compongono la *GAN*.

3.7.2 Generatore

Il generatore aggiunge in testa alla stuttura del decoder un layer denso che mappa un input randomico in una codifica di dominio, come mostrato in figura 3.15. All'interno della sottorete convoluzionale è presente l'aggiunta di Batch Normalization, Leaky Relu al posto di ReLU per l'attivazione dei livelli convoluzionali.

Il Generator laye è formato da un generatore di rumore randomico di distribuzione gaussiana, la quale è stata provata come ottimale nel caso di GAN da [29]. Tale distribuzione, di dimensione fissata, mima un possibile dominio codificato da encoder come descritto in sezione 3.5. A partire da tale rumore è possibile per il decoder realizzare una codifica di dominio via via più realistica con l'avanzare delle epoche di training della GAN.

3.7.3 Discriminatore

Il discriminatore implementa in uscita un layer denso che esegua regressione logistica sui domini codificati dalla sottorete encoder. (Figura 3.16). Tale regressione viene attuata tramite un layer denso che effettua classificazione tra domini reali, codificati durante la fase di encoding attuata in 3.5. La funzione di attivazione utilizzata all'interno del regressore logistico è la funzione sigmoide come descritto all'interno della sezione 3.2.2. Con il passare delle epoche di training, il discriminatore è sempre più in grado di distinguere domini reali da domini sintetici, forniti dal generatore. Tramite un insieme di tecniche sperimentali, è

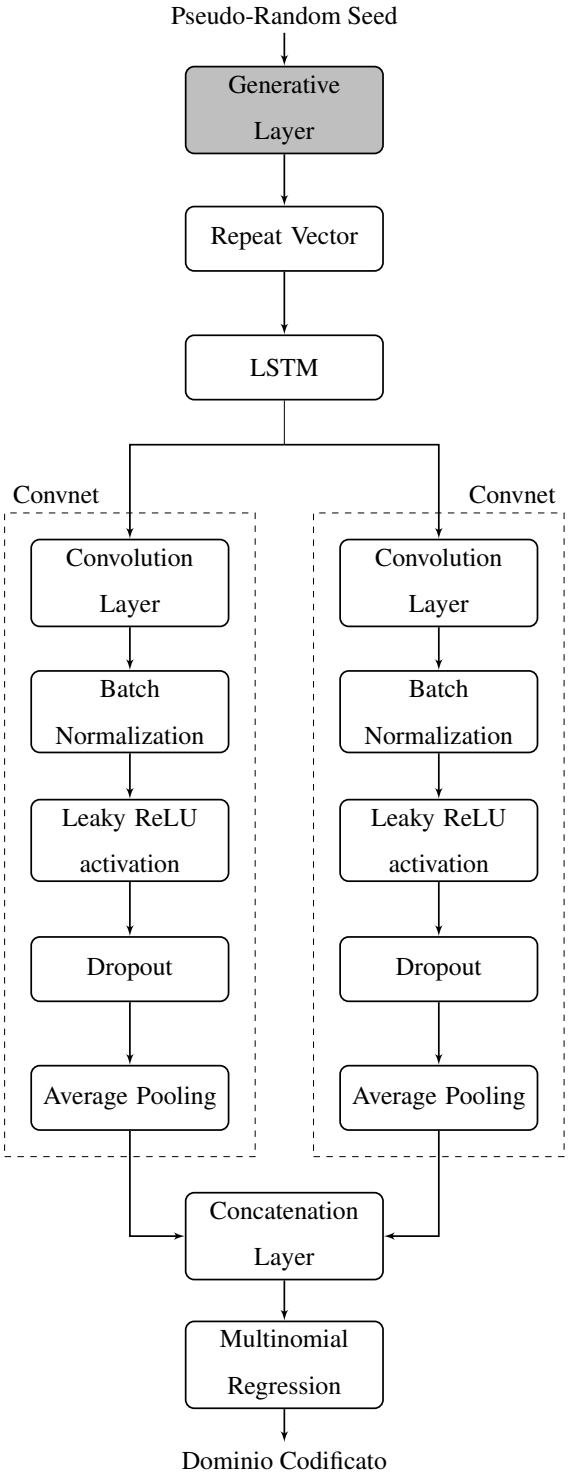

Figura 3.15: Struttura del generatore.

possibile evitare che il discriminatore prevalga sul generatore durante il training, mantenendo un equilibrio tra le due parti.

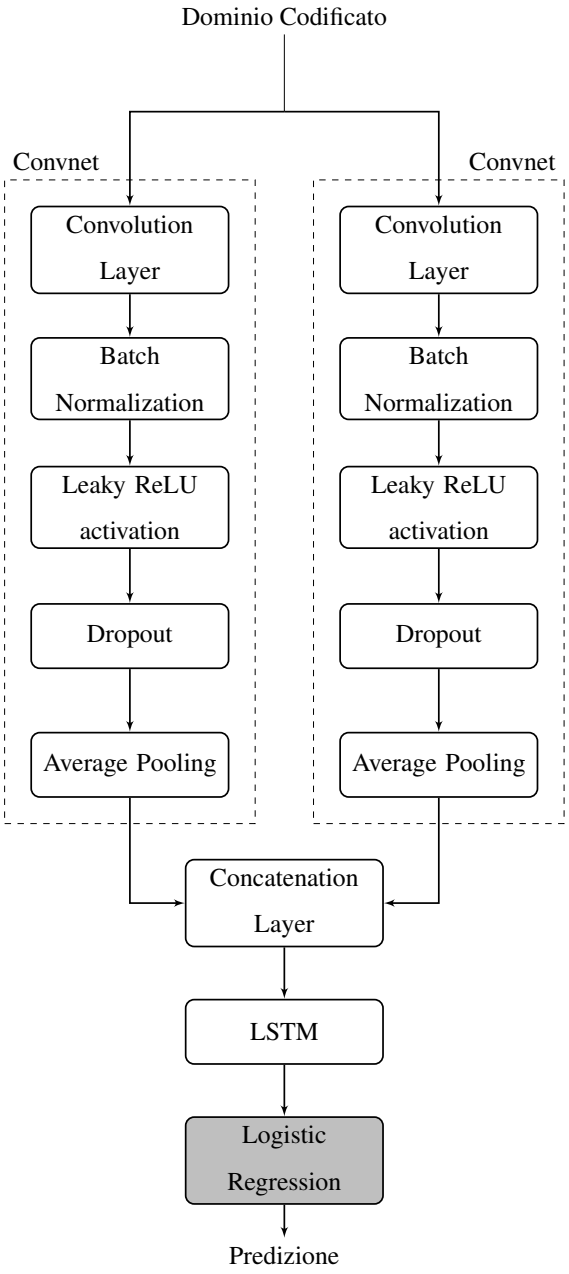

Figura 3.16: Struttura del discriminatore.

3.7.4 Funzionamento

Durante la fase di training della GAN, il discriminatore ed il generatore vengono allenati in maniera concorrente. Per ogni step di training si estrae un subset fissato di domini dal dataset di domini reali e si genera un tensore di rumore generato randomicamente di lunghezza

equivalente. Tale tensore viene dato in ingresso al generatore che restituisce una serie di domini sintetici codificati.

Successivamente il discriminatore viene allenato a distinguere i due subset estratti precedentemente, grazie ad un vettore di *target* creato ad hoc, che identifica domini reali dai sintetici.

In ultima fase, i pesi del discriminatore vengono congelati e l'intero impianto della GAN viene allenato fornendo un nuovo tensore di rumore randomico ed un vettore di target ingannevoli, i quali identificano tutti i domini generati sinteticamente come reali. L'output di tale procedura è la funzione di loss del discriminatore, il quale non aggiorna la sua matrice di pesi durante questa fase. Tale funzione di loss indica quindi quanto "reali" sono stati i domini creati dal generatore.

Tramite questo espediente è possibile far si che la fase di training consista in fasi in cui il discriminatore impara a distinguere con più precisione domini reali da domini sintetici ed il generatore impari a fornire domini sintetici via via più realistici. Questa fase procede in maniera indefinita, fin tanto che l'equilibrio tra le due parti rimane tale.

Capitolo 4

Implementazione

In questa sezione è descritta l'implementazione del progetto definito nel capitolo precedente. I principali strumenti utilizzati sono Python come linguaggio di programmazione e le librerie Scikit-learn [30] per attuare machine learning e Keras [31] per realizzare reti neurali. La libreria per reti neurali su cui opera Keras è Tensorflow [32]. Di seguito sono elencati i dettagli implementativi delle singole entità, ed i parametri usati durante la fase sperimentale.

4.1 Classificatore Random Forest

Il classificatore Random Forest è stato implementato tramite l'uso della libreria python Scikit-learn [30]. Al fine di avere un ambiente di testing facilmente fruibile, è stata creata una classe $MyClassifier$ in grado di gestire la serie di esperimenti effettuata sul classificatore. Tali esperimenti hanno permesso di testare i differenti parametri di funzionamento e la composizione delle features che compongo il dataset del classificatore. In figura 4.1 è mostrata la struttura della classe $MyClassifier$, la quale contiene le funzioni principali di inizializzazione, salvataggio e caricamento dei parametri, salvataggio e caricamento dei risultati oltre che alle funzioni necessarie per allenare il classificatore (funzione fit), eseguire cross-validation (funzione $cross\_validate$), produrre grafici di risultati ($classification\_report$ e $plot_AUC$) ed eseguire predizione su di una lista di domini in input (a fini di testing).

