

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

| | | |
|----------|---|----------|
| 1 | Introduzione | 2 |
| 1.1 | Citazioni | 2 |
| 1.2 | Oggetti float | 2 |
| 1.2.1 | Figure | 2 |
| 1.2.2 | Tabelle | 3 |
| 1.3 | Compilazione | 3 |
| 2 | Stato dell'arte | 4 |
| 2.1 | Machine Learning | 4 |
| 2.2 | Reti neurali | 4 |
| 2.2.1 | Percettrone Multistrato | 4 |
| 2.2.2 | Adversarial Networks | 4 |
| 2.3 | Adversarial Learning | 5 |
| 3 | Progetto | 8 |
| 3.1 | Classificatore Random Forest | 9 |
| 3.1.1 | Dataset | 9 |
| 3.1.2 | Features | 12 |
| 3.1.3 | Output | 13 |
| 3.2 | Classificatore Neurale | 14 |
| 3.2.1 | Input | 15 |
| 3.2.2 | Architettura Classificatore Neurale | 15 |
| 3.2.3 | Output | 17 |
| 3.3 | Realizzazione Adversarial Learning | 17 |
| 3.4 | Autoencoder | 18 |

| | | |
|----------|--|-----------|
| 3.4.1 | Dataset Autoencoder | 20 |
| 3.4.2 | Architettura Autoencoder | 20 |
| 3.5 | Encoder | 21 |
| 3.5.1 | Rete Convoluzionale | 21 |
| 3.5.2 | Pooling | 23 |
| 3.5.3 | Concatenazione | 24 |
| 3.5.4 | Long Short-term Memory | 24 |
| 3.5.5 | Output Encoder | 26 |
| 3.6 | Decoder | 26 |
| 3.6.1 | Regressione Multinomiale | 27 |
| 3.7 | Generative Adversarial Network | 28 |
| 3.7.1 | Architettura GAN | 29 |
| 3.7.2 | Generatore | 31 |
| 3.7.3 | Discriminatore | 31 |
| 4 | Implementazione | 34 |
| 5 | Risultati | 35 |
| 6 | Conclusioni | 36 |

Todo list

| | |
|---|----|
| indicare SVM e GNB solo nei risultati? | 9 |
| ampliare questa sezione? | 13 |
| indicare il capitolo coi risultati di Random Forest | 13 |
| come ampliare????????????? | 17 |
| inserire tabella di esempio. | 20 |
| parlare del pretraining | 34 |

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.

EXAMPLE

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:

```
1 $ pdflatex main.tex
2 $ bibtex main.aux
3 $ pdflatex main.tex
4 $ 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 \mathbf{x} , we define a prior on input noise variables $p_z(\mathbf{z})$, then represent a mapping to data space as $G(\mathbf{z}; \theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters θ_g . We also define a second multilayer perceptron $D(\mathbf{x}; \theta_d)$ that outputs a single scalar. $D(\mathbf{x})$ represents the probability that \mathbf{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 $\log(1 - D(G(\mathbf{z})))$:

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

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

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, $\log(1 - D(G(z)))$ saturates. Rather than training G to minimize $\log(1 - D(G(z)))$ we can train G to maximize $\log D(G(z))$. 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?

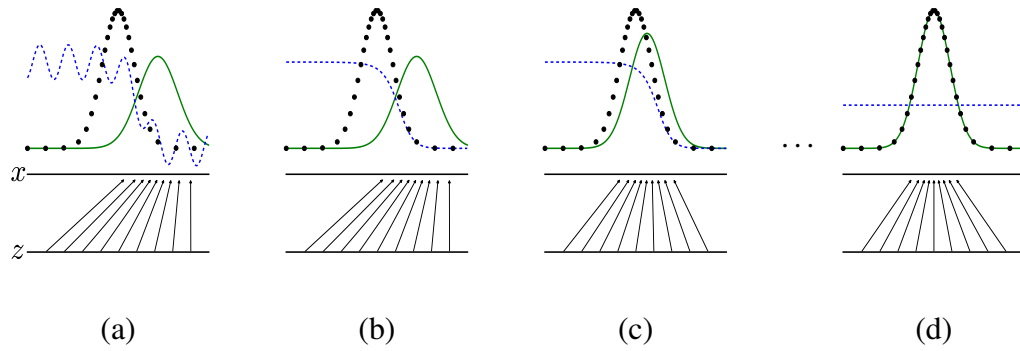


Figura 2.1: Generative adversarial nets are trained by simultaneously updating the discriminative distribution (D , blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) p_x from those of the generative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of x . The upward arrows show how the mapping $x = G(z)$ 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 $D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$. (c) After an update to G , gradient of D has guided $G(z)$ 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. $D(x) = \frac{1}{2}$.

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 $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

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

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

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

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 alitmicamente da domini reali. Si è successivamente progettato un classificatore più raffinato basato su rete neurale che superasse le precedenti mancanze. Il passo successivo ha riguardato la creazione di una *Generative Adversarial Network* basata sul progetto di un *Autoencoder* in grado di generare domini sintetici realistici col fine di rafforzare il classificatore neurale.

