Università degli Studi di Roma Tre

DIPARTIMENTO DI MATEMATICA E FISICA
Corso di Laurea Magistrale in Matematica

Tecniche di furto d'identità basate su Deep Learning

Relatore:
Prof. Marco Liverani

Candidato:
Federica Fino

Anno Accademico 2019–2020

1 Introduzione

Le password, come strumento di accertamento dell'identità di un utente di un sistema informatico, sono utilizzate dagli anni '60 per l'accesso a sistemi multiutente centralizzati. Successivamente, con la diffusione dell'informatica personale, se ne è in parte abbandonato l'utilizzo, per riprenderlo poi con la diffusione delle reti, dei nuovi sistemi operativi multi-utente e dei servizi online. Oggi la password è certamente un elemento essenziale nella vita di tutti i giorni, per la sicurezza delle nostre risorse e delle nostre informazioni.

Attualmente, nonostante la consapevolezza che non rappresentino di certo il metodo più robusto per tenere al sicuro le informazioni, le password sono ancora il fattore di sicurezza maggiormente utilizzato per proteggere l'accesso ai sistemi informatici e non solo. L'accesso di un utente ad un sistema si svolge introducendo uno *username*, attraverso cui l'utente si identifica, dichiara la propria identità, e una password segreta, attraverso cui l'utente fornisce una prova della propria identità; il sistema autentica l'identità dell'utente, verificando che la password corrisponda con quella registrata sul sistema stesso. Ovviamente la memorizzazione di queste password sul sistema non può avvenire *in chiaro* e tanto meno in forma cifrata (e quindi decifrabile!): al fine di impedire la rivelazione della password a persone diverse dal titolare dell'account di accesso al sistema, quello che viene memorizzato è quindi un *hash* della password, in modo tale che sia particolarmente complicato ricavare la password accedendo al file degli hash; tale file, per garantire il corretto funzionamento del sistema e consentire operazioni di gestione del sistema stesso, è accessibile agli amministratori del sistema e, su alcune configurazioni, anche ai singoli utenti.

Un hash è una funzione non invertibile che mappa una stringa di lunghezza variabile in una stringa di lunghezza predefinita. Generalmente si chiede che la funzione hash abbia le seguenti proprietà:

- *resistenza alla preimmagine*: sia computazionalmente oneroso cercare una stringa in input che dia un hash uguale ad un altro hash dato;

- *resistenza alla seconda preimmagine*: sia computazionalmente oneroso cercare una stringa in input che dia un hash uguale a quello prodotto da una stringa data;
- *resistenza alle collisioni*: sia computazionalmente oneroso cercare una coppia di stringhe in input che diano lo stesso hash.

Numerosi sono gli algoritmi che realizzano funzioni hash e ognuno possiede i seguenti requisiti:

1. restituisce una stringa di caratteri a partire da un qualsiasi flusso di bit di qualsiasi dimensione (l'input non è necessariamente una stringa, ma può essere anche un file). L'output è detto *digest*.
2. non è invertibile, ossia non è possibile tornare indietro e ricostruire il documento originale conoscendo la stringa di output, ovvero si tratta di una funzione unidirezionale.

Un sistema di password guessing, letteralmente un meccanismo automatico per indovinare le password, è un programma con il quale si cerca di scoprire una password a partire dal file degli hash delle password degli utenti registrati su un determinato sistema. Questi sistemi non tentano di decifrare l'hash della password (il che è virtualmente impossibile), bensì procedono emulando il processo di autenticazione: generano, seguendo qualche criterio euristico o puramente combinatorio, password verosimili, ne calcolano l'hash e lo confrontano con quelli presenti sul file degli hash delle password di un determinato sistema.

I criteri che utilizzano sono i più vari: dall'inefficiente ricerca esaustiva (di fatto impraticabile, a meno che le password non siano molto corte e basate su un alfabeto ristretto), a quelle basate sull'uso di dizionari, a tecniche *rule-based*, che utilizzano algoritmi specifici per costruire stringhe con metodi spesso usati per definire password complesse ma allo stesso tempo facili da ricordare (es.: «`p@ssw0rd`»). Esistono anche dei software *open source* in grado di eseguire in modo abbastanza efficiente la ricerca delle stringhe corrispondenti agli hash,

dotati anche di un buon livello di affidabilità; tra questi sono molto utilizzati i software noti con il nome di *HashCat* [3] e *John the Ripper* [5].

Recentemente anche l'intelligenza artificiale e il *deep learning* stanno avanzando all'interno del mondo della sicurezza informatica e una delle recenti applicazioni si è proprio concentrata sul password guessing. È stata infatti proposta una nuova tecnica, che sembra essere molto più efficiente delle precedenti, basata sulle reti neurali GAN (*Generative Adversarial Network*) [2]. Questa tecnica, battezzata dagli ideatori Briland Hitaj, Paolo Gasti, Giuseppe Ateniese e Fernando Perez-Cruz con il nome di *PassGAN* [4], sfrutta due reti neurali, una per la generazione di stringhe e l'altra per verificare il grado di bontà della stringa generata.

Nel sistema PassGAN la struttura utilizzata sia per la rete generativa che per quella discriminativa è la rete neurale convoluzionale (CNN). L'architettura di questa tipologia di reti le rende particolarmente efficienti in contesti con dati molto complessi, come le immagini, i video o i suoni. I dati di input nell'ambito del *password guessing* sono invece più semplici, perché composti da stringhe di caratteri alfanumerici. La mia idea è stata quindi quella di provare a costruire un sistema, con lo stesso obiettivo di PassGAN, ma più efficiente perché caratterizzato da un'architettura meno articolata.

L'obiettivo della tesi è dunque quello di studiare questo contesto applicativo, approfondire i vari elementi che entrano in gioco e implementare un prototipo di sistema che, utilizzando una rete GAN, possa ricavare le password da un file di hash fornito in input. Nell'ambito della parte sperimentale del mio lavoro ho pertanto realizzato una rete GAN originale per l'implementazione di tale sistema; al termine dell'addestramento ho poi inserito la rete generativa risultante all'interno di un programma che, ricevuto un file di utenti con il relativo hash della password, cerca di individuarne il maggior numero possibile.

2 Password Guessing

Il furto d'identità digitale è un attacco informatico in cui l'attaccante riesce ad entrare in possesso di informazioni riservate utilizzate per l'autenticazione degli utenti. Queste informazioni vengono spesso aggiunte a enormi database e rese disponibili sul dark web, così da poter essere utilizzate da criminali per compiere azioni illegali o fraudolente. Tra i metodi più diffusi di furto d'identità digitale si trovano la clonazione della carta di credito o il phishing; i sistemi di Password Guessing sfruttano invece il data breach (dall'inglese «violazione dei dati») e i grandi furti di informazioni dai database con le credenziali di autenticazione degli utenti.

Sono due gli approcci che un attaccante ha a disposizione per compiere il suo attacco di password guessing: online e offline. Nel primo scenario l'attaccante invia una password direttamente al sistema che intende violare, esattamente come se fosse un utente che vuole accedere a quel determinato servizio. Questo è il caso in cui l'attaccante non ha accesso diretto al file di hash delle password di quel determinato sistema, cosa che invece caratterizza un attacco di tipo offline.

