SAPIENZA
UNIVERSITÀ DI ROMA

Analisi e implementazione di reti neurali per la sicurezza informatica

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Simone Valente
Matricola 1651220

Relatore

Prof. Emiliano Casalicchio

Anno Accademico 2024/2025

Analisi e implementazione di reti neurali per la sicurezza informatica
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: simone.valente95@gmail.com

Sommario

Questa relazione presenta il lavoro di tirocinio interno svolto presso l'Università La Sapienza di Roma sotto la guida del Professor Emiliano Casalicchio nell'ambito di ***Sistemi di rilevamento di intrusioni*** basati sull'apprendimento automatico.

L'obiettivo principale del tirocinio è stato quello di ottenere un modello di classificatore basato su reti neurali performante e in grado di ricevere dei precisi dati rappresentanti lo stato di una qualsiasi macchina per poi predire se su quest'ultima sia in corso un attacco e se si, di quale tipologia così da permettere di adottare le appropriate contromisure.

Oltre allo sviluppo di tale sistema, il tirocinio ha riguardato anche il confronto con alcuni problemi tutt'ora aperti nella ricerca scientifica riguardo i modelli basati su reti neurali e più in generale sull'apprendimento automatico.

Il sistema è stato realizzato mediante una rete neurale che riceve in input dei dati inerenti lo stato di una macchina e che, con una certa probabilità, classifica questi dati in un tipo di attaccho.

Per perseguire tale obiettivo però, in mancanza di dataset di addestramento particolarmente validi e aggiornati su tutti i tipi di attacchi, una parte consistente del lavoro è stata dedicata anche alla creazione di un dataset artificiale, sempre usando tecniche basate su reti neurali, con il fine di migliorare l'accuratezza del classificatore.

Contents

Chapter 1

Introduzione

Sono sempre maggiori le minacce informatiche che le macchine di tutto il mondo ricevono quotidianamente. Secondo il CyberSecurity Magazine, l'anno 2020, segnato dalla crisi del Covid-19, ha visto un incremento delle minacce ai dispositivi sfruttando lo smartworking e molte piccole e medie imprese sull'orlo della crisi, per non investire ulteriore denaro, hanno trascurato l'aspetto della sicurezza informatica trovandosi di fronte a perdite ben maggiori [1].

Basti pensare che il 43% di tutte le violazioni colpiscono piccole e medie imprese e che il 40% delle piccole imprese ha subito un grave attacco informatico con almeno 8 ore di inattività e quindi ingenti perdite [1].

Inoltre l'83% delle piccole e medie imprese non è finanziariamente preparato a riprendersi da un attacco informatico ma nonostante questo il 91% delle piccole imprese ritiene non necessario dotarsi di contromisure e investire nella sicurezza informatica e questo evidenzia quanto questo importante argomento sia ancora sottovalutato e fonte di confusione tra le imprese [1].

Con il passare del tempo, diventa sempre più importante che le imprese vengano informate sulle principali minacce e che si dotino non solo di sistemi automatici in grado di rilevare gli attacchi ma anche di personale preparato in grado di rispondere agli attacchi.

La presente relazione si propone di risolvere in parte questo problema, permettendo la costruzione di un sistema di rilevamento intrusioni (IDS) per rilevare attacchi in corso su una determinata macchina.

L'obiettivo principale di tale lavoro riguarda quindi l'addestramento, mediante un particolare dataset [7], di un modello di classificatore basato su rete neurale, installabile su qualsiasi macchina in grado di supportare il linguaggio Python e le relative librerie utilizzate nel progetto come Keras, Pandas, Numpy, Joblib, Sklearn.

Tale classificatore, al termine dell'addestramento è stato testato con una parte

del dataset non usata nell'addestramento, ottenendo la percentuale di campioni etichettati correttamente (accuracy) ed una volta salvato in formato h5 questo modello risulta in grado di ricevere in input dei dati rappresentanti lo stato di una macchina e di determinare se questi evidenziano o meno la presenza di un attacco in corso.

Altre ricerche hanno trattato lo sviluppo di sistemi di questo tipo basandosi però prevalentemente su tecniche di machine learning di diversa tipologia ed offrendo una panoramica di risultati di accuracy concentrati sui singoli tipi di attacchi. Ad esempio per gli attacchi come *DOS* e *Probe*, sono stati ottenuti ottimi risultati superiori al 99% mentre su altri tipi di attacchi i risultati sono nettamente più bassi a seconda della tecnica di machine learning utilizzata. [3]

Questa relazione invece si concentra sull'ottenimento di un singolo risultato di accuracy che possa evidenziare quanto il classificatore riesce a riconoscere in generale tutte le tipologie di attacco presenti nel dataset di apprendimento e soprattutto presenta un sistema basato esclusivamente sull'impiego di reti neurali, anziché su varie tecniche di machine learning.

Una delle sfide più importanti di questo lavoro è stata quella di ricercare un dataset di partenza ed arricchirlo mediante la costruzione di un altro modello basato su reti neurali ma di natura generativa così da creare nuovi campioni ed aggiungerli a quello di partenza per poi osservare le differenze nei risultati di accuracy.

I capitoli di questo documento sono organizzati nel seguente modo:

Nel Capitolo 2 viene descritta la ricerca di un dataset di sicurezza informatica valido ed il suo raffinamento per poi essere usato per l'addestramento del classificatore.

Nel Capitolo 3 viene descritta la costruzione di un classificatore di partenza basato su rete neurale e viene eseguita un'indagine sui risultati ottenuti da test automatici.

Nel Capitolo 4 vengono descritti due metodi per ricercare un'architettura di rete neurale che garantisca risultati migliori.

Nel Capitolo 5 viene descritta la generazione di nuovi campioni, per potenziare il dataset di partenza, mediante la realizzazione di un modello GAN e vengono affrontati alcuni dei problemi tutt'ora aperti nella ricerca scientifica per la generazione di nuovi campioni.

Nel Capitolo 6 vengono descritti i test del classificatore ultimato e vengono fatte alcune analisi sulle sue prestazioni.

Nel Capitolo 7 vengono descritte le conclusioni e viene spiegato come impiegare il sistema in un contesto reale.

Chapter 2

Preparazione dei dati

Prima di costruire il modello basato su rete neurale per classificare i dati del sistema in possibili attacchi, ovvero il classificatore, sono state intraprese alcune attività di preparazione:

- Ricerca di un dataset: questa fase serve a trovare un insieme di campioni contenenti i dati su cui poi il classificatore deve essere addestrato.
- encoding: questa fase serve a codificare i dati che costituiscono il dataset poiché alcuni di questi sono rappresentati in forma non numerica mentre una rete neurale può processare soltanto dati numerici [2]
- scaling: questa fase serve a distribuire i valori delle caratteristiche costituenti i campioni, su scale simili aiutando così la rete neurale ad apprendere meglio.
- suddivisione del dataset: questa fase serve a scegliere quanti campioni dedicare all'addestramento e quanti dedicarne al testing.

2.1 Ricerca del dataset di addestramento

Nel contesto della sicurezza informatica, una delle maggiori difficoltà, consiste nel riuscire a trovare dei dataset aggiornati e contenenti un gran numero di campioni di dati per ogni tipologia di attacco, con l'obiettivo di usare poi questi dati per addestrare un classificatore che si occupi di riconoscere attacchi informatici in corso.

I dati di interesse sono informazioni riguardanti le macchine su cui sono stati raccolti e la relativa situazione, ovvero se tali macchine in quel momento si trovassero sotto attacco o meno.

Tali dati possono essere rappresentati come un array di valori non necessariamente numerici, di conseguenza è stata dapprima avviata la ricerca di un dataset di questa

tipologia, dove ogni riga rappresentasse un campione di questi dati con la relativa etichetta di attacco.

Ad ogni riga del dataset corrisponde quindi un preciso "stato" in cui una determinata macchina si è trovata nel tempo e una tipologia di attacco se in quel momento su tale macchina era in corso quel tipo di attacco.