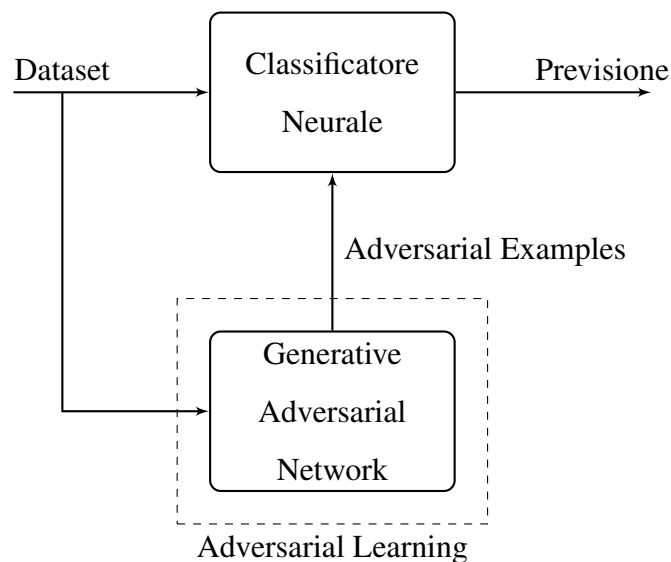


Figura 3.1: Diagramma di progetto

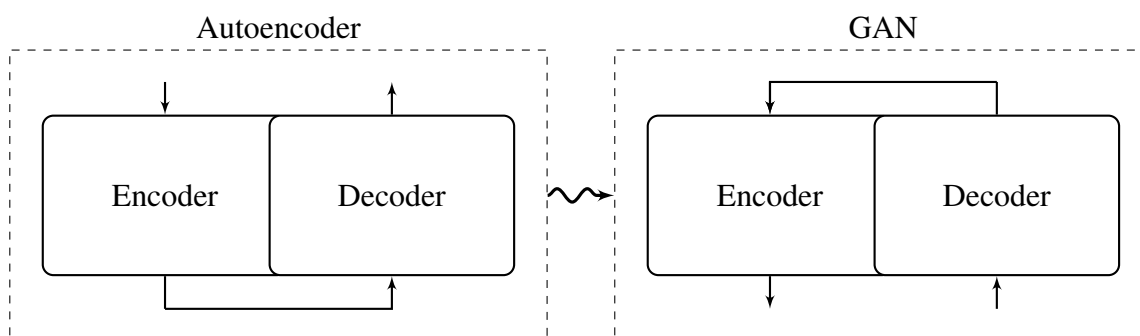


Figura 3.2: Schema generale della progettazione da Autoencoder a Gan

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.

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* [?], 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 divisione 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.

indicare
SVM
e
GNB
solo
nei
risul-
tati?

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.* [1] , per un totale di 1 milione di siti realmente esistenti; mentre grazie

al repository fornito da [2] 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 antipodi, *suppobox* crea domini concatenando due parole scelte in maniera pseudo-casuale da un piccolo dizionario Inglese. In tabella ?? vengono mostrati alcuni esempi di domini generati alitmicamente 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 stringa 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 | fwulvdmdyt.com |
| | gybuisybe.cc |
| | gyinkvye.net |
| lockyv2 | btlwubflhflshn.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 alitmicamente 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 alitmicamente. 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: [4] [5] [6] [7]. Di seguito viene illustrato l'insieme di tali *features*:

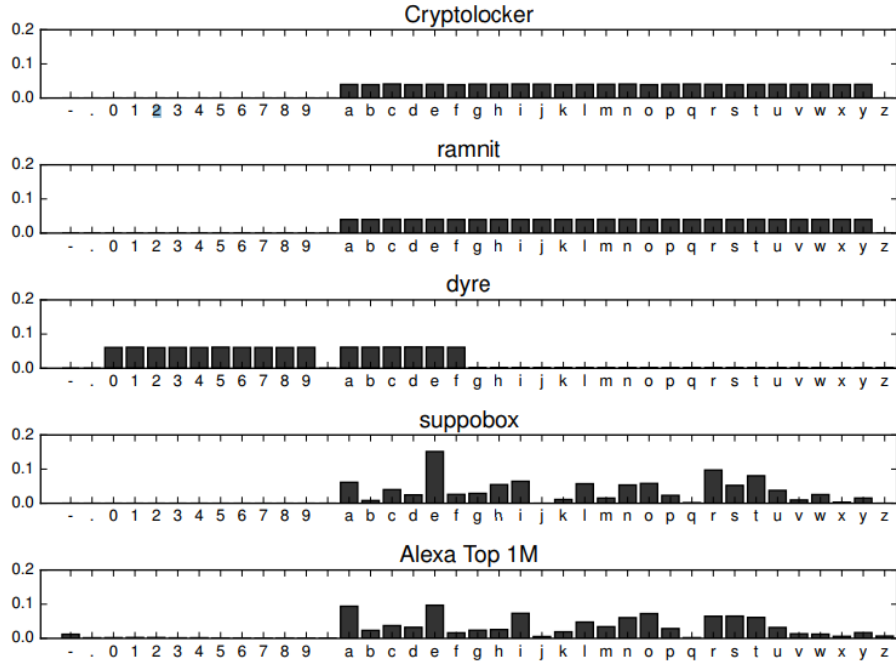


Figura 3.3: Distribuzione dei singoli caratteri in *cryptolocker*, *ramnit*, *dyre*, *suppobox* (dictionary-based DGA) ed il primo milione di domini più visitati secondo Alexa [1]. *fonte:* [3]

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 sottoparole significative w_i di almeno 3 caratteri: $|w_i| \geq 3$ cercando di lasciare fuori meno caratteri possibili:

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

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

- **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 aliticamente. 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 :

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

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

- **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 3.1.1. 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. A tale proposito ???

Durante la fase di sperimentazione il classificatore si è rivelato efficace rispetto la maggior parte delle famiglie di *DGA*. Il caso particolare della famiglia *suppobox* [8] 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* del dizionario inglese di 384

ampliare
que-
sta
se-
zio-
ne?