I sistemi di *password guessing* sono certamente uno strumento di attacco informatico, progettato per violare la sicurezza di un sistema attraverso il cosiddetto «furto d'identità», ma possono essere anche uno strumento assai utile nell'ambito dell'informatica forense, nel corso di un'indagine giudiziaria, o per risalire ad una password indispensabile per gestire un sistema, quando la password di un determinato utente è andata irrimediabilmente persa.

I due strumenti più popolari di password guessing già citati nell'introduzione sono HashCat [3] e John the Ripper [5].

Entrambi utilizzano diverse strategie per costruire delle password verosimili, ma il loro punto di forza è rappresentato dall'utilizzo del modello di Markov [10], il quale riduce significativamente lo spazio delle possibili password. Lo

spazio utilizzato per generare le password è $X = L \cup U \cup N \cup S$

$L = \{a, b, c, \ldots, z\}$	26 lettere minuscole;
$U = \{A, B, C, \ldots, Z\}$	26 lettere maiuscole;
$N = \{0, 1, 2, \ldots, 9\}$	10 caratteri numerici;
$S = \{!, @, \#, \ldots, \$\}$	32 caratteri speciali.

Un modello di Markov definisce una distribuzione di probabilità sull'insieme di tutti i caratteri X in base alla frequenza, in altre parole considera solamente sequenze con determinate proprietà:

- In un *modello di Markov di ordine zero*, ogni carattere è generato secondo la probabilità di distribuzione e indipendentemente dai caratteri precedentemente generati. Questo modello produce password che apparentemente non sono naturali, ovvero è raro che venga prodotta una parola di senso compiuto, tuttavia riduce di molto la dimensione dello spazio di ricerca ed è ottimo per trovare le password generate considerando le iniziali delle parole all'interno di una frase.

- In un *modello di Markov di primo ordine*, invece, ogni carattere è generato tenendo conto dei caratteri precedenti; ciò che si fa è quindi assegnare un valore ad ogni coppia di caratteri: il criterio con cui questo valore viene assegnato è in base alla probabilità di transizione dal primo al secondo carattere della coppia, ossia la probabilità che il secondo carattere si possa trovare dopo il primo all'interno di una parola. Questo modello, a differenza di quello di ordine zero, è in grado di produrre parole che se non sono di senso compiuto, sono almeno pronunciabili verbalmente e riduce drasticamente lo spazio delle possibili password da provare.

Per ottenere lo stesso risultato, PassGan [4] utilizza invece tecniche di deep learning e in particolar modo le Generative Adversarial Network (note anche con il nome di «reti GAN»).

3 Deep Learning

L'approccio alla risoluzione del problema del *password guessing* che abbiamo proposto all'interno di questo lavoro è basato su tecniche di *Deep Learning*, ossia un approccio metodologico che rientra nell'ambito più generale dell'*Intelligenza Artificiale*. In questo ambito sono state sviluppate tecniche ed algoritmi basate sul modello di rete neurale artificiale che vanno sotto il nome di tecniche e modelli di *Machine Learning*, ossia sistemi di apprendimento automatico con cui la macchina, sulla base di dati ricevuti in input, impara a classificare le informazioni.

Il primo a dare una definizione informale di *machine learning* fu Arthur Samuel nel 1959, il quale lo definì proprio come il campo di studi che fornisce al computer l'abilità di imparare senza essere stati esplicitamente programmati. Una definizione più formale fu invece formulata da Tom Mitchell nel 1997 [9], il quale definì il concetto fondamentale alla base del machine learning, l'*apprendimento*:

Definizione. Un programma apprende da una certa esperienza E in relazione ad una certa classe di abilità T con una misura di prestazione P, se la sua prestazione nell'abilità T misurata da P migliora con l'esperienza E.

L'apprendimento può essere *supervisionato*, qualora l'input fornito all'algoritmo in grado di compiere una determinata classificazione delle informazioni, sia assistito da un utente, che etichetta le informazioni di training. Al contrario l'apprendimento è *non supervisionato* quando tale etichettatura dei dati in input forniti alla macchina nella fase di apprendimento è assente.

La rete neurale artificiale è un modello di machine learning ed è rappresentata in questo contesto come un grafo bipartito, con un insieme di vertici indipendenti che rappresenta lo strato di input ed un secondo insieme di vertici che rappresenta lo strato di output. I pesi numerici assegnati agli spigoli del grafo sono frutto del processo di apprendimento (supervisionato o meno) e indirizzeranno la classificazione delle informazioni ricevute in input verso il più probabile esito in output. Una rete neurale è quindi un modello com-

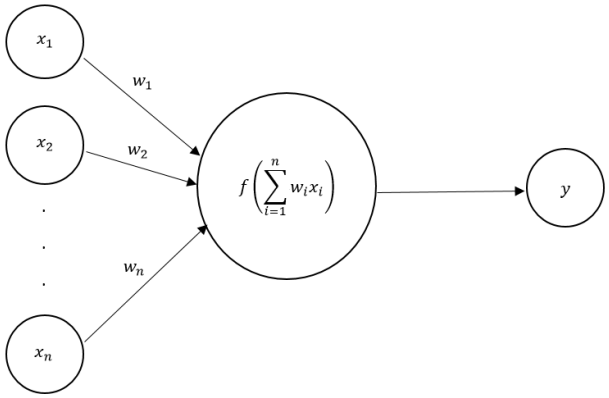

Figura 1: *Rappresentazione di un neurone artificiale.* In entrata riceve n dati in input, in uscita fornisce la somma pesata attraverso la funzione f

putazionale su più livelli composto da elementi simili ai neuroni del sistema nervoso umano. L'elemento di elaborazione di base è infatti il neurone neurale artificiale, che può essere rappresentato con lo schema in Figura 1: il neurone non è altro che una funzione matematica che riceve diversi dati in input e restituisce come output un unico valore reale. Questa funzione, nota anche come funzione di attivazione, è una funzione della somma dei valori in ingresso moltiplicati per i relativi pesi (ovvero $\sum_{i=1}^{n} w_i x_i$). In figura 2 riporto le scelte più diffuse per le funzioni di attivazioni.

Durante la costruzione di una rete neurale sono due gli aspetti fondamentali su cui bisogna porre l'attenzione:

- La sua **architettura**: rappresenta la scelta della struttura della rete (se *feedforward*, *recurrent*, *convolutional*, ecc.), del numero di strati e del numero di neuroni in ogni strato.

- L'**apprendimento**: rappresenta la scelta delle tecniche con cui costruire la fase di addestramento della rete. La scelta più comune, anche se non l'unica possibile, è quella di utilizzare il *gradient descent* e la *back propagation*.