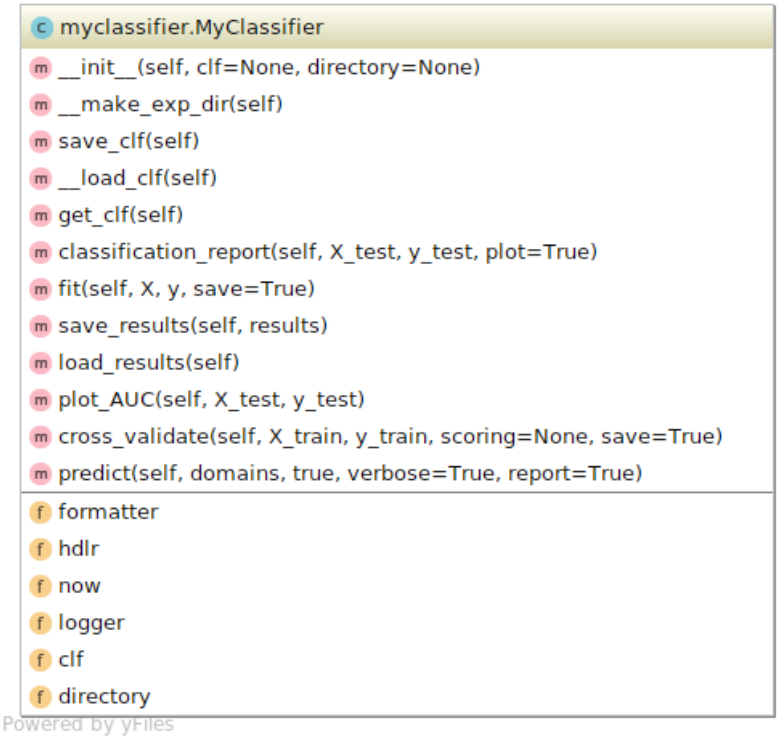

Figura 4.1: Diagramma UML della classe MyClassifier

4.1.1 Dataset

Il dataset utilizzato per le fasi di training e testing del classificatore è stato definito nella sezione 3.1.1. A livello operativo il dataset è stato generato a partire dai soli nomi di dominio in formato stringa, per i quali sono stati estratti le singole features. Segue una descrizione dettagliata delle features utilizzate per generare il dataset.

- **MCRExtractor** si occupa di fornire il rapporto tra caratteri significativi. È caratterizzato dal parametro che identifica la minima sottostringa da analizzare, composta da 3 caratteri. L'algoritmo cicla per ogni dimensione di n-gramma ≥ 3 , estrae le tuple di dimensione i dalla stringa e per ogni tupla s presente in una $word$ del dizionario inglese, viene aggiunta la sua lunghezza alla somma dei caratteri significativi della stringa. Il risultato è il rapporto tra la somma della lunghezza dei caratteri significativi e la lunghezza totale della stringa.

```
1 min_subtr = 3
2 maxl = 0
3 for i in range(min_subtr, len(str(domain_name))):
```

```
    tuples = zip(*[str(domain_name)[j::i] for j in range(i)])
    split = [''.join(t) for t in tuples]
    tmpsum = 0
    tmps = []
    for s in split:
      for words in eng_dict:
        if s in words:
          tmpsum += len(s)
          tmps.append(s)

    if tmpsum > maxl:
      maxl = tmpsum
return maxl / int(len(str(domain_name)))
```

- **NormalityScoreExtractor** estrae il punteggio di normalità degli n-grammi. A differenza del lavoro propsoto in [8], si è scelto di estrarre il *normality score* per gli n-grammi di lunghezza 1,2,3,4,5 caratteri, in quanto in fase sperimentale hanno dimostrato di fornire un apporto significativo alla precisione del classificatore. È definito dal seguente algoritmo, dove il valore $self.n$ identifica la lunghezza dell'n-gramma da valutare. L'algoritmo separa in n-grammi di dimensione n la stringa e per ogni tupla di caratteri s, somma il numero di occorrenze di tale tupla all'interno del vocabolario inglese, ritornando il rapporto tra il valore complessivo delle occorenze delle tuple nel vocabolario inglese e la lunghezza della stringa meno $n + 1$.

```
if len(str(domain_name)) < self.n:
            return 0
tuples = ngrams(str(domain_name), self.n)
myngrams = (''.join(t) for t in tuples)
scoresum = 0
for s in myngrams:
  counter = 0
    for words in eng_dict:
      if s in words:
          counter += 1
    scoresum += counter
```

```
return scoresum / (len(str(domain_name)) - self.n + 1)
```

- **NumCharRatio** estrae il rapporto di caratteri numerici rispetto alla lunghezza della stringa

```
counter = Counter(domain_name)
ncr = 0
for key, value in counter.iteritems():
  if key.isdigit():
    ncr += value

return ncr / len(domain_name)
```

- **DomainNameLength** estrae la lunghezza del nome di dominio.

```
return (len(domain_name))
```

- **VowelConsonantRatio** estrae il rapporto tra vocali e consonanti all'interno della stringa.

```
count = 0
vowels = set("aeiou")
for letter in domain_name:
  if letter in vowels:
    count += 1
return count / (len(domain_name) - count)
```

L'insieme di tali features è stato usato per generare un dataset contenente per ogni nome di dominio le 9 features linguistiche generate a partire dalle stringhe dei nomi di dominio.

Assieme a tale dataset è stato generato un vettore binario "*target*" da cui il classificatore può verificare la categoria di appartenenza (malevolo o reale) di ogni dominio.

4.1.2 Ambiente di training

Durante la fase di training il dataset è stato mescolato e separato in due subset: *train* e *test*. Il subset di testing ha la dimensione pari al 10% dell'intero dataset.

La fase di training è stata eseguita tramite la tecnica di convalida incrociata (*k-fold cross validation*) in cui il dataset di training è stato separato in 10 subset *(fold)* e ad ogni passo la *k-esima* parte è stata usata come validazione per la fase di training. Questa tecnica evita il problema dell'overfitting, di cui generalmente soffrono i modelli di machine learning.

Il restante subset di test, mai applicato nella fase di training, è stato utilizzato per valutare il modello del classificatore.

4.1.3 Parametri Classificatore

In particolare i parametri del classificatore utilizzato che sono stati modificati dal loro valore di default sono:

- **`n_estimators`** = 100. Numero di stimatori da utilizzare nel processo. Il valore è stato deciso come ottimale dopo una serie di test sperimentali preliminari.
- **`min_samples_leaf`** = 50. Numero minimo di campioni per un nodo foglia. il valore è stato deciso come ottimale dopo una serie di test sperimentali preliminari.
- **`oob_score`** = True. Implica l'uso di campioni *out-of-bag* per ottenere una stima facilitata dell'errore. Con questa tecnica una parte del campione di stima viene esclusa per essere utilizzata come insieme di verifica.

4.2 Classificatore Neurale

L'implementazione del classificatore neurale è stata eseguita in ambiente Python, con l'utilizzo della libreria Keras [31] per le reti neurali. Per motivi di semplicità si è scelto di copiare l'ambiente di testing definito per il classificatore Random Forest mostrato in figura 4.1. Tale classe permette di gestire i diversi esperimenti, utilizzati per decidere quale composizione del *Multi Layer Perceptron* (abbr. MLP) in base ai risultati sperimentali. Il punto saliente

del classificatore è formato dalla sua baseline, la quale contiene i livelli che effettivamente compongono la rete neurale.

4.2.1 Baseline

Come definito all'interno della sezione 3.2.2, la composizione del MLP è stata decisa dopo un confronto tra tre modelli differenti nella conformazione:

- un primo modello ridotto contenente un input a dimensione 15 (fissata inizialmente come dimensione dei domini codificati), un livello intermedio di dimensione dimezzata rispetto all'input ed un livello finale con un singolo neurone per ottenere classificazione binaria (listato 4.1).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
dense_1 (Dense)              (None, 15)                225
_________________________________________________________________
batch_normalization_1 (Batch (None, 15)                60
_________________________________________________________________
dropout_1 (Dropout)          (None, 15)                0
_________________________________________________________________
activation_1 (Activation)    (None, 15)                0
_________________________________________________________________
dense_2 (Dense)              (None, 7)                 105
_________________________________________________________________
batch_normalization_2 (Batch (None, 7)                 28
_________________________________________________________________
dropout_2 (Dropout)          (None, 7)                 0
_________________________________________________________________
activation_2 (Activation)    (None, 7)                 0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 7
_________________________________________________________________
batch_normalization_3 (Batch (None, 1)                 4
_________________________________________________________________
dropout_3 (Dropout)          (None, 1)                 0
```

```
25 _________________________________________________________________
26 activation_3 (Activation)     (None, 1)                 0
27 =================================================
28 Total params: 429
29 Trainable params: 383
30 Non-trainable params: 46
31 _________________________________________________________________
```

Listing 4.1: Baseline Ridotta

- un secondo modello ampliato contenente input equivalente al precedente, due livelli interni di dimensione 128 ed una uscita con singolo neurone. (listato 4.2).

