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 2016/2017

Indice

Capitolo 1

Introduzione

Capitolo 2

Stato dell'arte

All'interno di questo capitolo vengono presentate le basi teoriche che compongono il lavoro mostrato in questo elaborato. In sezione 2.1 viene presentata una breve introduzione alle botnet, la loro tassonomia ed il funzionamento del Command & Control. In sezione 2.2 si espone l'argomento dell'Adversarial Learning e le problematiche che esso comporta nell'ambito dell'information security. In Sezione 2.3 viene introdotto il campo di studio del Machine Learning, mostrando una panoramica delle problematiche affrontabili con tale strumento ed una tassonomia delle tecniche possibili di utilizzo. In Sezione 2.4 viene presentato il sottocampo di applicazione Machine Learning: Deep Learning, assieme ad una spiegazione dei fondamenti di funzionamento. All'interno della sezione 2.5 vengono mostrate le tecniche di Generative Deep Learning che saranno utilizzate all'interno di questo elaborato nei capitoli successivi quali Autoencoder e Generative Adversarial Network.

2.1 Botnet

Una botnet è composta da svariati dispositivi connessi a Internet, come host, smartphone o dispositivi IoT, ciascuno dei quali esegue uno o più bot. I proprietari delle botnet controllano queste ultime tramite software C&C (Command and Control) per svolgere svariate attività, solitamente dannose, che richiedono un livello di automazione su vasta scala, tra cui:

- Attacchi DDoS (Distributed Denial-of-Service) che causano downtime non pianificati delle applicazioni
- Verifica di elenchi di credenziali divulgate (attacchi di compilazione delle credenziali) che portano al controllo degli account
- Attacchi alle applicazioni web allo scopo di sottrarre dati
- Fornire a utenti malintenzionati accesso a un dispositivo e alla relativa connessione a una rete

Le botnet vengono sempre più affittate dai cyber criminali per gli scopi più diversi e rappresentano una minaccia reale per qualsiasi azienda presente in Internet. Questo significa che non è più necessario che gli autori degli attacchi siano in possesso delle conoscenze necessarie per realizzare le proprie botnet, in quanto possono utilizzare botnet già create da altri. Il numero di bot varia notevolmente da una botnet all'altra e dipende dall'abilità del proprietario della botnet di infettare dispositivi non protetti.

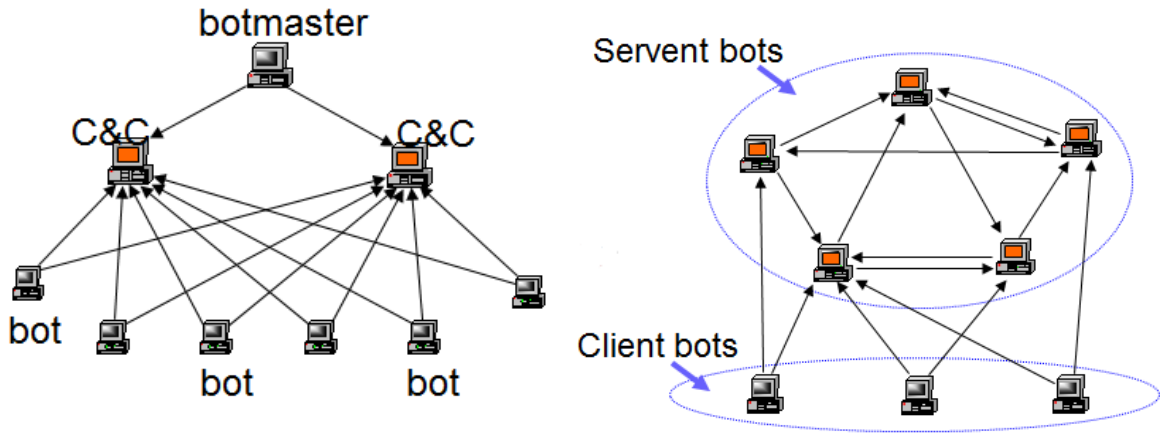
Le botnet possono essere divise in due classi, operando la suddivisione per architettura (Figura 2.1):

- Botnet Centralizzate: il tipo di architettura è più semplice, tutti i bot sono connessi direttamente al C&C. Il server C&C gestisce la lista di tutte le macchine infettate, controlla il loro status e da loro informazioni operative. Questa tipologia di botnet è estremamente vulnerabile, una volta "convertito" un bot, si hanno tutte le informazioni necessarie per poter effettuare un attacco al C&C e smantellare la rete; esistono alcune estensioni a questa struttura, in cui si usano più server di comando e controllo, ma i vantaggi che si ottengono sono trascurabili.

- Botnet Decentralizzate o P2P: in questo schema i bot non sono necessariamente connessi al server C&C ma tutti insieme costituiscono una rete di comunicazione in cui i comandi sono trasmessi anche da zombie a zombie. Ogni nodo della rete è un bot, il quale comunica solo con una lista di nodi "vicini". In questo schema per gestire la botnet è necessario l'accesso ad almeno un client. Il punto di forza di questi schemi è la difficoltà che s'incontra nel momento in cui si è interessati a smantellare l'intera rete; il reverse engineering di un singolo bot non è più sufficiente a individuare tutti i computer coinvolti né tantomeno a smantellare i server C&C.

2.1.1 Command and Control

Durante la fase C&C una macchina appena infettata diventa attivamente a far parte della botnet. Un bot tradizionale client-server, una volta installato su una nuova macchina, tenterà immediatamente il collegamento attraverso una rete IRC o contattando il server C&C via HTTP. I bot P2P decentralizzati ecentralised sono distribuiti con un metodo di bootstrapping predeficino per connettersi ad un DHT rilevante. Una volta connesse, le macchine compromesse chiedereanno ad uno deipropri peer riguardo gli ultimi comandi richiesti. Alcuni bot P2P richiedono che una specifica porta sia aperta ai peer per abilitare la comunicazione tra peer [1]. Vi sono due tecniche principali per la propagazione di comandi in un sistema bot-



Botnet Centralizzata Botnet Decentralizzata

Figura 2.1: Anatomia di Botnet

net: push e pull. In botnet basate su comunicazione IRC viene utilizzato il sistema push, in quanto i bot risiedono in una chat-room, in ascolto per nuovi comandi. In botnet basate su HTTP, i bot controllano periodicamente il server per verificare nuovi comandi. Reti basate su P2P possono agire in entrambe le modalità in quanto ricevono ed inviano comandi in maniera paritaria.

Molte famiglie di botnet contengono algoritmi di generazione di domini (DGA) per rendere le difese preventive inefficaci. I domini vengono generati algoritmicamente in maniera pseudo-randomica (decine di migliaia al giorno) da un campione iniziale. La macchina infetta successivamente invia una richiesta HTTP ad una larga parte di tali domini contemporaneamente, nella speranza che uno di questi domini sia effettivamente registrato e ospiti il C&C della botnet. L'attacker ha solamente il compito di registrare un piccolo numero di questo domini, avendo a disposizione gli stessi domini generati. Dal lato dei difensori invece è necessario che tutti i domini generati da tali algoritmi siano rilevati e inseriti in una lista di esclusione per ottenere una difesa efficace della rete. Tale metodo di difesa risulta via via più inefficiente man mano che il numero di domini DGA aumenta.

2.2 Adversarial Learning

Le tecniche di Machine Learning possono essere utilizzate per prevenire azioni vietate o illegali e nel caso vi siano degli incentivi economici, vi possono essere probabili antagonisti intenzionati ad eludere tali restrizioni. Un esempio tipico è il caso del filtraggio anti spam, nel quale gli spammers forgiano messaggi ad hoc per eludere le più recenti tecniche di filtraggio. Tali tecniche possono essere identificate come *adversarial learning* (apprendimento antagonista).

Le vulnerabilità dei metodi di Machine Learning rispetto a manipolazione antagonista non possono venire semplicemente ignorate con la richiesta di tecniche più "robuste". I fondamenti teorici del Machine Learning odierno sono largamente costruiti sull'assunto che i dati di allenamento descrivano adeguatamente la realtà del fenomeno studiato. Questo assunto viene chiaramente violato nel caso che le distribuzioni di allenamento o di test vengano alterate, assumendo che gli antagonisti utilizzino qualsiasi mezzo possibile per disturbare l'algoritmo di apprendimento. I metodi di apprendimento in caso di ambienti antagonisti dovrebbero quindi essere in grado di sostenere una distorsione nei propri dati.

Come dimostrato da ricerche precedenti ([2], [3], [4]) un antagonista senza vincoli che può arbitrariamente alterare dati e target, può introdurre un errore fino al 100%. Tuttavia nei casi pratici gli attackers devono atteneresi a determinati vincoli; ad esempio una email di spam deve recapitare il suo messaggio, un malware inviato ad un host deve eseguire correttamente e sfruttare una vulnerabilità e antagonisti che cercano di alterare i motori di ricerca possono controllare solo una frazione di tutti i domini raggiungibili. In alcuni casi si può mostrare come certi vincoli possono rendere la rilevazione di un attacco computazionalmente intrattabile [5].

L'investigazione di metodi di Machine Learning per ambienti ostili è stata attuata in tre aree di ricerca largamente distinte: Machine Learning, sicurezza informatica e spam filtering. Nel caso di Machine Learning ricerche precedenti si sono incentrate su metodi di minimo-massimo con l'obiettivo di ottenere robustezza rispetto alla incertezza dell'input. Ad esempio classificatori più robusti sono stati sviluppati per gestire casi come il feature deletion [6] ; oppure ricerche hanno dimostrato che equilibri unici di Nash esistono per alcune tipologie di situazioni antagoniste [7] .

Una differente visione dei problemi di apprendimento antagonista è emersa nel campo della sicurezza informatica, specialmente per quel che riguarda il rilevamento di intrusioni (*intrusion detection*). Diversi metodi sono stati proposti per rilevare pacchetti di rete anomali o per generare automaticamente *signatures* strettamente legate a metodi di Machine Learning (ad esempio i lavori eseguiti in [8] e [9]).

L'area dello spam filtering ha richiesto sforzi fondamentali da parte degli esperti in sicurezza; la costante ricerca di tecniche di evasione dei filtri anti-spam ha portato a studi estensivi riguardo vincoli e tecniche robuste di filtering [10].

Il Machine Learning può fornire soluzione a difficili problemi di sicurezza di rete, compresi filtri anti-spam, rilevazione di vari tipologie di attacchi contro server e host, rilevazione di pagine web deliberatamente modificate per manipolare i motori di ricerca sfruttandone le vulnerabilità. Tutte queste problematiche riguardano la manipolazione di grandi quantità di dati in un ambiente altamente variabile e il bisogno di tecniche di classificazione veloci ed accurate. L'esistenza di antagonisti che possono trarre profitto in queste aree complica l'applicazione di Machine Learning e crea una "corsa agli armamenti" tra chi sviluppa classificatori sempre piu robusti e antagonisti che cercano di manipolare tali classificatori. Fortunatamente ogni area di applicazione fornisce vincoli riguardo le azioni antagoniste che rendono la classificazione fattibile.

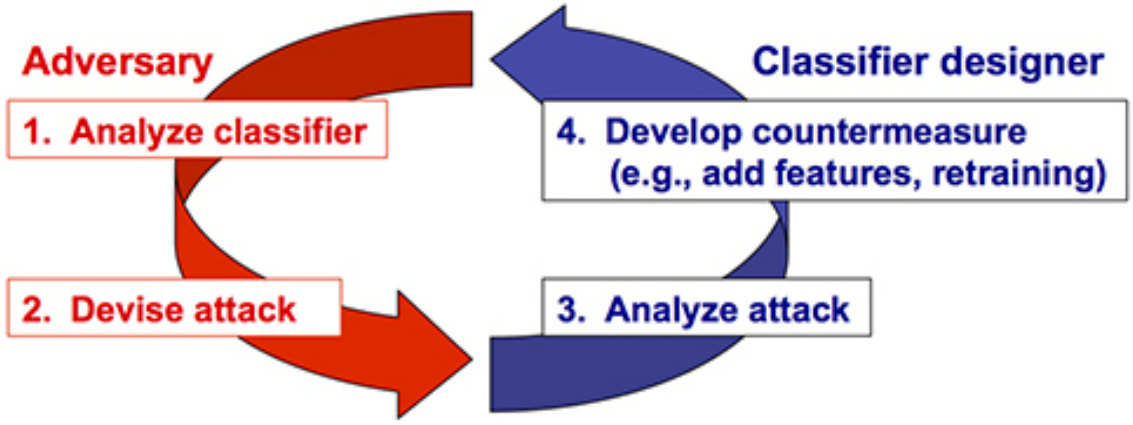


Figura 2.2: Schema concettuale del rapporto tra difensori ed antagonisti. *fonte* [11]

2.3 Machine Learning

Il Machine Learning è un sottocampo dell'intelligenza artificiale (IA) è stato definito da Arthur Samuel nel 1959 [12]. L'obiettivo del Machine Learning generalmente è capire la struttura dei dati e adattare tali dati in modelli in grado di essere capiti e utilizzati da utenti.

Nonostante il Machine Learning sia un campo informatico, si differenzia dai tradizionali approcci computazionali. Nell'informatica tradizionale, gli algoritmi sono serie di istruzioni programmate esplicitamente usate dai calcolatori per elaborare dati o risolvere problemi. Gli algoritmi Machine Learning, al contrario, permettono ai calcolatori di allenarsi su input di dati ed usare analisi statistica in modo da produrre valori che ricadono all'interno di intorni specifici. Grazie a questo, il Machine Learning facilita ai calcolatori la costruzione di modelli che siano in grado di automatizzare processi decisionali basati su campioni di dati.

Molti campi tecnologici odierni sfruttano i benefici del Machine Learning; ad esempio tecnologie di riconoscimento facciale permettono alle piattaforme di social networks di aiutare gli utenti nel tag delle foto di amici. Altro esempio sono le tecnologie di Optical Character Recognition (OCR) in grado di convertire immagini di testi stampati in documenti testuali modificabili da un word processor. I recommendation engines, basati su Machine Learning, suggeriscono agli utenti i programmi televisivi o film basati sulle loro preferenze. Le auto a guida autonoma si basano su Machine Learning in molte applicazioni, ad esempio per il riconoscimento della segnaletica stradale.

Nel Machine Learning i compiti sono generalmente classificati in grandi categorie. Tali categorie sono basate su come l'apprendimento è impostato o su come il feedback dell'apprendimento viene dato al sistema.

Due dei metodi di apprendimento più comunemente adottati sono l'apprendimento supervisionato in cui si allenano algoritmi basandosi su input e output di esempio, categorizzati da esseri umani e l'apprendimento non supervisionato, in cui non vengono forniti dati categorizzati agli algoritmi in modo da permettere al sistema di trovare autonomamente la struttura che compone i dati di input. Nelle sezioni seguenti si mostrano tali metodi in dettaglio.

2.3.1 Tipologie di Problemi

Il Machine Learning è impiegato principalmente per la risoluzione di tre tipologie di problemi:

- **classificazione**, ovvero identificare la classe di un nuovo obiettivo sulla base di conoscenza estratta da un training set. I classificatori estraggono dal dataset un modello che utilizzano poi per classificare le nuove istanze. Se una singola istanza può essere espressa come un vettore in uno spazio numerico R^n il problema della classificazione può essere ricondotto alla ricerca delle superfici chiuse che delimitano le classi. Esistono molte tecniche per costruire un classificatore, in questa tesi si tratterà in particolare l'uso di Alberi di Decisione. Si tratta di un classificatore implementato attraverso una struttura ad albero; è quindi costituito da una struttura gerarchica che applica una strategia "dividi et impera" per classificare i dati. Data la sua struttura di albero può essere facilmente convertito in un set di regole, permettendo così di estrarre conoscenza. La ricerca delle curve che delimitano le classi viene ricondotta alla ricerca ricorsiva di punti di split. Ogni nodo interno rappresenta un vincolo che porta a una scelta determinando uno split, e ogni foglia indica la classe di appartenenza dell'elemento. Gli alberi sono tipicamente costruiti con strategie Greedy, che individuano di volta in volta il punto più conveniente in cui effettuare lo split. (Figura 2.3).

- **raggruppamento** (clustering), quando si vuole raggruppare i dati che presentano caratteristiche simili. Ad esempio, un sistema può raggruppare immagini di figure geometriche separando figure con 4 lati e i cui angoli sono a 90 gradi oppure tutte le figure con quattro lati in cui solo due sono paralleli, ecc. In marketing, ad esempio, il raggruppamento viene utilizzato per l'individuazione di clienti e mercati potenziali.

- **regressione**, cioè prevedere il valore futuro di un dato avendo noto il suo valore attuale. Un esempio è la previsione della quotazione delle valute o delle azioni di una società. Nel marketing viene utilizzato per prevedere il tasso di risposta di una campagna sulla base di un dato profilo di clienti; nell'ambito commerciale per stimare come varia il fatturato dell'azienda al mutare della strategia.

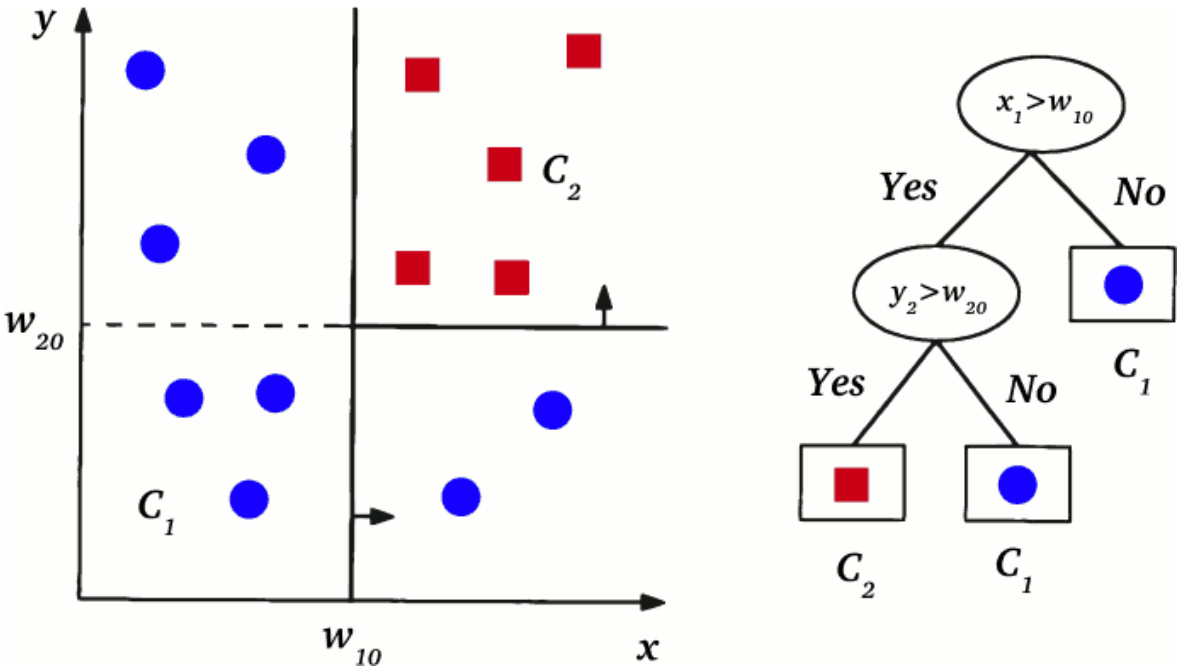


Figura 2.3: Utilizzo di un albero per la classificazione di un dataset

2.3.2 Apprendimento Supervisionato

Nell'apprendimento supervisionato, vengono forniti al sistema dei dati di input già etichettati con l'output desiderato. Lo scopo di tale metodo è permettere all'algoritmo di apprendimento di imparare comparando il suo output con quello fornito in ingresso, in modo da rilevare errori e modificare il modello di conseguenza. L'apprendimento supervisionato perciò usa patterns per predire i valori delle "etichette" su dati non etichettati.