Si parla infine di *Deep Learning* quando il sistema di machine learning implementa una rete neurale con più di due strati: nella struttura del grafo, fra

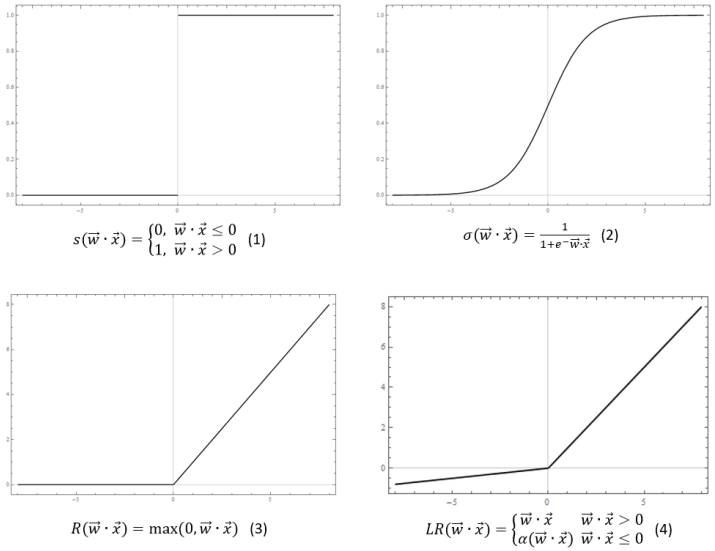

Figura 2: *Funzione di Attivazione.* Rappresentazione grafica delle quattro scelte più diffuse per la funzione di attivazione. Sulla sinistra si trova la *funzione soglia*, sulla destra la *funzione sigmoide* e in basso la *Rectified Linear function (ReLu)* con la sua variante *Leaky ReLu function*

lo strato di input e quello di output, sono presenti uno o più strati nascosti.

Numerose sono le applicazioni del *deep learning*: con esso è possibile insegnare ai computer a riconoscere un'immagine, a comprendere cosa qualcuno dice, a tradurre dei testi in varie lingue e ad aiutare un robot ad esplorare l'ambiente intorno a se. Diffusissimo è il suo utilizzo nel campo medico, con lo studio automatizzato delle cartelle cliniche passate o con l'utilizzo della *computer vision* per la visualizzazione delle immagini utili per le analisi; o anche nell'ambito della sicurezza alla guida, di cui ne sono esempi la frenata d'emergenza automatizzata e il *lane keeping assist*, che consente al veicolo di rimanere nella corsia in cui si trova senza la necessità di alcun intervento da parte dell'autista.

In base alla struttura si possono distinguere numerose tipologie di reti neurali differenti, tra le più comuni nell'ambito del *deep learning* si trovano:

- **Multi-layer perceptron (MLP)**: una rete neurale totalmente connessa e con almeno uno strato nascosto.

- **Convolutional neural network (CNN)**: una rete neurale con diversi tipi di strati speciali, costruita organizzando i percettroni in modo simile a come le cellule biologiche sono organizzate nella corteccia visiva del cervello. Questo tipo di reti, infatti, trovano molte applicazioni nel riconoscimento o creazione di imagini, video e suoni.

- **Recurrent neural network (RNN)**: una rete neurale caratterizzata dalla presenza di uno stato interno (o memoria), che si basa su tutti o solo in parte i dati di input già precedentemente forniti alla rete. L'output di una rete ricorrente è una combinazione del suo stato interno (memoria di input) e l'ultimo campione immesso come input.

4 Multi-layer perceptron

Il neurone descritto nel paragrafo precedente non è altro che una rete neurale con un singolo strato. Il passo successivo è introdurre ulteriori strati tra quello di input e quello di output: questi strati prendono il nome di *hidden layers*. Ogni neurone moltiplica l'input ricevuto in base a determinati pesi predefiniti, esegue il calcolo e trasferisce il risultato allo strato successivo (l'output è uno solo e viene fornito in input a tutti i neuroni del livello seguente); quest'ultimo riaggrega i dati ricevuti dai diversi nodi del livello precedente e li utilizza come input. Questo processo appena descritto prende il nome di *forward propagation* e rappresenta in un certo senso la *funzione ipotesi* della rete neurale multistrato.

Il passo successivo è quello di determinare l'errore che la rete ha commesso, calcolando la "distanza" tra il dato ottenuto come output e il risultato reale che ci si aspetta. Per fare ciò si definisce una funzione, nota anche come *funzione costo*, che misuri la precisione della rete. Una volta calcolato l'errore finale come differenza tra l'output trovato e quello desiderato, si procede ridiffondendo all'indietro l'errore tra i vari nodi della rete per comprendere il grado di responsabilità di ogni percettrone sull'errore ottenuto. Questo processo prende il nome di *error back propagation* e permette di aggiornare i pesi in modo tale

da minimizzare la perdita, dando ai nodi con errore maggiore pesi inferiori e viceversa.

Una volta minimizzato l'errore la fase di addestramento termina e si ottengono i pesi definitivi; a questo punto la rete è pronta per ricevere in input nuovi dati e produrre un output verosimile utilizzando unicamente la *forward propagation*.

4.1 Forward Propagation

Consideriamo una rete neurale multi-strato generica composta da L strati (uno di input, uno di output e $L-2$ strati "nascosti"). Ogni strato può avere un numero arbitrario di nodi, quindi sia s_l il numero di unità nello strato l, $\forall l = 1, \ldots, L$. Siano:

- $a_i^{(l)}$ l'attivazione del nodo i-esimo del livello l;
- $a^{(l)} = (a_0^{(l)}, a_1^{(l)}, \ldots, a_{s_l}^{(l)})$, il vettore dei nodi al livello l con l'aggiunta del *bias* $a_0^{(l)}$, che si pone uguale a 1 $\forall l = 1, \ldots, L-1$;
- $w^{(l)}$ la matrice dei pesi relativa alla funzione di attivazione tra il livello l e il livello $l+1$.

In generale, l'elemento $w_{ij}^{(l)}$ altro non è che il peso associato all'arco che va dal nodo i dello strato l al nodo j dello strato $l+1$.

La rete neurale riceve in input $x = (x_1, \ldots, x_{s_1})$, le cui componenti rappresentano ciascuna un nodo nel layer di input. Si pone dunque:

$$(a_1^{(1)}, \ldots, a_{s_1}^{(1)}) = (x_1, \ldots, x_{s_1})$$

Per tutti gli altri strati i nodi di attivazione si ottengono nel modo seguente:

$$(z_1^{(l)}, \ldots, z_{s_l}^{(l)}) = (w^{(l-1)})^T \cdot a^{(l-1)} \tag{1}$$

$$(a_1^{(l)}, \ldots, a_{s_l}^{(l)}) = (f(z_1^{(l)}), \ldots, f(z_{s_l}^{(l)})) \tag{2}$$

Quindi:

$$a^{(l)} = (a_0^{(l)}, a_1^{(l)}, \ldots, a_{s_l}^{(l)})$$

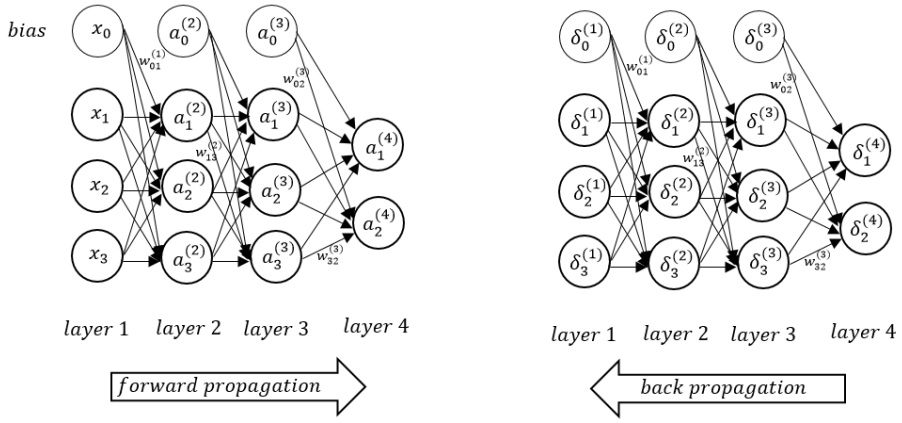

Figura 3: Rappresentazione grafica dei processi di *forward propagation* e *back propagation* su una rete neurale a 4 strati

Una volta giunti all'ultimo strato, seguendo le stesse regole dei livelli precedenti, si ottiene in output l'ipotesi $a^{(L)}$.