Dopo un'attenta ricerca, sono stati considerati i seguenti dataset pubblicamente disponibili online, analizzandone vantaggi e svantaggi in funzione dell'obiettivo finale di sviluppo del classificatore:

- KDD99
- UNSW-NB15

2.2 KDD99

La versione pubblicamente disponibile del dataset, contiene quasi 500,000 campioni ed ognuno di questi è composto da 42 valori che si riferiscono a 42 caratteristiche (etichetta inclusa) che rappresentano sia traffico di rete registrato, sia possibili stati anomali del sistema come ad esempio:

- Numero di login falliti in un certo lasso di tempo
- Azioni critiche rilevate (non necessariamente malevole)
- Numero di accessi effettuati in un certo lasso di tempo
- Dettagli su shell e privilegi

Ogni campione è etichettato con uno fra 23 possibili attacchi (incluso il caso "normale", ovvero quando non è in corso alcun attacco) tuttavia un punto debole di questo dataset risiede nel fatto che è fortemente sbilanciato. Infatti per alcune classi di attacco sono disponibili moltissimi campioni, ad esempio nel caso di attacchi "Smurf" sono disponibili più di 280,000 campioni (oltre il 50%, dell'intero dataset) mentre per altre classi di attacco ne sono disponibili numeri insignificanti come ad esempio i campioni etichettati come "Spy" che sono solo 2 e quindi inutili in quanto

nessun modello sarebbe in grado di imparare a riconoscere degli attacchi dopo aver visto solo 2 campioni.

Viene di seguito riportata la suddivisione dei campioni del dataset in base all'etichetta:

Etichetta	**Numero di campioni**
normal	97277
bufferoverflow	30
loadmodule	9
perl	3
neptune	107201
smurf	280790
guesspasswd	53
pod	264
teardrop	979
portsweep	1040
ipsweep	1247
land	21
ftpwrite	8
back	2203
imap	12
satan	1589
phf	4
nmap	231
multihop	7
warezmaster	20
warezclient	1020
spy	2
rootkit	10

Table 2.1. Suddivisione dei campioni KDD99 per etichetta

2.3 UNSW-NB15

La versione pubblicamente disponibile di questo dataset contiene circa 175,000 campioni che riguardano però solo una ristretta tipologia di attacchi informatici ed ogni campione è costituito da 45 valori che si rifericono a 45 caratteristiche (etichetta inclusa) riguardanti esclusivamente traffico di rete [8] e che non considerano possibili anomalie del sistema a differenza del dataset KDD99.

In totale, i tipi di attacco in cui sono mappati i campioni, sono solo 10 (incluso il caso normale).

Questo dataset è meno sbilanciato ma ha molti meno campioni, di cui quasi 60,000 (oltre il 30%) etichettati come "normale". Inoltre anche in questo dataset alcuni tipi di attacco sono rappresentati con pochi campioni rispetto al totale [8].

Di seguito la suddivisione completa:

Etichetta	**Numero di campioni**
Analysis	2000
Backdoor	1746
Dos	12264
Exploit	33393
Fuzzers	18184
Generic	40000
Normal	56000
Reconnaissance	10491
Shellcode	1133
Worms	130

Table 2.2. Suddivisione dei campioni UNSW-NB15 per etichetta

2.4 Scelta del dataset

Dopo aver analizzato le differenze fra i due dataset mostrati nella sezione precedente, è stato scelto di proseguire utilizzando il dataset KDD99 perché contiene più dati inerenti a più tipi di attacchi e contiene dati di più varia tipologia.

2.5 Encoding

Solitamente i dataset, contengono sia valori numerici che valori non numerici (categorici) ed anche il dataset KDD99 non fa eccezione.

Ad esempio per quanto riguarda i protocolli utilizzati, i possibili valori vengono sempre rappresentati mediante stringhe del tipo "tcp", "udp" e così via.

Di conseguenza è stato necessario convertire questi valori poiché per poter addestrare il classificatore e più in generale qualunque modello basato su reti neurali, è necessario che tutti i valori di ogni campione siano numeri.

Questo fondamentale requisito è dovuto al fatto che, se alcuni valori fossero categorici, il modello non riuscirebbe ad assegnare loro un ruolo nella determinazione dell'etichetta.

Sono stati presi in considerazioni due tipi di encoding:

- LabelEncoding
- OneHotEncoding

Usando il **LabelEncoding**, ogni valore categorico di una precisa caratteristica, viene trasformato in un intero positivo e questo va inevitabilmente a creare un ordine fra i valori [2].

Usando il **OneHotEncoding** invece, la colonna rappresentante i possibili valori della caratteristica, viene sostituita con un numero n di nuove colonne, dove n è la cardinalità dell'insieme di valori presenti nella colonna [2].

In totale vengono quindi aggiunte n-1 colonne ognuna contente come valore 0 oppure 1 e ciò serve ad identificare ogni valore senza però stabilire un ordine e dunque dare pesi diversi ai valori.

La scelta del tipo di encoding categoriale può determinare risultati di addestramento diversi, per questo motivo è stato necessario valutare il significato delle caratteristiche interessate per poter decidere quale dei due tipi di econding applicare.

Tali caratteristiche, sono:

- protocollo
- servizio
- flag

Un fattore che accomuna queste caratteristiche è l'assenza di un ordine naturale tra i valori. Ad esempio nel caso della caratteristica "protocollo", i possibili valori non sono legati da una qualche relazione che possa stabilirne un ordine naturale.

Per questo motivo, usare il labelEncoding potrebbe portare il modello ad imparare ordini inesistenti tra i dati e dunque peggiorare la sua accuratezza nel riconoscere le minacce informatiche, di conseguenza è stato scelto di usare il OneHotEncoding.

Lo svantaggio nell'uso del OneHotEncoding risiede nell'aumento del numero di colonne del dataset processabile e ciò va a rendere più computazionalmente oneroso l'addestramento del modello.

Nel caso specifico del dataset KDD99, applicando questo encoding si passa da un totale di 41 caratteristiche (l'etichetta è esclusa) ad un totale di 118 caratteristiche poiché la colonna protocollo viene sostituita con 3 nuove caratteristiche, mentre il servizio con 66 caratteristiche ed infine la flag con 11 caratteristiche.

Al termine del processo, è stato quindi creato un dataset raffinato costituito da 118 caratteristiche anziché 42 come in origine.

2.6 Scaling

Oltre all'encoding delle caratteristiche non numeriche, è necessario preparare i dati numerici affinché siano su scale adatte ad essere processate da una rete neurale. Questo perché, i modelli basati su reti neurali apprendono meglio se le caratteristiche del dataset di apprendimento sono distribuite in modo simile.

Ogni caratteristica del dataset originale, ha i valori distribuiti in una certa maniera ed è proprio questo il fattore principale che è stato preso in considerazione per decidere quale algoritmo di scaling applicare.

Analizzando il dataset KDD99 con uno script creato appositamente, è stato possibile osservare che i valori delle sue caratteristiche non seguono una distribuzione normale bensì fortemente asimmetrica, come riportato nelle seguenti immagini riguardanti 2 delle 41 caratteristiche:

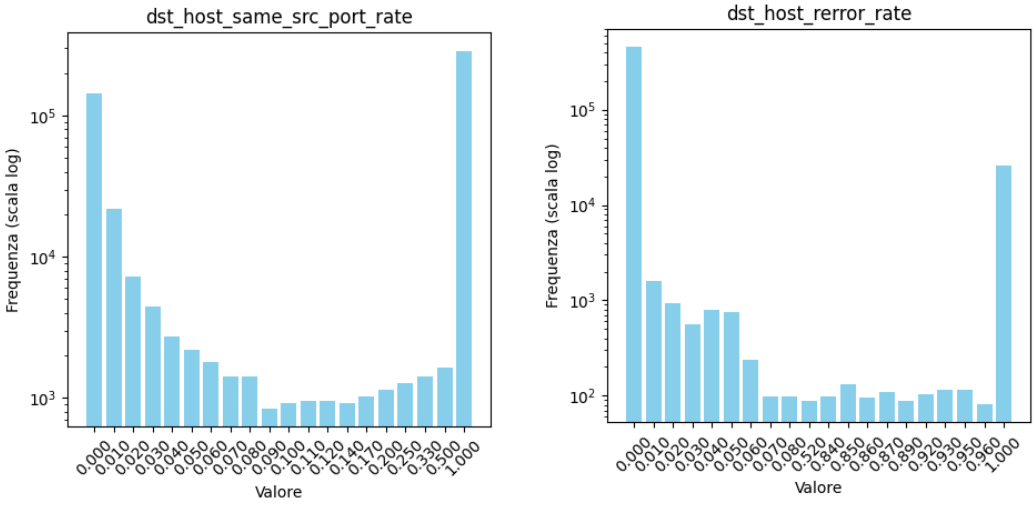

Figure 2.1. Distribuzione valori caratteristiche

Applicare un algoritmo di scaling come lo "Standard Scaling" non avrebbe un impatto positivo sulla trasformazione dei dati perché, per funzionare bene, richiede che essi seguano una distribuzione simile a quella standard.