```
1  _________________________________________________________________
2  Layer (type)                 Output Shape              Param #
3  =================================================
4  dense_1 (Dense)              (None, 15)                225
5  _________________________________________________________________
6  batch_normalization_1 (Batch (None, 15)                60
7  _________________________________________________________________
8  dropout_1 (Dropout)          (None, 15)                0
9  _________________________________________________________________
10 activation_1 (Activation)    (None, 15)                0
11 _________________________________________________________________
12 dense_2 (Dense)              (None, 128)               1920
13 _________________________________________________________________
14 batch_normalization_2 (Batch (None, 128)               512
15 _________________________________________________________________
16 dropout_2 (Dropout)          (None, 128)               0
17 _________________________________________________________________
18 activation_2 (Activation)    (None, 128)               0
19 _________________________________________________________________
20 dense_3 (Dense)              (None, 128)               16384
21 _________________________________________________________________
22 batch_normalization_3 (Batch (None, 128)               512
23 _________________________________________________________________
24 dropout_3 (Dropout)          (None, 128)               0
25 _________________________________________________________________
26 activation_3 (Activation)    (None, 128)               0
```

```
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 128
_________________________________________________________________
batch_normalization_4 (Batch (None, 1)                 4
_________________________________________________________________
dropout_4 (Dropout)          (None, 1)                 0
_________________________________________________________________
activation_4 (Activation)    (None, 1)                 0
=================================================
Total params: 19,745
Trainable params: 19,201
Non-trainable params: 544
_________________________________________________________________
```

Listing 4.2: Baseline ampliata

- un modello intermedio, contenente ingresso equivalente al precedente, due livelli interni da 128 e 64 e uscita con singolo neurone. (listato 4.3).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
dense_1 (Dense)              (None, 15)                240
_________________________________________________________________
dropout_1 (Dropout)          (None, 15)                0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 65
=================================================
Total params: 10,609
Trainable params: 10,609
Non-trainable params: 0
```

```
21 _________________________________________________________________
```

Listing 4.3: Baseline intermedia.

Tutti i modelli contengono al loro interno oltre che i dense layer, dei layer aggiuntivi per ottimizzare la fase di training, in particolare i layer Dropout e Batch Normalization per contrastare il fenomeno dell'*overfitting*. Dopo una fase sperimentale si è deciso di optare per il modello intermedio, in quanto ha espresso i risultati migliori a fronte di un ridotto consumo di risorse.

4.2.2 Dataset

Come definito all'interno della sezione 3.2.1 il dataset utilizzato per il classificatore neurale è differente dal caso precedente in quanto i domini vengono codificati in forma di vettori numerici, mappati rispetto ad un dizionario di caratteri ammissibili per i domini di rete (tabella 4.1).

Key	Value	Key	Value	Key	Value	Key	Value
1	a	11	k	21	u	31	4
2	b	12	l	22	v	32	5
3	c	13	m	23	w	33	6
4	d	14	n	24	x	34	7
5	e	15	o	25	y	35	8
6	f	16	p	26	z	36	9
7	g	17	q	27	0	37	-
8	h	18	r	28	1	38	.
9	i	19	s	29	2		
10	j	20	t	30	3		

Tabella 4.1: Dizionaro di caratteri mappati.

Tale mapping permette al MLP di estrarre features a partire dai caratteri che compongono i domini.

4.2.3 Ottimizzatore

I modelli testati in questa sezione sono stati compilati utilizzando l'ottimizzatore *Adaptive Moment Estimation (abbr. adam)* [33]. Tale ottimizzatore fornisce generalmente le prestazioni migliori nel caso di classificazione binaria e pertanto si è optati per tale soluzione.

4.2.4 Fase di Training

Durante la fase di training si sono testati gli iperparametri principali che pilotano la performance del classificatore: in particolare i valori della dimensione del tensore di ingresso batch_size ed il numero di epoche epochs sul quale eseguire la fase di training.

Dopo una fase sperimentale i valori che hanno mostrato i risultati migliori in fase di valutazione sono stati:

- **batch_size** = 35
- **epochs** = 60

4.3 Autoencoder

L'implementazione dell'autoencoder descritto in sezione 3.4 è stata eseguita in ambiente Python con l'ausilio della libreria Keras per reti neurali.

4.3.1 Dataset

Il dataset utilizzato per il training dell'autoencoder è stato definito nella sezione 3.4.1. Tale dataset risulta essere un'ulteriore elaborazione della mappatura utilizzata all'interno del classificatore neurale precedentemente descritto: i domini codificati in vettori numerici sono stati ulteriormente codificati con la tecnica one-hot, in modo da ottenere, per ogni dominio, un tensore di valori binari contenente un vettore per ogni *step temporale* che compone il dominio. tale vettore, contenente la lettera i-esima del dizionario è formato da zeri tranne l'elemento alla posizione i-esima, impostato ad 1.

Questa ulteriore codifica ha permesso successivamente di poter trasformare l'autoencoder in una GAN, inviando l'output del decoder come input dell'encoder senza effettuare operazioni di campionamento, fonte di rottura del gradiente del modello.

La differenza principale con il dataset usato precedentemente per il classificatore neurale è la totale mancanza di domini DGA e l'uso esclusivo di domini forniti dal database di Alexa [2]. Il fine infatti è ottenere un sistema che possa riprodurre al meglio i domini reali.

4.3.2 Encoder

La struttura dell'Encoder, definita all'interno della sezione 3.5 è stata implementata con l'ausilio delle API funzionali fornite da Keras. In questo modo è stato possibile implementare un modello più complesso del precedente classificatore, in grado di eseguire l'operazione convoluzionale con due diverse conformazioni in contemporanea rispetto allo stesso tensore di ingresso. Di seguito sono mostrati gli iperparametri che definiscono il modello Encoder.

```
dropout_value = 0.5
cnn_filters = [20, 10]
cnn_kernels = [2, 3]
cnn_strides = [1, 1]
leaky_relu_alpha = 0.2
timesteps = 15
word_index = 38
latent_vector = 20
```

Listing 4.4: Iperarametri Encoder.

È possibile notare quali parametri definiscono le due reti convoluzionali:

- Convnet formata da 20 filtri, kernel di dimensione 2 e passo 1.
- Convnet formata da 10 filtri, kernel di dimensione 3 e passo 1.

La bontà di tali valori è stata confermata da [4] e per tanto utilizzati in maniera identica. I valori timesteps, word_index, latent_vector definiscono le dimensioni dei tensori di ingresso ed uscita, così come definiti dalle dimensioni del dataset. La dimensione di latent_vector è arbitraria: in questo caso si è optato per ridurre ulteriormente la dimensione del tensore di

uscita dalle due reti convoluzionali per estrarre le features più caratterizzanti dalla fase di encoding.

I valori dropout_value e leaky_relu_alpha sono il risultato di una lunga fase di *tuning* degli iperparametri, particolarmente critici per il mantenimento dell'equilibrio della successiva GAN.

Per completezza di seguito è elencato il codice essenziale che compone l'encoder.

```

discr_inputs = Input(shape=(timesteps, word_index),
           name="Discriminator_Input")
for i in range(2):
  conv = Conv1D(cnn_filters[i],
           cnn_kernels[i],
           padding='same',
           strides=cnn_strides[i],
           name='discr_conv%s' % i)(discr_inputs)
  conv = BatchNormalization()(conv)
  conv = LeakyReLU(alpha=leaky_relu_alpha)(conv)
  conv = Dropout(dropout_value, name='discr_dropout%s' % i)(conv)
  conv = AveragePooling1D()(conv)
  enc_convs.append(conv)

discr = concatenate(enc_convs)
discr = LSTM(latent_vector)(discr)

E = Model(inputs=discr_inputs, outputs=discr, name='Encoder')
```

Listing 4.5: Struttura livelli Encoder.

4.3.3 Decoder

La composizione del decoder è pressochè identica a quella mostrata dall'encoder e ricalca il progetto mostrato in sezione 3.6. Di seguito sono elencati gli iperparametri che compongono il modello decoder. Si può notare come siano identici a quelli mostrati dall'encoder a differenza del dropout_value, il quale gioca un ruolo maggiore nel pilotare l'equilibrio tra i due modelli.

```
dropout_value = 0.4
cnn_filters = [20, 10]
cnn_kernels = [2, 3]
cnn_strides = [1, 1]
dec_convs = []
leaky_relu_alpha = 0.2
latent_vector = 20
timesteps = 15
word_index = 38
```

Listing 4.6: Iperarametri Decoder.

Per completezza si riporta l'implementazione del decoder:

```

dec_inputs = Input(shape=(latent_vector,),
           name="Generator_Input")
decoded = RepeatVector(timesteps, name="gen_repeate_vec")(dec_inputs)
decoded = LSTM(word_index, return_sequences=True,
    name="gen_LSTM")(decoded)
decoded = Dropout(dropout_value)(decoded)
for i in range(2):
  conv = Conv1D(cnn_filters[i],
           cnn_kernels[i],
           padding='same',
           strides=cnn_strides[i],
           name='gen_conv%s' % i)(decoded)
  conv = LeakyReLU(alpha=leaky_relu_alpha)(conv)
  conv = Dropout(dropout_value, name="gen_dropout%s" % i)(conv)
  dec_convs.append(conv)

decoded = concatenate(dec_convs)
decoded = TimeDistributed(Dense(word_index, activation='softmax'),
    name='decoder_end')(decoded)

D = Model(inputs=dec_inputs, outputs=decoded, name='Decoder')
```

Listing 4.7: Struttura livelli Decoder.

4.3.4 Training

Durante la fase di training si è realizzato il modello generale che compone l'autoencoder, sottoforma di modello sequenziale Keras keras.models.Sequential. Tale modello è composto da due livelli: Encoder e Decoder Il modello è compilato con l'uso dell'ottimizzatore *adam* e *Categorical Cross Entropy* come funzione di *loss*. Tale scelta è dovuta alla natura dell'autoencoder, al quale viene dato come target per il training lo stesso dataset da input al fine di mimare il più correttamente possibile l'input iniziale passando per una codifica a minore dimensione.

```
1  Layer (type)                 Output Shape              Param #
2  =================================================
3  Encoder (Model)         (None, 20)                6890
4  _________________________________________________________________
5  Decoder (Model)             (None, 15, 38)            12836
6  =================================================
7  Total params: 19,726
8  Trainable params: 19,666
9  Non-trainable params: 60
10 _________________________________________________________________
```

Listing 4.8: Composizione di massima dell'autoencoder

I valori di batch_size ed epochs sono stati definiti tali dopo una breve fase sperimentale:

- **batch_size** = 128
- **epochs** = 200

Durante la fase di training l'indicatore principale di performance è il valore di *loss*, il quale al suo decrescere indica una migliore capacità di riproduzione dei domini.