Ad esempio, con l'apprendimento supervisionato, si può fornire ad un algoritmo un insieme di immagini di squali, etichettati come $pesce$ e immagini di oceani etichettati come $acqua$. Allenandosi su tali dati, l'algoritmo supervisionato dovrebbe essere in grado di riconoscere immagini di squali ed etichettarle come $pesci$" ed immagini di oceani ed etichettarle come $acqua$.

Un comune uso dell'apprendimento supervisionato è l'uso di dati storici per predire probabili eventi futuri. Può usare informazioni statistiche sull'andamento del mercato azionario ed anticipare fluttuazioni future o utilizzato per filtrare email di spam. E' possibile utilizzare foto di cani etichettate per identificare la razza di un cane unicamente dalla sua foto.

2.3.3 Apprendimento non Supervisionato

Nell'apprendimento non supervisionato, i dati non sono etichettati. L'algoritmo viene lasciato libero di rilevare affinità tra i dati di input. Siccome i dati non classificati sono molto piu comuni che grandi quantità di dati etichettati, i metodi di Machine Learning che facilitano l'apprendimento non supervisionato sono considerabili come molto pregiati.

L'obiettivo dell'apprendimento non supervisionato può essere diretto, come la ricerca di pattern nascosti all'interno dei dati, ma può anche avere l'obiettivo di *feature learning*: l'apprendimento di caratteristiche particolari che permette al sistema di scoprire automaticamente le rappresentazioni necessarie a classificare i dati grezzi.

L'apprendimento non supervisionato viene comunemente utilizzato per dati transazionali. Ad esempio, nel caso reale di un grosso dataset di clienti e dei loro acquisti, un essere umano difficilmente riuscirebbe a ricavare gli attributi che legano i profili clienti e le tipologie dei loro acquisti. Tuttavia fornendo tali dati ad un algoritmo di apprendimento non supervisionato è possibile rilevare caratteristiche che legano determinati gruppi di clientela a determinati prodotti e di conseguenza prendere decisioni di marketing mirate.

L'apprendimento non supervisionato, grazie alla mancanza di un target da raggiungere, è in grado di analizzare dati che apparentemente non mostrano relazioni ed estrarne dati significativi. Viene spesso usato per eseguire *anomaly detection*, come l'utilizzo fraudolento di carte di credito; oppure è possibile eseguire classificazione di immagini (come nel precedente caso di razze canine) in maniera autonoma partendo da immagini non categorizzate.

2.3.4 Apprendimento Semi Supervisionato

Posto a metà tra il supervisonato e il non-supervisionato, l'apprendimento parzialmente supervisionato si basa su dati misti in cui una minima parte è già etichettata e una larghissima maggioranza è costituita da dati non etichettati. Questo approccio viene utilizzato per migliorare le previsioni fatte dalla macchina sui dati non etichettati e richiede, normalmente, l'intervento di un analista. L'approccio è principalmente usato nei problemi di classificazione e di raggruppamento o nella descrizione delle relazioni causa-effetto tra le variabili.

2.3.5 Apprendimento per Rinforzo

L'apprendimento per rinforzo (Reinforcement learning) è una tecnica di apprendimento automatico che punta a realizzare sistemi in grado di apprendere ed adattarsi ai cambiamenti dell'ambiente in cui sono immersi attraverso la distribuzione di una "ricompensa" detta rinforzo, data dalla valutazione delle prestazioni. Il suo funzionamento è attuato da tre componenti:

- un sistema logico di esecuzione (definito A), che sulla base dei dati in ingresso (Input) riesce a restituire un risultato (Output)
- un sistema logico di valutazione (definito B) che assegna un premio (se il risultato è corretto) o una penalità (se il risultato non è corretto) al sistema logico A
- un sistema logico di ottimizzazione (definito C) che osserva il comportamento di A e B e modifica il modello utilizzato da A per aumentare il premio e ridurre le penalità che B assegna ad A.

L'apprendimento per rinforzo è utilizzato in tutti quei campi in cui è essenziale che la macchina risponda ai cambiamenti dell'ambiente. Per questo è utilizzato frequentemente nella robotica, per controllare i movimenti degli automi, ma anche nelle auto a guida autonoma. Trova anche applicazione in ambiti industriali nella produzione e nel controllo qualità e in diversi altri settori.

2.4 Deep Learning e Reti neurali

Il Deep Learning è uno specifico sottocampo del Machine Learning, un diverso modo di apprendere rappresentazioni di dati, principalmente orientato verso l'apprendimento di successivi strati (*layers*) di rappresentazione via via più significativi. Il termine "deep" indica la presenza di questa moltitudine di strati che compongono il modello di rappresentazione dei dati; infatti il deep learning odierno comprende generalmente modelli composti da decine o centinaia di strati successivi di rappresentazione, i quali apprendono in maniera automatica tramite l'esposizione ai dati di allenamento. Per confronto, gli approcci di Machine Learning descritti in sezione 2.3 generalmente coinvolgono uno o due strati di rappresentazione dei dati. Difatti tali modelli vengono definiti anche come *shallow learning*.

All'interno del deep learning, queste rappresentazioni a più strati sono quasi unicamente apprese attraverso modelli definiti **reti neurali**, strutturati letteralmente in strati sovrapposti gli uni agli altri. Il termine "rete neurale" deriva dal campo della neurobiologia, tuttavia non si tratta di veri e propri modelli del funzionamento del cervello umano.

Come è possibile vedere in figura 2.5, la rete neurale trasforma l'immagine di una cifra numerica disegnata a mano in una rappresentazione via via più informativa riguardo il risultato finale. Si può considerare il deep leraning come una operazione di distillazione di

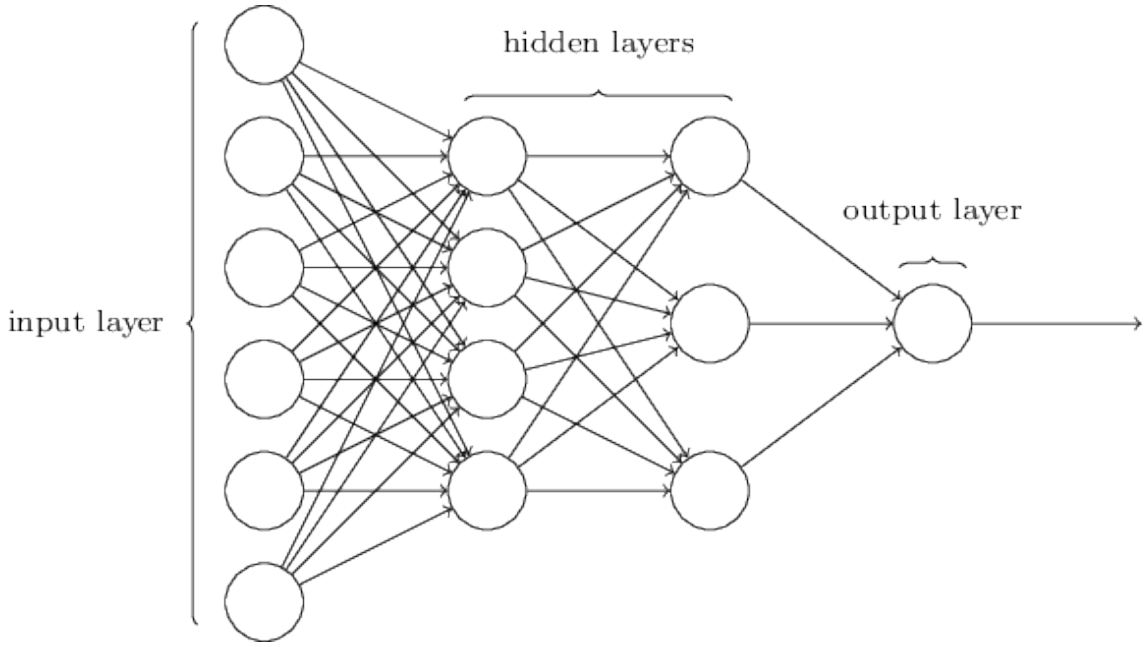


Figura 2.4: Schema generico di una rete Neurale

informazioni composta da più fasi in cui i dati vengono filtrati in maniera sempre più fine.

2.4.1 Funzionamento Reti Neurali

Le informazioni specifiche riguardo quali operazioni un livello apporta ai dati di input sono immagazzinate all'interno degli strati stessi, nella matrice dei "pesi". La trasformazione implementata da un livello è parametrizzata da tali pesi. Generalmente la fase di apprendimento consiste nel trovare un set di valori per i quali i pesi di tutti gli strati che compongono una rete neurale mappino correttamente i dati di esempio ai loro target associati. Tuttavia l'operazione di ricerca dei valori ottimali non è triviale, in quanto una rete neurale può contenere fino a decine di milioni di parametri e rilevarne i valori corretti risulta in una operazione complessa.

Per superare tale vincolo, la fase di apprendimento di una rete neurale è composta da più fasi di osservazione, nel quale si misura quanto distante si trova l'output da quanto atteso. Tale è il compito della funzione di *loss*. La funzione di loss utilizza la predizione della rete neurale e l'output atteso e computa un punteggio di distanza, catturando quindi la performance della rete sull'esempio analizzato.

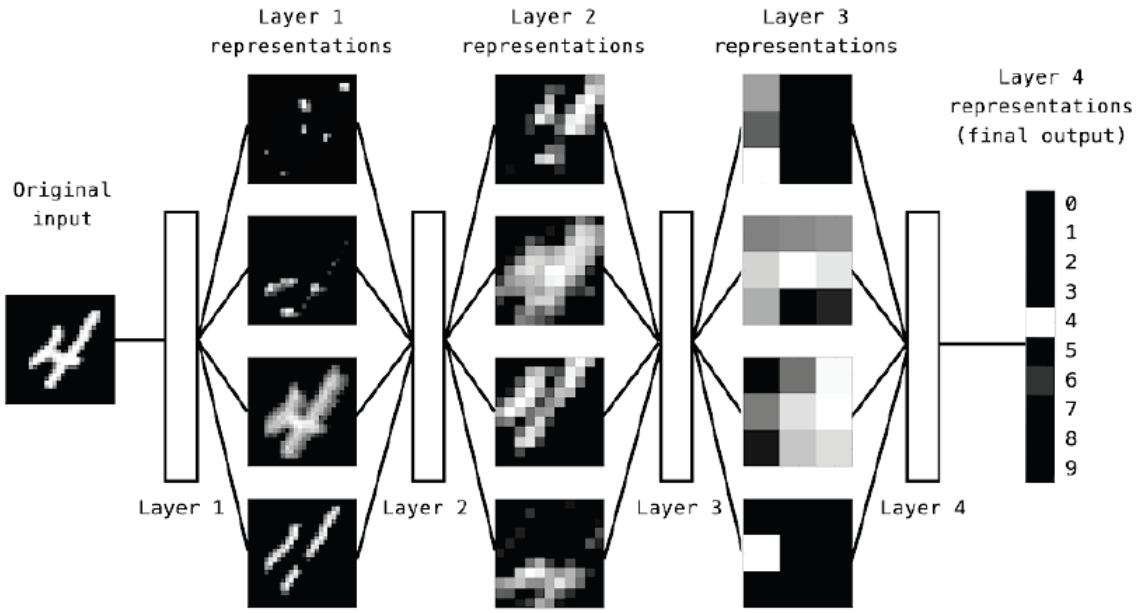


Figura 2.5: Esempio di rappresentazione di una cifra numerica tramite rete neurale. *fonte* [13]

Il meccanismo principale nel deep learning è l'uso di questo punteggio come segnale di *feedback* per aggiustare il valore dei pesi in maniera da diminuire il valore del punteggio di loss per l'esempio analizzato. Questo aggiustamento è compito dell'*ottimizzatore*, il quale implementa l'algoritmo di *backpropagation* che attua la modifica dei pesi durante la fase di apprendimento.

Inizialmente i pesi della rete neurale sono assegnati a valori randomici, causando per le prime fasi di apprendimento un valore di loss particolarmente alto. Via via che le fasi di apprendimento mostrano diversi esempi alla rete i pesi vengono aggiustati di una piccola parte nella direzione corretta, causando una decrescita nel punteggo di loss. Questo *training loop* viene ripetuto per un numero sufficiente di volte fino al raggiungimento di un valore minimo di loss e alla produzione di output il più possibile simile a quello desiderato. In figura 2.6 è mostrato uno schema di alto livello.

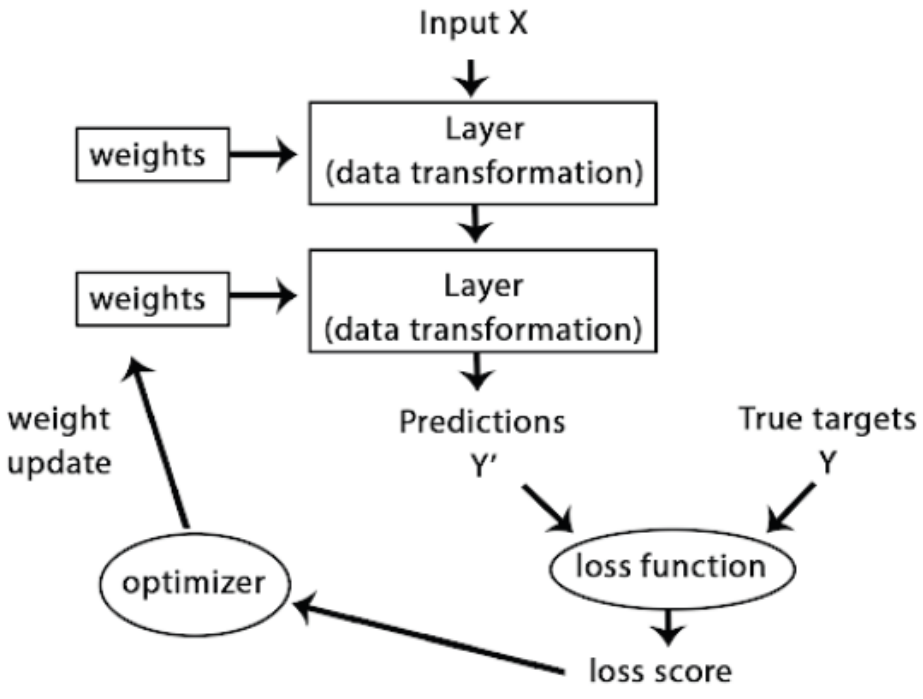


Figura 2.6: Schema di alto livello del funzionamento di una rete neurale. *fonte* [13]

2.5 Generative Deep Learning

I recenti risultati ottenuti nell'ambito del Machine Learning e del deep learning hanno permesso di ottenere algoritmi e architetture in grado di modellare complesse strutture di dati come immagini, suoni e testo. Questi avanzamenti sono stati incentrati principalmente in algoritmi di apprendimento supervisionato in grado di stimare una distribuzione di probabilità $p(x|y)$ dato un input ed un output di target. Tali modelli vengono definiti come modelli discriminativi o predditivi.

I modelli Generativi, al contrario, tentano di apprendere una funzione $p(y, x)$ nota come *probabilità congiunta*. Tali funzioni di probabilità sono correlate secondo la relazione:

$$p(x, y) = p(x)p(x|y)$$

dove $p(x)$ indica la densità di probabilità per l'evento x. Nel caso di modelli generativi, il modello ha accesso alla probabilità di input ed output allo stesso istante; permettendo ad esempio di generare immagini di animali, campionando specie di animali y e nuove immagini x a partire da $p(x, y)$. Il passo successivo è l'abilità di apprendere solo la funzione di densità $p(x)$, dipendente unicamente dallo spazio degli input. Tali algoritmi sono definiti come Modelli Generativi non Supervisionati (*Unsupervised Generative Models*), dove il termine *generativo* indica l'abilità di campionare dalla distribuzione catturata dal modello.

L'idea fondamentale alla base dei modelli generativi è la volontà di convertire il problema di generazione di dati in un problema di predizione ed usare l'insieme di tecniche di deep learning già note per risolvere tale problema. Gli algoritmi di deep learning odierni sono in grado di modellare mappature molto complesse e offrono flessibilità nel definire problemi in termini di grafi computazionali che possono essere ottimizzati da varianti di algoritmi di back-propagation.

2.5.1 Autoencoder

La forma più semplificata di conversione da problema generativo a problema discriminativo è l'apprendimento di una mappatura a partire dallo spazio degli input stesso. Sia dato un input x, si vuole ottenere una identità di tale input $x = f(x)$ dove f rappresenta il modello predittivo. Il modello generativo può essere definito come l'insieme formato da un modello

encoder $q_e(h|x)$ in grado di mappare l'input in un nuovo spazio, definito come *spazio latente* rappresentato da h ed un modello *decoder* $q_d(x|h)$ in grado di apprendere la mappatura inversa a partire dallo spazio latente. Queste due componenti possono essere connesse assieme, a formare un modello end-to-end allenabile, caratterizzato da un vincolo di dimensione nello spazio latente h; generalmente definito come una riduzione di dimensione con lo scopo di far apprendere al modello un vettore di caratteristiche fondamentali dell'input. (figura 2.7

Una volta che il modello è stato appreso, decoder ed encoder possono essere utilizzati indipendentemente, ad esempio per generare nuovi campioni a partire dallo spazio latente.

2.5.2 Generative Adversarial Networks

La naturale evoluzione delle architetture Autoencoder è la Generative Adversarial Network in cui encoder e decoder vengono rimodellati due reti :discriminatore e generatore.

Con l'obiettivo di apprendere la distribuzione del *generatore* p_g rispetto ai dati $\boldsymbol{x}$, si definisce precedentemente una variabile di rumore input $p_{\underline{\mathbf{Z}}}(\boldsymbol{z})$, e successivamente si rappresenta

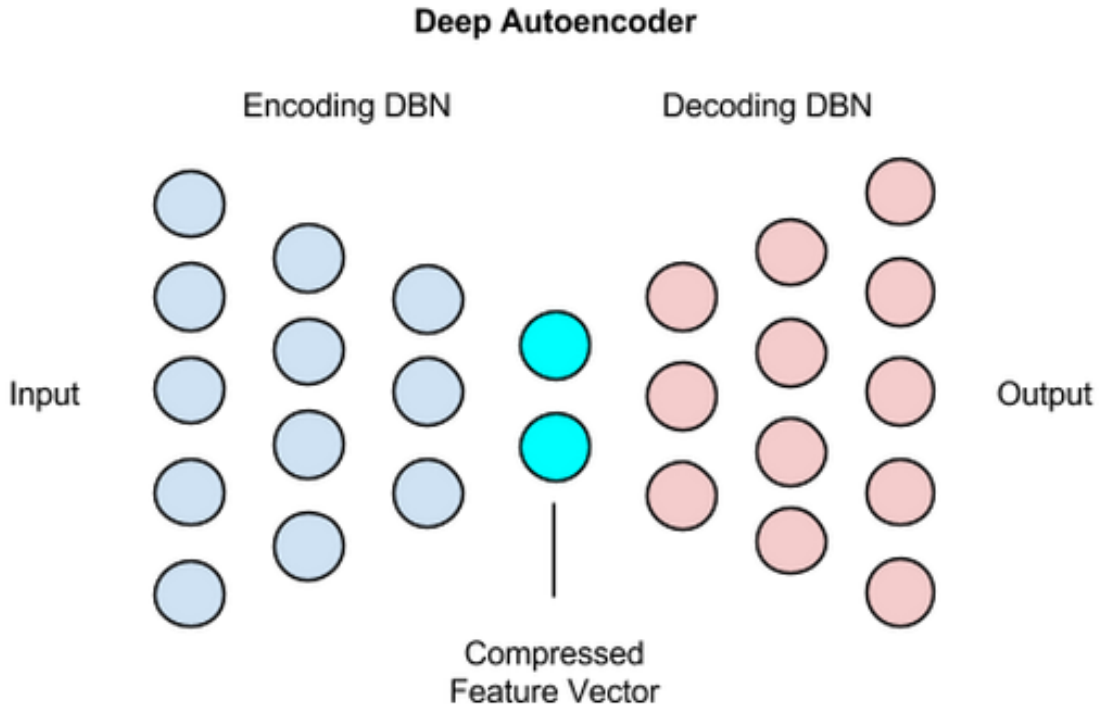


Figura 2.7: Schema concettuale di un Autoencoder.

una mappatura dello spazio dei dati come $G(\boldsymbol{z}; \theta_g)$, dove G è una funzione differenziabile rappresentata da una rete neurale a parametri θ_g. Si definisce inoltre una seconda rete neurale *discriminatore* $D(\boldsymbol{x}; \theta_d)$ che emette un singolo scalare. $D(\boldsymbol{x})$ rappresenta la probabilità che tale $\boldsymbol{x}$ provenga dai dati reali piuttosto che da p_g. A tal scopo si allena D per massimizzare la probabilità di assegnare l'etichetta corretta sia ai campioni di allenamento che a quelli provenienti da G. Simultaneamente si allema G per minimizzare $\log(1 - D(G(\boldsymbol{z})))$:

In altri termini, D e G eseguono una partita a due giocatori di minimo-massimo con funzione di valore $V(G, D)$:

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

In pratica è necessario implementare la partita in maniera iterativa. L'ottimizzazione di D fino al completamento in un training a se stante risulta computazionalmente proibitivo e su dataset finiti risulta nel fenomeno di *overfitting*. In maniera differente si alternano k passi di ottimizzazione di D ed un passo di ottimizzazione di G. Questo risulta nel mantenimento di D vicino alla sua soluzione ottimale, cambiando gradualmente G. Questa strategia è mostrata formalmente all'interno dell'algoritmo 1.