indicare
il ca-

parole. Tale caratteristica fa sì 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 Input

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 [9] (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) [10] definita dalla funzione sottostante e mostrato in figura 3.4

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

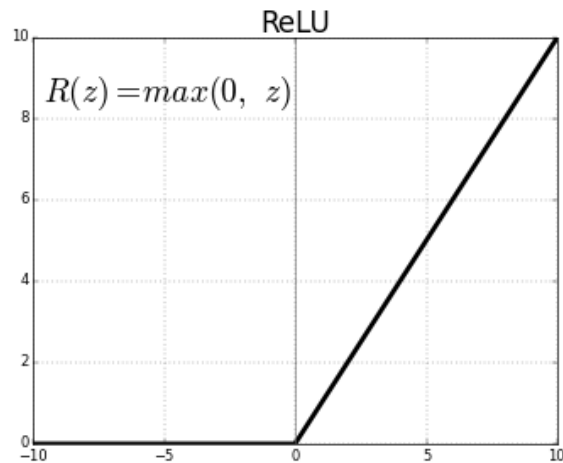


Figura 3.4: *fonte:* [11]

dove x rappresenta l'*input* del neurone. I vantaggi di tale funzione sono una migliorata *performance* rispetto ad altre funzioni simili 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

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

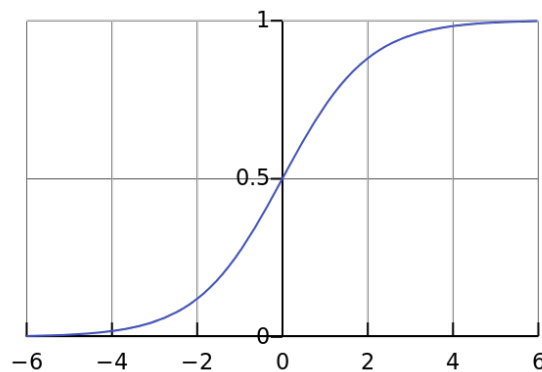


Figura 3.5: *fonte:* [12]

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 alitmicamente, 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* [13], [14]. In particolare la ricerca proposta in [14] introduce il metodo del *fast gradient sign* per scoprire *adversarial examples* perturbando un campione noto x con una piccola quantità $\Delta x \in \text{sign}(\nabla_x J(\theta, x, y))$ dove θ rappresenta i parametri del modello e J il costo necessario a classificare x come y . Separatamente [15] propone l'uso di *Generative Adversarial Network* (abbr. GAN) come *framework* in grado di generare campioni artificiali provenienti dalla stessa distribuzione del

come
am-
plia-
re????????

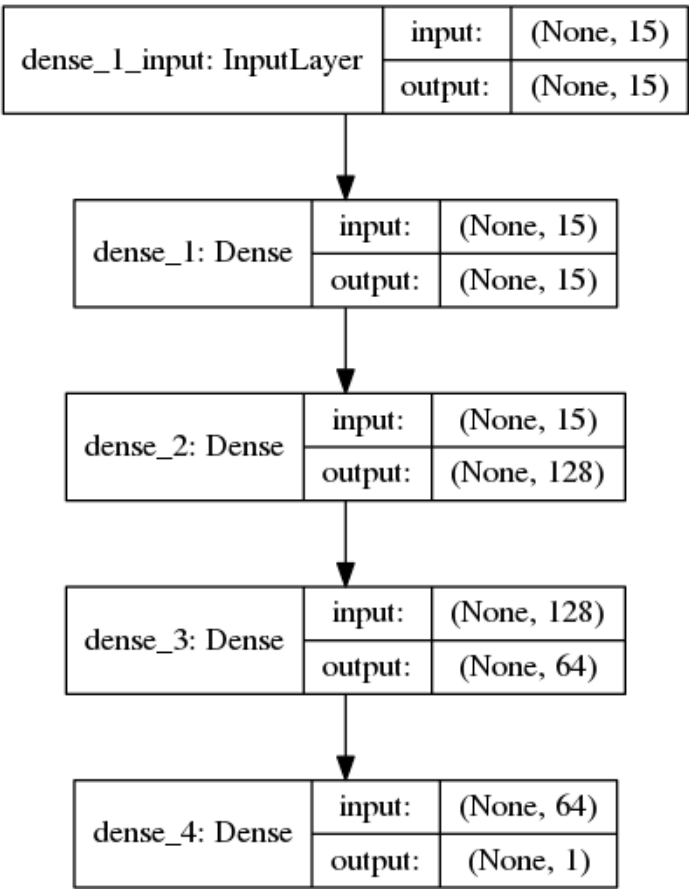


Figura 3.6: Grafico del modello intermedio. Escluso il layer di input iniziale, si notino gli *hidden layer* dense2 e dense3 di dimensioni rispettivamente 128 e 64

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 [3].

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 [16] [17]. Si supponga di avere un set di training $\{x^{(1)}, x^{(2)}, x^{(3)}, \dots\}$ 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 $x \approx h_{W,b}(x)$. Un *autoencoder* tipicamente consiste in due macro-componenti:

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

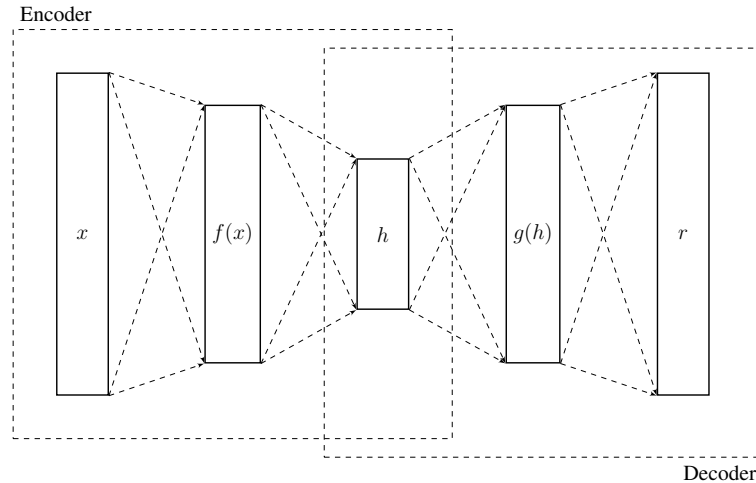


Figura 3.7: Struttura generica di un autoencoder, il quale mappa l'input x in un output r attraverso una rappresentazione codificata h

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* [18] 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*.