4.4 Generative Adversarial Network

L'implementazione della Generative Adversarial Network (sezione 3.7, è stata evoluta a partire da quella eseguita per l'autoencoder.

4.4.1 Dataset

La composizione del dataset di domini è identica a quella fornita per l'autoencoder. Durante la fase di training si alternano mini-batch provenienti da tale dataset e mini-batch create sinteticamente dal generatore a partire da un vettore randomico di valori compresi tra $[-1.0, 1.0]$.

4.4.2 Discriminatore

Il modello del discriminatore è derivato dal modello definito dall'encoder: in coda all'encoder si è aggiunto un layer denso in grado di operare classificazione binaria, definito da:

```
Dense(1, activation='sigmoid', kernel_initializer='normal')
```

Questo layer aggiuntivo trasforma la funzione dell'encoder in un classificatore in grado di discriminare tra domini reali e domini generati sinteticamente.

4.4.3 Generatore

La struttura del generatore è identica alla struttura definita per il decoder. La differenza principale sta nel tipo di input dato in ingresso: Nel caso decoder l'input proviene dall'uscita dell'encoder, mentre nel caso generatore l'input è fornito in forma di vettore randomico creato con l'ausilio della libreria *numpy* [34] np.random.normal con valori compresi tra $[-1.0, 1.0]$.

4.4.4 Training

La fase di training comporta più fasi, che coinvolgono separatamente i sottosistemi della GAN.

In prima fase, come indicato da [4] si è scelto di eseguire un pre-training tramite l'autoencoder. Si è eseguita una fase di training efficace su tale sistema, copiando successivamente i pesi aggiornati all'interno della GAN, al fine di fornire una base di partenza stabile per entrambi i sottosistemi Encoder/Discriminatore e Decoder/Generatore.

Successivamente a tale pre-training il dataset viene caricato in memoria, i modelli di Discriminatore e Generatore sono compilati con i seguenti valori:

- **Discriminatore**: utilizza l'ottimizzatore RMSprop [35] con tali valori:
  - learning rate $= 0.01$
  - clipvalue $= 1.0$
  - decay $= 10^{-8}$
- **Generatore**: utilizza l'ottimizzatore Adam con tali valori:
  - learning rate $= 0.0001$
  - beta_1 $= 0.9$
  - beta_2 $= 0.999$
  - $\epsilon = 10^{-8}$
  - decay $= 10^{-8}$
  - clipvalue $= 1.0$

Tali valori sono il frutto di una fase sperimentale molto lunga, durante la quale si è cercato di calibrare attentamente i valori di learning rate di entrambi gli ottimizzatori al fine di ottenere un equilibrio tra i due sottosistemi della GAN: ha infatti dimostrato una forte propensione a degenerare verso un sistema o l'altro, fornendo valori di loss estremi. Si è cercata infatti una impostazione tale che mantenesse valori di loss più o meno stabili durante la durata della fase di training.

Entrambi i sottosistemi sono stati compilati con l'uso di *binary crossentropy* come funzione di loss, in quanto l'output di entrambi è rappresentato dal layer denso, in uscita dal discriminatore.

Nella fase successiva il dataset è stato suddiviso secondo la dimensione impostata dal valore di batch_size, in modo da poter garantire una passata completa a tutto il dataset per ogni epoca di training. Per ogni epoca e per ogni minibatch del dataset si è proceduto con il seguente metodo:

- Si genera un vettore di valori randomici, di dimensione identica all'input accettato dal Generatore.
- A partire da tale vettore si sono generati domini sintetici.
- In maniera alternata, per ogni ciclo di mini-batch, si è fornito come training al discriminatore un mini-batch formato unicamente da domini reali provenienti dal dataset oppure un mini-batch di domini sintetici prodotti dal generatore. Entrambi gli input sono accompagnati da un vettore di target che indica al discriminatore la natura di tali domini (reali o sintetici).
- Si congelano i pesi del discriminatore. Tale procedura permette al generatore di procedere alla sua fase di training, utilizzando il discriminatore come mezzo per ottenere un indice di qualità dei domini generati sinteticamente.
- Si genera un nuovo vettore randomico.
- Si procede al training del generatore, utilizzando l'intera GAN come modello. In ingresso si forniscono il vettore randomico ed un vettore di target ingannevole, il quale indica che i valori in uscita forniti sono reali. Questo metodo fa si che il generatore si spinga via via verso la creazione di domini via via più simili a quelli reali. L'output del generatore viene dato in ingresso al livello successivo, composto dal discriminatore che in questo caso non può aggiornare i propri pesi ma solo fornire una predizione sulla natura dei dati di ingresso.
- Si riattiva l'aggiornamento dei pesi per il discriminatore, per il prossimo ciclo di training.

Tale procedura è ripetuta per tutti i mini-batch e per il numero di epoche di training impostato.

Capitolo 5

Risultati

In questo capitolo si presentano i risultati della fase sperimentale effettuata su tutti i sistemi mostrati precedentemente: classificatore *random forest*, classificatore neurale, autoencoder e GAN. Le metriche di valutazione utilizzate sono enunciate all'interno della sezione 5.1. In sezione 5.2 vengono mostrati i risultati sperimentali ottenuti dal classificatore random forest, punto di partenza di questo studio. In sezione 5.3 vengono mostrati i risulati sperimentali ottenuti dal classificatore neurale, evoluzione del classificatore random forest, e le problematiche incontrate durante tale fase. In sezione 5.4 viene mostrata la fase sperimentale che ha coinvolto l'autoencoder, per durante la quale si sono ottenuti i pesi utilizzati successivamente come punto di partenza per la fase di training della Generative Adversarial Network. All'interno dell'ultima sezione (5.5) sono mostrati i risultati ottenuti dal training della GAN e la lunga fase di tuning richiesta da quest'ultima.

5.1 Metriche di valutazione

Le metriche di valutazione utilizzate per testare la qualità dei modelli sono state:

- **Precision**: il rapporto $\frac{t_p}{t_p+f_p}$ dove t_p è il numero di veri positivi e f_p il numero di falsi positivi. Intuitivamente è l'abilità del classificatore di non marcare come positivo un campione negativo.

- **Recall**: il rapporto $\frac{t_p}{t_p+f_n}$ dove f_n sono i falsi negativi. Intuitivamente è l'abilità del classificatore di trovare tutti i campioni positivi.

- **F-score**: è definito come la media armonica tra *precision* e *recall*:

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

- **Area sottesa da Receiver Operating Characteristic**: metodo grafico per la valutazione della qualità di un classificatore binario al variare della soglia di discriminazione. E' creata graficando la frazione dei veri positivi rispetto ai campioni positivi (tpr = True positive rate) contro la frazione dei falsi positivi rispetto ai campioni negativi (fpr = False positive rate). L'area sottesa dalla curva ROC equivale alla probabilità che il classificatore predica un campione positivo casuale rispetto ad un campione negativo casuale. Formalmente è definita da:

$$A = \int_{\infty}^{-\infty} \text{TPR}(T)\,(-\text{FPR}'(T))\;dT = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I(T' > T) f_1(T') f_0(T)\,dT'\,dT = P(X_1 > X_0)$$

  dove X_1 è il punteggio per un'istanza positiva e X_0 è il punteggio per un'istanza negativa, mentre f_0 e f_1 sono densità di probabilità che un campione sia negativo (1) o positivo (0)

5.2 Classificatore Random Forest

Il classificatore Random Forest è stato testato in diverse configurazioni. La configurazione che è stata presentata nella sezione 4.1 ha mostrato i migliori risultati in ogni test.

Il classificatore è stato testato dapprima testato con le seguenti famiglie di malware, contro un subset di dimensione simile di domini provenienti dalla classifica Alexa.

Malware Families
legit
cryptolocker
zeus
pushdo
rovnix
tinba
conficker
matsnu
ramdo

Tabella 5.1

Il dataset così riunito è stato separato in due parti disuguali: il 90% è stato utilizzato come dataset di training mentre il restante 10% come testing in maniera da evitare il fenomeno di overfitting. I risultati della predizione sul dataset di testing sono mostrati in figura 5.1 e figura 5.2. A fianco dell'etichette *legit* e *DGA* è indicato il numero di campioni utilizzati per le due categorie. Come si può notare la performance del classificatore è molto positiva. Si ipotizza che tale risultato sia dovuto alla forte differenza linguistica tra domini reali e domini DGA, pertanto il classificatore soffre di un errore praticamente nullo.

In fase preliminare si è effettuato un confronto con altri due algoritmi di classificazione: SVC (C-Support Vector) e Gaussian Naive-Bayes. La loro performance non è stata altrettanto eccellente e pertanto si è deciso di accantonarli e proseguire con l'utilizzo di Random Forest. I report di classificazione sono mostrati in figura 5.3 e 5.4.