In pratica l'equazione 2.1 può non fornire un gradiente sufficiente per permettere a G di apprendere. Nelle fasi iniziali dove G è debole, D può rifiutare i campioni con buona certezza in quanto chiaramente differenti dai dati di training. In tal caso, $\log(1 - D(G(\boldsymbol{z})))$ satura. Piuttosto che allenare G al fine di minimizzare $\log(1 - D(G(\boldsymbol{z})))$ si allena per massimizzare $\log D(G(\boldsymbol{z}))$. Questa funzione obiettivo risulta nello stesso punto fisso che forma la dinamica di G and D ma fornisce un gradiente molto più stabile nelle fasi iniziali.

In figura 2.8 viene mostrata la fase di apprendimento di una GAN. Tale architettura è allenata aggiornando simultaneamente il gradiente della distribuzione discriminativa (D, linea blu tratteggiata) in modo che discrimini tra campioni della distribuzione generatrice di dati (linea nera tratteggata) $p_{\boldsymbol{x}}$ e i campioni provenienti dalla distribuzione del generatore p_g (G) (linea verde). La linea nera orizzontale da cui $\boldsymbol{z}$ è campionato è in questo caso uniforme; mentre la linea nera superiore è parte del dominio di $\boldsymbol{x}$. Le frecce indicano come la mappatura $\boldsymbol{x} = G(\boldsymbol{z})$ imponga una distribuzione non uniforme p_g sui campioni trasformati. G

si contrae in regioni ad alta densità ed espande in regioni a bassa densità p_g. Di seguito la descrizione di quanto mostrato in figura 2.8

- (a) si considera una coppia avversaria prossima alla convergenza: p_g è simile a p_{data} e D è un classificatore parzialmente accurato.
- (b) Nell'iterazione interna dell'algoritmo, D è allenato a discriminare dati convergenti da:

$$D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

- (c) dopo un aggiornamento a G, il gradiente di D ha guidato $G(\boldsymbol{z})$ verso regioni dove sia più probabile che sia classificato come dato.
- (d) dopo numerosi step di training se G e D hanno sufficiente capacità, raggiungono entrambi un punto in cui entrambi non possono migliorarsi a causa di

$$p_g = p_{\text{data}}$$

  Il discriminatore non è in grado di differenziare tra le due distribuzioni. (ad esempio: $D(\boldsymbol{x}) = \frac{1}{2}$).

Il minimo globale del criterio di training virtuale $C(G)$ è ottenuto solo se $p_g = p_{\text{data}}$. A tal punto, si raggiungerà il valore $C(G) = -\log 4$. Se G e D hanno sufficiente capacità e ad ogni step dell'algoritmo 1 il discriminatore è in grado di raggiungere il suo ottimo dato G, ed inoltre se p_g venga aggiornato così da migliorare il criterio

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D^*_G(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D^*_G(\boldsymbol{x}))]$$

allora p_g convergerà a p_{data}.

2.5.2.1 Vantaggi e Svantaggi

Questo framework moderno presenta vantaggie e svantaggi relativi ai precedenti framework. Gli svantaggi primariamente sono che non esiste una esplicita rappresentazione di $p_g(\boldsymbol{x})$, e che D deve essere ben sincronizzato con G durante la fase di training (in particolare,

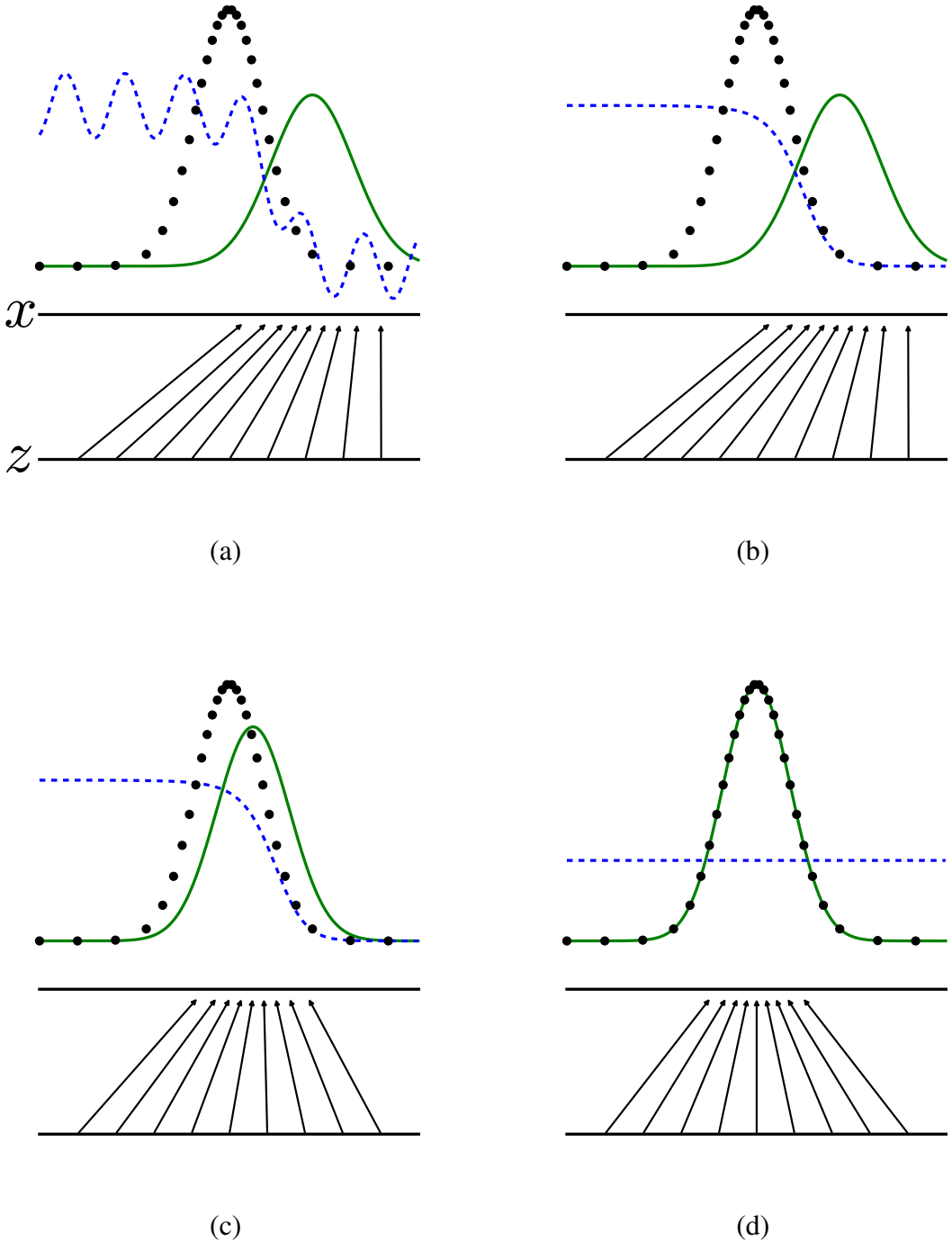


Figura 2.8

G non deve essere allenato troppo senza aggiornare D, in modo da evitare uno scenario degenerativo in cui G collassa troppi valori di $\mathbf{z}$ verso lo stesso valore di $\mathbf{x}$ per ottenere abbastanza diversità da modellare p_{data}). I vantaggi rispetto a modelli alternativi come le Catene di Markov sono principalmente di tipo computazionale. I modelli antagonisti possono trarre un vantaggio statistico non dovendo aggiornare la rete generatrice con campioni di dati ma solamente tramite il flusso di gradiente che attraversa il discriminatore. Questo significa che le componenti dell'input non sono copiate direttamente nei parametri del generatore. Un ulteriore vantaggio delle reti antagoniste è la capacità di rappresentare distribuzioni molto nette mentre metodi alternativi basati su Catene di Markov richiedono che la distribuzione sia meno distinta in modo da garantire alle catene la capacità di mischiare le modalità.

Algorithm 1 Discesa stocastica del gradiente durante la fase di training di una GAN. il numero di passi applicati al discriminatore, k, è un iperparametro della rete.

for numero di iterazioni di trainingons **do**

for k passi **do**

• Campionare un mini-batch di m campioni generati da rumore $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ dal rumore precedente $p_g(\boldsymbol{z})$.

• Campionare un mini-batch di m esempi $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ dalla distribuzione di dati reali $p_{\text{data}}(\boldsymbol{x})$.

• Aggiornare il discriminatore aumentando il proprio gradiente stocastico:

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

end for

• Campionare un mini-batch di m campioni generati da rumore $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ dal rumore precedente $p_g(\boldsymbol{z})$.

• Aggiornare il generatore, aumentando il proprio gradiente stocastico:

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

end for

Gli aggiornamenti di gradiente possono utilizzare qualsiasi legge di apprendimento.

Capitolo 3

Progetto Classificatore

riscrivere

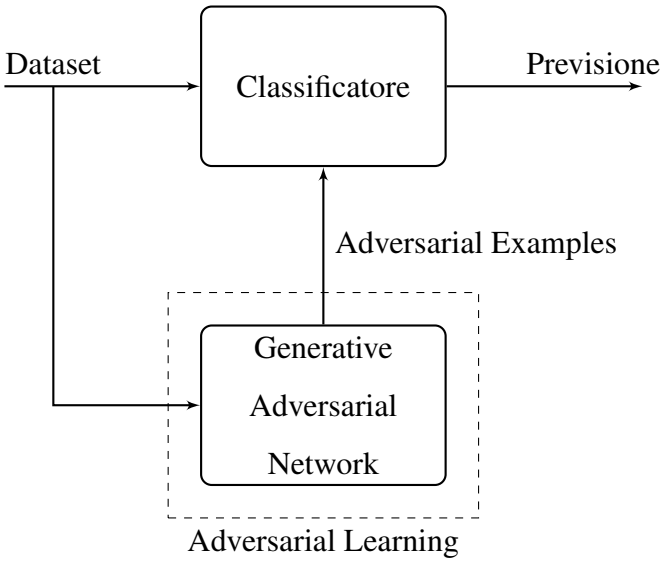


Figura 3.1: Diagramma generale di progetto.

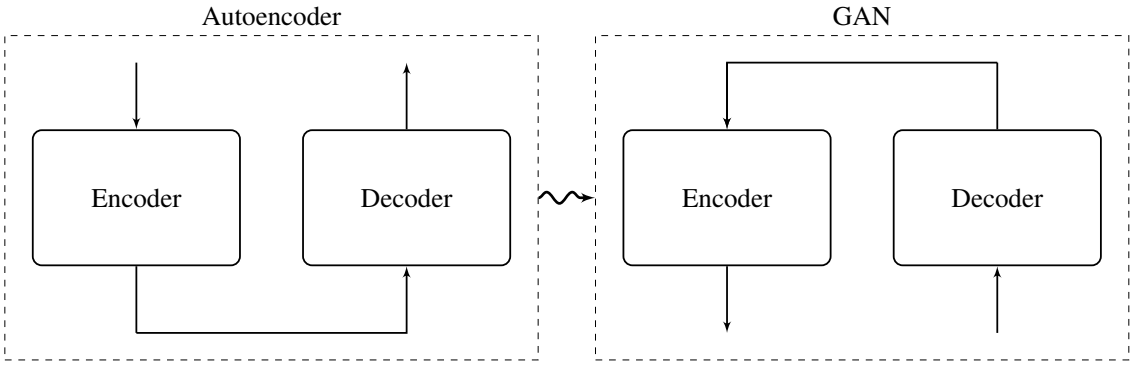


Figura 3.2: Schema generale della progettazione da Autoencoder a Gan.

3.1 Classificatore Machine Learning

La prima fase di questo studio è stata quella di progettare 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 dalle 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 mes-

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.* [15] , per un totale di 1 milione di siti realmente esistenti; mentre grazie al repository fornito da [16] 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.

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: [18] [19] [20] [21]. Di seguito viene illustrato l'insieme di tali *features*:

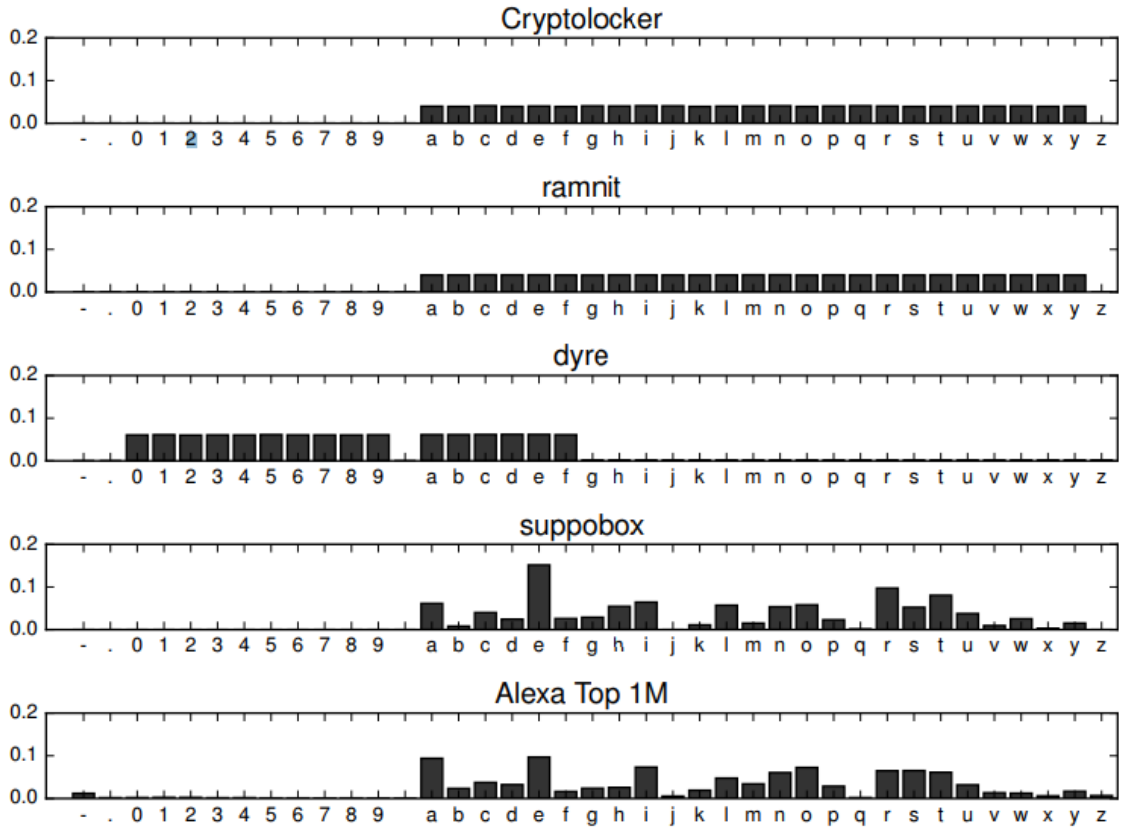


Figura 3.3: Distribuzione dei singoli caratteri in *cryptolocker, ramnit, dyre*, *suppobox* (dictionary-based DGA) ed il primo milione di domini più visitati secondo Alexa [15]. *fonte:* [17].

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.

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 Architettura Interna

La scelta della tipologia di architettura da utilizzare per il classificatore è stata ridotta a tre modelli di classificazione differenti:

- **Random Forest** [14]: un metodo di *ensemble learning* [22] supervisionato per la classificazione che opera tramite la costruzione di un insieme di alberi decisionali durante la fase di training ed emette la classe che rappresenta la moda statistica delle classi individuate dai singoli alberi. Formalmente è definita dalla seguente funzione: Per un punto x, sia $v_j(x)$ il nodo terminale a cui x è assegnato durante la discesa dell'albero T_j $(j = 1, 2, \ldots, t)$. Sia indicata da $P(c|v_j(x))$ la probabilità a posteriori che x appartenga alla classe $c(c = 1, 2, \ldots, n)$ come:

  $$P(c|v_j(x) = \frac{P(c, v_j(x))}{\sum_{l=1}^{n} P(c_t, v_j(x))}$$

  Tale probabilità può essere stimata dalla frazione di punti di classe c rispetto a tutti i punti che sono assegnati a $v_j(x)$. La funzione di discriminazione è definita da:

  $$g_c(x) = \frac{1}{t} \sum_{j=1}^{t} \hat{P}(c|v_j(x))$$

  e la regola decisionale è assegnare x alla classe c per cui $g_c(x)$ è massimo.

- **Support Vector Machine** [23]: un metodo di apprendimento supervisionato per la classificazione basati sul concetto di piani decisionali, che definiscono i limiti di decisione. L'algoritmo costruisce un iperpiano o un set di iperpiani in uno spazio multidimensionale, usato per la classificazione. Un esempio è mostrato in figura **??**

Formalmente è definito da: Dati i vettori di training $x_i \in \mathbb{R}^l, i = 1, \ldots, n$ in due classi e un vettore $y \in \{1, -1\}^n$ il classificatore risolve il problema:

$$\min_{\omega, b, \zeta} \frac{1}{2}\omega^T\omega + C\sum_{i=1}^{n}\zeta_i$$

$$\text{soggetto a } y_i(\omega^T\phi(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, \ldots, n$$

il cui duale è definito da:

$$\min_{\alpha} \frac{1}{2}\alpha^T Q\alpha - e^T\alpha$$

$$\text{soggetto a } y^T\alpha = 0$$

$$0 \leq \alpha_i \leq C, i = 1, \ldots, n$$

dove e è un vettore di 1, $C > 0$ è il limite superiore, Q è una matrice $n \times n$ positiva semidefinita, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, dove $K(x_i, x_j) = \phi(x_i)^T\phi(x_j)$ è il kernel. I vettori di training sono implicitamente mappanti in uno spazio dimensionale più grande dalla funzione ϕ. La funzione di decisione è:

$$sgn\left(\sum_{i=1}^{n} y_i\alpha_i K(x_i, x) + \rho\right)$$

- **Naive Bayes**: Metodo di classificazione supervisionata basati sul teorema di Bayes con un assunto "*naive*" di indipendenza tra ogni coppia di feature. Data una variabile di classe y e un vettore di features indipendenti $x_1 \ldots, x_n$ il teorema di Bayes dimostra la seguente relazione:

$$P\left(y|x_1, \ldots, x_n\right) = \frac{P(y)P(x_1, \ldots, x_n|y)}{P(x_1, \ldots, x_n)}$$

Siccome $P(x_1, \ldots, x_n)$ è una costante data in ingresso è definita la seguente regola di classificazione:

$$\hat{(}y) = arg, \max_y P(y)\prod_{i=1}^{n} P(x_i|y)$$

Il classificatore è stato implementato e testato con i tre modelli. Si confrontino i risultati all'interno del capitolo 6.