inserire
tabel-
la di
esem-
pio.

3.4.2 Architettura Autoencoder

L'architettura dell'*encoder* in esame è ispirato al lavoro mostrato in [19] 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 l'asimmetrico 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 Convolutionale accoppiata ad una LSTM. A differenza del lavoro proposto in [3], 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 Convolutionale

Una Rete Convolutionale 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

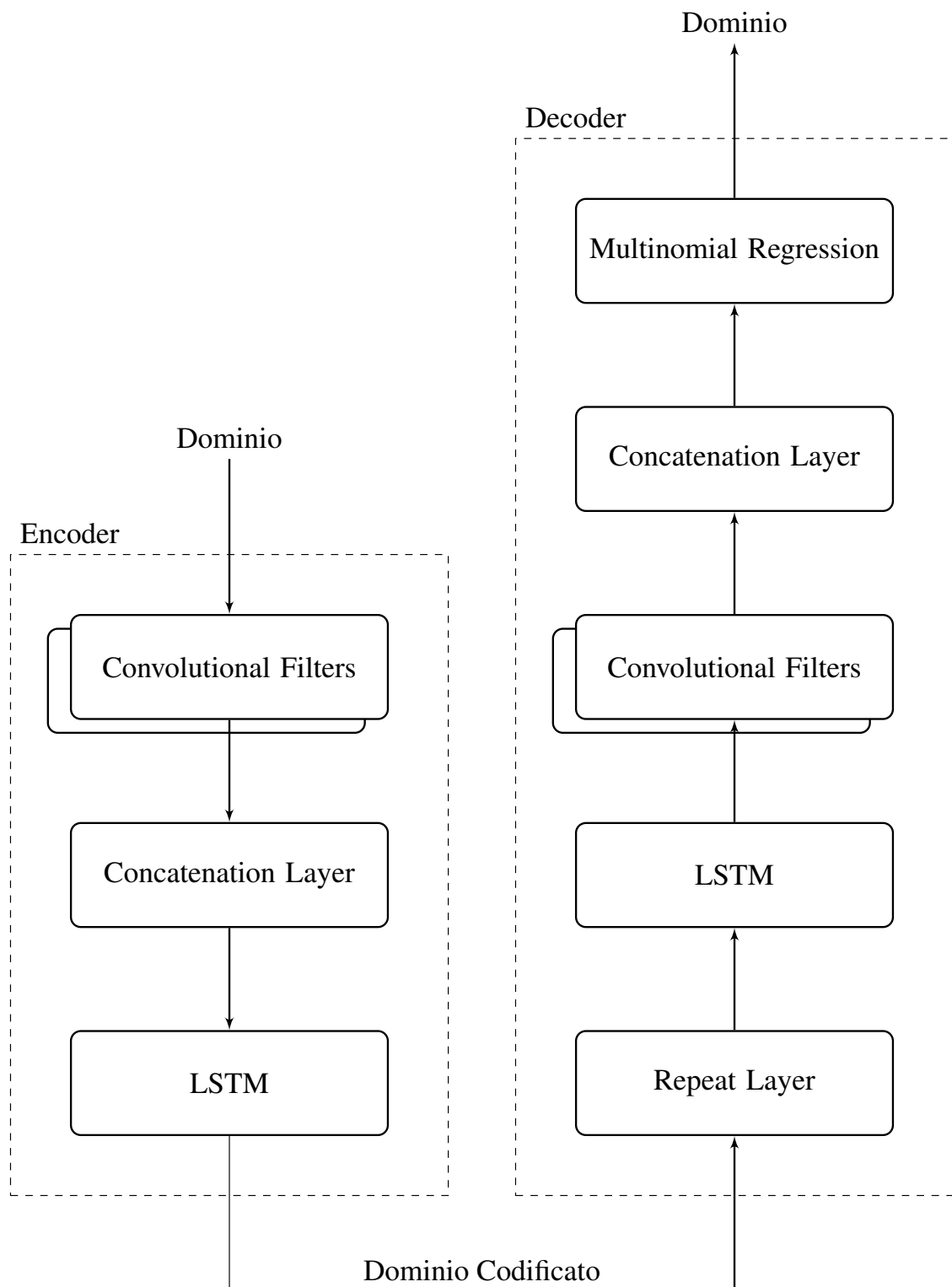
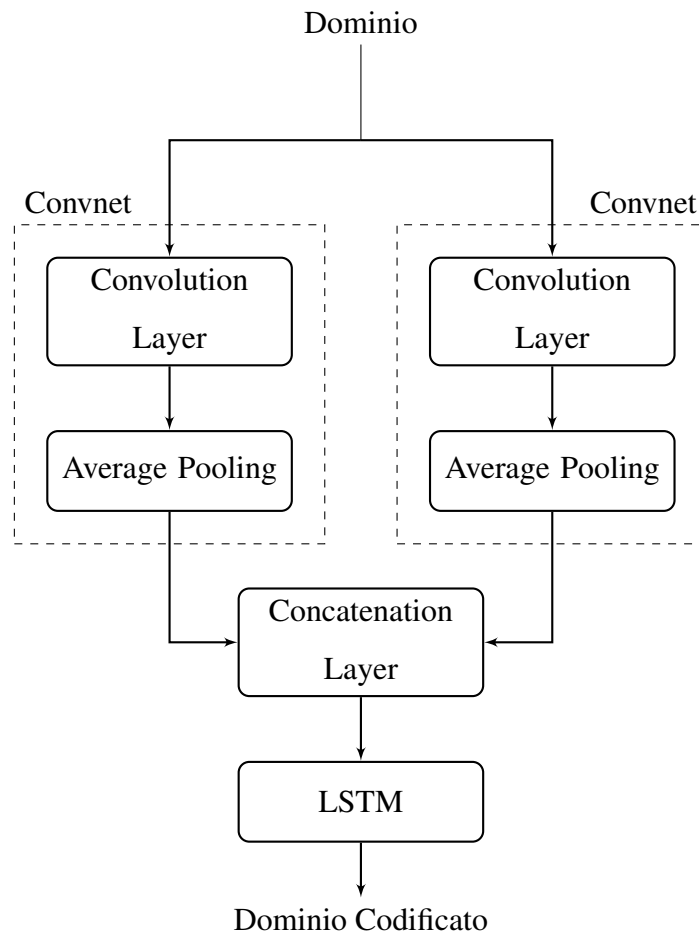


Figura 3.8: Struttura dell'*autoencoder* in esame. L'input in ingresso dato dai domini viene codificato attraverso l'*encoder* e dato in ingresso al *decoder* che ne genera una approssimazione

**Figura 3.9:** Struttura dell'*encoder*

i valori dei *filtri*. I filtri codificano determinati aspetti caratteristici dell'input, analizzando 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'insieme 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 anziché