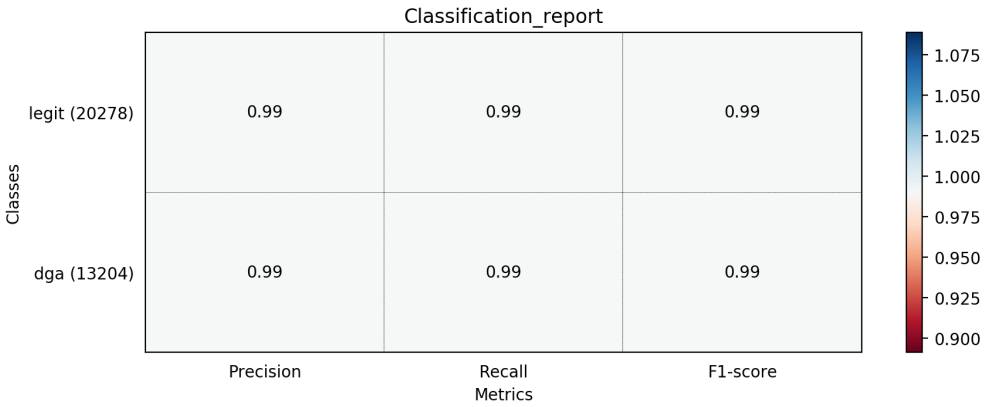

Figura 5.1: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malevoli (DGA).

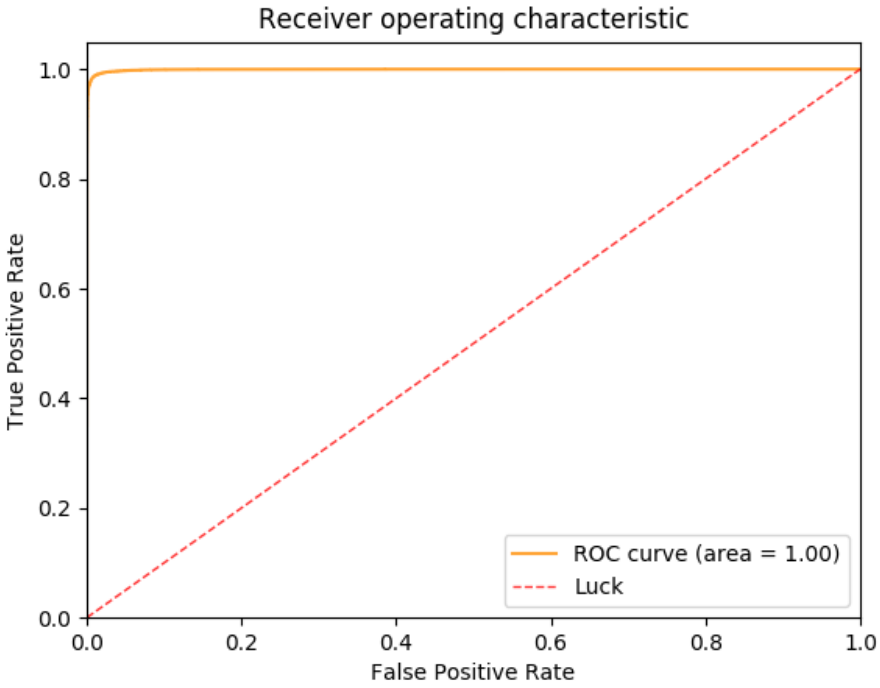

Figura 5.2: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con domini di tabella 5.1.

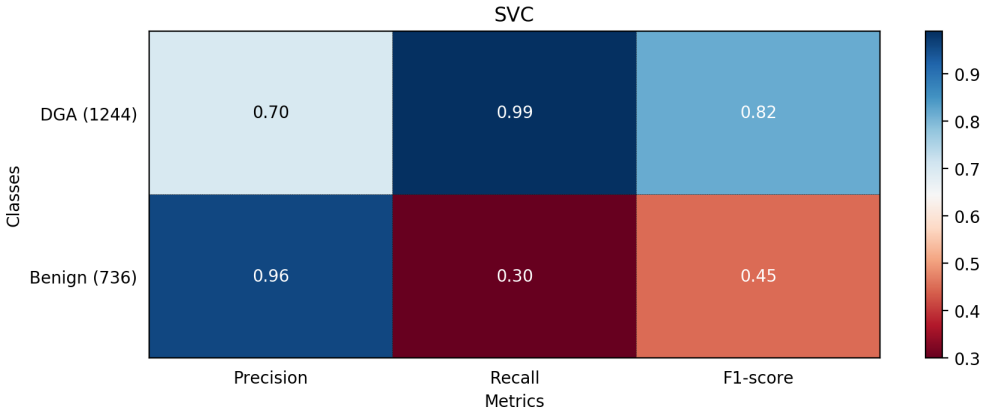

Figura 5.3: Report di classificazione per l'algoritmo SVC.

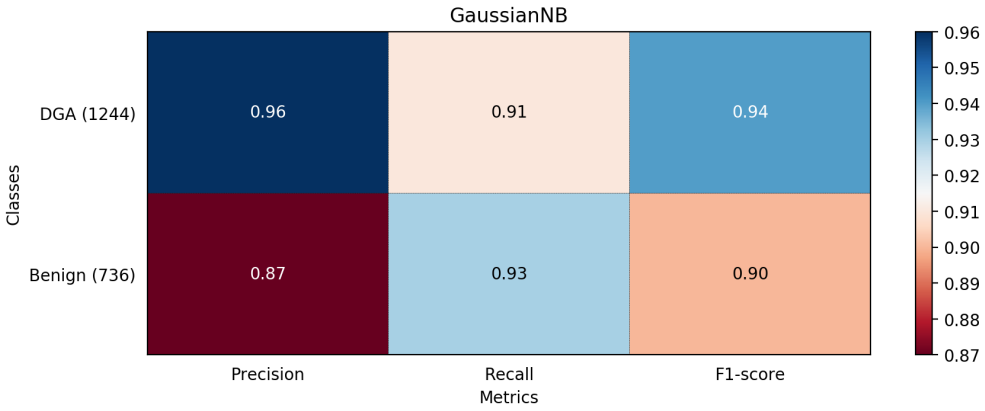

Figura 5.4: Report di classificazione per l'algoritmo Gaussian Naive-Bayes.

Il classificatore random forest è stato testato inserendo *suppobox* tra le famiglie DGA già presenti. Si è scelto tale malware come campione esterno in quanto presenta la maggiore differenza rispetto alle famiglie mostrate in tabella 5.1. I risultati si possono vedere in figura 5.5 e 5.6 e come si può notare la performance ne è fortemente influenzata, introducendo una grande percentuale di falsi nelle predizioni effettuate dal classificatore.

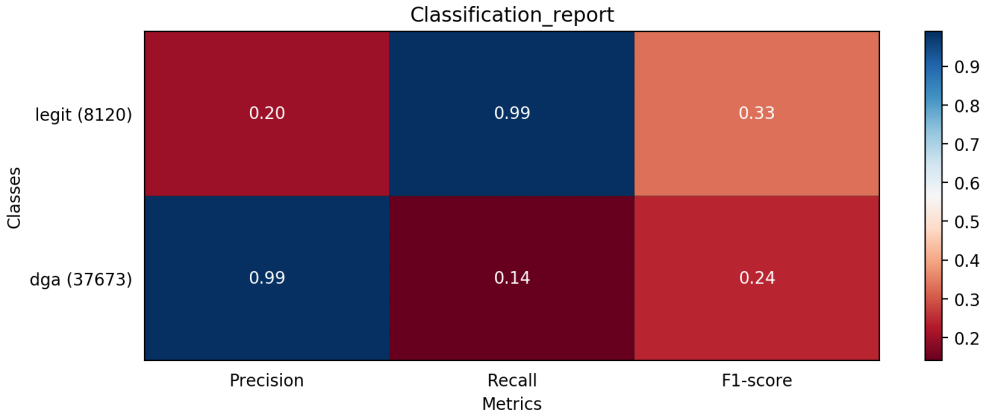

Figura 5.5: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

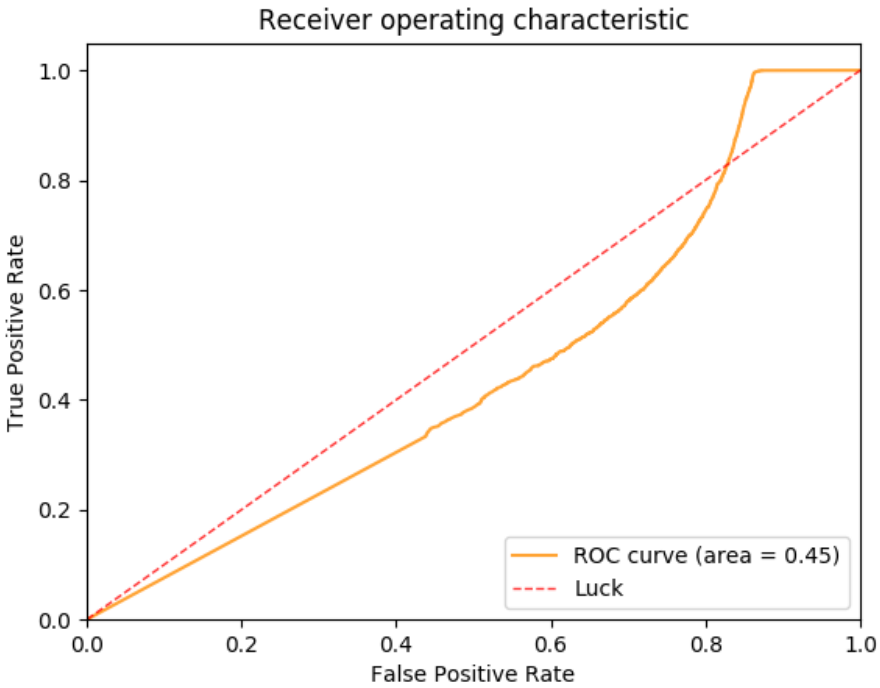

Figura 5.6: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con suppobox.

Come ultimo test è stato eseguito il training aggiungendo al precedente dataset di training una parte di domini generati da suppobox (Figura 5.7 e 5.8). Come si può notare la per-

formance è migliorata sensibilmente, non raggiungendo comunque i risultati eccellenti del primo test.