In questo caso dunque, dato che per molte delle caratteristiche sono stati individuati anche diversi outlier, è stato scelto di applicare il "Robust scaler" che funziona bene proprio nei casi in cui sono presenti gli outlier e nei casi in cui i dati non seguono distribuzioni standard.

2.7 Suddivisione del dataset

Dato che, la valutazione di un modello addestrato su un dataset deve essere fatta con diverse combinazioni di campioni, è stata innanzitutto decisa la percentuale di dati dedicata al training e quella dedicata al testing, durante la fase di valutazione sono stati alternati in modo automatico e pseudo-casuale l'insieme dei campioni usati per l'addestramento e l'insieme dei campioni usati per il testing. Per farlo correttamente, è stato necessario suddividere il dataset KDD99 in 23 parti, ognuna contenente i soli dati etichettati in uno dei 23 tipi di attacco.

Ciò è stato fatto perché altrimenti non sarebbe possibile garantire che i dati di training e di test coprano tutte le etichette e nella percentuale specificata. Ad esempio potrebbe accadere che vengano selezionati per il training tutti i campioni di un certo tipo di etichetta e che quindi quelli relativi a questa etichetta non vengano mai testati, oppure potrebbe accadere che i campioni di una certa etichetta vengano usati tutti per il testing.

Dopo aver suddiviso correttamente il dataset, ogni parte ottenuta è stata gestita

come un dataset a sé stante durante il training.

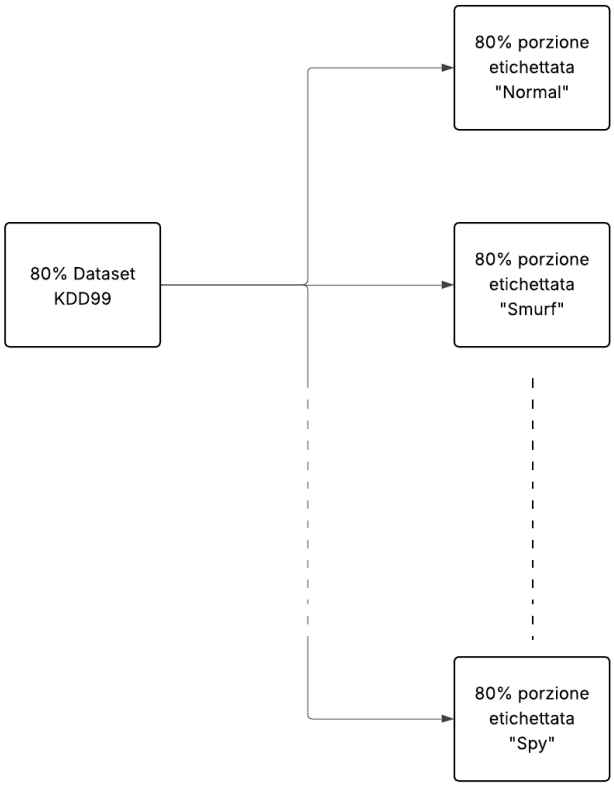

Figure 2.2. Suddivisione per il training

Chapter 3

Costruzione del modello

3.1 Scelte primarie

Per costruire il classificatore basato su rete neurale ed addestrarlo sul dataset KDD99 [7] di cui si è discussa la raffinazione e suddivisione nel precedente capitolo, è stato necessario fare delle scelte di partenza per creare la rete neurale adatta, con particolare attenzione a migliorare via via le scelte sulla base dei risultati ottenuti.

Le prime scelte da considerare sono state le seguenti:

- Percentuali di suddivisione del dataset per il training-testing
- Tipologia di modello
- Numero di strati
- Numero di neuroni per ogni strato
- Funzione di attivazione per ogni strato
- Funzione di perdita
- Funzione di ottimizzazione
- Parametri di addestramento

3.2 Percentuali training-testing

Come scelta iniziale per le percentuali di dataset da utilizzare per il training-testing del classificatore, è stata scelta la regola 80-20, quindi l'80% dei campioni (di ognuna delle 23 porzioni del dataset) è stato riservato al training e il restante 20% (di ognuna delle 23 porzioni del dataset) riservato al testing.

Nella seguente immagine viene mostrato uno schema dell'addestramento e del testing.

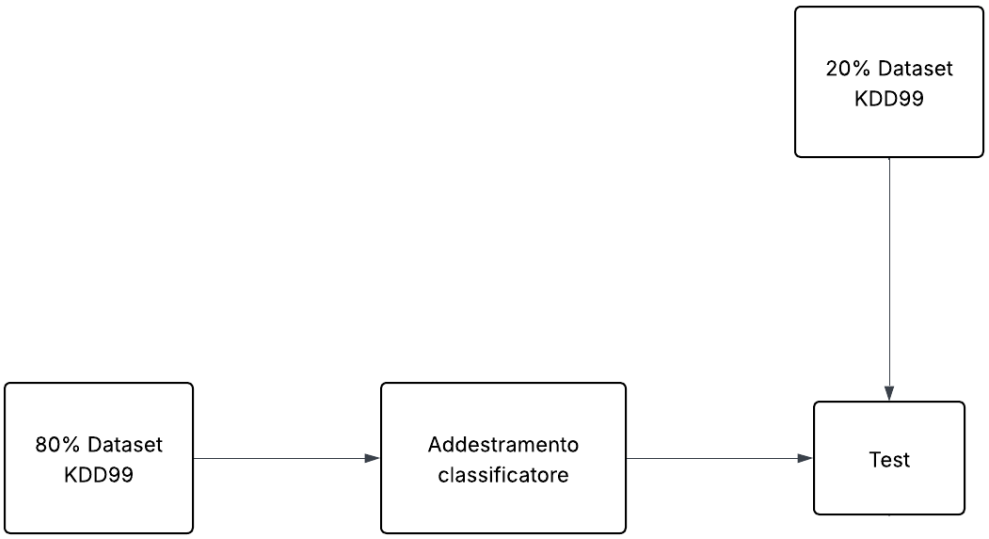

Figure 3.1. Schema delle percentuali di dati KDD99 usate per l'addestramento ed il testing

3.3 Tipologia di modello del classificatore

Per costruire il classificatore in grado di riconoscere i possibili attacchi dai dati, è stato scelto di usare il modello Multi-layer Perceptron a 3 strati: un primo strato di input composto da un numero di neuroni pari al numero di caratteristiche in

ingresso ovvero 118 elabora i valori delle caratteristiche del dataset, un secondo strato nascosto composto da 236 neuroni rielabora le informazioni ricevute dallo strato di input ed infine un terzo strato di output composto da 23 neuroni stabilisce per ogni tipo di attacco la probabilità che i dati in ingresso evidenzino tale attacco.