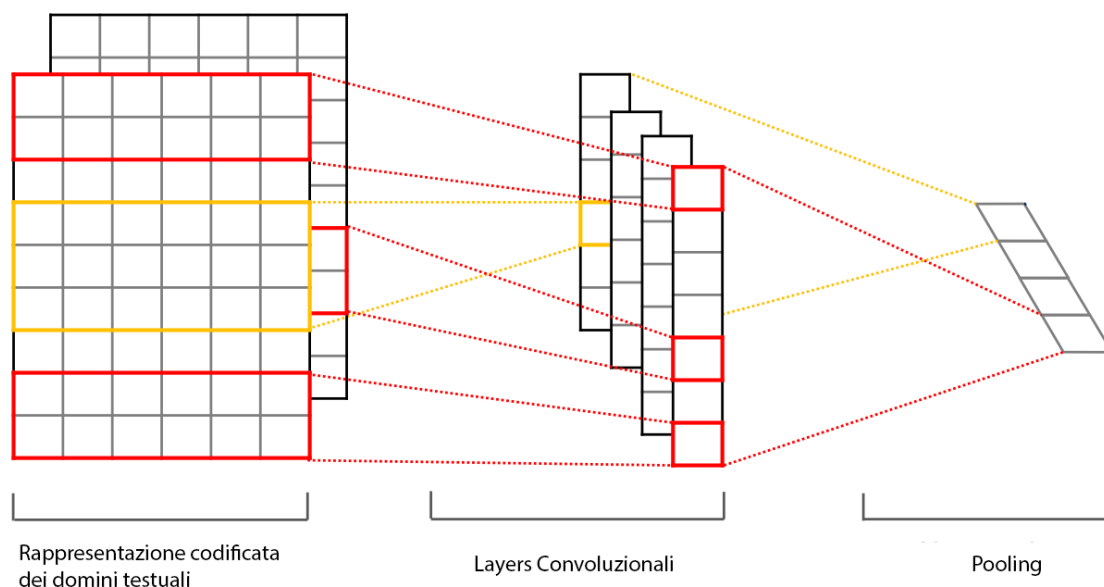


Figura 3.10: Funzionamento di una Convolutional Neural Network, *fonte* [20]

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

3.5.3 Concatenazione

Il layer aggiuntivo di *concatenazione* funge per mette 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*) [21]. Si tratta di un modello di *Recurrent Neural 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 input 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:

$$i_t = \sigma(W^i x_t + U^i h_{t-1} + b^i)$$

$$f_t = \sigma(W^f x_t + U^f h_{t-1} + b^f)$$

$$o_t = \sigma(W^o x_t + U^o h_{t-1} + b^o).$$

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.

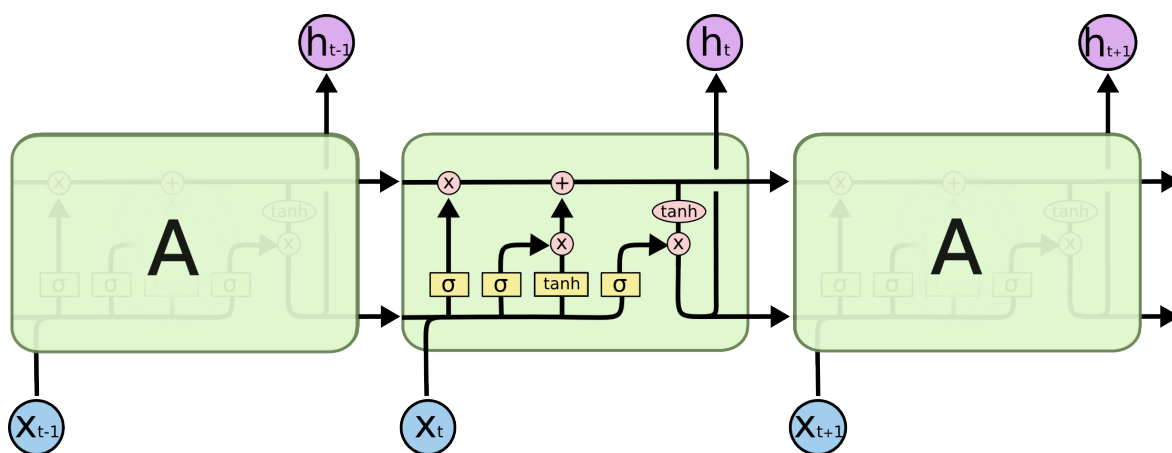


Figura 3.11: Struttura di una cella LSTM. *fonte:* [22]

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 dimensione 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*.