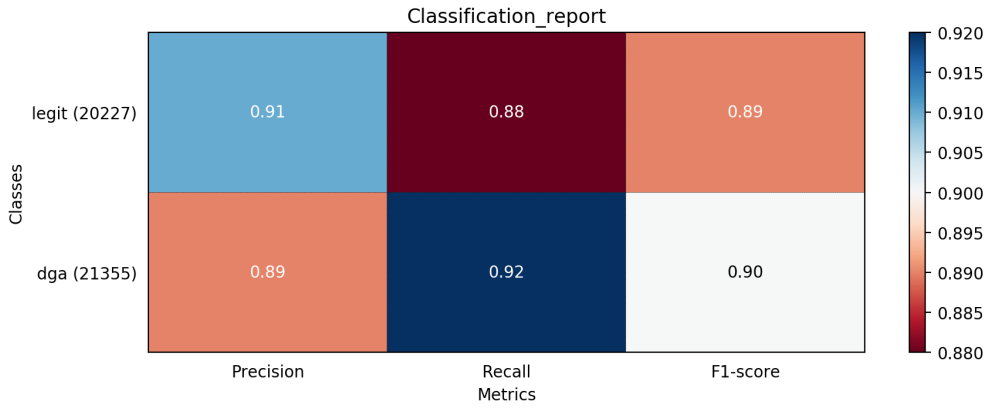

Figura 5.7: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

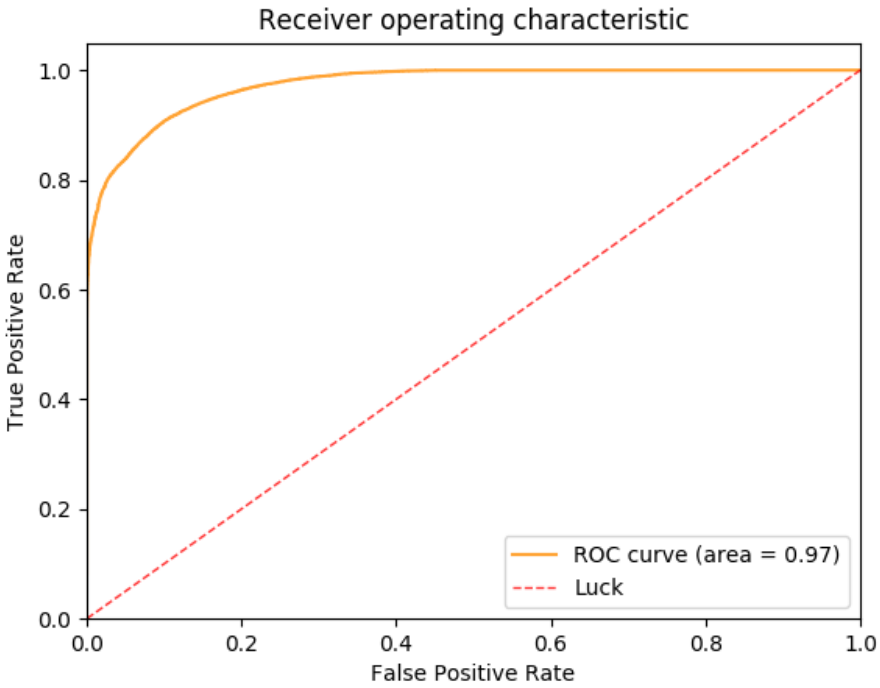

Figura 5.8: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con domini reali e malware (comprendenti suppobox).

5.3 Classificatore Neurale

Il classificatore neurale, nato per superare le mancanze del classificatore random forest, è stato testato nelle stesse condizioni utilizzate precedentemente: in particolare è stato utilizzato lo stesso dataset mostrato in sezione 5.2 e diviso ancora una volta in $\frac{9}{10}$ per la fase di training e $\frac{1}{10}$ per la fase di testing.

In prima fase si sono messe a confronto le tre architetture presentate in sezione 3.2.2. Tali architetture hanno dimostrato tre andamenti simili; tuttavia a fronte dei risultati mostrati e della minore richiesta di risorse, il modello intermedio è risultato vincente rispetto agli altri testati. (Figura 5.9)

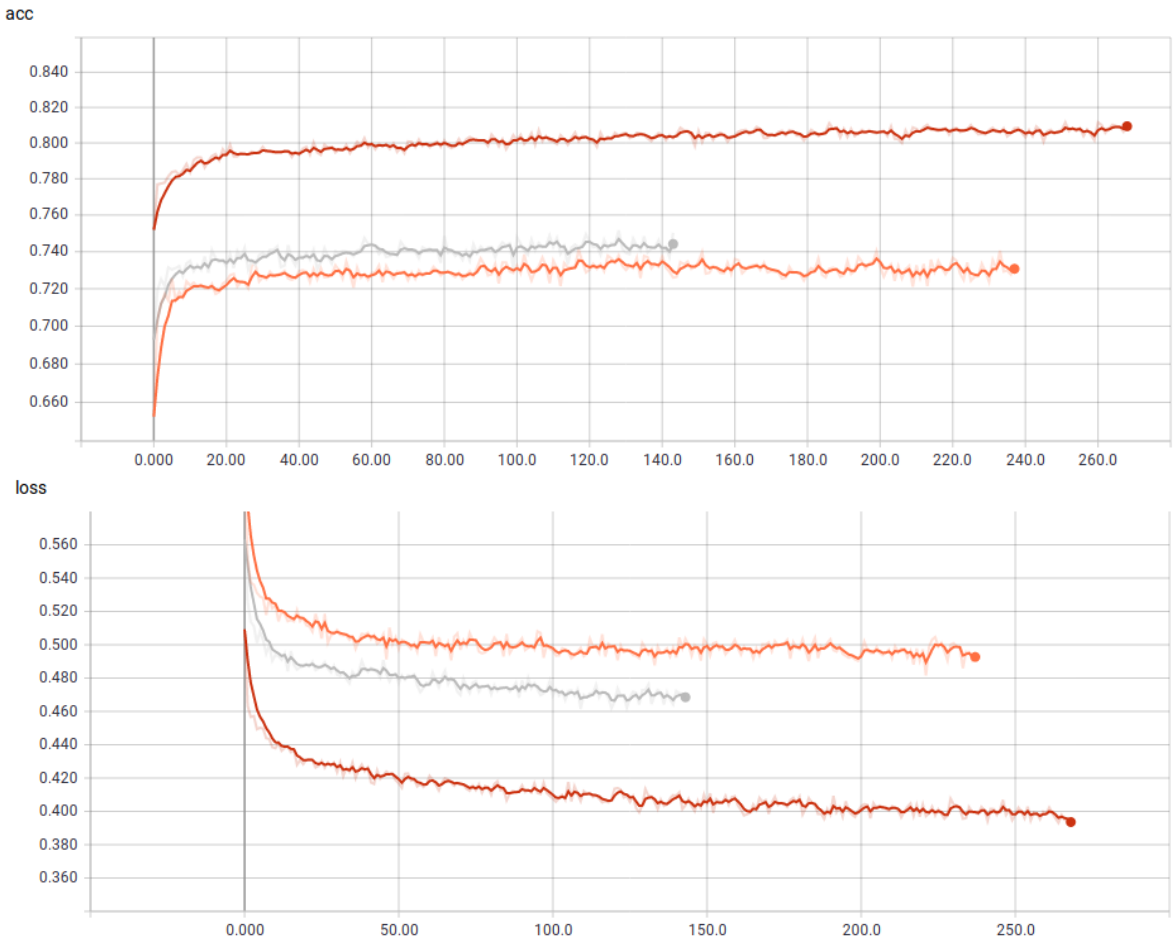

Figura 5.9: Classificatore Neurale: Grafici di Accuracy e Loss di confronto fra i tre modelli. La curva di colore grigio rappresenta il modello ingrandito, la curva di colore arancione rappresenta il modello ridotto mentre la curva di colore rosso rappresenta il modello intermedio; vincente tra i tre.

Motivo di confronto è stata l'introduzione o meno di Batch Normalization [26]. Come si

può vedere dai grafici mostrati in figura 5.10 vi è una differenza di prestazione dovuta alla normalizzazione dei mini-batch, pertanto si è scelto di mantenere tale funzione all'interno dei livelli nonostante l'aumento di costi prestazionali.