4.2 Back-propagation

Si consideri ora la *funzione costo* $J(w)$; l'obiettivo è quello di minimizzare tale funzione e di trovare quindi il

$$\min_{w} J(w)$$

Per utilizzare uno degli algoritmi di minimizzazione bisogna calcolare le derivate parziali di tale funzione rispetto ai pesi:

$$\frac{\partial J(w)}{\partial w_{ij}^{(l)}}$$

La *back propagation* fornisce un metodo computazionalmente efficiente per calcolare tali derivate.

Per calcolare tutti gli errori si parte, questa volta, dallo strato di output, calcolando la differenza tra il risultato ottenuto dalla *forward propagation* e il corretto output $y = (y_1, \ldots, y_{s_L})$:

$$\delta^{(L)} = a^{(L)} - y$$

Per tutti gli altri strati, come dimostrato nella tesi, utilizzando l'algoritmo di *back propagation*, si ottiene:

$$\delta_j^{(l)} = f'(z_j^{(l)})(w^{(l)} \cdot \delta^{(l+1)}), \text{ per } 0 \leq j \leq s_l$$

dove $\delta^{(l+1)}$ è il vettore degli errori relativi ai nodi del livello $l+1$.

Si possono distinguere due fasi fondamentali nel processo di *back propagation*:

- la propagazione dell'errore a ritroso lungo la rete con l'obiettivo di calcolare le derivate;
- l'aggiustamento dei pesi utilizzando le derivate calcolate.

La prima fase può essere applicata a molti altri tipi di rete e non solo al *multi-layer perceptron*, può coinvolgere funzioni di errore diverse e valutare anche altre tipologie di derivate, come le matrici Jacobiana e Hessiana. Allo stesso modo, la seconda fase può essere affrontata utilizzando una diversa varietà di schemi di ottimizzazione: la versione più semplice, nonché la prima proposta, coinvolge il *gradient descent*, comunque non è l'unica possibile.

5 Convolutional neural networks

Alla base delle *convolutional neural networks* ci sono tre concetti fondamentali:

- i campi ricettivi locali;
- la condivisione dei parametri;
- il sottocampionamento.

5.1 I campi ricettivi locali

Nel modello *multi-layer perceptron* descritto nelle pagine precedenti, gli input erano rappresentati come una linea verticale di neuroni. In una *convolutional*

neural network invece l'input si può immaginare come un quadrato $n \times n$ di neuroni, i cui valori corrispondono alle intensità degli $n \times n$ pixel che compongono l'immagine di input.

Nelle pagine seguenti, per ragioni di sintesi e per rendere più agevole la comprensione del modello, faremo riferimento ad una rete neurale convolutiva impiegata nel caso dell'elaborazione di immagini digitali.

Per creare uno strato di neuroni nascosti si procede, non collegando ogni pixel di input a ogni neurone nascosto, ma creando connessioni solo in aree piccole e localizzate dell'immagine di input. Per essere più precisi, ogni neurone nel primo strato nascosto sarà connesso a una piccola regione di neuroni di input di dimensioni $m \times m$, corrispondente a $m \times m$ pixel di input. Quella regione nell'immagine di input è chiamata *campo ricettivo locale* per il neurone nascosto; una sorta di piccola finestra sui pixel di input. Ogni connessione ha un peso associato, ed inoltre il neurone nascosto è caratterizzato anche da un *bias*. Si può pensare a quel particolare neurone nascosto come se stesse imparando ad analizzare il suo campo ricettivo locale. Quindi si procede facendo scorrere il campo ricettivo locale sull'intera immagine di input. Ad ogni campo ricettivo locale, corrisponde un diverso neurone nascosto nello strato successivo.

5.2 La condivisione dei parametri

Come già visto nel paragrafo precedente, ogni neurone nascosto ha un bias e $m \times m$ pesi collegati al suo campo ricettivo locale. La condivisione dei parametri consiste nel riutilizzare gli stessi pesi e bias per ciascuno dei neuroni nascosti appartenenti allo strato successivo. Ovvero, per il neurone in posizione (j,k) dello strato nascosto, l'output è:

$$f\left(b + \sum_{i=0}^{m-1}\sum_{l=0}^{m-1} w_{i,l} a_{j+i,k+l}\right)$$

dove f è la funzione di attivazione del neurone, b è il valore del bias, $w_{i,l}$ è la matrice $m \times m$ dei pesi e infine $a_{x,y}$ è l'attivazione di input in posizione (x,y).

In altre parole i pesi e il bias sono tali da permettere al neurone nascosto di stabilire se una determinata caratteristica è presente o meno nel campo ricettivo locale a cui è collegato. Per questo motivo si parla di condivisione dei parametri, perché gli stessi pesi e bias vengono utilizzati su tutti i campi ricettivi locali così da cercare una stessa caratteristica lungo tutta la superficie dell'immagine. La mappa dallo strato di input al primo strato nascosto è infatti nota con il nome di *feature map*, ovvero mappa delle caratteristiche. I pesi e il bias condivisi altro non fanno che definire un filtro, o *kernel*, per una determinata caratteristica.

Un livello convoluzionale completo sarà quindi costituito da diverse mappe delle caratteristiche, così da cercare diversi aspetti all'interno di una stessa immagine.

5.3 Il sottocampionamento

Oltre agli strati convoluzionali appena descritti, le *convolutional neural networks* sono caratterizzate anche dalla presenza di *pooling layers*, o strati di raggruppamento. Questo tipo di strati vengono tipicamente utilizzati immediatamente dopo gli strati convoluzionali e il loro scopo è quello di semplificare le informazioni ricevute da essi.

In particolare, un *pooling layer* prende l'output di ciascuna *feature map* dal layer convoluzionale precedente e prepara una sorta di *feature map* "condensata". Ciascuna unità nello strato di raggruppamento riassume una regione di neuroni appartenenti allo strato precedente.

Per portare un esempio concreto, una procedura comune è il cosiddetto *max-pooling*, in cui l'unità nello strato di raggruppamento emette semplicemente l'attivazione massima nella regione di input. Un'altra tecnica molto diffusa è anche il *raggruppamento* L_2. In questo caso invece di prendere il massimo tra le attivazioni di una regione di neuroni, si prende la radice quadrata della somma dei quadrati delle attivazioni nella regione che si sta considerando.