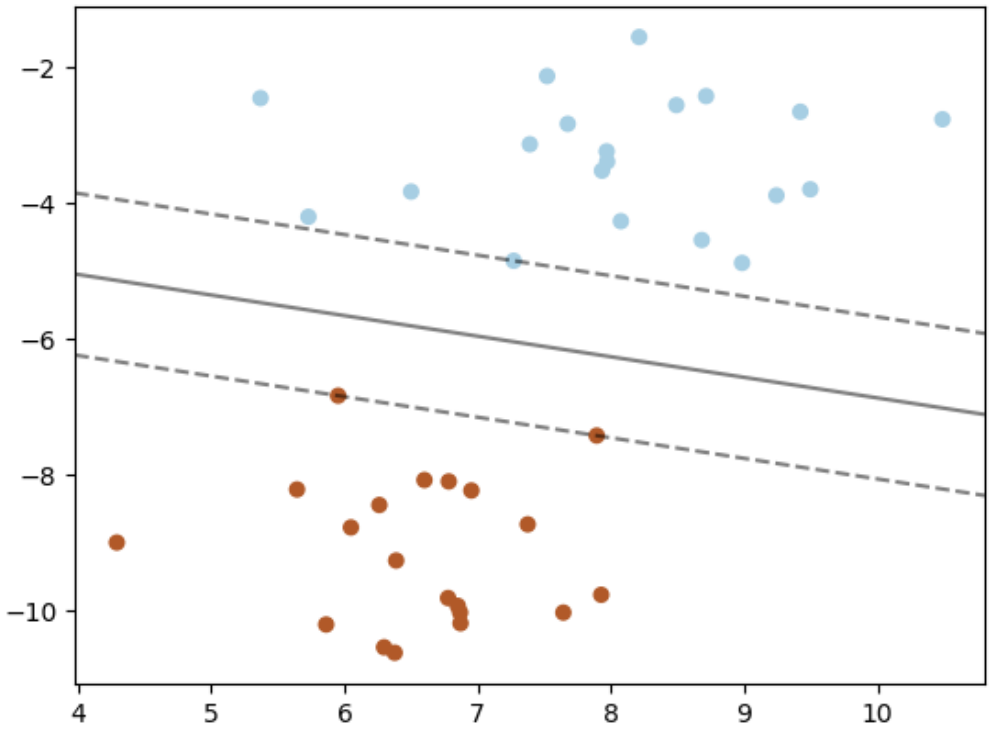


Figura 3.4: Esempio di iperpiano di una SVM.

3.1.4 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* [24] 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 [25] (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) [26] definita dalla funzione sottostante e mostrato in figura 3.5

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

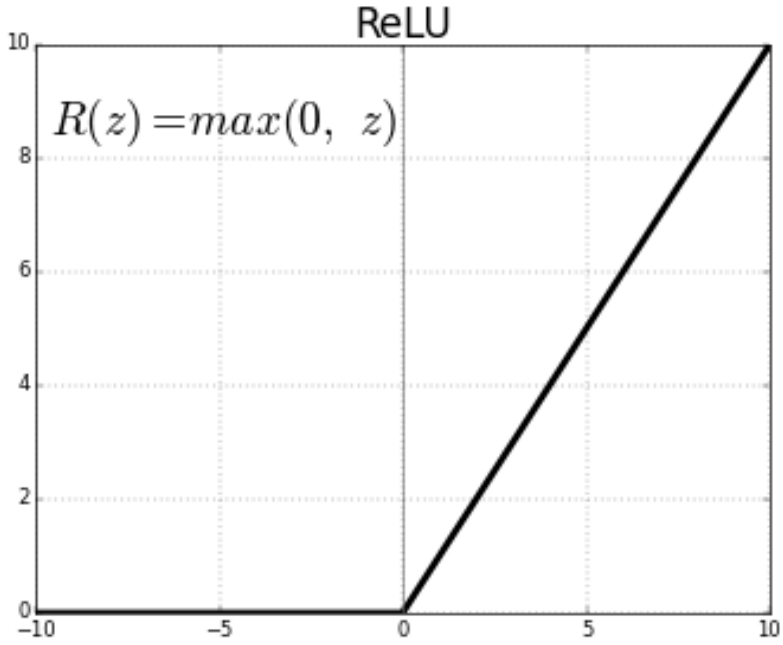


Figura 3.5: *fonte:* [27].

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

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

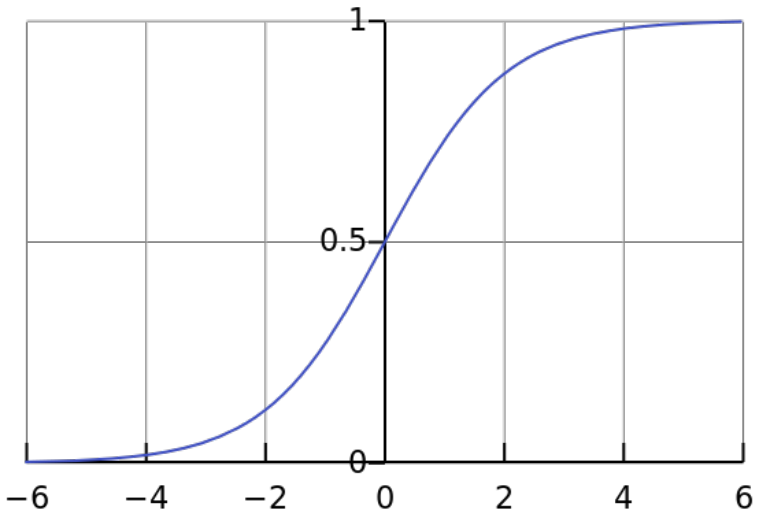


Figura 3.6: *fonte:* [28]

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, due layer intermedi di dimensioni comprese tra i valori ottenuti dal modello ridotto ed il modello allargato ed un layer finale di dimensione 1.

I tre modelli sono stati messi a confronto in fase sperimentale, dovendo procedere ad una fase di tuning dei parametri che compongono il modello intermedio in quanto non è possibile

in maniera automatica determinare quali siano i parametri ottimali per il MLP utilizzato nel caso in esame.

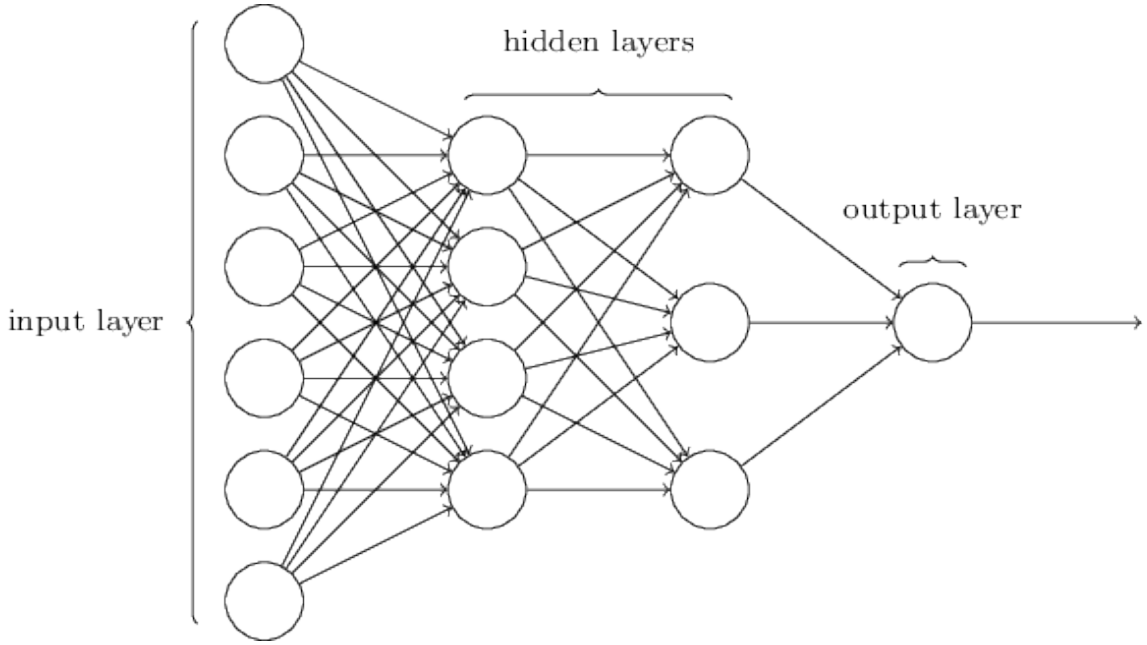


Figura 3.7: Schema generico di un Multilayer Perceptron

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.

Capitolo 4

Progetto Adversarial Learning

Ricerche precedenti hanno dimostrato che molti modelli di machine learning, incluse le reti neurali, sono vulnerabili agli *adversarial examples* [29], [30]. In particolare la ricerca proposta in [30] introduce il metodo del *fast gradient sign* per scoprire *adversarial examples* perturbando un campione noto x con una piccola quantità definita

$$\Delta x =\in sign(\nabla_x J(\theta, x, y))$$

dove θ rappresenta i parametri del modello e J il costo necessario a classificare x come y. Separatamente [31] propone l'uso di *Generative Adversarial Network (abbr. GAN*) come *framework* in grado di generare campioni artificiali provenienti dalla stessa distribuzione del 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 [17].

4.1 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 [32] [33]. 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.

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.

4.1.1 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

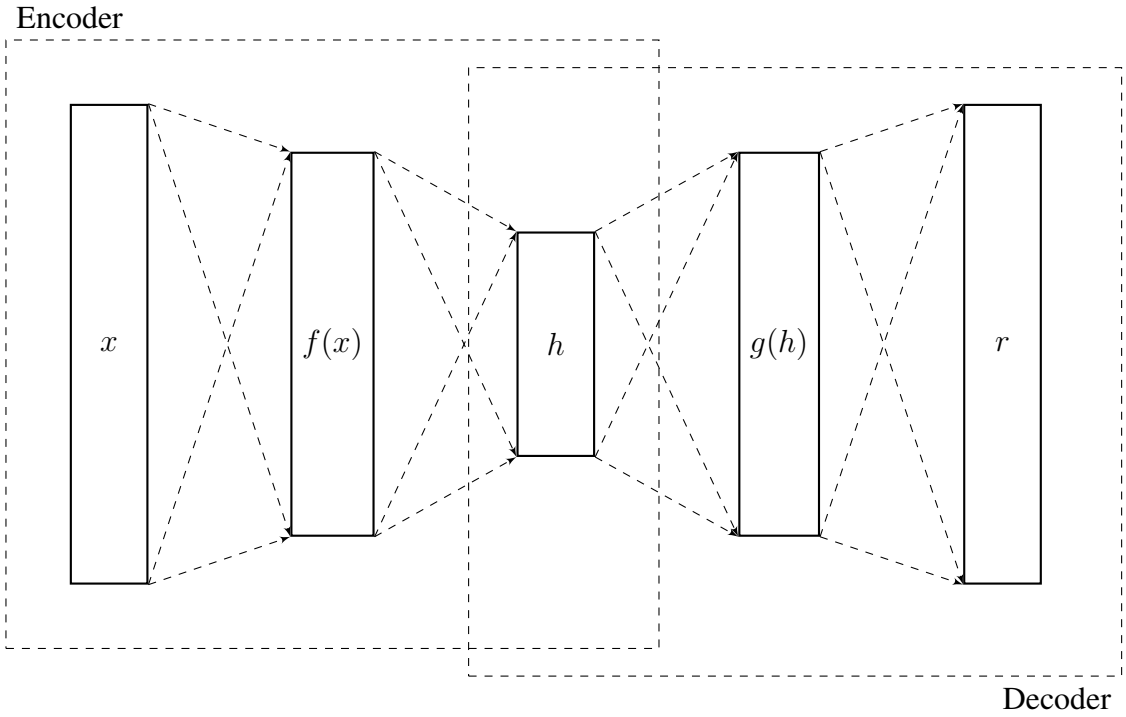


Figura 4.1: Struttura generica di un autoencoder, il quale mappa l'input x in un output r attraverso una rappresentazione codificata h.

preprocessing: i domini codificati in sequenze di valori interi sono stati ulteriormente codificati tramite il *one hot encoding* [34] 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*.

4.1.2 Architettura Autoencoder

L'architettura dell'*encoder* in esame è ispirato al lavoro mostrato in [35] 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 4.2 è mostrata la struttura di massima dell'autoencoder. Di seguito vengono illustrati in dettaglio le principali componenti che compongono l'*autoencoder*.

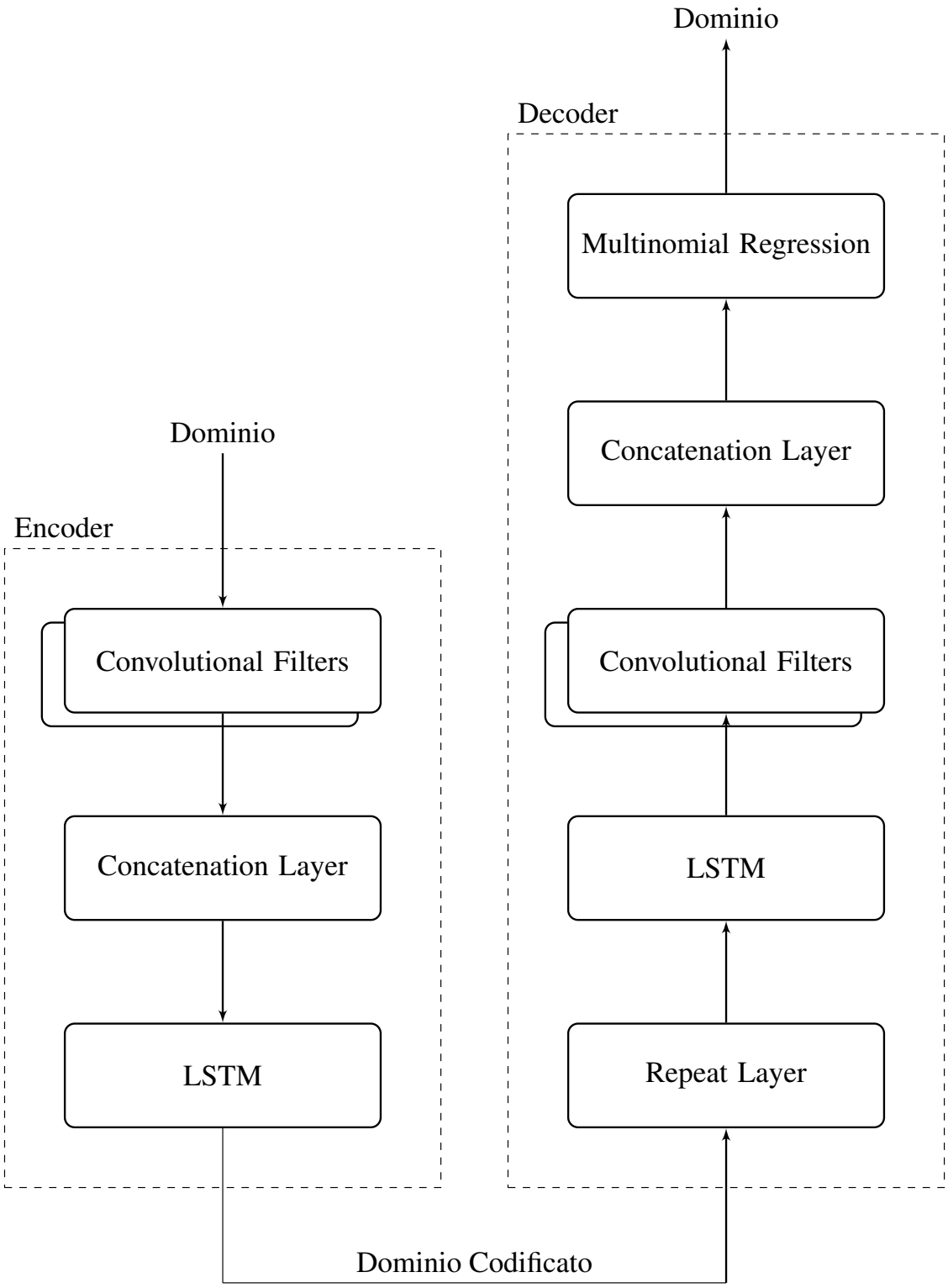


Figura 4.2: Struttura dell'*autoencoder* in esame. L'input in ingresso dato dai domini viene codificato attraverso l'*encoder* e dato in ingresso al *decoder* che ne genera una approssimazione.

4.2 Encoder

La composizione interna dell'*encoder* è formata da una Rete Convoluzionale accoppiata ad una LSTM. A differenza del lavoro proposto in [17], 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 4.3 viene mostrata la struttura semplificata dell'encoder.

4.2.1 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

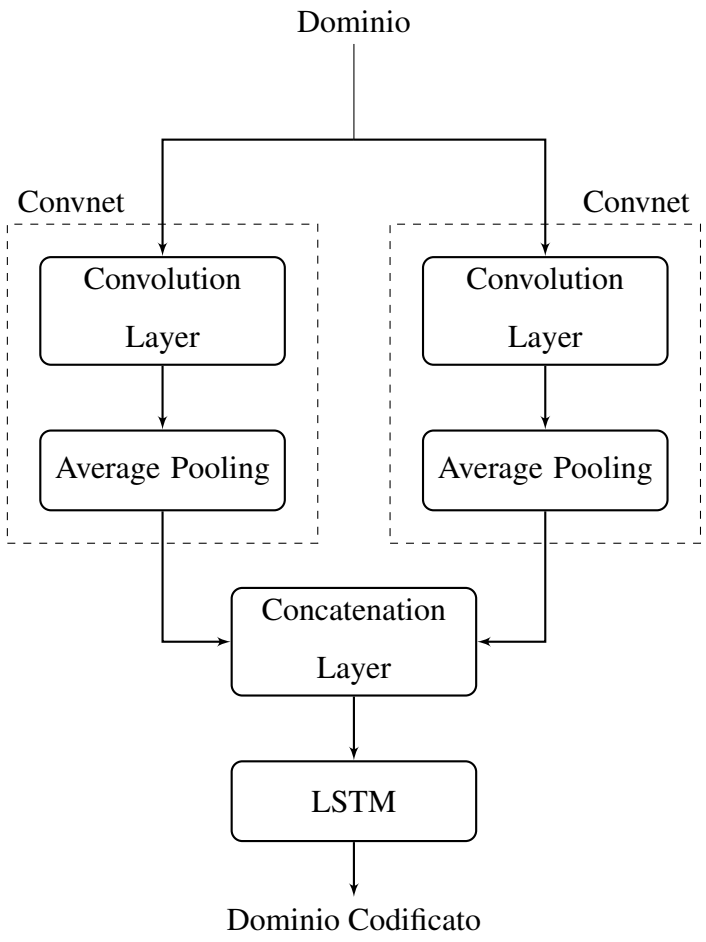


Figura 4.3: Struttura dell'*encoder*.

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 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 4.4 è possibile vedere un diagramma del funzionamento di una rete convoluzionale.