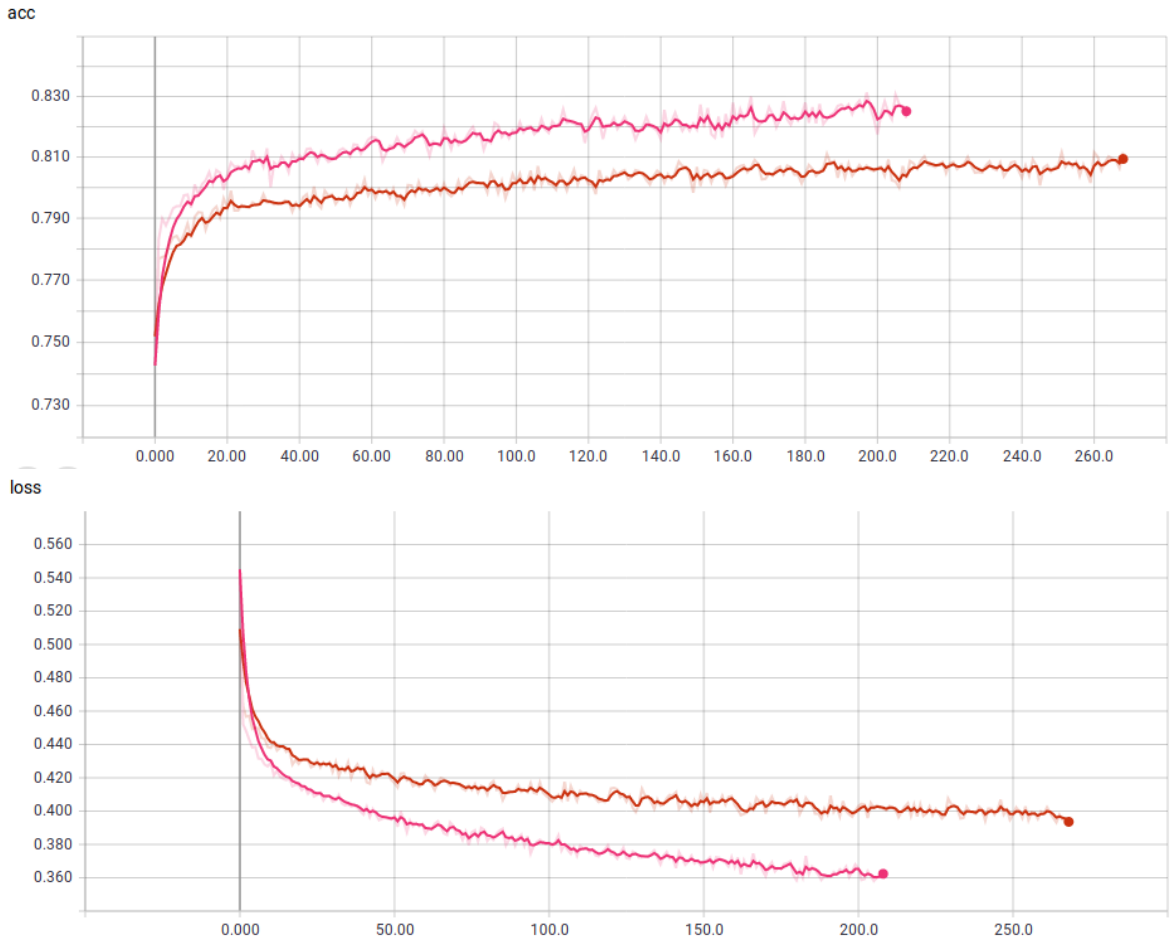

Figura 5.10: Classificatore Neurale: Grafici di Accuracy e Loss per il modello intermedio. La curva rossa identifica il modello senza l'ausilio di Batch Normalization mentre la curva fuchsia rappresenta lo stesso modello con l'inserimento di Batch Normalization per ogni livello densamente connesso che compone il Multilayer Perceptron.

Particolarmente difficoltoso si è dimostrato il tuning degli iperparametri di numero epoche e dimensione mini-batch per ottenere valori ottimali. Dopo una serie di test sperimentali che hanno messo a confronto diversi valori, si sono rilevati i valori

- **numero epoche** $= 60$
- **dimensione minibatch** $= 35$

Tali valori hanno dimostrato di fornire la migliore performance durante la fase di training.

I test effettuati sul dataset hanno mostrato i risultati mostrati in figura 5.11 e 5.12. Come si può vedere dai grafici il comportamento del classificatore è pressoché identico a quello mostrato dal classificatore random forest (figure 5.7 e 5.8)

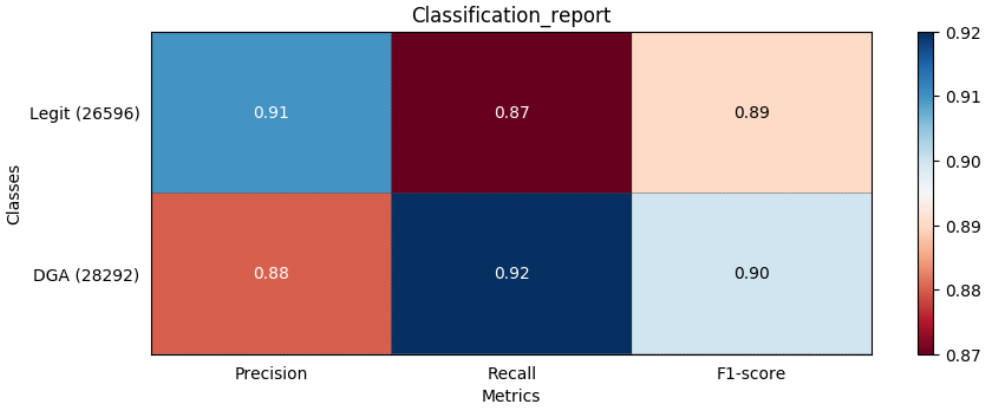

Figura 5.11: Classificatore Neurale: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

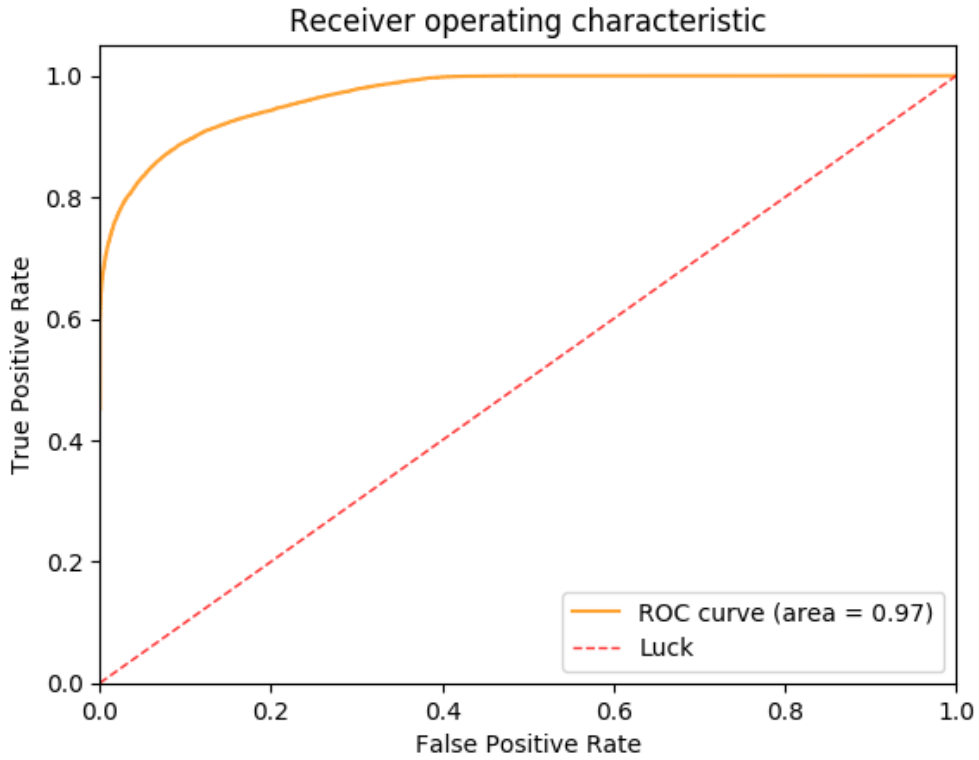

Figura 5.12: Classificatore Neurale: Area sottesa dalla curva ROC per il test con domini reali e malware (comprendenti suppobox).

A partire da questo classificatore neurale stabile è stato possibile implementare un sistema di adversarial learning tramite la GAN derivante da un autoencoder.

5.4 Autoencoder

L'autoencoder come presentato in sezione 3.4 è stato testato con il dataset mostrato in sezione 4.3.1. In fase sperimentale si è proceduto alla quantificazione della configurazione ottimale dell'autoencoder. In particolare si sono messi a confronto i valori di dropout presenti all'interno di encoder e decoder ed i valori di learning rate dei rispettivi compilatori. I valori finali di tali compilatori sono indicati all'interno della sezione 4.3.2 e 4.3.3

Di seguito è possibile notare i risultati della fase di training, per il quale si è trattenuto $\frac{1}{3}$ del training set come subset di validazione. Come è possibile notare l'ultima iterazione (colore verde) ha mostrato i risultati migliori e pertanto è stata utilizzata come base di partenza per il training della successiva GAN. (Figure 5.13, 5.14).