Questo passaggio è noto con il nome di sottocampionamento perché, se da uno strato convoluzionale si produce ad esempio un output di 24×24 neuroni, dopo un *pooling* che considera una regione di dimensione 2×2 si ottengono 12×12 neuroni. Ovvero si sta riducendo il numero di neuroni e quindi in pratica si sta in un certo senso "sottocampionando".

6 Recurrent neural networks

Tutte le reti neurali discusse fino ad ora sono caratterizzate da una struttura di tipo *feedforward*, ovvero reti in cui l'output di un livello viene utilizzato come input per il livello successivo e le informazioni vengono sempre trasmesse in avanti e mai due volte nello stesso neurone.

Ciò implica che la rete non può in nessun modo presentare dei *loop*, perché se ci fossero dei cicli, si presenterebbero situazioni in cui l'input per la funzione di attivazione f di un determinato neurone dipende anche dall'output dello stesso e sarebbe quindi difficile dare un senso alla rete.

Tuttavia, ci sono altri modelli di reti neurali artificiali in cui è possibile considerare delle strutture cicliche. Questi modelli sono chiamati *recurrent neural networks*, o reti neurali ricorrenti. Alla base di questa tipologia di reti c'è la considerazione del tempo e il concetto di memoria. Ovvero, oltre ai nuovi dati ricevuti in input, i neuroni all'interno di una rete ricorrente possono dipendere anche dal loro output prodotto precedentemente a partire da un dato differente. I loop non causano problemi in un modello di questo tipo, perché l'output di un neurone influisce sul suo input solo in un secondo momento e non istantaneamente.

6.1 La struttura

Partendo da un esempio molto semplice, se si considera una *recurrent neural network* composta da un unico neurone, essa non è altro che una funzione ricorrente:

$$s_t = f(s_{t-1}, x_t)$$

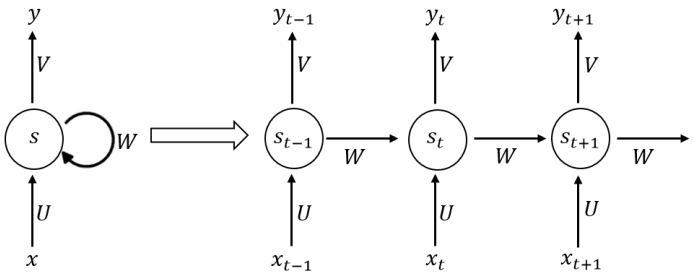

Figura 4: Visualizzazione grafica della relazione di una *recurrent neural network*. Sulla sinistra una vista compatta del neurone, mentre sulla destra uno schema del suo sviluppo sulla sequenza di tempi $t-1$, t e $t+1$

dove f è una funzione differenziabile, s_t è un vettore di valori che rappresenta lo stato interno della rete al tempo t, e x_t è l'input fornito alla rete al passo t. A differenza delle reti viste sin'ora, dove la funzione di attivazione di un neurone dipende unicamente dall'input corrente e dai pesi e il bias associati, in questo caso s_t è una funzione sia dell'input corrente che dello stato precedente s_{t-1}. Questo tipo di relazione appena descritta può essere rappresentata con lo schema in Figura 4.

Una *recurrent neural network* ha quindi bisogno di tre insiemi di parametri:

1. U, utilizzato per trasformare l'input x_t nello stato s_t;
2. W, utilizzato per trasformare lo stato precedente s_{t-1} nel nuovo stato s_t;
3. V, utilizzato per mappare il nuovo stato interno appena calcolato s_t nell'output y_t.

Lo stato interno della rete si può quindi riscrivere nel modo seguente:

$$s_t = f(s_{t-1} * W + x_t * U)$$
$$y_t = s_t * V$$

Ovviamente, anche nel caso delle *recurrent neural networks*, è possibile ampliare la rete appena descritta considerando più strati di neuroni. Lo stato $s_t^{(l)}$ di una cella ricorrente appartenente allo strato l al tempo t, sarà una funzione

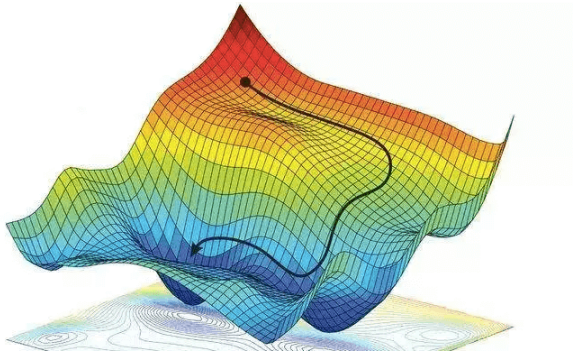

Figura 5: *Gradient Descent*. Grafico della funzione costo con in nero una possibile traiettoria del *gradient descent*. Partendo da un altro punto la traiettoria sarebbe sicuramente stata differente e avrebbe potuto raggiungere un diverso minimo della funzione costo

dell'output $y_t^{(l-1)}$ della cella ricorrente appartenente allo strato $l-1$ al tempo t e dello stato $s_{t-1}^{(l)}$ della cella dello stesso strato l al tempo precedente $t-1$:

$$s_t^{(l)} = f(s_{t-1}^{(l)}, y_t^{(l-1)})$$

7 Gradient Descent

Alla base degli algoritmi di *Machine Learning* c'è il concetto di *ottimizzazione*. Per apprendere autonomamente dai dati, infatti, solitamente si definisce una funzione che rappresenti l'errore commesso dall'algoritmo nell'eseguire le sue previsioni. Questa funzione prende il nome di *funzione costo* e l'obiettivo del problema di ottimizzazione è quello di minimizzarla. Uno dei metodi di ottimizzazione più diffusi è il *gradient descent*: l'algoritmo riceve in input una funzione convessa e ad ogni iterazione modifica i suoi parametri con lo scopo di minimizzarla, fintanto che un minimo locale non viene raggiunto.

Data la funzione costo $J \in C^1$, $J : \mathbb{R}^n \to \mathbb{R}$, la regola di aggiornamento è la seguente:

$$w^{(k+1)} = w^{(k)} - \alpha \cdot \nabla J(w^{(k)})$$

dove α è il valore di apprendimento e $\nabla J(w^{(k)})$ è il gradiente della funzione costo.

Il valore di apprendimento α rappresenta la lunghezza del passo che si sta compiendo verso il minimo; se α è troppo piccolo, l'algoritmo può risultare troppo lento, se invece è troppo grande può con un passo "saltare" il punto di minimo e far addirittura divergere l'algoritmo. La scelta del valore di α deve quindi essere presa assicurandosi che l'algoritmo converga e che lo faccia in un tempo ragionevole. Generalmente α è fisso, però si può anche far variare tra un'iterazione e la successiva, diminuendo il suo valore nel tempo, così da ottenere una convergenza migliore al minimo. Ad esempio

$$\alpha = \frac{\text{costante}_1}{\text{iterazione} + \text{costante}_2}$$

In base a come vengono gestiti i dati di input si distinguono diversi tipi di gradient descent:

- ***Batch Gradient Descent***: lavora calcolando l'errore per ognuno dei dati forniti in input e aggiornando i pesi considerandoli tutti ogni volta.
- ***Stochastic Gradient Descent***: anche questo metodo considera tutti i dati forniti in input, però questa volta i parametri vengono aggiornati considerando un singolo dato alla volta.
- ***Mini-Batch Stochastic Gradient Descent***: in questa tecnica l'insieme dei dati viene diviso in sottoinsiemi di un numero fissato *m* di elementi (*mini-batch*), così da sfruttare entrambe le strategie precedenti.