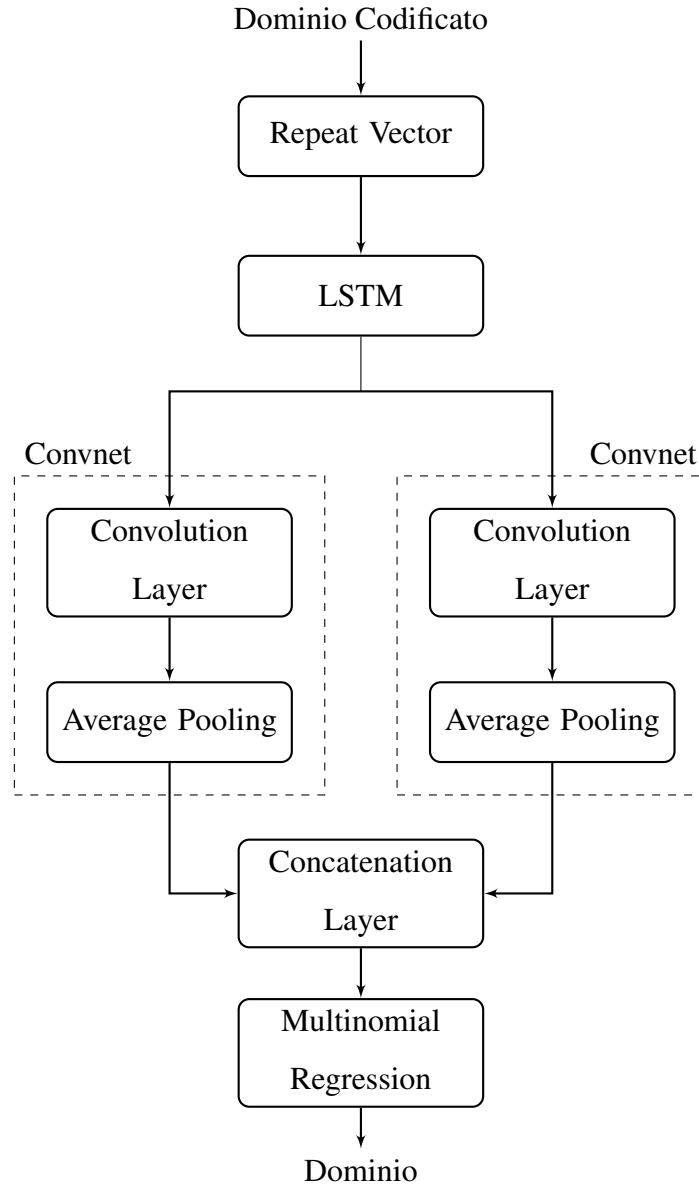


Figura 3.12: Struttura del *decoder*

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:

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

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 stata usata una funzione *softmax* utilizzata nel campo dell'apprendimento per rinforzo [23], 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" ($\tau \rightarrow \infty$), tutte le azioni hanno la stessa probabilità e più bassa la temperatura, più la "ricompensa" influisce sulla probabilità. Per basse temperature ($\tau \rightarrow 0^+$), la probabilità di azione con la ricompensa più alta tende ad 1.

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

3.7 Generative Adversarial Network

La *Generative Adversarial Network* (abbr. GAN) è un 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 difficile la fase di *training*: a differenza di reti neurali più semplici, non è possibile fissare un minimo da ottimizzare. Normalmente si attua una 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 sì 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.

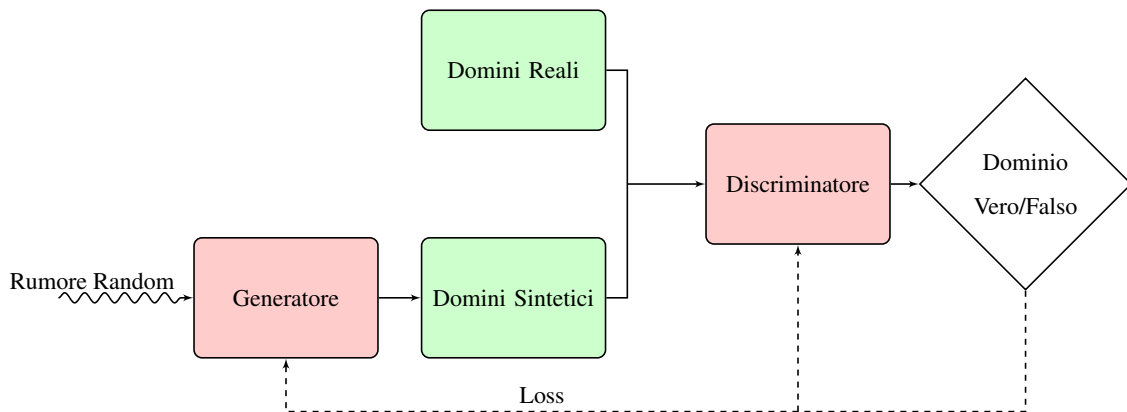


Figura 3.13: Struttura di una *Generative Adversarial Network*

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 [3], si è deciso di implementare una architettura più semplificata. Come mostrato in [24] 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 [25], l'utilizzo di Leaky Relu [26] come layer di attivazione all'interno della sottorete Convoluzionale e Dropout come strumento per evitare il fenomeno di *overfitting* [27]. L'uso di Leaky ReLU come funzione di attivazione è giustificata dalla riduzione della sparsità del gradiente, la quale generalmente causa instabilità.