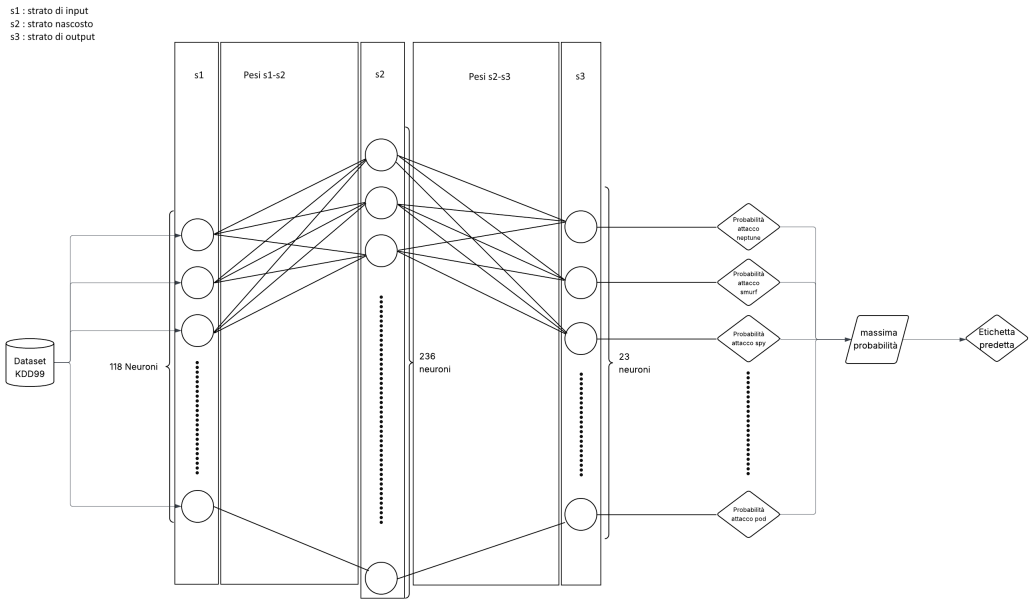

Figure 3.2. schema della prima rete neurale realizzata

3.4 Funzioni di attivazione

Riguardo le funzioni di attivazione negli strati della rete neurale del classificatore, è stato scelto di usare la funzione **Relu** per il primo ed il secondo strato, poiché tale funzione è generalmente indicata per questo tipo di strati e statisticamente ha buone prestazioni su problemi di classificazione come questo.

Per lo strato finale invece, è stato scelto di usare la funzione **Softmax** perché in tal modo i dati ricevuti dallo strato superiore vengono trasformati in probabilità per ciascun tipo di attacco, come mostrato nel seguente esempio di esecuzione di una previsione:

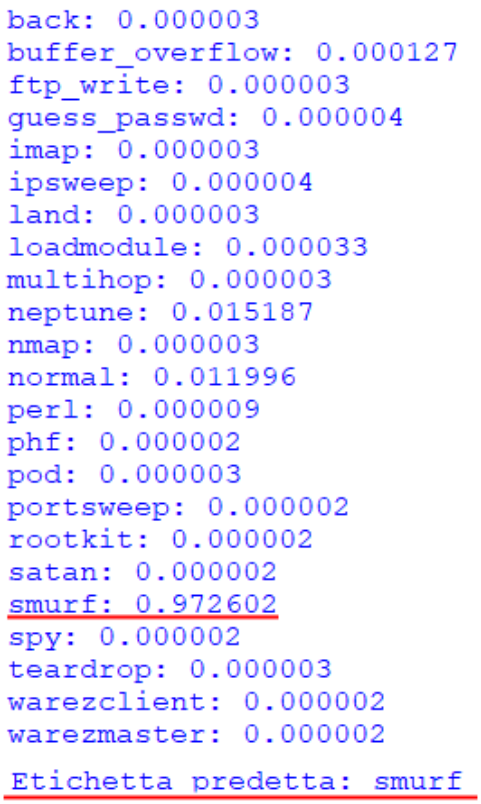

Figure 3.3. Esempio di stampa probabilità classificazione ultimo strato

Come si può notare dalla figura, la rete neurale dopo aver processato uno dei campioni ricevuti, ha dato in output per ogni tipologia di attacco, la probabilità che tale campione indicasse quel preciso attacco. In questo caso con il 97% di probabilità il classificatore ha ritenuto che si trattasse di un attacco di tipo "smurf".

3.5 Funzione di perdita

Per quanto riguarda la funzione di perdita, usata per calcolare l'errore sulle previsioni, è stato scelto di usare la **CategoricalCrossEntropy** perché tale funzione si presta bene in casi di classificazione multiclasse come questo. La perdita calcolata da tale funzione viene poi usata nella fase di backpropagation della rete in cui, dopo aver calcolato l'errore sulle probabilità delle previsioni effettuate, tale errore viene propagato all'indietro fra gli strati per aggiornare i pesi e permettere dunque alla rete di valutare meglio i successivi campioni.

3.6 Funzione di ottimizzazione

Per la funzione di ottimizzazione per l'aggiornamento dei pesi invece, è stata scelta la funzione "Adam" per via della sua generica validità.

3.7 Parametri di addestramento

Sono stati poi considerati dei parametri inerenti il processo di addestramento del classificatore relativi al tasso di apprendimento (learning rate), il numero di volte che i campioni vengono processati (numero di epoche) ed il numero di campioni dopo i quali aggiornare i pesi (batch size).

Quindi ricapitolando, la rete neurale del classificatore è così costituita:

- Numero di neuroni nello strato di input (s1) = 118 con attivazione Relu
- Numero di neuroni nello strato nascosto (s2) = 236 con attivazione Relu
- Numero di neuroni nello strato di output (s3) = 23 con attivazione Softmax
- Funzione di perdita Categorical CrossEntropy
- Funzione di ottimizzazione Adam
- Parametri:
  - Learning rate = 0.0001
  - Epoche = 1
  - Batch size = 64

3.8 Implementazione Python

Per l'implementazione è stato scelto di usare il linguaggio Python con l'uso di apposite librerie, quali pandas per avere a disposizione algoritmi efficienti per processare il dataset, sklearn per l'encoding e lo scaling ed infine Keras per creare il modello e gestire l'apprendimento ed il testing.

```
originale = pd.read_csv(training)

# DataFrame vuoti per train e test
d1 = pd.DataFrame(columns=originale.columns)
d2 = pd.DataFrame(columns=originale.columns)

# Itera su ogni etichetta
for label in originale['label'].unique():
    subset = originale[originale['label'] == label]

    test_perc_h = test_perc

    if (1-test_perc_h) * len(subset) < 1.0:
        train_t = 1.0/len(subset)
        test_perc_h = 1-train_t

    # Split
    train_part, test_part = train_test_split(
        subset,
        test_size=test_perc_h
    )

    # Aggiungi ai DataFrame finali
    d1 = pd.concat([d1, train_part], ignore_index=True)
    d2 = pd.concat([d2, test_part], ignore_index=True)

train_data = d1
df_test = d2

# Separa le caratteristiche e l'etichetta per il dataset
   di addestramento
X_train = train_data.drop('label', axis=1)
y_train = train_data['label']

# Calcolo delle frequenze delle etichette
class_counts = y_train.value_counts()
class_weights = {i: 1.0 / count for i, count in
   class_counts.items()}

# Normalizzazione dei pesi
total_weight = sum(class_weights.values())
class_weights = {k: v / total_weight for k, v in
   class_weights.items()}
```

```
# Codifica le variabili categoriali
categorical_cols = [col for col in train_data.columns if
    not pd.api.types.is_numeric_dtype(train_data[col])]

categorical_cols.remove('label')
encoder = OneHotEncoder(handle_unknown='ignore',
    sparse_output=False, drop='first')
X_train_encoded = encoder.fit_transform(X_train[
    categorical_cols])

# Crea DataFrame per le colonne codificate
X_train_encoded_df = pd.DataFrame(X_train_encoded, columns
    =encoder.get_feature_names_out(categorical_cols))

# Rimuovi le colonne originali e aggiungi quelle
    codificate
X_train = pd.concat([X_train.drop(categorical_cols, axis
    =1).reset_index(drop=True), X_train_encoded_df.
    reset_index(drop=True)], axis=1)

# Codifica le etichette in formato one-hot encoding
y_train = pd.get_dummies(y_train).values

# Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)



num_neuroni_output = y_train.shape[1]

# Crea il modello
model = Sequential([
    Dense(num_neuroni_input, activation='relu',
        input_shape=(X_train.shape[1],)),
    Dense(num_neuroni_nascosti, activation='relu'),
    Dense(num_neuroni_output, activation='softmax')
])

# Funzione di perdita
loss = CategoricalCrossentropy()

# Compila il modello
```

```
ottimizzatore = Adam(learning_rate=learning_rate)
model.compile(optimizer=ottimizzatore, loss=loss, metrics
    =['accuracy'])

# Addestra il modello con pesi per classe
model.fit(X_train, y_train, epochs=epoche, batch_size=
    batch, validation_split=test_perc, class_weight=
    class_weights)

# Salva il modello, encoder, scaling ed etichette
model.save('modello_addestrato.h5')
joblib.dump(scaler, 'scaler.joblib')
joblib.dump(encoder, 'encoder.joblib')

labels = list(pd.get_dummies(train_data['label']).columns)
joblib.dump(labels, 'labels.joblib')
```