mostrare e commentare i nomi di dominio generati dall'autoencoder

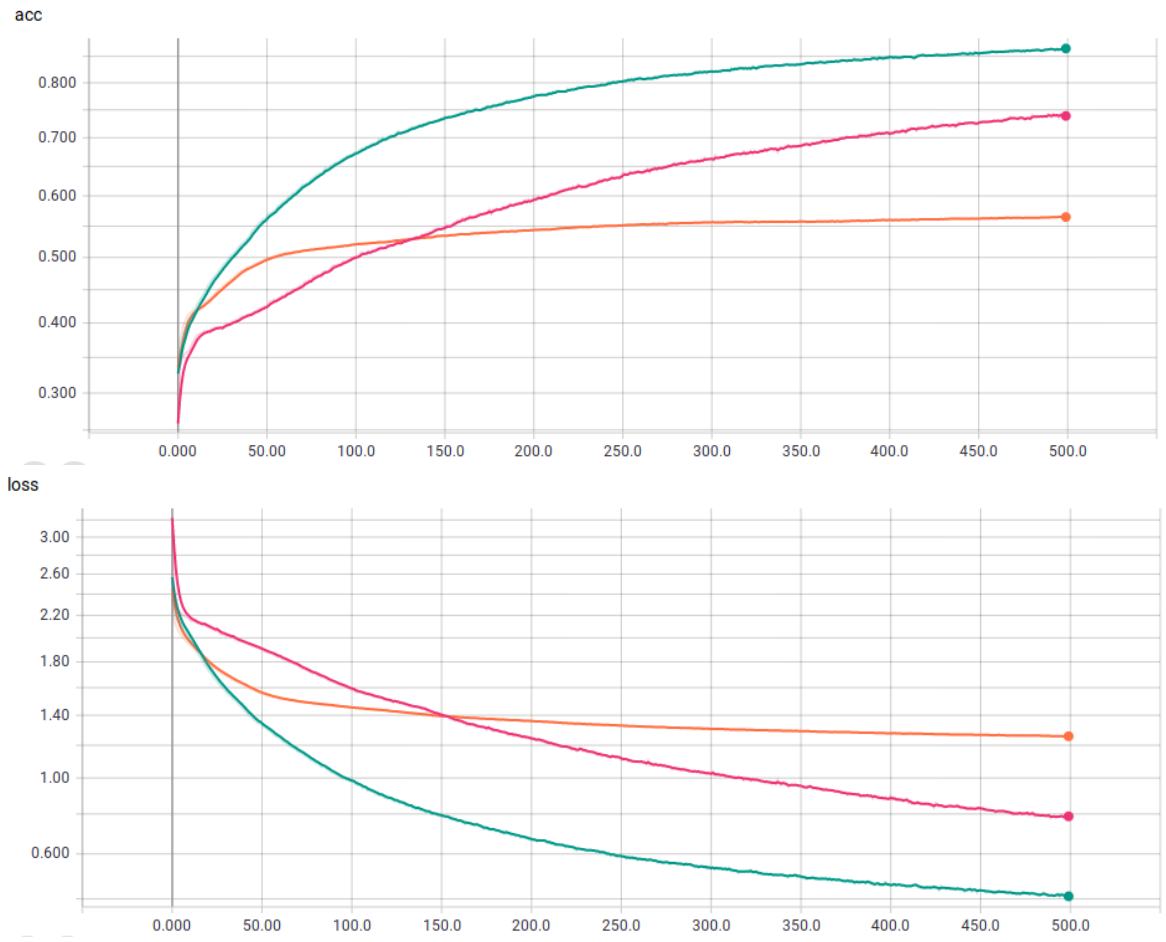

Figura 5.13: Grafici di Accuracy e Loss per la fase di training dell'autoencoder. La prima fase è rappresentata dalla curva arancione, la seconda dalla curva fuchsia mentre la terza fase è rappresentata dalla curva di colore verde. Si può notare come la terza iterazione raggiunga buoni valori di accuracy e loss; pertanto stato scelto come configurazione vincente per la GAN

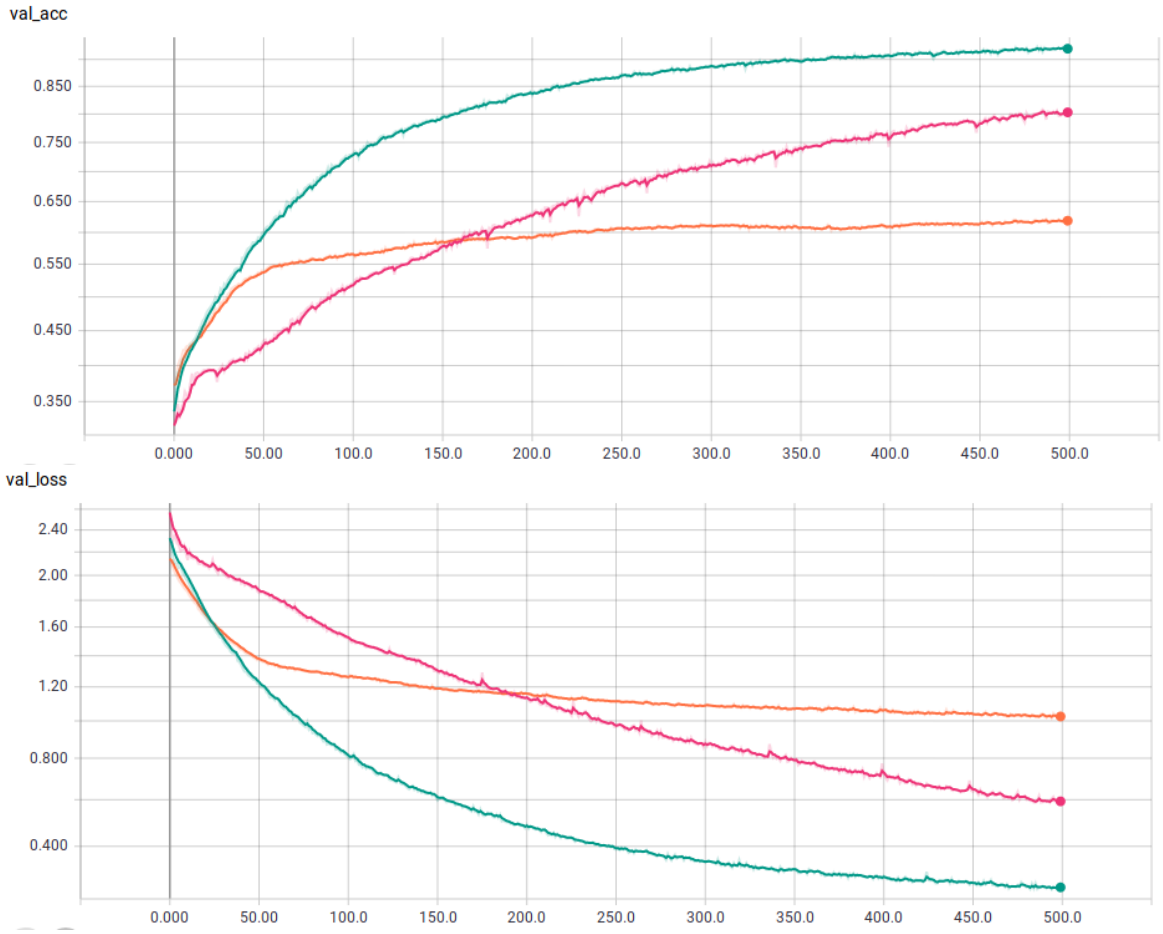

Figura 5.14: Grafici di Accuracy e Loss rispetto al subset di validazione per la fase di training dell'autoencoder. Si può notare come i valori raggiunti siano molto simili a quelli ottenuti sul dataset di training, indice di qualità del modello rispetto a valori mai visti.

5.5 Generative Adversarial Network

La Generative Adversarial Network descritta in sezione 3.7 e implementata come in sezione 4.4 ha richiesto una lunga fase sperimentale nella quale è stato necessario trovare la giusta combinazione di iperparametri per i quali i due sottosistemi generatore e discriminatore potessero rimanere in equilibrio durante la durata necessaria per completare la fase di training.

Capitolo 6

Conclusioni

In questo capitolo si propongono degli esempi per gli oggetti utilizzati più di frequente in latex: la Sezione 1.1 descrive come scrivere citazioni, la Sezione 1.2 propone degli esempi di oggetti float, la Sezione 1.3 descrive come compilare questo documento.

Bibliografia

[1] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, pp. 278–282 vol.1, Aug 1995.

[2] Amazon, "Alexa." `https://www.alexa.com/`, visited in Sep. 2017.

[3] A. Abakumov, "Dga." `https://github.com/andrewaeva/DGA`, visited in Sep. 2017.

[4] H. S. Anderson, J. Woodbridge, and B. Filar, "Deepdga: Adversarially-tuned domain generation and detection," 2016.

[5] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: Detecting the rise of dga-based malware," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 491–506, USENIX, 2012.

[6] S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan, "Detecting algorithmically generated malicious domain names," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, (New York, NY, USA), pp. 48–61, ACM, 2010.

[7] S. Yadav, A. K. K. Reddy, A. L. N. Reddy, and S. Ranjan, "Detecting algorithmically generated domain-flux attacks with dns traffic analysis," *IEEE/ACM Trans. Netw.*, vol. 20, pp. 1663–1677, Oct. 2012.

[8] S. Schiavoni, F. Maggi, L. Cavallaro, and S. Zanero, *Phoenix: DGA-Based Botnet Tracking and Intelligence*, pp. 192–211. Cham: Springer International Publishing, 2014.

[9] J. Geffner, "End-to-end analysis of a domain generating algorithm malware family," *Black Hat USA*, vol. 2013, 2013.

[10] ICANN. `https://www.icann.org/`.

[11] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[12] "Neural networks." `https://ml4a.github.io/ml4a/neural_networks/`.

[13] Qef, "The logistic curve." `https://commons.wikimedia.org/w/index.php?curid=4310325`.

[14] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013.

[15] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014.

[16] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[17] Y. Bengio, "Learning deep architectures for ai," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[18] C.-Y. Liou, J.-C. Huang, and W.-C. Yang, "Modeling word perception using the elman network," *Neurocomput.*, vol. 71, pp. 3150–3157, Oct. 2008.

[19] D. Harris and S. Harris, *Digital Design and Computer Architecture, Second Edition*, p. 129. Morgan Kaufmann, 2012.

[20] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, "Character-aware neural language models," 2015.

[21] D. Britz, "Understanding convolutional neural networks for nlp." http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/, 2015.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," vol. 9, pp. 1735–80, 12 1997.

[23] C. Olah, "Understanding lstm networks." http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[24] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.

[25] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," 2016.

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[27] S. Chintala, E. Denton, M. Arjovsky, and M. Mathieu, "How to train a gan? tips and tricks to make gans work." https://github.com/soumith/ganhacks.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[29] T. White, "Sampling generative networks," 2016.

[30] "Scikit-learn." http://scikit-learn.org/stable/, first visited in Aug 2017.

[31] "Keras." https://keras.io/, first visited in Oct 2017.

[32] "Tensorflow." https://tensorflow.org/, first visited in Oct 2017.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[34] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, pp. 22–30, March 2011.

[35] G. Hinton. `https://www.coursera.org/learn/neural-networks`, 2013.