8 Le reti GAN

Nell'ambito di questa tesi, la tecnica di *deep learning* utilizzata nel sistema di *password guessing* implementato è quella delle *generative adversarial networks*. In questa tecnica vengono addestrate in maniera competitiva due reti neurali: una rete generativa *G* in grado di imparare a generare nuovi dati aventi la

stessa distribuzione dei dati usati in fase di addestramento, e una rete discriminante D in grado di imparare a determinare se un campione proviene dalla distribuzione di G o da quella dei dati. La competizione alla base di questo modello, porta le due reti a migliorare le loro capacità fintanto che i dati artificiali non diventano indistinguibili dai dati reali.

Entrando più nel dettaglio, la rete generativa G prende in input una variabile z con densità di probabilità p_z e restituisce $x_g = G(z)$ che, al termine dell'addestramento, dovrebbe seguire la densità di probabilità desiderata, ovvero quella dei dati.

La rete discriminante D prende in input un dato x e restituisce la probabilità $D(x)$ che x sia un dato reale. x può essere un dato reale, in questo caso si denota con x_t (dall'inglese *true*) e segue la densità p_t, oppure può essere un dato generato da G, in questo caso si denota con x_g, la cui densità p_g è indotta da p_z attraverso G.

L'obiettivo della rete generativa è di ingannare la rete discriminativa il cui obiettivo è invece quello di distinguere i dati reali da quelli generati. La procedura di addestramento per D consiste quindi nel massimizzare la probabilità di assegnare la giusta etichetta ai dati ricevuti in input; contemporaneamente però si addestra G in modo tale da minimizzarla. Il problema di ottimizzazione che si ottiene per la rete GAN può essere scritto nel modo seguente:

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_t(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \quad (3)$$

In breve, il generatore cerca di minimizzare la funzione obiettivo, mentre il discriminatore cerca di massimizzarla. Ciò che ne risulta è quindi che durante il processo di apprendimento, i pesi della rete discriminante sono aggiornati in modo tale da aumentare la funzione V, mentre i pesi della rete generativa sono aggiornati in modo tale da ridurla.

La procedura appena descritta è formalmente presentata nell'Algoritmo 1 [2]. Nel ciclo più interno dell'algoritmo si addestra D a distinguere dati reali e dati generati. In questo modo D tenta di convergere al miglior discriminatore

possibile, ovvero

$$D_G^*(x) = \frac{p_t(x)}{p_t(x) + p_g(x)}$$

Terminato il ciclo più interno si aggiorna anche G che tenderà a spostarsi verso le zone che hanno maggiore probabilità di essere classificate da D come dati reali.

Al termine dell'addestramento G e D raggiungeranno un punto in cui nessuna delle due reti può più migliorarsi, perché $p_g = p_d$ e quindi il discriminatore non è più in grado di distinguere le due distribuzioni e pertanto $D(x) = \frac{1}{2}$.

Il motivo per cui l'aggiornamento di D avviene prima e più volte rispetto a quello di G è per prevenire uno dei più diffusi problemi delle *generative adversarial network*: il *mode collapse*, noto anche con il nome di *helvetica scenario*. Questo evento si presenta quando la rete generativa identifica una soluzione (una sola delle tante possibili) come la migliore per riuscire ad ingannare la rete discriminativa. La rete G inizia quindi a generare troppi dati con le stesse caratteristiche della soluzione individuata, limitando in questo modo l'apprendimento. Fondamentale per prevenire questo tipo di problematica è l'utilizzo del *mini-batch*.

Numerosi sono gli esempi di applicazione di queste reti GAN, tra cui si trovano: la trasposizione di un'immagine in un'altra, come ad esempio la trasformazione di una mela in un'arancia o di un cavallo in una zebra; la generazione di immagini a partire da foto; la generazione di immagini a partire da una descrizione; la ricostruzione di immagini a partire da frammenti di esse; la costruzione di modelli tridimensionali; la produzione di testi e frasi di senso compiuto; la predizione e la generazione di un frammento di video.

9 Un sistema basato sul deep learning

L'impiego di un modello basato sul *machine learning* per l'automazione di un compito, non consiste nella semplice implementazione di un algoritmo, ma nella realizzazione di un intero sistema, costituito da funzioni e programmi

Algoritmo 1 Addestramento di una *Generative Adversarial Network* utilizzando il *Mini-batch stochastic gradient descent*. Il termine k è il numero di step da applicare al discriminante e viene fornito in input.

1: **per** numero di iterazioni di addestramento **ripeti**
2: **per** k passi **ripeti**
3: campiona un minibatch di m elementi $\{z^{(1)}, \ldots, z^{(m)}\}$ dalla distribuzione $p_g(z)$
4: campiona un minibatch di m elementi $\{x^{(1)}, \ldots, x^{(m)}\}$ dalla distribuzione $p_t(x)$
5: aggiorna i pesi della rete discriminante D ascendendo il suo gradiente stocastico:

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

6: **fine-ciclo**
7: campiona un minibatch di m elementi $\{z^{(1)}, \ldots, z^{(m)}\}$ dalla distribuzione $p_g(z)$
8: aggiorna i pesi della rete generativa G discendendo il suo gradiente stocastico:

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

9: **fine-ciclo**

che implementano le diverse fasi del processo di apprendimento e di automazione del compito che intendiamo affidare al sistema stesso (nel nostro caso il *password guessing* a partire da un file di hash di password):

1. **Acquisizione dei dati in input**: alla base di ogni algoritmo di *machine learning* ci sono i dati. Questi dati possono provenire da diverse fonti e avere formati differenti. Per esempio se si sta lavorando su un progetto di riconoscimento o creazione di immagini, allora i dati saranno delle foto, mentre nella maggior parte dei casi i dati sono di tipo alfanumerico e vengono raccolti in una forma tabellare, simile ad un foglio di calcolo.

2. **Preelaborazione dei dati**: i dati grezzi sono spesso disordinati, incompleti o soggetti a errori; hanno quindi bisogno di essere ripuliti e uniformati prima di essere forniti in input all'algortimo. A volte si stravolge completamente la loro natura, ad esempio da attributi grafici o testuali si trasformano in numeri, mentre in altri casi è sufficiente standardizzarli, tramite ad esempio la normalizzazione o il *riscalamento*, affinché l'algoritmo non dia troppa enfasi ad alcuni dati rispetto ad altri. Molto spesso i dati presi dal mondo reale sono incompleti e in questo caso si può agire in diversi modi: la via più semplice è di eliminare le righe e/o le colonne che presentano delle mancanze; altrimenti, nel caso di dati numerici, una possibilità è quella di sostituire le informazioni mancanti con la media di quelle che già si conoscono; o un'altra scelta ancora potrebbe essere quella di predire il valore sconosciuto basandosi sulle informazioni che si possiedono. Infine, l'ultimo passaggio della preelaborazione dei dati è la suddivisione del dataset in due sottoinsiemi: il *training set*, utilizzato durante tutta la fase di addestramento, e il *test set*, utilizzato per verificare le performance dell'algoritmo.