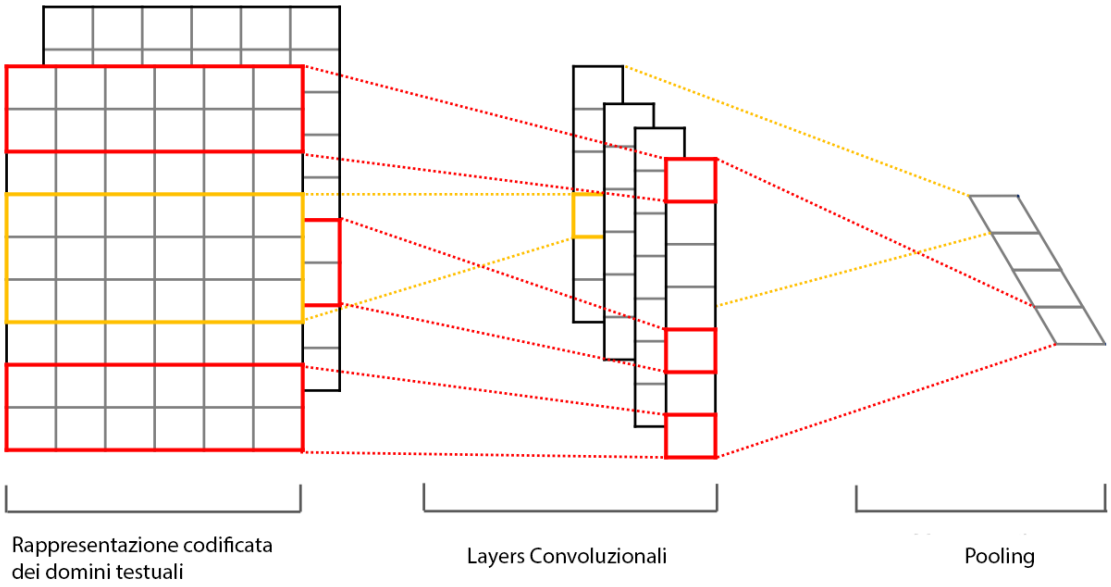


Figura 4.4: Funzionamento di una Convolutional Neural Network, *fonte* [36].

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

4.2.3 Concatenazione

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

4.2.4 Long Short-term Memory

Seconda fase della sottorete Encoder è la presenza di una Long Short-term Memory Network *(abbr LSTM)* [37]. 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 4.5 è indicata la struttura di una cella LSTM.

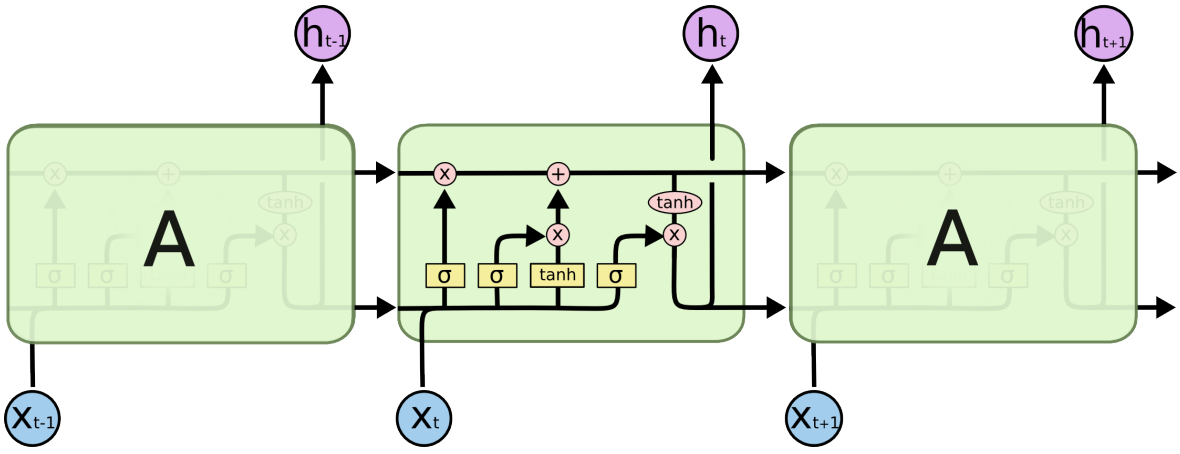


Figura 4.5: Struttura di una cella LSTM. *fonte:* [38].

4.2.5 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.

ampliare

4.3 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 4.6 viene mostrato lo schema di massima del *decoder*.

4.3.1 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.

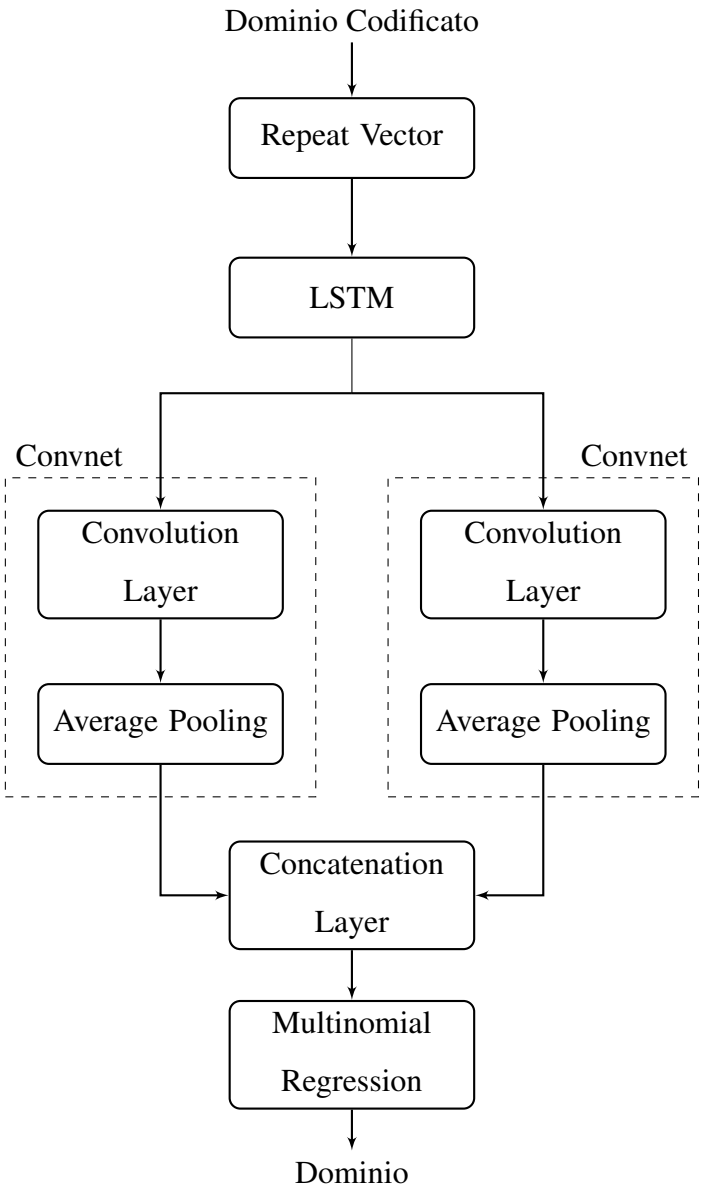


Figura 4.6: Struttura del *decoder*.

Per il campionamento è stato usata una funzione *softmax* utilizzata nel campo dell'apprendimento per rinforzo [39], 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.

4.4 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 4.7 si può vedere una struttura di massima di una GAN.

La struttura dell'autoencoder descritto a partire dalla sezione 4.1 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

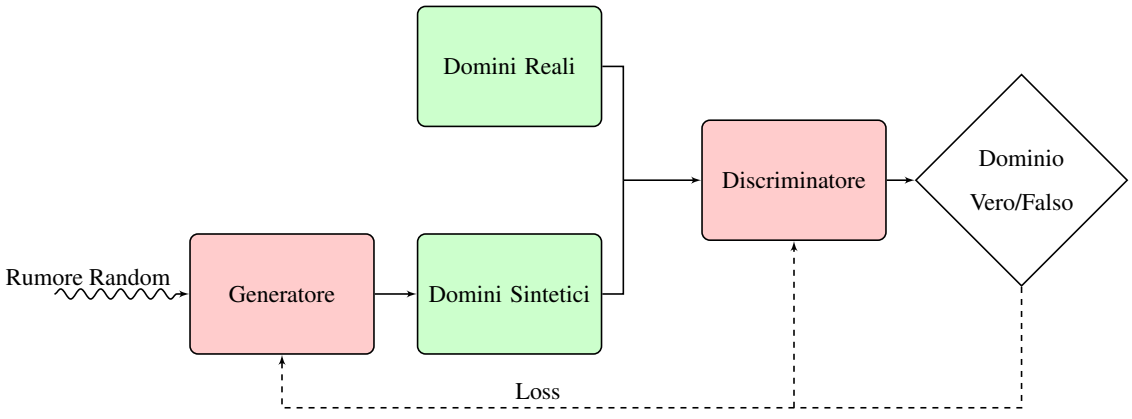


Figura 4.7: Struttura di una *Generative Adversarial Network*.

domini reali.

4.4.1 Architettura GAN

Come mostrato in figura 4.8, 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 [17], si è deciso di implementare una architettura più semplificata. Come mostrato in [40] 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 [41], l'utilizzo di Leaky Relu [42] come layer di attivazione all'interno della sottorete Convoluzionale e Dropout come strumento per evitare il fenomeno di *overfitting* [43]. L'uso di Leaky ReLU come funzione di attivazione è giustificata dalla riduzione della sparsità del gradiente, la quale generalmente causa instabilità.

4.4.1.1 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 [41]. 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*.

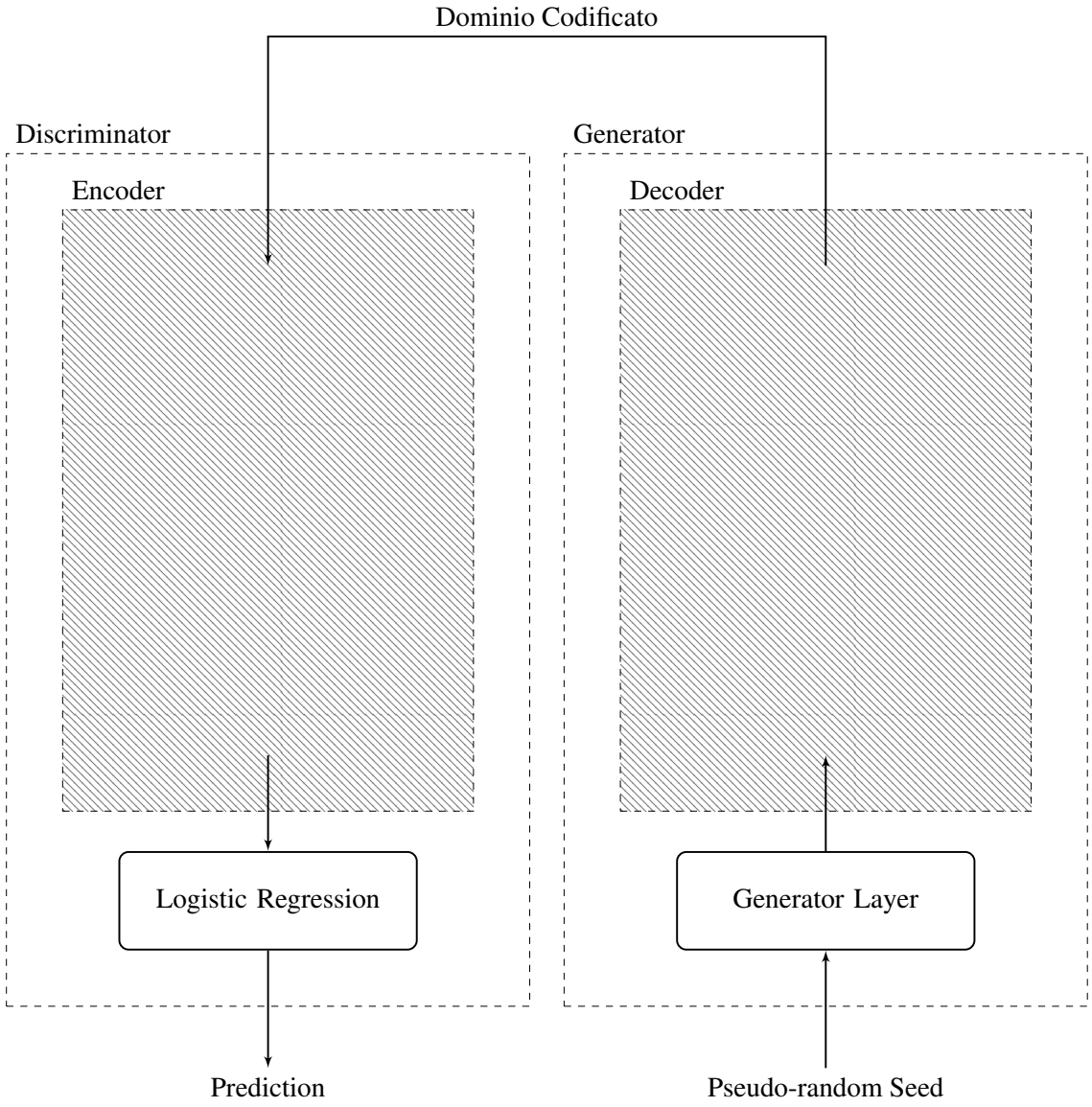


Figura 4.8: Struttura della GAN evoluta dell'autoencoder.

4.4.2 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 4.9. 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 [44]. Tale distribuzione, di dimensione fissata, mima un possibile dominio codificato da encoder come descritto in

sezione 4.2. 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.

4.4.3 Discriminatore

Il discriminatore implementa in uscita un layer denso che esegua regressione logistica sui domini codificati dalla sottorete encoder. (Figura 4.10). Tale regressione viene attuata tramite un layer denso che effettua classificazione tra domini reali, codificati durante la fase di encoding attuata in 4.2. 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 possibile evitare che il discriminatore prevalga sul generatore durante il training, mantenendo un equilibrio tra le due parti.

4.4.4 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.

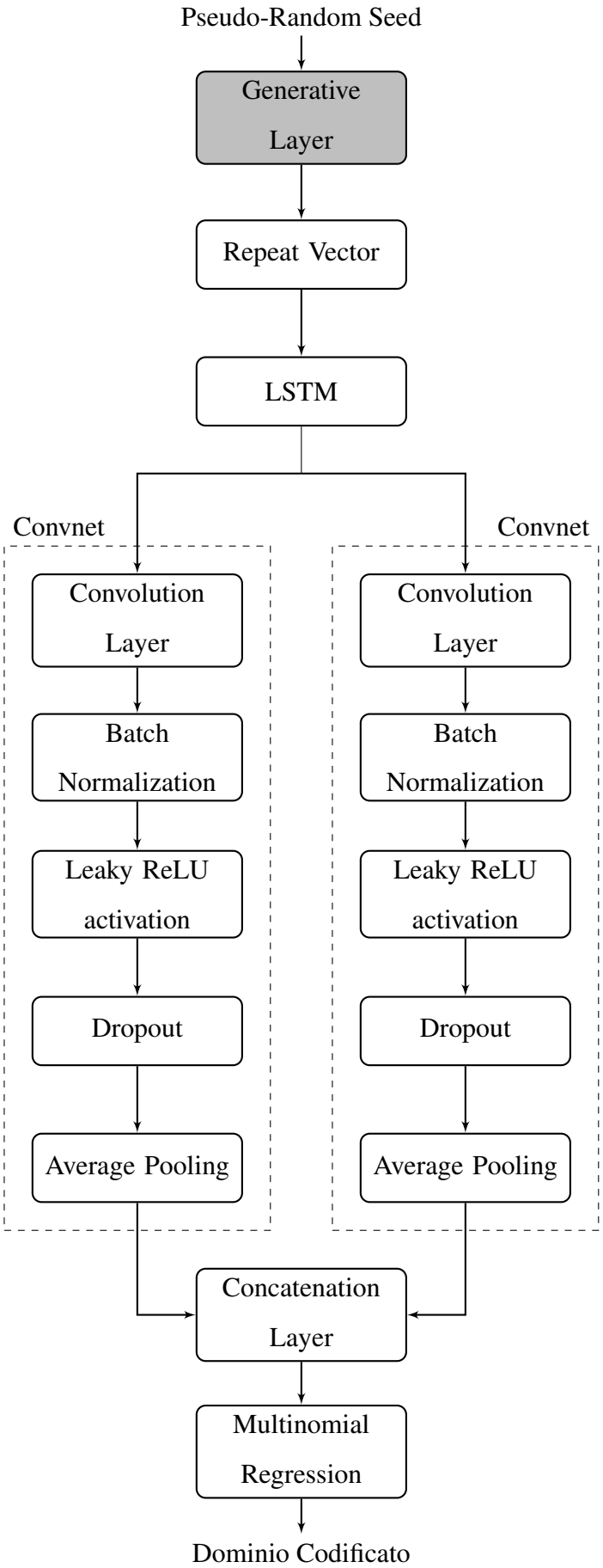


Figura 4.9: Struttura del generatore.

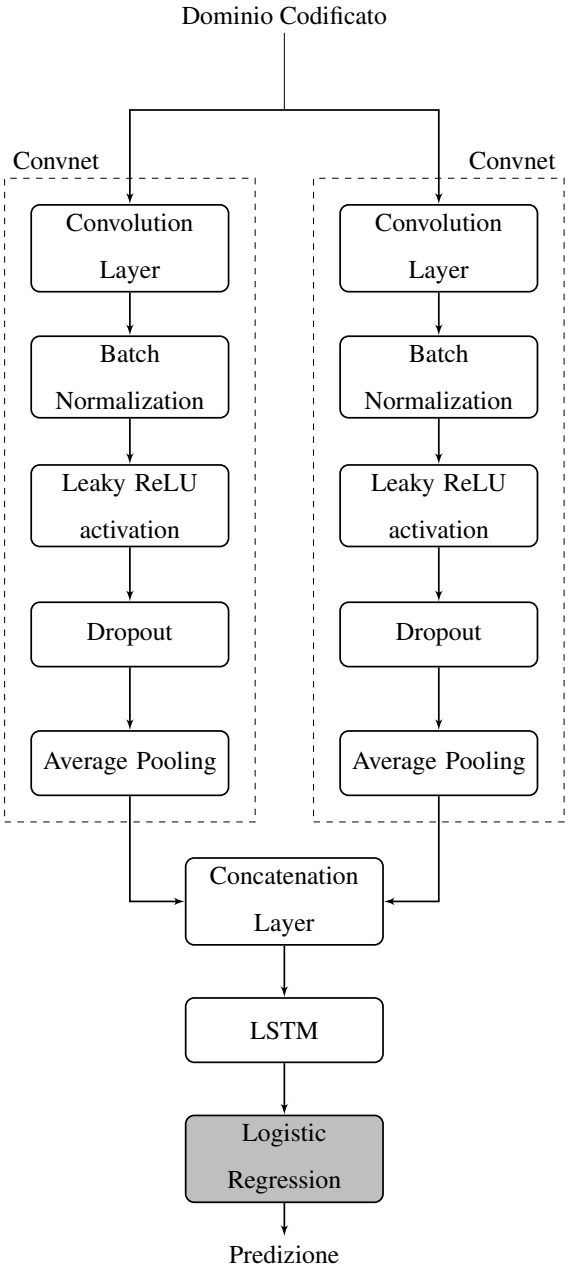


Figura 4.10: Struttura del discriminatore.

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 5

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 [45] per attuare machine learning e Keras [46] per realizzare reti neurali. La libreria per reti neurali su cui opera Keras è Tensorflow [47]. Di seguito sono elencati i dettagli implementativi delle singole entità, ed i parametri usati durante la fase sperimentale.

5.1 Classificatore Supervisionato

Il classificatore Random Forest è stato implementato tramite l'uso della libreria python Scikit-learn [45]. 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 5.1 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).