Listing 3.1. Estratto codice Python: Training del modello

3.9 Primi risultati

La valutazione del modello descritto precedentemente, è stata eseguita conducendo più di 1000 esperimenti in modo automatico mediante uno script, cambiando di volta in volta, i campioni scelti casualmente per training e testing e salvando alcuni dati per ogni esperimento. Ogni esperimento è costituito da un addestramento ed un testing mirato a raccogliere dati come ad esempio la percentuale di successo da parte del modello nell'indovinare la categoria di attacco dei campioni riservati al testing.

```
# Carica il modello, lo scaler e l'encoder
model = keras.models.load_model('modello_addestrato.h5')
scaler = joblib.load('scaler.joblib')
encoder = joblib.load('encoder.joblib')
labels = joblib.load('labels.joblib')


df = df_test
df['label'] = df['label'].str.strip("\n")
```

```
# Contatore delle occorrenze
occorrenze = df['label'].value_counts().to_dict()
corrette = {label: 0 for label in occorrenze}

# Trasformazione in batch
encoded = encoder.transform(df[categorical_cols])
encoded_df = pd.DataFrame(encoded, columns=encoder.
    get_feature_names_out(categorical_cols))
final_df = pd.concat([df.drop(categorical_cols+['label'],
    axis=1).reset_index(drop=True),
                      encoded_df.reset_index(drop=True)],
                         axis=1)
final_df = final_df.reindex(columns=scaler.
    feature_names_in_)
scaled_data = scaler.transform(final_df)

# Predizioni in batch
predictions = model.predict(scaled_data)
predicted_indices = predictions.argmax(axis=-1)
predicted_labels = [labels[idx].strip("\n") for idx in
    predicted_indices]

# Calcola le previsioni corrette
for actual, predicted in zip(df['label'], predicted_labels
    ):
     if actual == predicted:
         corrette[actual] += 1


#conta tutti i campioni
allsample = df['label'].count()

totalcorr = 0
for attack_type in corrette:
     totalcorr += corrette[attack_type]

result = totalcorr/allsample
```

Listing 3.2. Estratto codice Python : Testing del modello

I dati raccolti al termine della serie di esperimenti riguardanti l'accuracy, ovvero la percentuale di campioni etichettati correttamente nella fase di testing (post-addestramento), sono presenti nella seguente figura:

Numero di esperimenti	1010
Media di accuracy	98.215653
Varianza di accuracy	0.000016
Deviazione standard di accuracy	0.003971
Valore minimo di accuracy	98.187467
Valore massimo di accuracy	98.223900
Range dei valori di accuracy	0.036433

Table 3.1. Risultati iniziali accuracy classificatore

Come si può notare guardando la voce "Media di accuracy", per il 98.215% dei campioni sottoposti al testing, il classificatore etichetta correttamente i campioni, di conseguenza nell'1.8% circa dei casi il modello fallisce e non riesce ad etichettare correttamente i dati, confondendo un attacco con un altro tipo o addirittura scambiandolo per traffico normale.

D'altra parte però, i risultati ottenuti sono stabili in quanto il valore minimo e il valore massimo dell'accuracy sono abbastanza vicini ed il numero di esperimenti effettuati comporta un'alta probabilità che i risultati siano attendibili e che l'apprendimento del classificatore sia avvenuto con successo.

3.10 Indagine sul tasso di fallimento

Per approfondire le ragioni del tasso di fallimento dell'1.8% nel riconoscere gli attacchi da parte del classificatore, è stato scelto di calcolare il tasso di fallimento separatamente su ogni porzione del dataset, con lo scopo di identificare su quali tipi di attacco il modello sbagliasse più spesso.

I risultati di questi altri esperimenti, hanno evidenziato che si possono distinguere quattro gruppi di tipi di attacco sulla base della difficoltà che il classificatore possiede nel riconoscerli.

- **Primo gruppo**: questo gruppo è composto dai tipi "smurf", "neptune", "normal". Per tali categorie di attacco infatti, la percentuale di riconoscimento da parte del modello è molto alta e mediamente vicina al 100%.
- **Secondo gruppo**: questo gruppo è composto dalla tipologia di attacco "bufferoverflow".

  Per questa categoria di attacco, in base a quali campioni vengono usati per il training ed il testing si hanno risultati molti diversi. Si ha un valore medio di riconoscimento di poco superiore al 50% ma si ha anche una deviazione

standard abbastanza alta, vicina a 0.2. In alcuni esperimenti il riconoscimento è stato pari al 100%, in molti altri invece, è stato pari a 0%.

- **Terzo gruppo**: questo gruppo è composto da due tipologie di attacco: "loadmodule" e "perl". Per entrambe, la percentuale media di riconoscimento è bassissima, nel caso di "loadmodule" è dello 0.14%, mentre nel caso di "perl" è dello 0.09%. Nel caso di "loadmodule" ci sono stati alcuni esperimenti in cui il riconoscimento ha raggiunto al massimo il 50%, mentre nel caso di "perl" in alcuni esperimenti è stato raggiunto il 100% nonostante la sua media sia più bassa.

- **Quarto gruppo**: questo gruppo è composto da tutte le restanti categorie di attacco, ovvero "back", "ftpwrite", "guesspasswd", "imap", "ipsweep", "land", "multihop", "nmap", "phf", "pod", "portsweep", "satan", "rootkit", "spy", "teardrop", "warezclient", "warezmaster". Per tali categorie la percentuale di riconoscimento è sempre pari allo 0% in tutti gli esperimenti effettuati quindi il modello non è riuscito mai a riconoscerli.

3.11 Motivazioni

Le motivazioni di queste differenze nel riconoscimento da parte del modello, risiedono principalmente nella quantità di campioni disponibili per ogni attacco. Infatti, per gli attacchi in cui il modello riesce ad avere buone prestazioni vicine al 100% è possibile notare che sono disponibili grandi quantità di campioni, ad esempio per la porzione costituita da attacchi di tipo "smurf" sono disponibili 280.000 campioni che rappresentano oltre il 50% di tutto il dataset completo ed è per questo che il modello riesce poi a riconoscerli facilmente.

Al contrario invece, per gli attacchi che il modello non riesce mai a riconoscere, si può notare che sono disponibili pochissimi campioni.

Tuttavia, questo non sembra essere l'unico fattore che determina il fallimento del modello in quanto per gli attacchi bufferoverflow (secondo gruppo) sono disponibili solo 30 campioni e nonostante ciò, il modello riesce comunque a riconoscerli in molti degli esperimenti effettuati, a differenza invece degli attacchi "back" che pur consistendo di oltre 2000 campioni, non sono mai stati riconosciuti in alcuno degli oltre 1000 esperimenti effettuati.

Dopo aver predisposto uno script per automatizzare addestramento e testing su ogni porzione, è stato anche costruito un analizzatore di dati per osservare meglio tutte le informazioni raccolte negli esperimenti .