3. **Analisi esplorativa dei dati** (EDA, dall'inglese *Exploratory Data Analysis*): in questa fase l'obiettivo è quello di comprendere la vera natura dei dati, a livello qualitativo, tramite una rappresentazione grafica. Si utilizzano infatti diverse tipologie di grafici, come ad esempio gli istogrammi

o gli *scatter plot*. Questo processo permette quindi di scegliere le caratteristiche dei dati che sono utili all'apprendimento e di costruirne di nuove (*feature engineering*).

4. **Costruzione del modello**: in base alla natura del problema, si sceglie l'algoritmo di *machine learning* da utilizzare. In questa fase un passaggio fondamentale è l'ottimizzazione degli *iperparametri*, ovvero di tutti quei parametri scelti in modo tale da favorire il processo di apprendimento, come ad esempio in una rete neurale il numero di strati, o il numero di attivazioni in ciascuno strato. Fanno parte di questa fase anche l'addestramento del modello e la valutazione delle sue performance.

5. **Utilizzo del modello**: una volta eseguiti tutti i passaggi precedenti, il sistema è pronto per essere utilizzato su nuovi dati.

Il programma implementato, scritto in linguaggio Python, utilizza diverse librerie tipiche dei software per il calcolo numerico ed altre specifiche per l'implementazione di sistemi di machine learning:

- **pandas**: è la libreria per l'analisi dei dati e pertanto l'ho utilizzata in fase di preelaborazione e di analisi esplorativa dei dati (EDA) per lo studio e la sistemazione dei dati in mio possesso;

- **numpy**: è il package che raccoglie le funzioni matematiche di calcolo numerico;

- **random**: è la libreria utilizzata per la generazione di numeri casuali;

- **tensorflow**: è la libreria open source, utilizzata e implementata da Google, per l'implementazione di sistemi di machine learning; **keras** è un insieme di funzioni e oggetti definiti sulla base di tensorflow per accelerare l'implementazione di modelli di deep learning;

- **hashlib**: è la libreria che offre numerose funzioni di hash, utilizzate in alcune procedure accessorie del mio programma.

Il programma è articolato su diversi sotto-programmi, utilizzati per la preparazione dei dati e la fase di pre-processing, la fase esplorativa dei dati, l'implementazione dei modelli, la fase di addestramento, quella di testing ed infine per l'implementazione dell'algoritmo di *password guessing* vero e proprio. Di seguito un dettaglio dei sotto-programmi sviluppati per la realizzazione del sistema:

- *Fase di Preprocessing*: questa fase è implementata da due programmi distinti, **preprocessing.py** e **prep-data-tf.py**. Il primo contiene l'analisi dei dati presenti nell'insieme RockYou e l'eliminazione delle righe caratterizzate da dati mancanti o incompleti. Il secondo programma ha invece come obiettivo quello di produrre come output i due file «`train.txt`» e «`test.txt`», utilizzati rispettivamente in fase di training e testing del sistema. In un primo momento filtra l'insieme di partenza tenendo unicamente le password composte da non più di 10 caratteri; successivamente mescola le password per ottere una distribuzione casuale e infine le suddivide in training set (l'80%) e test set (il restante 20%).

- *Analisi esplorativa dei dati (EDA)*: l'analisi esplorativa dei dati è implementata dal programma **eda.py**, in cui analizzo i dati utilizzati in fase di addestramento del sistema. Inoltre aggiungo l'informazione aggiuntiva della lunghezza della stringa, così da studiarne anche la distribuzione all'interno del train set.

- *Modelli*: il programma **models.py** definisce il sistema GAN. In questo programma ho infatti indicato le strutture delle reti neurali generativa e discriminativa e le ho poi concatenate per ottenere la rete GAN. In particolare per la rete discriminativa *D* ho scelto una struttura semplice con tre strati: quello di input con 10 nodi, ognuno rappresentante un carattere della stringa, il secondo con 10 e l'ultimo con un unico neurone, quello di output. La rete generativa *G* presenta una struttura molto simile a quella della rete discriminativa, infatti anche in questo caso abbiamo 3 strati: quello di input con 10 nodi, il secondo con 80 e l'ulti-

Iperparametro	**Rete D**	**Rete G**
Numero di strati	3	3
Numero di neuroni nello strato nascosto	10	80
Funzioni di attivazione utilizzate	*ReLu* e *sigmoide*	*ReLu* e *Leaky ReLu*
Metodi di regolarizzazione utilizzati	*dropout*	/
Funzione di ottimizzazione	*stochastic gradient descent*	*stochastic gradient descent*
Funzione costo	*binary cross entropy*	*mean square error*

Tabella 1: Iperparametri scelti per la rete discriminativa *D* e la rete generativa *G*

mo con 10 neuroni, ognuno rappresentante un carattere della stringa di output. Per entrambe le reti riporto una sintesi delle impostazioni degli iperparametri utilizzati in Tabella 1.

- *Fase di Addestramento*: la fase di addestramento della rete neurale è implementata mediante il programma **train.py**: prima di tutto si costruisce il sistema GAN. Si entra poi all'interno del ciclo di addestramento vero e proprio in cui si chiede alla rete *G* di generare 128 password che si utilizzano, etichettate con 0, insieme a 128 password prese dal test set e etichettate con 1, per addestrare la rete *D*. Questa operazione si ripete per 5 volte prima di addestrare anche la rete *G* e procedere con un'altra iterazione. Al termine del ciclo di addestramento il programma salva le reti *G*, *D* e *GAN*, così da poter riutilizzare le reti addestrate.

- *Fase di Testing*: la fase di testing consiste nel verificare l'efficacia della rete generativa addestrata. Le operazioni utilizzate per tale scopo sono implementate nel programma **testing.py**, in cui per 10.000 iterazioni viene chiesto alla rete *G* (precedentemente caricata) di generare 200 password, che vengono confrontate con quelle contenute nel file «`test.txt`», per vedere quante di esse è in grado di generarne. Inoltre le password generate in questa fase vengono tutte scritte sul file «`generated_password.txt`», così da poter osservare in un secondo momento le stringhe prodotte.

- *Programma di password guessing*: il programma **PasswordGuessing.py** riceve in input un file contente nomi utenti e digest delle rispettive password e cerca di individuarne il più possibile. È strutturato sulla base di tre blocchi principali: nel primo blocco tenta con una lista di password molto comuni (es. *password*, *1234*), nel secondo tenta con il nome utente e nell'ultimo utilizza la rete *G* risultante dal sistema GAN per produrre le stringhe da utilizzare nei vari confronti. Per questa terza fase abbiamo impostato un massimo di 10.000 iterazioni così da imporre un termine al programma per evitare che l'elaborazione continui potenzialmente in eterno.