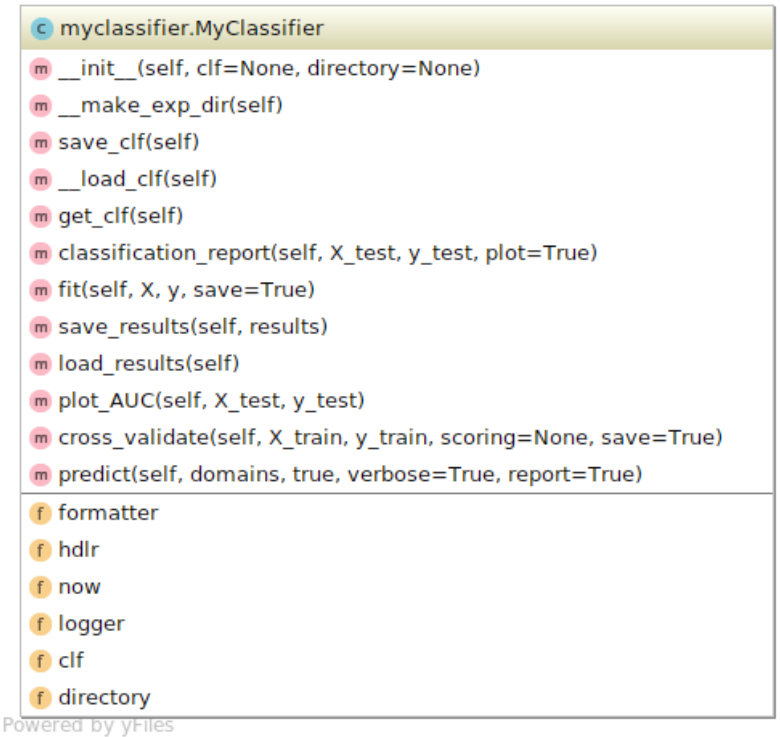


Figura 5.1: Diagramma UML della classe MyClassifier

5.1.1 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.

```
min_subtr = 3
maxl = 0
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 [21], 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 dimo-

strato 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.

5.1.2 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.

5.1.3 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.

5.2 Classificatore Neurale

L'implementazione del classificatore neurale è stata eseguita in ambiente Python, con l'utilizzo della libreria Keras [46] 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 5.1. 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.

5.2.1 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 5.1).

```
_________________________________________________________________
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
_________________________________________________________________
activation_3 (Activation)    (None, 1)                 0
=================================================
Total params: 429
Trainable params: 383
Non-trainable params: 46
_________________________________________________________________
```

Listing 5.1: Baseline Ridotta

- un secondo modello ampliato contenente input equivalente al precedente, due livelli interni di dimensione 128 ed una uscita con singolo neurone. (listato 5.2).

```
_________________________________________________________________
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, 128)               1920
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
activation_2 (Activation)    (None, 128)               0
```

```
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16384
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
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 5.2: Baseline ampliata

- un modello intermedio, contenente ingresso equivalente al precedente, due livelli interni da 128 e 64 e uscita con singolo neurone. (listato 5.3).

```
_________________________________________________________________
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
```

```
13 _________________________________________________________________
14 dropout_3 (Dropout)            (None, 64)                0
15 _________________________________________________________________
16 dense_4 (Dense)                (None, 1)                 65
17 =================================================
18 Total params: 10,609
19 Trainable params: 10,609
20 Non-trainable params: 0
21 _________________________________________________________________
```

Listing 5.3: Baseline intermedia.

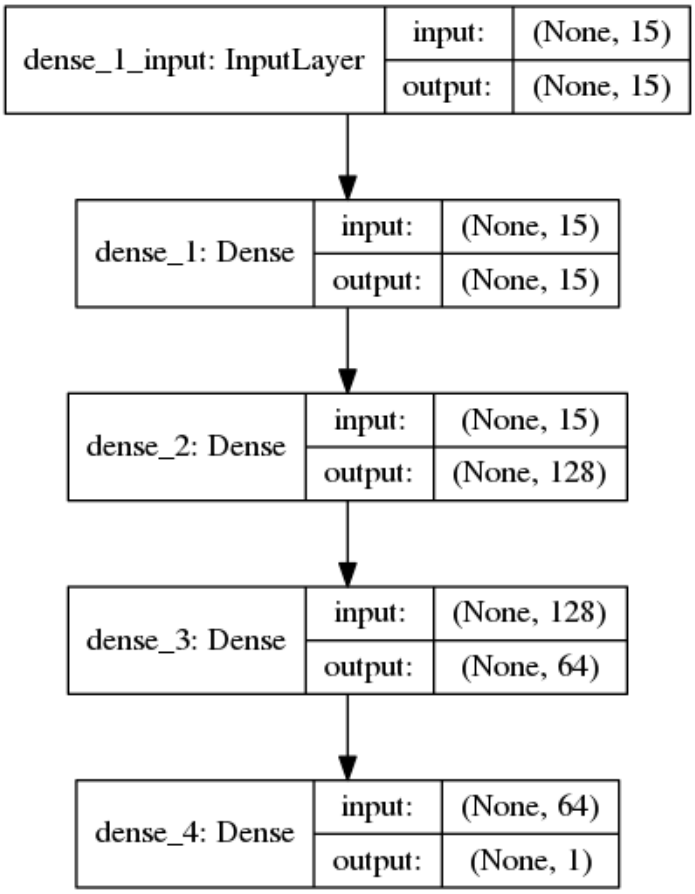


Figura 5.2: Grafico del modello intermedio. Escluso il layer di input iniziale, si notino gli *hidden layer* dense2 e dense3 di dimensioni rispettivamente 128 e 64.

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.

5.2.2 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 5.1).

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 5.1: Dizionaro di caratteri mappati.

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

5.2.3 Ottimizzatore

I modelli testati in questa sezione sono stati compilati utilizzando l'ottimizzatore *Adaptive Moment Estimation (abbr. adam)* [48]. Tale ottimizzatore fornisce generalmente le prestazioni migliori nel caso di classificazione binaria e pertanto si è optati per tale soluzione.

5.2.4 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

5.3 Autoencoder

L'implementazione dell'autoencoder descritto in sezione 4.1 è stata eseguita in ambiente Python con l'ausilio della libreria Keras per reti neurali.

5.3.1 Dataset

Il dataset utilizzato per il training dell'autoencoder è stato definito nella sezione 4.1.1. 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 [15]. Il fine infatti è ottenere un sistema che possa riprodurre al meglio i domini reali.

5.3.2 Encoder

La struttura dell'Encoder, definita all'interno della sezione 4.2 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.

```
1 dropout_value = 0.5
2 cnn_filters = [20, 10]
3 cnn_kernels = [2, 3]
4 cnn_strides = [1, 1]
```

```
leaky_relu_alpha = 0.2
timesteps = 15
word_index = 38
latent_vector = 20
```

Listing 5.4: 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 [17] 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)
```

```
14   enc_convs.append(conv)
15
16 discr = concatenate(enc_convs)
17 discr = LSTM(latent_vector)(discr)
18
19 E = Model(inputs=discr_inputs, outputs=discr, name='Encoder')
```

Listing 5.5: Struttura livelli Encoder.

5.3.3 Decoder

La composizione del decoder è pressochè identica a quella mostrata dall'encoder e ricalca il progetto mostrato in sezione 4.3. 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.

```
1 dropout_value = 0.4
2 cnn_filters = [20, 10]
3 cnn_kernels = [2, 3]
4 cnn_strides = [1, 1]
5 dec_convs = []
6 leaky_relu_alpha = 0.2
7 latent_vector = 20
8 timesteps = 15
9 word_index = 38
```

Listing 5.6: Iperarametri Decoder.

Per completezza si riporta l'implementazione del decoder:

```
1
2 dec_inputs = Input(shape=(latent_vector,),
3             name="Generator_Input")
4 decoded = RepeatVector(timesteps, name="gen_repeate_vec")(dec_inputs)
5 decoded = LSTM(word_index, return_sequences=True,
     name="gen_LSTM")(decoded)
6 decoded = Dropout(dropout_value)(decoded)
7 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 5.7: Struttura livelli Decoder.

5.3.4 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.

```
Layer (type)                 Output Shape              Param #
=================================================
Encoder (Model)         (None, 20)                6890
_________________________________________________________________
Decoder (Model)             (None, 15, 38)            12836
=================================================
Total params: 19,726
Trainable params: 19,666
Non-trainable params: 60
```

```
10 ____________________________________________________________________________________
```

Listing 5.8: 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.

5.4 Generative Adversarial Network

L'implementazione della Generative Adversarial Network (sezione 4.4, è stata evoluta a partire da quella eseguita per l'autoencoder.

5.4.1 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 $[-1.0, 1.0]$.