Tale analisi ha permesso di riscontrare che per alcune tipologie di attacchi su cui il classificatore ha scarso successo, i relativi campioni vengono comunque identificati

come attacchi.

Queste categorie sono:

Tipologia di attacco	percentuale confusione come altro attacco
satan	91
ipsweep	1.20
portsweep	29.33
teardrop	8.16
rootkit	50
imap	100

Table 3.2. Confusioni attacchi

Chapter 4

Miglioramento del modello mediante ricerca di nuove architetture

Con lo scopo di migliorare l'accuracy del classificatore, è stato ampliato lo script Python di cui è stato visto un estratto in precedenza, adottando due diverse tecniche con l'obiettivo di trovare i parametri di addestramento migliori quali il numero di epoche, la dimensione del batch, il tasso di apprendimento e i numeri di neuroni nei vari strati, per massimizzare l'accuracy.

4.1 Ricerca Random

Questa tecnica è basata sull'idea di scegliere casualmente le proprietà di un processo al fine di esplorare la dimensione dei possibili risultati. In questo contesto quindi, sono stati scelti i parametri di addestramento da far variare all'interno di precisi intervalli. Vengono mostrati di seguito gli intervalli di esplorazione:

- num neuroni input = random.randint(10, 500)
- num neuroni nascosti = random.randint(10, 800)
- epoche = random.randint(1, 10)
- batch = random.randint(32, 5000)
- learning rate = random.uniform(0.00001, 0.01)

Conducendo un certo numero di esperimenti ed applicando casualmente queste variazioni, è stato possibile misurare nuove accuracy del classificatore ed osservare dunque delle differenze nelle capacità di classificazione da parte del modello a seconda della configurazione scelta automaticamente dallo script.

Ad esempio, nell'esperimento numero 430 il modello è riuscito ad etichettare correttamente e dunque a riconoscere, circa il 13% dei dati etichettati con la categoria di attacco "pod", mentre nella precedente serie di oltre 1000 esperimenti descritta nel capitolo precedente effettuata con un'unica configurazione, non era stato possibile riconoscere alcun attacco di questo tipo.

In questo esperimento particolare, lo script automatico ha generato casualmente la seguente configurazione:

Neuroni input	Neuroni nascosti	Epoche	Batch size	learning rate
60	219	1	1118	2.2347752486710717e-05

Table 4.1. Configurazione particolare trovata

Quindi si può notare come sia stato scelto un numero più basso di neuroni in input, un numero molto alto di dimensione del batch (1118) ed un tasso di apprendimento più basso (0.0000223) rispetto alla configurazione standard usata nel capitolo precedente.

Questo ha suggerito che aumentando la dimensione del batch o usando un tasso di apprendimento più basso, il modello potesse avere più probabilità di essere addestrato con successo (in base ai campioni usati) su quelle porzioni di attacchi costituite da poco campioni.

Quindi è stata presa questa configurazione e sono stati effettuati nuovi esperimenti per cercare di riprodurre i risultati e di trovare nuovi miglioramenti su quelle porzioni su cui il modello non aveva avuto successo.

Infatti, eseguendo esperimenti mirati con questa configurazione, è stato sia confermato che è possibile riconoscere una piccola parte degli attacchi "pod" e sia che è possibile riconoscere molti degli altri tipi di attacchi costituiti da pochi campioni, sebbene con percentuali molto basse (ma comunque maggiori di 0%).

Purtroppo però, l'uso di queste configurazioni da un lato permette di ottenere leggeri miglioramenti per le classi con pochi campioni ma, dall'altro lato causa il peggioramento delle prestazioni per le classi con molti campioni, su cui erano stati raggiunti ottimi risultati, quali "smurf", "neptune" e "normal". In particolare per quest'ultima categoria, la percentuale di correttezza (media) nell'indovinare l'etichetta scende da circa 100% a circa 93%. Nei casi più gravi, questa può addirittura

scendere sotto il 10% se l'architettura usata è particolarmente sfavorevole.

Media	Varianza	Dev Standard	Valore minimo	Valore massimo	Range
0.933777	0.020901	0.144571	0.000000	1.000000	1.000000

Table 4.2. Risultati percentuali su etichetta Normal

Per quanto riguarda invece "smurf" il peggioramento (in media) è irrisorio, grazie all'enorme quantità di campioni disponibili che dà robustezza e stabilità alla rete nel riconoscere questo tipo di campioni.

4.2 Ricerca Bayesiana

Dato che la ricerca casuale dei parametri descritta nella sezione precedente non ha dato i risultati sperati, in quanto lo spazio dei parametri è troppo grande per essere esplorato casualmente, anche per via del fatto che ogni addestramento e testing ha un costo computazionale abbastanza alto, è stato scelto di usare un altro metodo per trovare un'architettura migliore con lo scopo di migliorare i risultati di accuracy del classificatore.

Tale metodologia descritta qui di seguito riguarda l'applicazione della teoria della ricerca bayesiana che consiste nel prevedere di volta in volta quali valori scegliere sulla base dei risultati ottenuti e a seconda dell'effetto desiderato (ad esempio massimizzare o minimizzare il risultato) [4].

In questo contesto quindi, ciò si traduce nel trovare architetture migliori analizzando i risultati precedenti ottenuti con altre architetture, senza dunque effettuare l'esplorazione in modo casuale come visto prima.

Il vantaggio di questa strategia consiste nel trovare rapidamente una combinazione ottima di parametri ma solo se aumentando e diminuendo uno o più di questi parametri è "evidente" il miglioramento o peggioramento del risultato da massimizzare o minimizzare.

Se ad esempio la funzione di addestramento e testing restituisse il risultato sulla base del seguente criterio.

```
def addestramentoTest(num_neuroni_input, num_neuroni_nascosti,
    num_epoche, batch, learning_rate, training,
   artificial_training):

    ...
```

```
    res = (num_epoche / 700) * 99.0

    if num_epoche == 500:
        res = 100.0

    return res
```

Listing 4.1. Esempio di codice Python

Ovvero se il massimo valore raggiungibile con una combinazione di parametri fosse determinato da un preciso valore di un singolo parametro (o anche più parametri), allora diventerebbe ancora più difficile trovare la soluzione ottima (in questo esempio decisa a priori e prodotta dall'uso di num epoche = 500) poiché tale soluzione è "nascosta" dal trend di risultati ottenuti facendo tentativi con valori sempre più alti di num epoche.

In un caso del genere quindi il vantaggio della ricerca bayesiana verrebbe perso perché nella migliore delle ipotesi tenderebbe a comportarsi come la ricerca casuale cercando di "indovinare" un parametro o più parametri che possano produrre una soluzione migliore delle precedenti, mentre nella peggiore delle ipotesi continuerebbe a scegliere di aumentare il valore di num epoche rendendo impossibile trovare la combinazione migliore, che prevede invece un valore specifico e più basso di num epoche.

Dal momento che, nel caso trattato in questa relazione non è possibile sapere in anticipo se uno dei parametri possieda una proprietà del genere, è stato deciso comunque di provare questa tecnica eseguendo altri 2000 esperimenti.

```
def objective(trial):
    # Parametri da ottimizzare
    num_neuroni_input = trial.suggest_int('num_neuroni_input',
        118, 1000)
    num_neuroni_nascosti = trial.suggest_int('
       num_neuroni_nascosti', 200, 2000)
    epoche = trial.suggest_int('epoche', 1, 100)
    batch = trial.suggest_int('batch', 8, 3000)
    learning_rate = float(trial.suggest_loguniform('
       learning_rate', 1e-6, 1e-3))

    # Chiamata addestramento
```

```
    result = addestramentoTest(num_neuroni_input,
        num_neuroni_nascosti, epoche, batch, learning_rate, "
        dataset.csv", "generated.csv")

    return result

# Specifica lo storage di optuna
storage_name = "sqlite:///Ricerca_IDS.db"

# Crea lo studio
study = optuna.create_study(
    study_name='hyperparam_ricerca',
    storage=storage_name,
    load_if_exists=True,
    direction='maximize'
)

try:
    best_trial = study.best_trial
    btv = best_trial.value
except:
    btv = 0

while True:
    # Esegui la ricerca
    study.optimize(objective, n_trials=1)
    best_trial = study.best_trial
    btv = best_trial.value
```

Listing 4.2. Codice Python Ricerca Bayesiana

Purtroppo la ricerca effettuata non ha prodotto i risultati sperati poiché ha permesso di trovare una combinazione di parametri che ha solo migliorato leggermente il risultato di accuracy medio facendolo aumentare da 98.2% a circa 98.4%.