- *Codice di altri programmi di utilità*: di seguito sono riportati i dettagli di alcuni programmi di utilità necessari per completare l'implementazione del sistema. Nel programma **dictionary.py** viene generato un dizionario utilizzato per tradurre le password in stringhe di numeri da fornire in input alla rete GAN. Inoltre qui vengono anche definite la funzione per criptare e decriptare le stringhe utilizzando il dizionario definito precedentemente. Il programma **utils.py** raccoglie una serie di funzioni richiamate più volte nei programmi già presentati. Contiene infatti la funzione `make_noise` utilizzata per creare il *random noise* da fornire in input alla rete generativa *G* e la funzione `load_dataset` utilizzata per caricare i dati a partire da un file di testo e contenente anche il processo di imbottitura (*padding*) necessario per uniformare i dati. Nel programma **Utils_hashes.py** invece abbiamo implementato una funzione che riceve una stringa di caratteri come input e produce un vettore contenente i digest di 5 algoritmi di *hash* distinti: MD5, SHA-224, SHA-256, SHA-384 e SHA-512. Inoltre in questo programma viene anche applicata questa funzione per produrre i digest di alcune password molto comuni che poi il programma di *password guessing* utilizza nei vari confronti.

Password generate
Arib82
deed12
Sard290
cangu3
pybot44

Tabella 2: Alcune password generate dalla rete generativa *G* al termine dell'addestramento

10 Risultati ottenuti

In questo lavoro di tesi, oltre a studiare l'applicazione di tecniche avanzate di deep learning come le reti GAN a problematiche di cybersecurity, abbiamo voluto verificare il comportamento di una rete neurale GAN originale, implementata in modo differente rispetto a quanto proposto in letteratura dall'articolo di Ateniese ed altri [4]. In particolare è stato tentato un approccio che non facesse uso del modello convoluzionale di rete neurale, a differenza di quanto invece avviene sul sistema PassGAN proposto nell'articolo già citato.

I risultati ottenuti si avvicinano a quelli che avremmo sperato di ottenere (in Tabella 2 riporto alcune password generate dalla rete implementata), tuttavia sono parzialmente insoddisfacenti. Il lavoro proposto all'interno della tesi potrebbe infatti essere ulteriormente sviluppato al fine di verificare se una scelta diversa degli iperparametri e della struttura della rete GAN (con l'utilizzo ad esempio di reti ricorrenti) potrebbe portare a migliorare sensibilmente le prestazioni del sistema implementato o se invece è proprio l'uso di reti convoluzionali a consentire di produrre soluzioni soddisfacenti al problema del password guessing.

In particolare abbiamo visto come le reti neurali sono in grado di apprendere e ricreare schemi di creazione di password, anche senza l'utilizzo di informazioni particolari riguardanti l'utente. Se in fase di addestramento venissero

utilizzati anche i dettagli individuali di ogni singolo utente, aumentando quindi significativamente gli attributi del dataset di apprendimento, si potrebbero certamente ottenere dei risultati ancora più soddisfacenti.

Inoltre è bene sottolineare che gli unici strumenti utilizzati nel lavoro presentato in queste pagine sono due semplici reti neurali messe in competizione e il risultato ottenuto è paragonabile ai modelli di generazione di password basati su regole, in cui la tipologia di password che il modello è in grado di generare è dettato dal numero di regole definite. Il successo del *machine learning* in questo campo è proprio dovuto al fatto che non presenta questa limitazione e che è quindi in grado di produrre un numero potenzialmente inesauribile di password.

Questa capacità di generare un elenco di password verosimili (potenzialmente di qualsiasi dimensione) rappresenta dunque uno strumento potente nell'ambito del furto d'identità digitale. Tuttavia può essere utilizzata anche per lo scopo esattamente inverso, per ridurre cioè il rischio di tali attacchi: infatti il programma può essere utilizzato per proteggere determinati sistemi, avvisando gli utenti nel caso in cui la password da loro scelta dovesse risultare troppo semplice (se presente in un elenco prodotto dal sistema), o suggerendo lui stesso una password più sicura.

In conclusione, se è vero che l'apprendimento automatico può senz'altro influire positivamente sulla sicurezza delle password e, più in generale, nel mondo della sicurezza delle informazioni, è pur vero, però, che l'uso di password complesse rende di fatto impossibile la loro memorizzazione da parte di utenti umani e proprio schemi di attacco basati sul *machine learning* come quello presentato nelle pagine precedenti, mettono in evidenza un limite intrinseco dei modelli di autenticazione basati su password: l'utilizzo di strumenti di autenticazione a più fattori (*multi-factor authentication*) integrati con sistemi di analisi in tempo reale del rischio (*risk based authentication*) e di analisi comportamentale degli utenti (*user behavior analysis*), basati anche questi su tecniche di *machine learning*, sono al momento delle opzioni facilmente applicabili per correre ai ripari e innalzare il livello di cyber-difesa, proprio dove

sistemi tradizionali basati su password hanno dimostrato di essere facilmente attaccabili con queste tecniche innovative.

Riferimenti bibliografici

[1] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[2] Ian J. Goodfellow, Jean Pouget-Abadiey, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozairz, Aaron Courville, Yoshua Bengio, *Generative Adversarial Nets*, in Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, K. Q. Weinberger, *Advances in Neural Information Processing Systems 27*, Curran Associates, Inc., pp. 2672–2680, 2014.

[3] *HashCat*, `https://hashcat.net`, 2017.

[4] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, Fernando Perez-Cruz, *PassGAN: A Deep Learning Approach for Password Guessing*, arXiv:1709.00440v3, 2019.

[5] *John the Ripper*, `http://www.openwall.com/john/`, 2017.

[6] Tero Karras, Samuli Laine and Timo Aila. *A style-based generator architecture for generative adversarial networks*, In Proc. CVPR, 2018

[7] James Loy, *Neural Network Projects with Python*, Packt Publishing, 2019.

[8] Warren McCulloch, Walter S. Pitts (1943). *A logical calculus of the ideas immanent in nervous activity*. Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/bf02478259.

[9] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

[10] Arvind Narayanan and Vitaly Shmatikov, *Fast dictionary attacks on passwords using time-space tradeoff*, In Proceedings of the 12th ACM

conference on Computer and communications security. ACM, 364–372, 2005.

[11] National Institute of Standards and Technology, *Secure Hash Standard*, FIPS PUB 180, 1993.

[12] Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015

[13] Ronald Rivest, *The MD5 Message-Digest Algorithm*, 1992.

[14] RockYou, 2010. `https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt`

[15] Frank Rosenblatt, *Perceptron Simulation Experiments*, in Proceedings of the IRE, vol. 48, no. 3, pp. 301-309, March 1960, doi: 10.1109/JRPROC.1960.287598.

[16] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15, 1929-1958, 2014.

[17] *This Person Does Not Exist*, `https://thispersondoesnotexist.com/`, 2019.

[18] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants and Valentino Zocca. *Python Deep Learning, Exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow*, Packt Publishing, 2019.