5.4.2 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:

```
1 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.

5.4.3 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* [49] np.random.normal con valori compresi tra $[-1.0, 1.0]$.

5.4.4 Training

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

In prima fase, come indicato da [17] 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 [50] 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 6

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 Generative Adversarial Network. Le metriche di valutazione utilizzate sono enunciate all'interno della sezione 6.1. In sezione 6.2 vengono mostrati i risultati sperimentali ottenuti dal classificatore random forest, punto di partenza di questo studio. In sezione 6.3 vengono mostrati i risultai sperimentali ottenuti dal classificatore neurale, evoluzione del classificatore random forest, e le problematiche incontrate durante tale fase. In sezione 6.4 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 (6.5) sono mostrati i risultati ottenuti dal training della Generative Adversarial Network e la lunga fase di affinamento degli iperparametri richiesta da quest'ultima.

6.1 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)

6.2 Classificatore Supervisionato

Il classificatore Random Forest è stato testato in diverse configurazioni. La configurazione che è stata presentata nella sezione 5.1 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.

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 6.1 e figura 6.2. 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 6.3 e 6.4.

Malware Families
legit
cryptolocker
zeus
pushdo
rovnix
tinba
conficker
matsnu
ramdo

Tabella 6.1

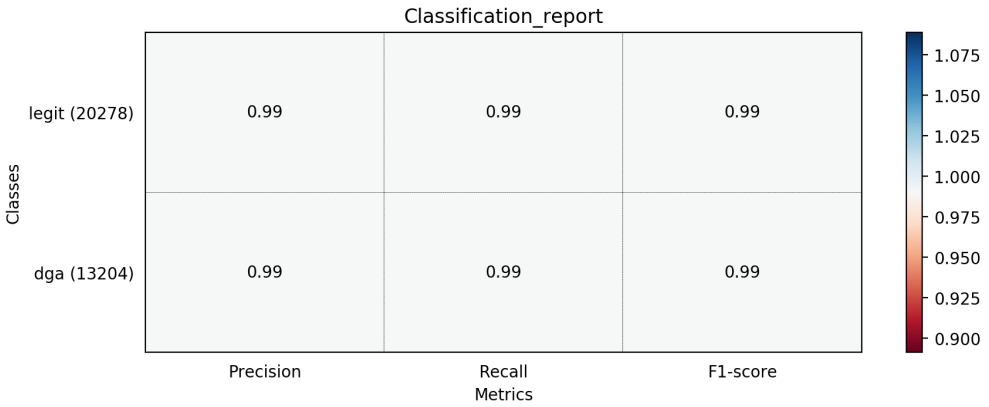


Figura 6.1: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malevoli (DGA).

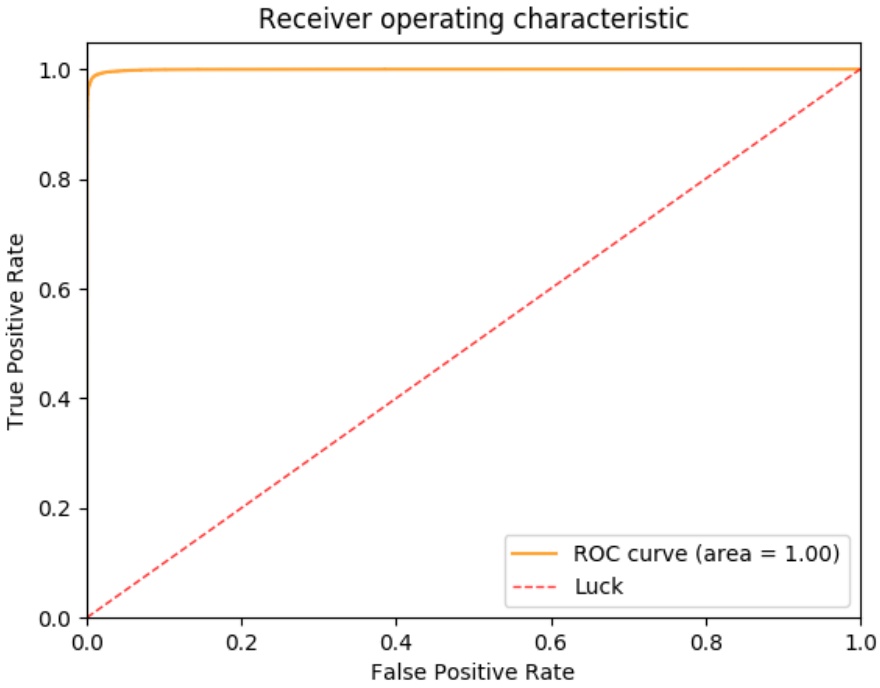


Figura 6.2: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con domini di tabella 6.1.

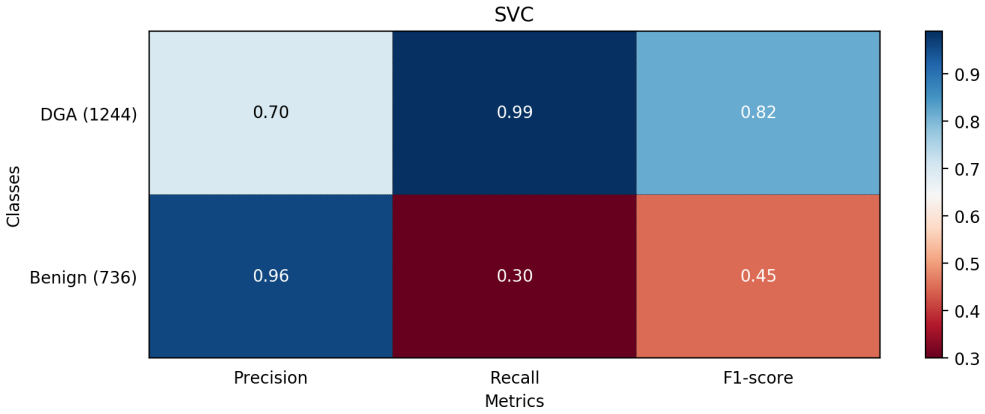


Figura 6.3: Report di classificazione per l'algoritmo SVC.

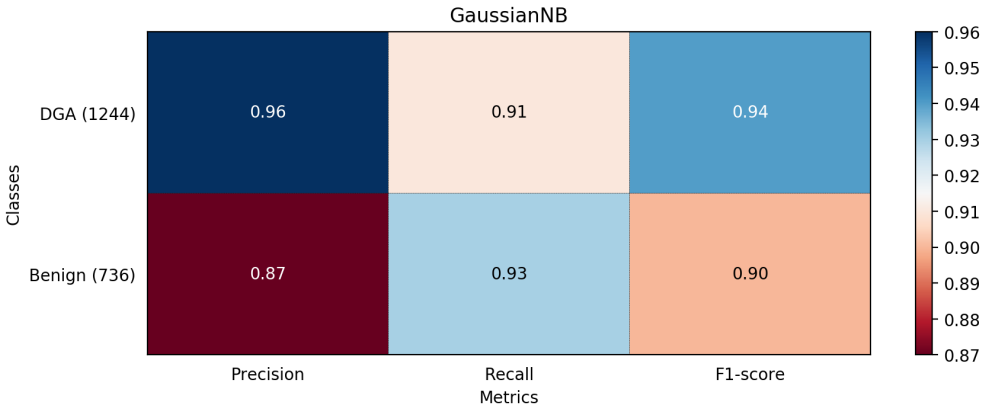


Figura 6.4: Report di classificazione per l'algoritmo Gaussian Naive-Bayes.

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 6.1. I risultati si possono vedere in figura 6.5 e 6.6 e come si può notare la performance ne è fortemente influenzata, introducendo una grande percentuale di falsi nelle predizioni effettuate dal classificatore.

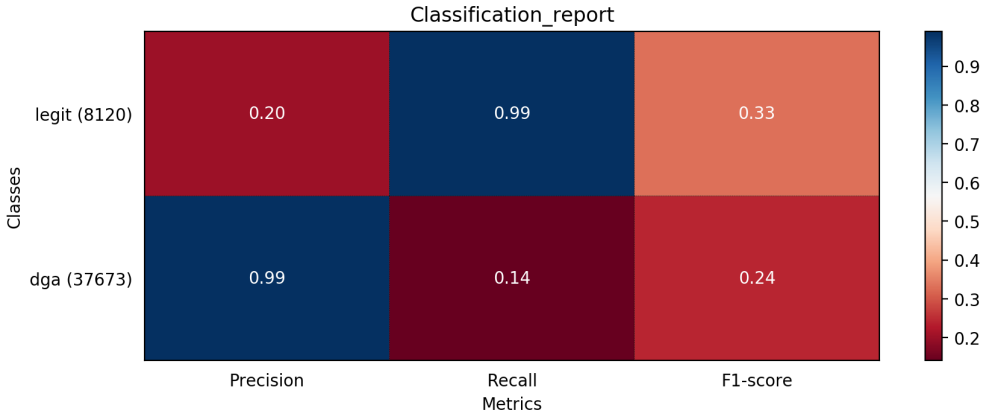


Figura 6.5: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

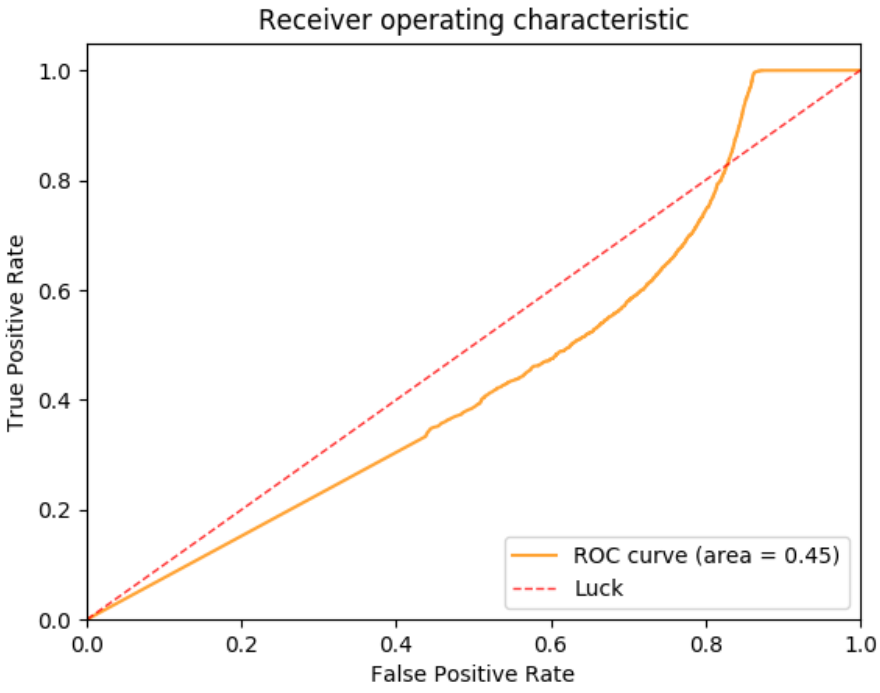


Figura 6.6: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con suppobox.

Come ultimo test è stato eseguito il training aggiungendo al precedente dataset di training una parte di domini generati da suppobox (Figura 6.7 e 6.8). Come si può notare la performance è migliorata sensibilmente, non raggiungendo comunque i risultati eccellenti del primo test.

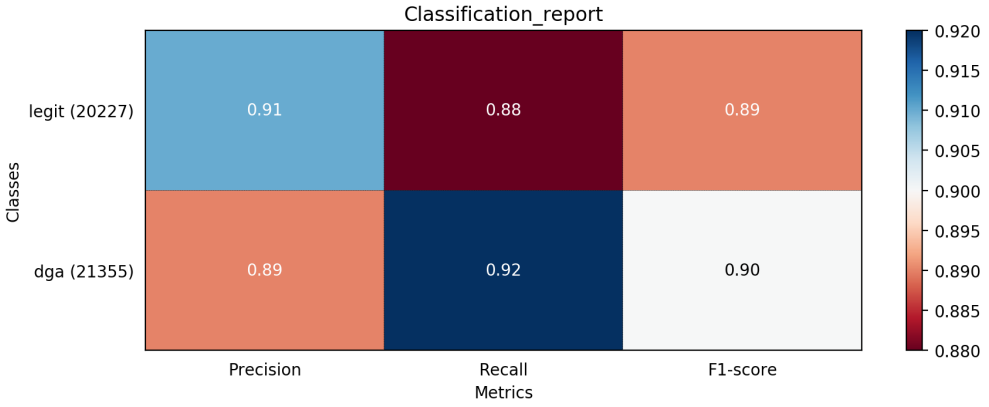


Figura 6.7: Classificatore Random Forest: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

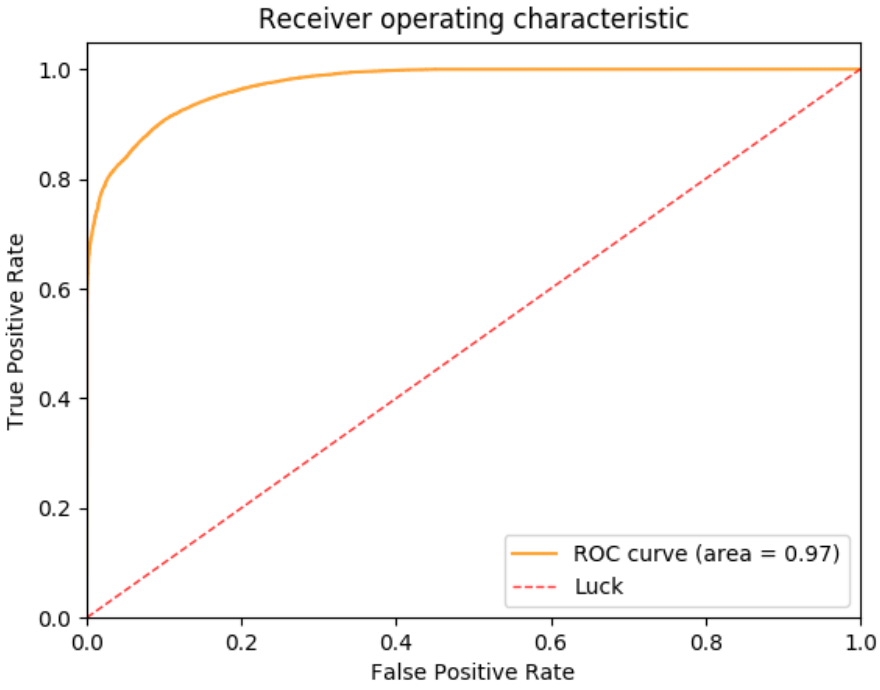


Figura 6.8: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con domini reali e malware (comprendenti suppobox).

6.3 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 6.2 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 6.9)

parlarne nella fase implementativa

Motivo di confronto è stata l'introduzione o meno di Batch Normalization [41]. Come si può vedere dai grafici mostrati in figura 6.10 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.

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 6.11 e 6.12. Come si può vedere dai grafici il comportamento del classificatore è pressoché identico a quello mostrato dal classificatore random forest (figure 6.7 e 6.8)

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

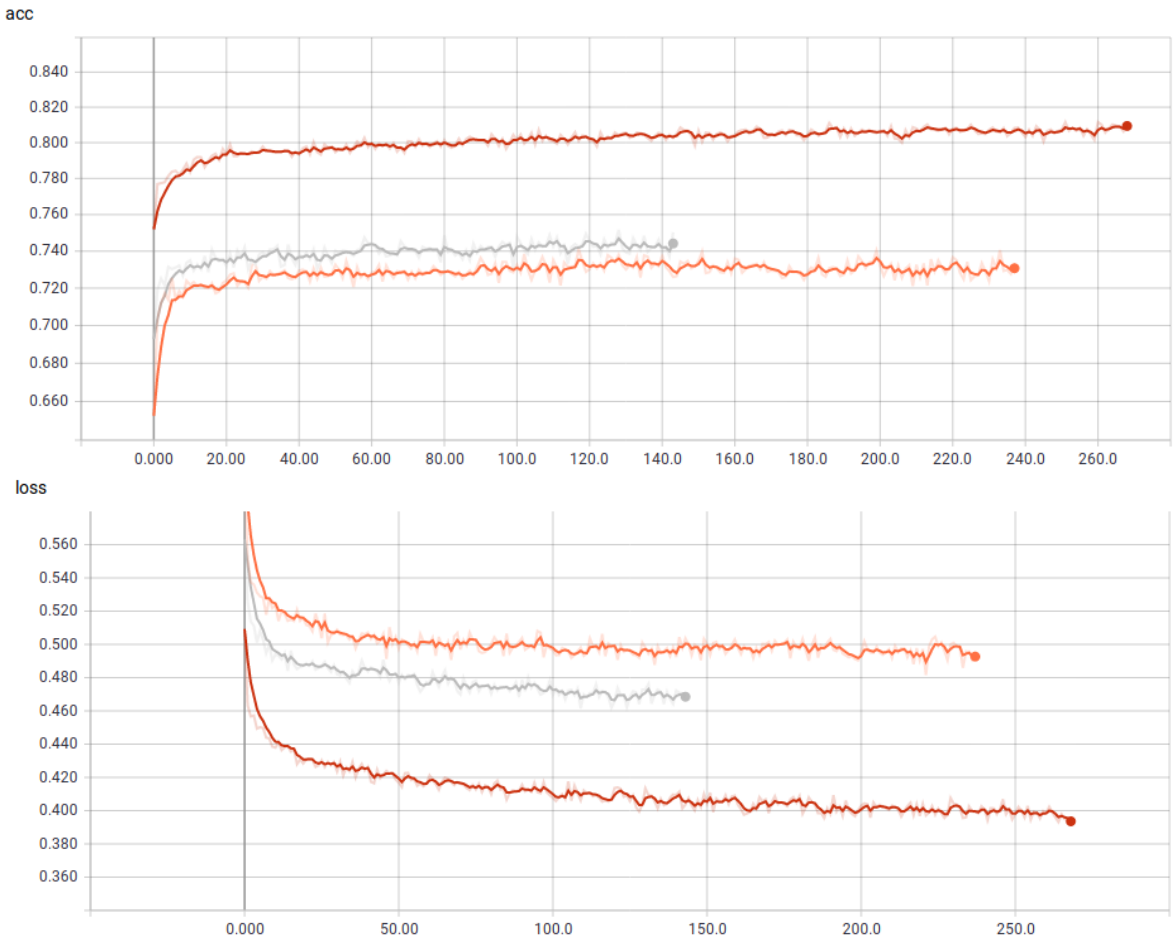


Figura 6.9: Classificatore Neurale: Grafici di Accuracy e Loss in funzione del tempo. confronto fra i tre modelli. La curva di colore grigio rappresenta il modello ingrandito, la curva di colore arancione rappresenta il modello ridotto mentre la curva di colore rosso rappresenta il modello intermedio; vincente tra i tre.

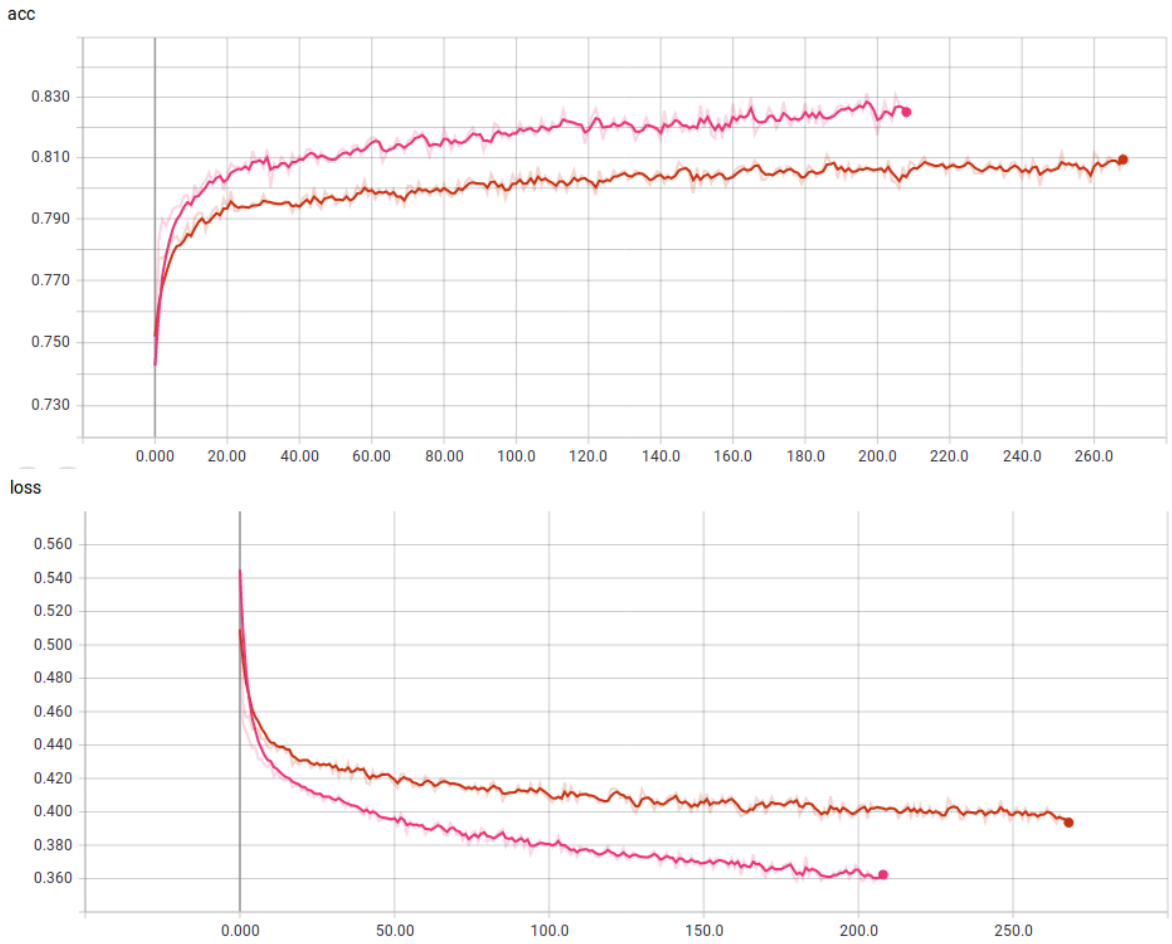


Figura 6.10: Classificatore Neurale: Grafici di Accuracy e Loss in funzione del tempo per il modello intermedio. La curva rossa identifica il modello senza l'ausilio di Batch Normalization mentre la curva fuchsia rappresenta lo stesso modello con l'inserimento di Batch Normalization per ogni livello densamente connesso che compone il Multilayer Perceptron.

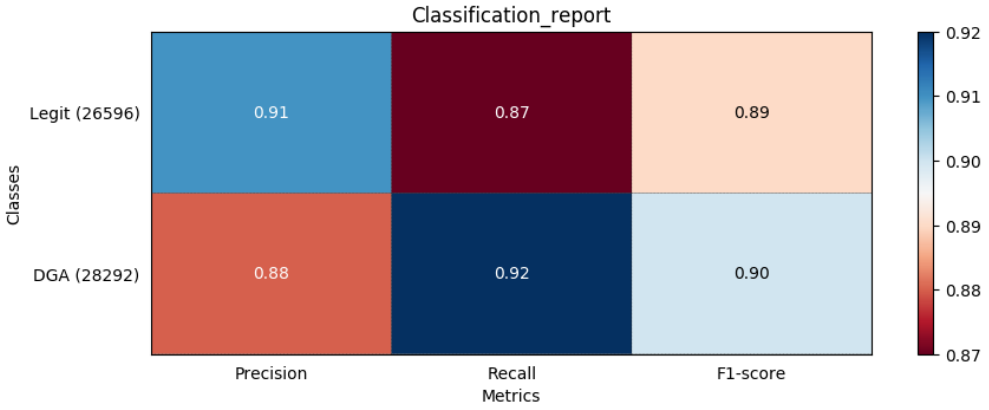


Figura 6.11: Classificatore Neurale: Report di classificazione su un subset di domini reali (legit) e malware, comprendenti suppobox (DGA).

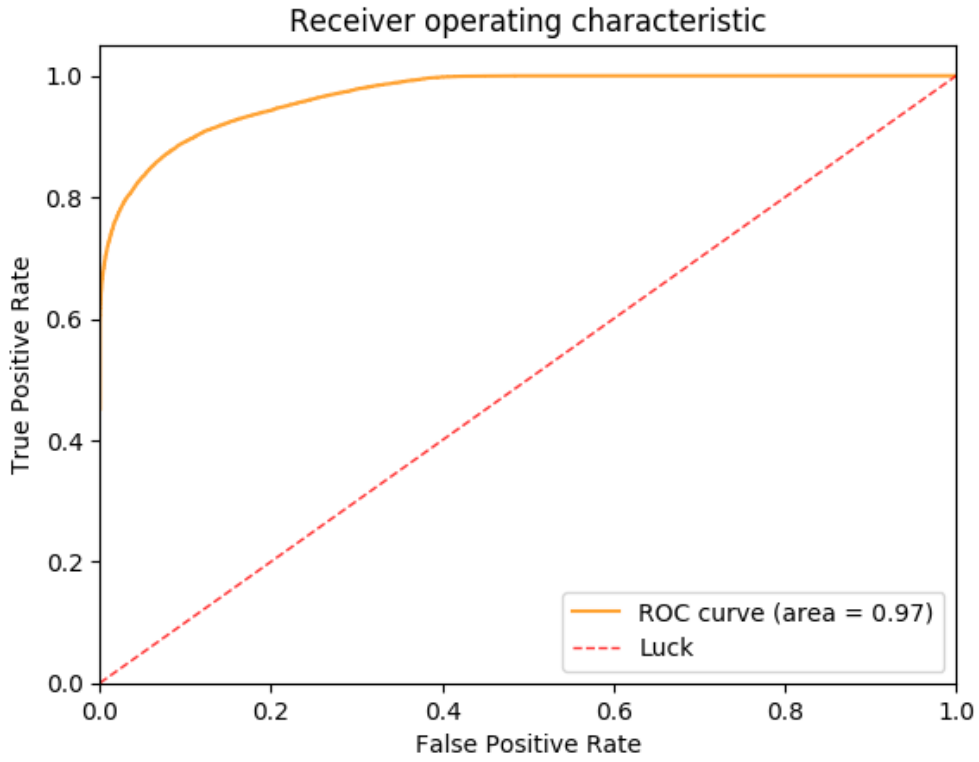


Figura 6.12: Classificatore Neurale: Area sottesa dalla curva ROC per il test con domini reali e malware (comprendenti suppobox).

6.4 Autoencoder

L'autoencoder come presentato in sezione 4.1 è stato testato con il dataset mostrato in sezione 5.3.1. 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 5.3.2 e 5.3.3

In figura 6.13 e 6.14 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.

L'utilizzo dei pesi dell'autoencoder come punto di partenza per il training della GAN ha contribuito fortemente a ridurre la instabilità di tale sistema, permettendo ai due sottosistemi generatore e discriminatore di partire da predizioni più precise rispetto all'utilizzo di pesi inizializzati in maniera randomica come generalmente attuato.

In tabella 6.2 è possibile vedere un esempio di quali domini l'autoencoder produce dato un subset del dataset Alexa in input. Si può notare come siano vagamente simili ai domini reali per quanto riguarda la distribuzione dei caratteri e la lunghezza media dei domini, tuttavia non presenta ulteriori caratteristiche tali da influenzare negativamente i classificatori neurali.

ehyt5tcncn3o5nw	d9ongedeo	2kbth	oldohizlioczzu
reknclkobg	meoomer	snd-drcepn	dodttiune
kne3xersl6npyr5	zggy1lbxgi1psir	sievd0	ahoinin3
moeaamutlrhsn	ypsanilwrox	ono5ponlanafhic	etiso9oo
5t7-iitnvtrm5en	bt5ennsl1zjchp0	mmd0-5-ile	qi8gtuyte-ssg-n
r-zeotn0t-wuf	runvpfcfrmaser	su1aojp52	mlsrp8gf
bgargtas	anhgnxracokimoa	eraveok	ktb1r2vb
vviadammpielw	atngsam	lfeubune	ptsdrqtanflog
7-aolelcfiextl	de-poaz9yiiu	ilnegban0	mcng5tsotnless
morehekb	nhntadt	uim-rca0ohxmsbi	rrhtsrceu

Tabella 6.2: Esempio di domini generati dall'autoencoder.

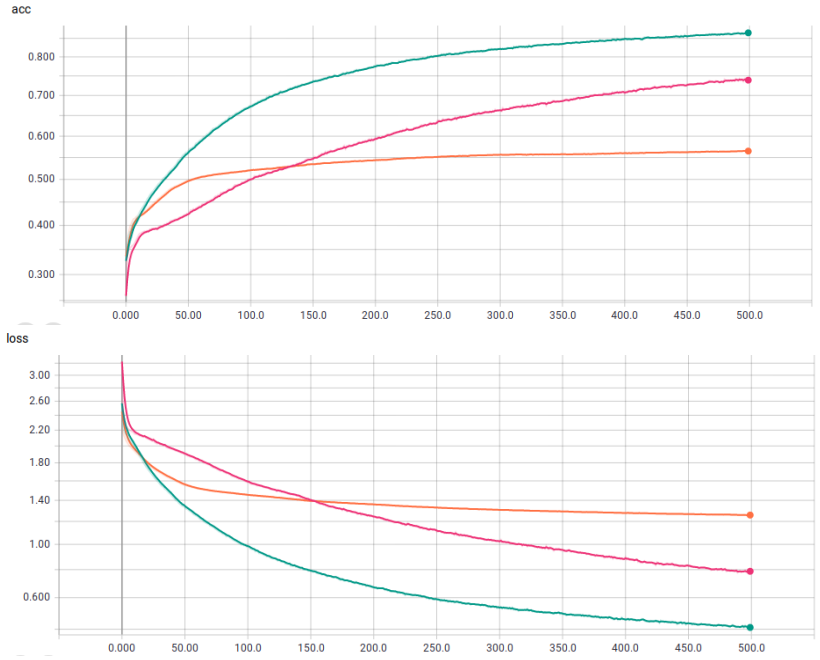


Figura 6.13: Grafici di Accuracy e Loss in funzione del tempo per la fase di training dell'autoencoder. La prima fase è rappresentata dalla curva arancione, la seconda dalla curva fuchsia mentre la terza fase è rappresentata dalla curva di colore verde. Si può notare come la terza iterazione raggiunga buoni valori di accuracy e loss; pertanto stato scelto come configurazione vincente per la GAN

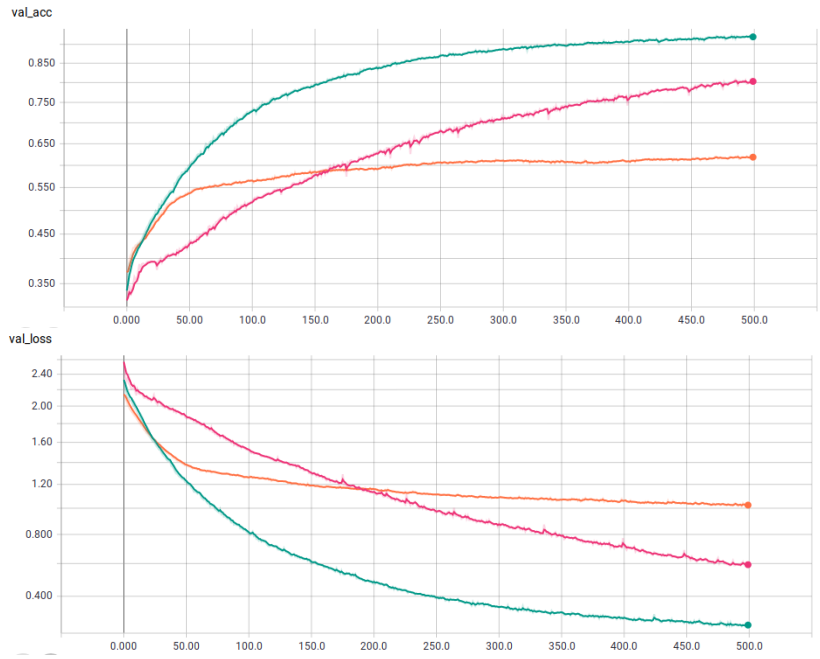


Figura 6.14: Grafici di Accuracy e Loss in funzione del tempo rispetto al subset di validazione per la fase di training dell'autoencoder. Si può notare come i valori raggiunti siano molto simili a quelli ottenuti sul dataset di training, indice di qualità del modello rispetto a valori mai visti.

6.5 Generative Adversarial Network

La Generative Adversarial Network descritta in sezione 4.4 e implementata come in sezione 5.4 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.

In particolare si sono incontrati due *failure modes*:

- Caso in cui il discriminatore prevale sul generatore, come mostrato in figura 6.15, si ottiene una curva di *loss* rasente lo zero, causando al generatore un incremento costante della propria curva di loss. Il significato di tale comportamento è l'impossibilità del generatore di generare domini sintetici sufficientemente realistici da poter mettere in crisi la predizione del discriminatore.

- Caso in cui il generatore degeneri, producendo "spazzatura", rendendo eccessivamente semplice la predizione del discriminatore. La degenerazione infatti avviene in forma di domini generati tutti uguali, contenente una singola lettera ripetuta per tutta la lunghezza di caratteri. Tale comportamento è dovuto alla mancata capacità del generatore di mimare realisticamente i domini realistici. Un esempio di tale comportamento è mostrato in figura 6.16

E' stato possibile ottenere la stabilità della GAN, grazie a numerose tecniche empiriche ottenute da [40] ed all'utilizzo del pre-training fornendo come inizializzazione i pesi ottenuti dalla fase di training dall'autoencoder. In figura 6.17 si può vedere l'andamento dei valori di loss di generatore e discriminatore nel caso di equilibrio tra le due reti neurali.