Tale combinazione è la seguente:

Neuroni input	Neuroni nascosti	Epoche	Batch size	learning rate
212	478	1	8	0.0001762435

Table 4.3. Configurazione particolare trovata

Chapter 5

Miglioramento mediante generazione di nuovi campioni

Per migliorare i risultati del classificatore è stato poi realizzato un modello GAN per generare nuovi campioni simili a quelli del dataset KDD99 e sono poi stati analizzati dal punto di vista della loro utilità effettiva e realisticità, confrontandoli con i campioni originali KDD99 da cui derivano mediante il calcolo della loss di addestramento del generatore e mediante il calcolo di matrici di correlazione fra le caratteristiche dei campioni originali e generati.

5.1 Generazione di campioni sintetici mediante GAN

Come detto in precedenza, alcune tipologie di attacchi sono poco presenti nel dataset KDD99 e questo impedisce al classificatore di imparare a riconoscerli con sufficiente accuracy, per questo è stato scelto di migliorare il classificatore non solo dal punto di vista architetturale ma anche dal punto di vista del dataset, arricchendolo di nuovi campioni ottenuti mediante l'impiego della generazione di campioni sintetici.

Questa tecnica consiste nel generare nuovi campioni basandosi sui pattern di cui sono costituiti quelli reali [5], quindi in questo caso i campioni presenti nel dataset KDD99.

Per eseguire questa tecnica sono stati considerati vari tipi di algoritmi generativi e alla fine è stato scelto di usare l'algoritmo GAN che si basa sull'impiego di reti neurali.

Questo tipo di algoritmo è spesso usato per generare campioni da dataset che rappresentano immagini ma si presta bene anche alla generazione di nuovi campioni partendo da un dataset di informazioni sulla sicurezza informatica come il KDD99.

Per implementarlo, è stato necessario creare altre due reti neurali con ruoli contrapposti:

- **Generatore**: il ruolo di questa rete è quello di generare nuovi campioni a partire da esempi (i campioni reali del dataset KDD99) aggiungendovi un opportuno rumore (variazioni) [5].
- **Discriminatore**: il ruolo di questa rete è invece quello di classificare i campioni come "veri" o "sintetici" [5].

L'addestramento di entrambe le reti neurali è stato gestito da un algoritmo che ne ha alternato gli sviluppi, facendo quindi generare nuovi campioni al generatore, dandoli poi in analisi al discriminatore e riportando i risultati al generatore così da aggiornarne i pesi e dunque la realisticità nel generare campioni.

Dopo un certo numero di iterazioni, il generatore si trova nello stato in cui riesce ad ingannare il discriminatore, generando campioni che il discriminatore reputa "veri" e quindi realistici.

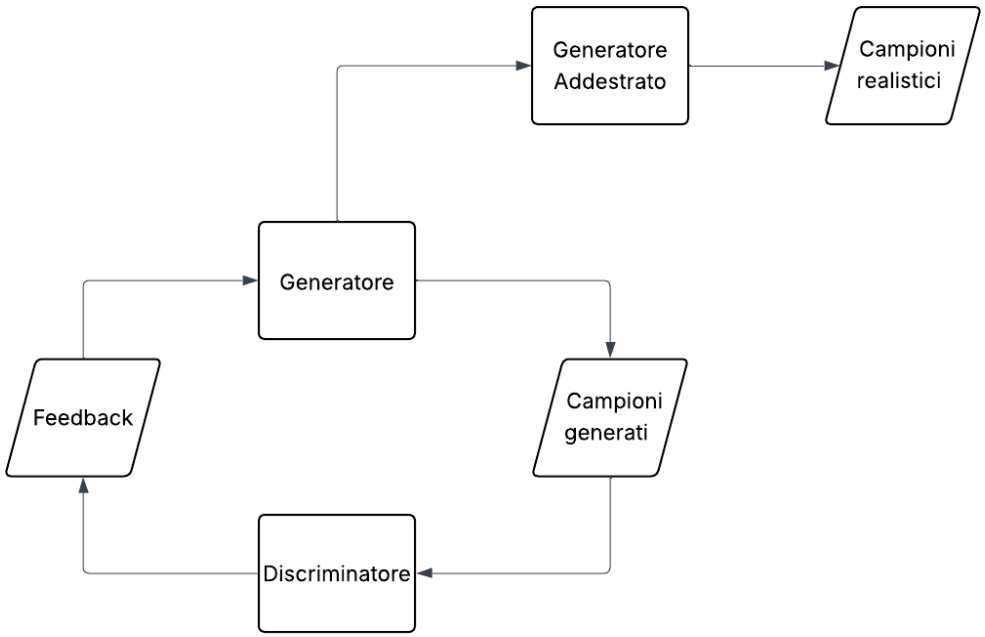

Figure 5.1. Schema di alto livello di funzionamento GAN

Ci sono diverse varianti con cui si può condurre l'addestramento, alcune ricerche affermano che in molti casi sia più conveniente congelare l'addestramento del discriminatore permettendogli solo di fornire feedback sui campioni generati, poiché

altrimenti si rischierebbe di renderlo più potente del generatore e quindi di non riuscire a fare in modo che il generatore possa produrre campioni realistici [6].

Il primo tentativo è stato eseguito dando come input tutto il dataset KDD99 (scandito nei relativi batch) ma ciò ha evidenziato un problema in quanto al termine dell'addestramento, eseguendo una nuova generazione di campioni, questi ultimi venivano generati sempre e solo con una delle etichette più largamente presenti nel dataset, ovvero le tipologie di attacco "smurf", "neptune" e "normal".

Questo tipo di algoritmo infatti fa molta difficoltà a generare l'etichetta dei campioni di un dataset, questo perché ha bisogno di grandi differenze tra i campioni etichettati diversamente così da apprenderne i diversi pattern, cosa impossibile da realizzare con un dataset "sensibile" come il KDD99 che ha solo 23 possibili valori di etichetta e in cui piccole variazioni tra i dati possono suggerire un attacco piuttosto che un altro.

5.2 Un'implementazione GAN per ogni porzione

Per ovviare al problema di generazione delle etichette dei campioni, è stato scelto di procedere addestrando una GAN diversa per ogni porzione di dataset, quindi per ogni subset avente una specifica etichetta, così da decidere in anticipo l'etichetta dei campioni ed eludendone quindi la generazione.

Anche in questo caso il dataset è stato suddiviso in porzioni e ogni porzione è stata data in input ad un'implementazione GAN ottimizzata per quella specifica porzione. Al termine dell'addestramento sono stati generati i nuovi campioni senza etichetta, la quale, come descritto, viene aggiunta automaticamente a seconda di quale porzione sia stata usata per l'addestramento.