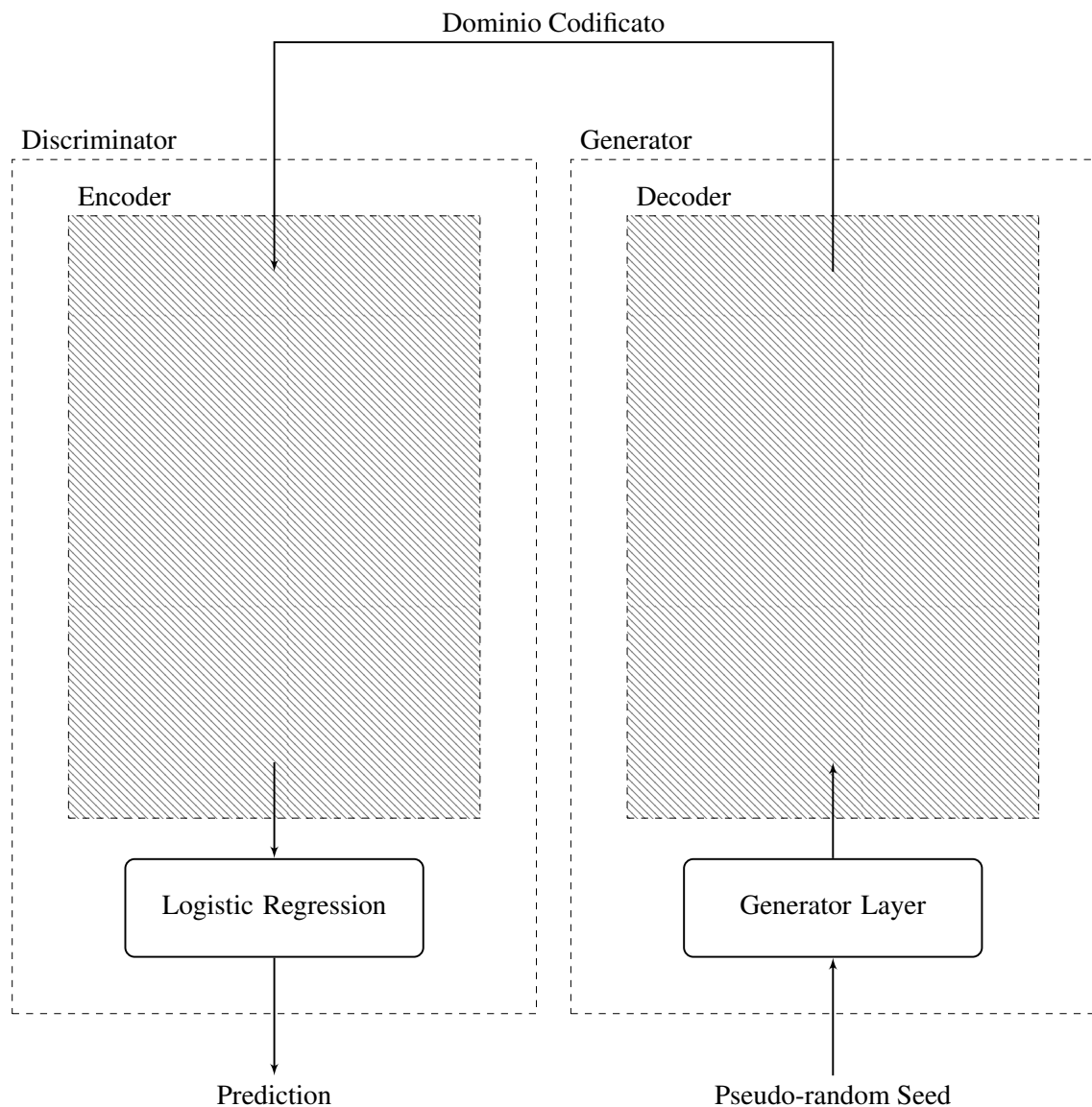


Figura 3.14: Struttura della GAN evoluta dell'autoencoder

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 [25]. 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.

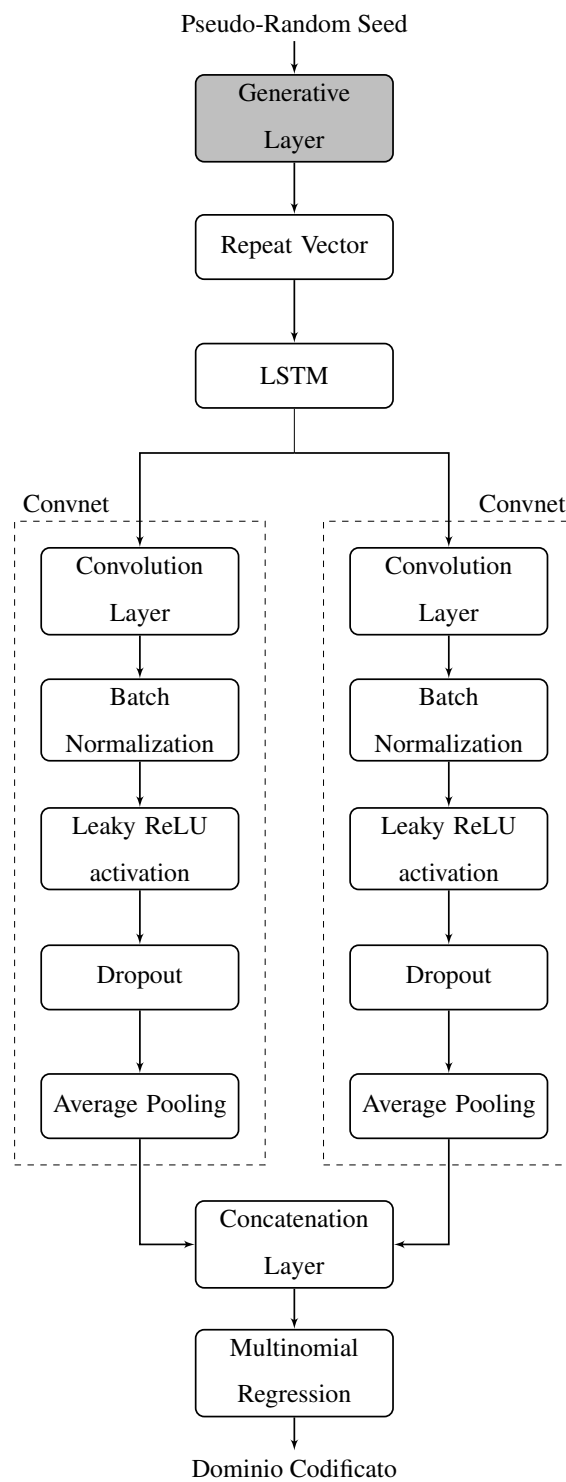
3.7.2 Generatore

Il generatore aggiunge in testa alla struttura 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.