Grazie a tale training è stato possibile infine generare un dataset di domini sintetici provenienti dalla GAN, che mimassero in maniera più precisa i domini realistici. Come si può notare in tabella 6.3 non si tratta di una rappresentazione di parole realmente esistenti, tuttavia si può notare come siano presenti n-grammi realmente esistenti oltre che ad una lunghezza di sequenza simile a domini reali.

Come si può notare dal confronto mostrato in figura 6.18 la distribuzione dei caratteri generata dalla GAN è molto simile a quella presente all'interno del dataset Alexa, dimostrando

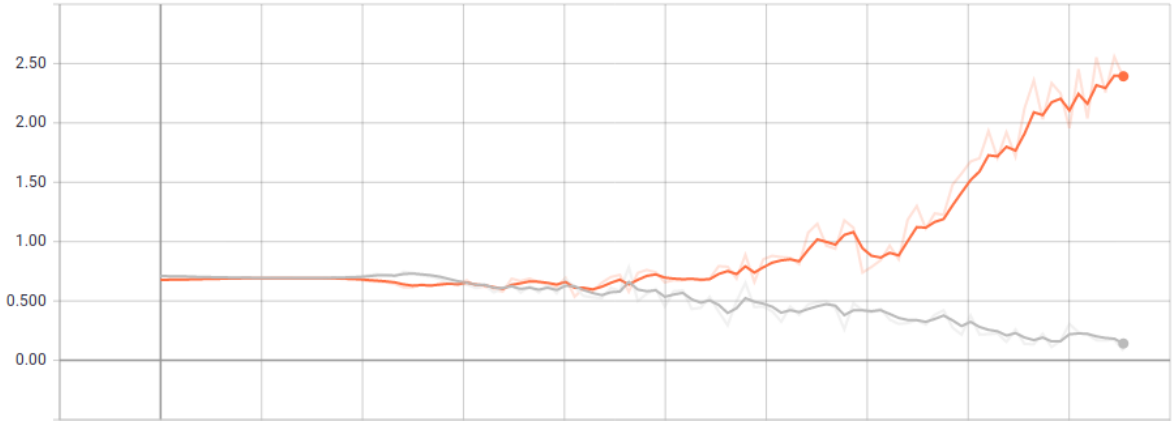


Figura 6.15: Caso di degenerazione 1. Grafico del valore di loss in funzione del tempo. Il discriminatore (curva di colore grigio) prevale sul generatore (curva arancione), il quale non riesce a migliorare il proprio valore di loss.

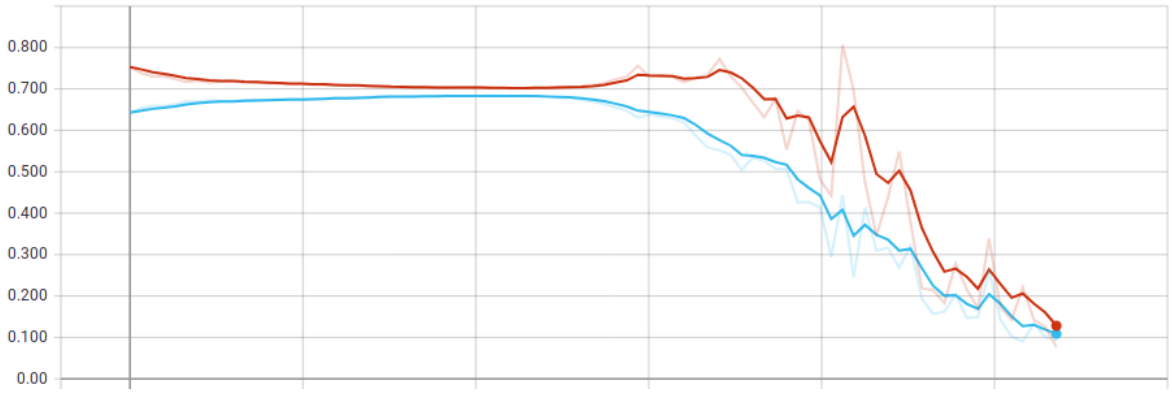


Figura 6.16: Caso di degenerazione 2. Grafico del valore di loss in funzione del tempo. Il generatore (curva di colore azzurro) non produce domini realistici, degenerando a dati inutilizzabili. Il discriminatore (curva di colore rosso) di conseguenza migliora la propria loss a causa della differenza sempre maggiore tra domini realistici e domini sintetici.

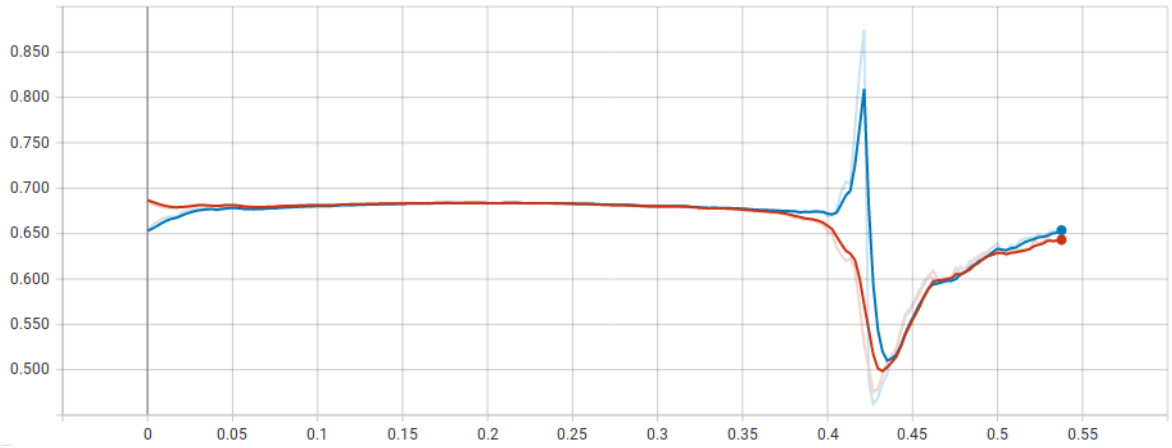


Figura 6.17: Grafico di loss in funzione del tempo per discriminatore (curva di colore rosso) e generatore (curva di colore azzurro). L'andamento rimane equilibrato fino al punto in cui il generatore non riesce a generare nuovi domini in grado di mettere in crisi il discriminatore.

edarareve	ssrarsone	horicicocr	avensdends
skonasesosarere	asccacca	sthonacorl	mwonwonerene
skaran-unar	monasheamc	raocjcacarcrarl	inihkkellgcrock
chicochophavock	itsusososse	vichitos	madoxto
dichoros	stlega	ogagagasuss	ljarlers
isherevores	ivortewrp	plerundinwoshn	maahofononoris
nillersosersrsp	sdesedlsss	odocococcocke	msusongere
rldicde	nggeneneres	tuccronpcs	scsacccca
escrarararuro	madesadk	mivorthitdhud	rrngajiagjonggk
aemjtup	cesasasrrrrrrs	mtuvocaro	ituutasisa

Tabella 6.3: Esempio di domini generati dalla GAN.

la natura dei domini sintetici rispetto a quelli reali. Il classificatore neurale presentato nelle sezioni precedenti è stato messo alla prova utilizzando un subset circa 10000 domini generati dalla GAN, etichettati come DGA, ed un subset di 10000 domini reali provenienti dal dataset Alexa. Come si può vedere da figura 6.19 il classificatore si trova in grave difficoltà nel distinguere i domini sintetici, raggiungendo un bassissimo valore di recall, l'abilità di riconoscere i campioni positivi forniti. Lo score medio del classificatore ne risulta molto basso rispetto a quello precedentemente mostrato in figura 6.11. In figura 6.20 è mostrata la curva ROC del classificatore neurale testato nelle medesime condizioni: la ridottissima area sottesa ($AUC = 0.39$) dimostra come il classificatore non sia in grado di distinguere in maniera efficiente i domini reali da quelli generati dalla GAN.

Tali risultati a confronto con quelli mostrati in sezione 6.2 e 6.3 dimostrano come i domini sintetici generati dalla GAN siano in grado di influenzare negativamente la performance di un classificatore DGA che in precedenza ha dimostrato buoni risultati.

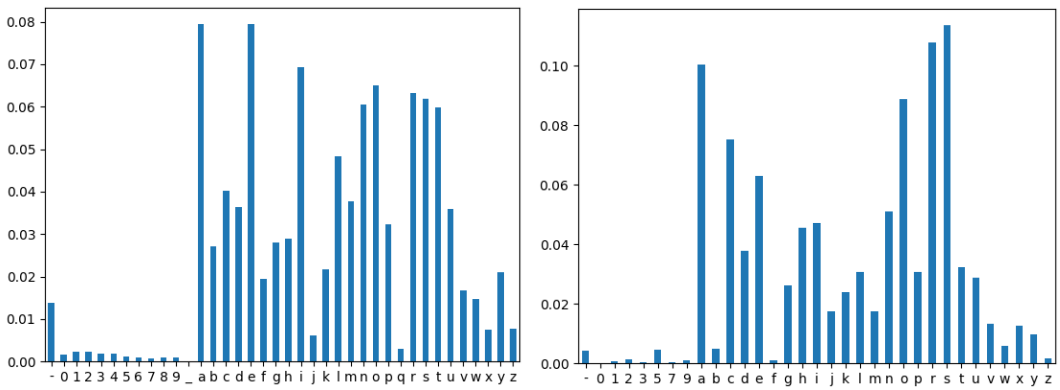


Figura 6.18: Confronto della distribuzione dei caratteri reali (grafico a sinistra) e generati algoritmicamente dalla GAN (grafico a destra)

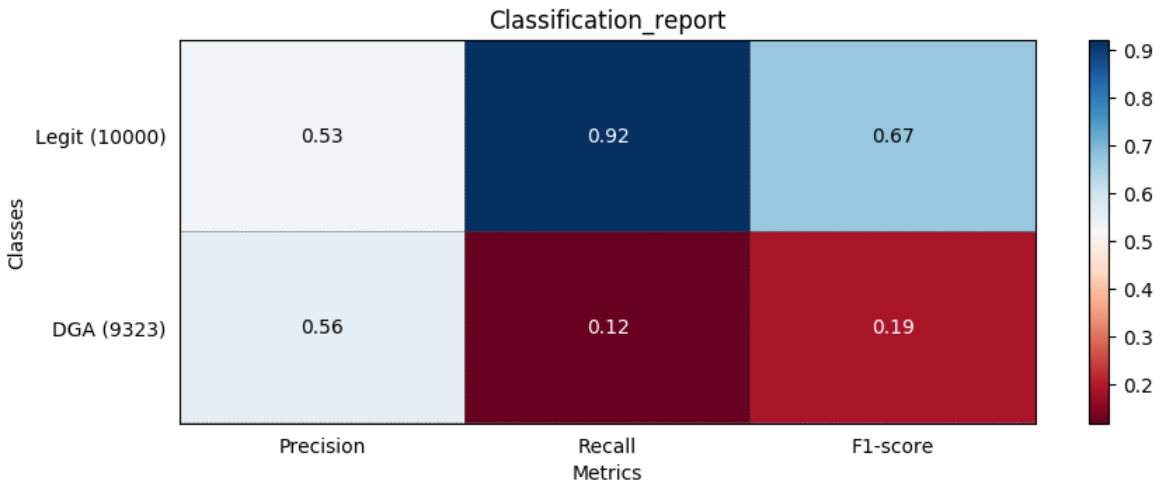


Figura 6.19: Classificatore Neurale testato su GAN: Report di classificazione su un subset di domini reali (legit) e generati da GAN (DGA).

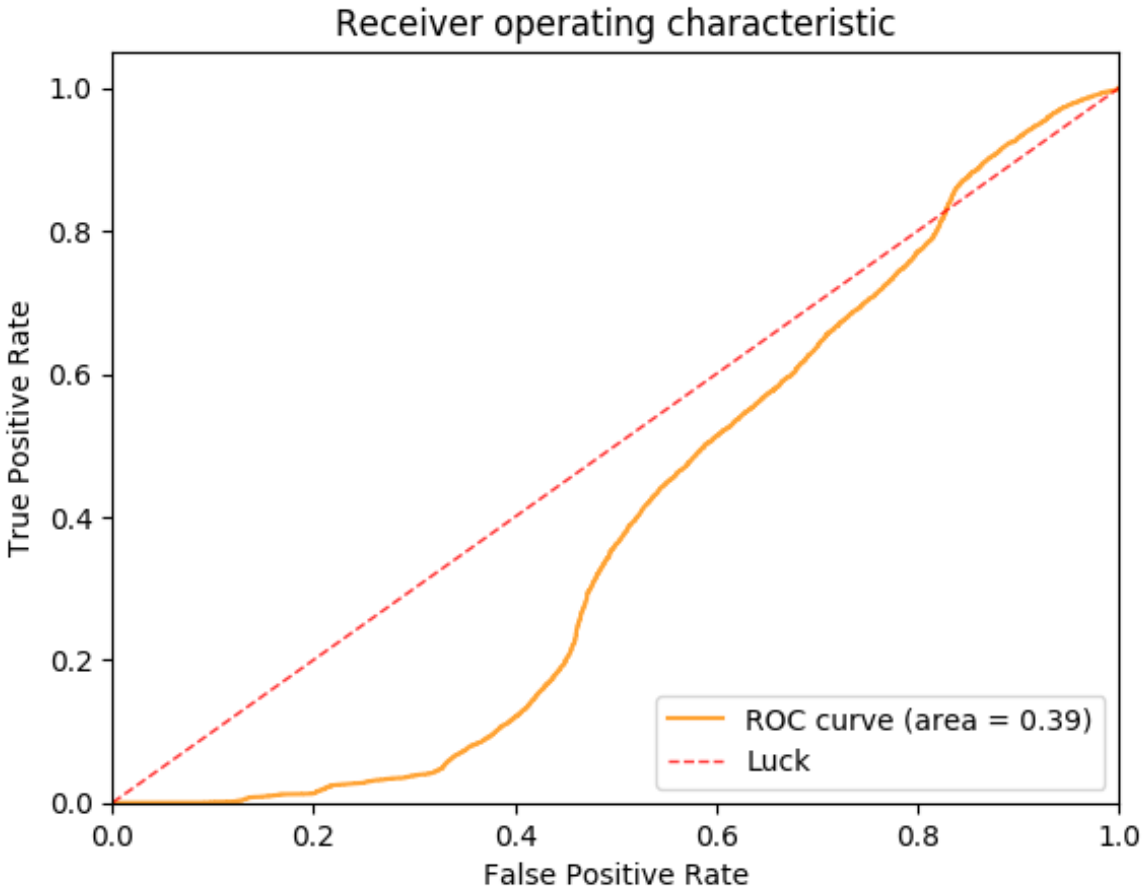


Figura 6.20: Classificatore Random Forest: Area sottesa dalla curva ROC per il test con domini reali e generati da GAN.

Capitolo 7

Conclusioni

1. Creato un classificatore in grado di discriminare tra domini reali e domini generati algoritmicamente tramite l'ingegnerizzazione di features linguistiche.

2. Creato un classificatore neurale in grado di eseguire la medesima discriminazione senza l'ausilio di features ingegnerizzate a partire unicamente da domini mappati numericamente e che dimostra la stessa performance sui medesimi casi.

3. Creato una GAN proveniente da un Autoencoder in grado di generare domini sintetici in grado di mettere in crisi la performance del classificatore neurale.

4. Allenato tale classificatore con i domini provenienti dalla GAN ed ottenuto robustezza riguardo tale caso, senza perdere performance per i casi precedenti.

Bibliografia

[1] P. Narang, J. M. Reddy, and C. Hota, "Feature selection for detection of peer-to-peer botnet traffic," in *Proceedings of the 6th ACM India Computing Convention*, Compute '13, (New York, NY, USA), pp. 16:1–16:9, ACM, 2013.

[2] M. Kearns and M. Li, "Learning in the presence of malicious errors," *SIAM Journal on Computing*, vol. 22, pp. 807–837, 1993.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, May 2002.

[4] N. H. Bshouty, N. Eiron, and E. Kushilevitz, "PAC learning with nasty noise," *Theoret. Comput. Sci.*, vol. 288, no. 2, pp. 255–275, 2002. Special issue for ALT '99.

[5] P. Fogla and W. Lee, "Evading network anomaly detection systems: Formal reasoning and practical techniques," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (New York, NY, USA), pp. 59–68, ACM, 2006.

[6] A. Globerson and S. Roweis, "Nightmare at test time: Robust learning by feature deletion," in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, (New York, NY, USA), pp. 353–360, ACM, 2006.

[7] M. Brückner and T. Scheffer, "Nash equilibria of static prediction games," in *Advances in Neural Information Processing Systems 22* (Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, eds.), pp. 171–179, Curran Associates, Inc., 2009.

[8] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *Recent Advances in Intrusion Detection* (E. Jonsson, A. Valdes, and M. Almgren, eds.), (Berlin, Heidelberg), pp. 203–222, Springer Berlin Heidelberg, 2004.

[9] K. Wang, J. J. Parekh, and S. J. Stolfo, "Anagram: A content anomaly detector resistant to mimicry attack," in *Recent Advances in Intrusion Detection* (D. Zamboni and C. Kruegel, eds.), (Berlin, Heidelberg), pp. 226–248, Springer Berlin Heidelberg, 2006.

[10] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, "Adversarial classification," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, (New York, NY, USA), pp. 99–108, ACM, 2004.

[11] Wikipedia, "Adversarial machine learning — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Adversarial%20machine%20learning&oldid=821367007`, 2018. [Online; accessed 25-January-2018].

[12] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, July 1959.

[13] F. Chollet, *Deep Learning with Python*. Manning Publications Company, 2017.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and Systems Magazine*, vol. 6, pp. 21–45, Third 2006.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, Sep 1995.

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

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

[26] 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.

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

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

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

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

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

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

[33] 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.

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

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

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

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

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

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

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

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

[42] 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`.

[43] 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.

[44] T. White, “Sampling generative networks,” 2016.

[45] “Scikit-learn.” `http://scikit-learn.org/stable/`, first visited in Aug 2017.

[46] “Keras.” `https://keras.io/`, first visited in Oct 2017.

[47] “Tensorflow.” `https://tensorflow.org/`, first visited in Oct 2017.

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

[49] 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.

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