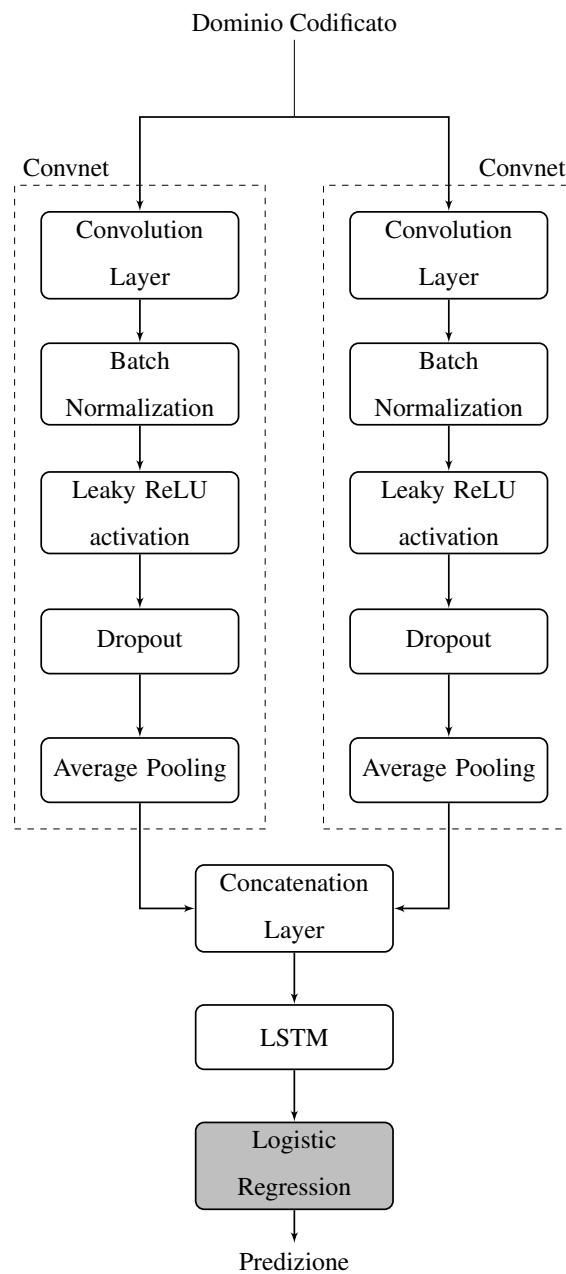
Il Generator layer è formato da un generatore di rumore randomico di distribuzione gaussiana, la quale è stata provata come ottimale nel caso di GAN da [28]. Tale distribuzione, di dimensione fissata, mima un possibile dominio codificato da encoder come descritto in sezione 3.5. A partire da tale rumore 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 empiriche, è pos-

**Figura 3.15:** Struttura del generatore

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

**Figura 3.16:** Struttura del discriminatore

Capitolo 4

Implementazione



parlare
del
pre-
train-
ning

Capitolo 5

Risultati

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.

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] Amazon, “Alexa.” <https://www.alexacom/>, visited in Sep. 2017.
- [2] A. Abakumov, “Dga.” <https://github.com/andrewaeva/DGA>, visited in Sep. 2017.
- [3] H. S. Anderson, J. Woodbridge, and B. Filar, “Deepdga: Adversarially-tuned domain generation and detection,” 2016.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] S. Schiavoni, F. Maggi, L. Cavallaro, and S. Zanero, *Phoenix: DGA-Based Botnet Tracking and Intelligence*, pp. 192–211. Cham: Springer International Publishing, 2014.
- [8] J. Geffner, “End-to-end analysis of a domain generating algorithm malware family,” *Black Hat USA*, vol. 2013, 2013.

- [9] ICANN. <https://www.icann.org/>.
- [10] 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.
- [11] “Neural networks.” https://ml4a.github.io/ml4a/neural_networks/.
- [12] Qef, “The logistic curve.”
<https://commons.wikimedia.org/w/index.php?curid=4310325>.
- [13] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, “Intriguing properties of neural networks,” 2013.
- [14] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” 2014.
- [15] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” 2014.
- [16] Y. Bengio, “Learning deep architectures for ai,” *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.
- [17] 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.
- [18] D. Harris and S. Harris, *Digital Design and Computer Architecture, Second Edition*, p. 129. Morgan Kaufmann, 2012.
- [19] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, “Character-aware neural language models,” 2015.
- [20] D. Britz, “Understanding convolutional neural networks for nlp.”
<http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/>, 2015.

- [21] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” vol. 9, pp. 1735–80, 12 1997.
- [22] C. Olah, “Understanding lstm networks.”
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015.
- [23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.
- [24] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” 2016.
- [25] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015.
- [26] 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>.
- [27] 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.
- [28] T. White, “Sampling generative networks,” 2016.