Quindi in totale sono state prodotte 23 GAN, ognuna delle quali addestrata su una certa porzione del dataset KDD99 e dunque specializzata nel produrre campioni di un certo tipo di attacco.

```
def build_generator(noise_dim, output_dim):
    model = Sequential([
        Dense(128, activation='relu', input_dim=noise_dim),
        Dense(256, activation='relu'),
        Dense(output_dim, activation='sigmoid')
    ])
    return model

def build_discriminator(input_dim):
```

```
    model = Sequential([
        Dense(100, input_dim=input_dim),
        Dropout(0.5),
        Dense(100, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=Adam(0.001, 0.5), loss='
       binary_crossentropy', metrics=['accuracy'])
    return model

def build_gan(generator, discriminator):
    #discriminator.trainable = False
    model = Sequential([generator, discriminator])
    model.compile(optimizer=Adam(0.001, 0.5), loss='
       binary_crossentropy')
    return model
```

Listing 5.1. Modello GAN

```
...

# Campioni reali provenienti dal batch
idx = np.random.randint(0, n_samples, batch_size)
real_samples = data[idx]
real_labels = np.ones((batch_size, 1))

start += batch_size

# Campioni sintetici
noise = np.random.normal(0, 1, (batch_size, noise_dim))
fake_samples = generator.predict(noise)
fake_labels = np.zeros((batch_size, 1))

# Calcolo perdita
d_loss_real = discriminator.train_on_batch(real_samples,
   real_labels)[0]
d_loss_fake = discriminator.train_on_batch(fake_samples,
   fake_labels)[0]
d_loss = 0.5 * (d_loss_real + d_loss_fake)

# Allena il generatore
noise = np.random.normal(0, 1, (batch_size, noise_dim))
g_loss = gan.train_on_batch(noise, np.ones((batch_size, 1)))
```

```
...
```

Listing 5.2. Fulcro di un addestramento GAN

5.3 Il problema dei range

Dopo aver addestrato la GAN, sono stati generati alcuni campioni di prova ed è stato notato che determinate caratteristiche di molti campioni generati assumevano valori fuori dai range rispetto a quelli delle stesse caratteristiche dei campioni del dataset KDD99.

Tale fenomeno è molto comune per questo tipo di generatori ed è tutt'ora un problema aperto nella ricerca scientifica.

Se la quota di valori corretti è piccola, ad esempio < 1–2%, l'inquinamento nei dati è tipicamente trascurabile; se invece è grande, ciò può suggerire che il generatore non stia modellando correttamente quelle caratteristiche.

Per comprendere meglio le ragioni di questo fenomeno è stato necessario analizzare i valori dei campioni e pertanto è stato realizzato uno script dedicato in grado di confrontare i campioni generati con quelli reali, il cui pseudocodice è il seguente:

- ottenere per ogni caratteristica del dataset originale il range di valori [min,max]
- ottenere per ogni caratteristica del dataset artificiale il range di valori [min,max]
- Per ogni caratteristica, calcolare la quantità di campioni artificiali che hanno i valori nel range rispetto al dataset originale

Dall'analisi è emerso che:

- Nessun campione aveva i valori di tutte le caratteristiche nei range corretti.
- 28 caratteristiche erano nel range appropriato nel 100% dei campioni generati
- 7 caratteristiche erano nel range appropriato in oltre il 99.5% dei campioni generati
- 1 caratteristica era nel range appropriato solo nel 42% dei campioni generati
- 2 caratteristiche erano nel range appropriato solo in

  - 0,027% dei campioni generati
  - 0.053% dei campioni generati

Inoltre è stato misurato quanto i valori generati si discostassero dal range (in percentuale rispetto all'ampiezza del range di riferimento) ed è emerso che:

- Le caratteristiche "lnum outbound cmds" e "is host login" si discostano al massimo del 50% rispetto ai range corretti, con un discostamento medio attorno al 42%. Questo significa che molti dei campioni generati hanno valori molto sbagliati su queste caratteristiche.
- La caratteristica "same srv rate" si discosta al massimo del 33% rispetto al range corretto con un discostamento medio pari al 17% circa
- Le due caratteristiche più problematiche ovvero "lnum outbounds cmds" e "is host login" hanno lo stesso range di riferimento ovvero [0,0] e tale range non è relativo ad altre caratteristiche del dataset.
- La caratteristica "same srv rate" ha lo stesso range [0,1] comune a tante altre caratteristiche ma per qualche motivo i suoi valori nei campioni generati tendono ad andare fuori dall'intervallo.

5.4 Soluzioni per il problema dei range

Sono state proposte alcune alternative per risolvere il problema:

- **1- Trasformare** i valori delle caratteristiche non in range mappandoli in valori in range.
  - **Vantaggi**: il problema viene sistemato velocemente in seguito all'applicazione di una funzione e alla sostituzione dei valori fuori range.
  - **svantaggi**: i dati generati vengono leggermente modificati.
- **2- Ripetere** la generazione finché non si ottiene il numero desiderato di campioni i cui dati rispettano i range
  - **Vantaggi**: i dati generati sono sempre in range e non è necessario eseguire un fix
  - **Svantaggi**: ogni generazione ha un costo computazionale molto alto
- **3- Incentivare** il generatore a produrre valori nei giusti range modificando la funzione di perdita del generatore

- **Vantaggi**: è sufficiente riscrivere la funzione di perdita senza modificare il resto
- **Svantaggi**: Aumentare la penalizzazione del feedback ricevuto dal generatore può influire sulla sua capacità di generare campioni realistici, seppure con valori in range.

Purtroppo è stato possibile applicare solo il metodo di trasformazione in quanto sia il secondo metodo che il terzo metodo non hanno prodotto i risultati sperati.

Nello specifico, applicare il metodo basato sulla ripetizione è stato infattibile come spiegato in seguito, mentre applicare il metodo basato sulla modifica della funzione di perdita è stato controproducente perché ha creato delle ulteriori complicazioni. Infatti, la modifica della funzione di perdita è stata implementata calcolando per ogni campione generato durante il processo competitivo generatore-discriminatore, il numero di caratteristiche fuori range (moltiplicato per la distanza dal range) e aggiungendo quindi una penalità complessiva alla perdita già normalmente calcolata. Dopo un numero molto alto di iterazioni il generatore riesce effettivamente a produrre campioni con valori in range ma riproducendo esattamente quelli visti dal dataset KDD99 rendendo quindi inutile l'applicazione di questa metodologia.

Nel caso delle caratteristiche "lnum outbounds cmd" e "is host login" dato che dall'analisi effettuata i valori generati sono quasi sempre fuori dai range appropriati, applicare il metodo di trasformazione è molto indicato, anche perché il range è costituito da un unico valore essendo [0,0] quindi non bisogna scegliere un altro valore diverso da 0.

Nella realtà è bene precisare che tali caratteristiche in una macchina potrebbero assumere altri valori oltre "0" (ad esempio nel caso di "lnum outbouns cmd" questo può essere un intero maggiore di "0"e nel caso di "is host login" un possibile valore potrebbe essere anche "1") tuttavia nel dataset KDD99 è presente solo il valore "0" su entrambe le colonne pertanto è stato scelto di rispettare questa limitazione.

Inoltre, se per assurdo fosse stato possibile applicare il metodo di ripetizione, ovvero se fosse stato generato almeno qualche campione con tutti i valori nei range corretti, sarebbe stato comunque particolarmente inefficiente, in quanto la probabilità di generare campioni con queste due caratteristiche in range, secondo l'analisi effettuata è molto bassa (circa 0.02% e 0.05%) e pertanto ciò avrebbe comportato un numero di tentativi di generazione altissimo e quindi computazionalmente troppo oneroso, considerando che ogni generazione richiede circa 5 minuti se eseguita su una macchina abbastanza prestante.

Per quanto riguarda invece le altre caratteristiche, sarebbe stato ideale applicare il metodo di ripetizione però come visto ci sono alcune porzioni di dataset per le quali è inutile ripetere la generazione di nuovi campioni fino a soddisfare le condizioni di range. Per tali porzioni infatti non vengono mai generati campioni con tutte le caratteristiche nei range appropriati e ciò è purtroppo una limitazione ben nota degli algoritmi basati su GAN.

5.5 Trasformazione

Come risultato finale è stato quindi applicato solo il metodo di trasformazione.

Una volta generati i campioni, questi vengono quindi processati da una funzione che ne corregge i valori non in range mappandoli nel range corretto. Inoltre per rendere più realistici i valori rispetto al dataset originale, è stata applicata un'ulteriore correzione che consiste nel rispettare il tipo di dato rispetto al dataset originale: se ad esempio una caratteristica nel dataset KDD99 assume solo valori interi, allora tutti i valori reali di tale caratteristica presenti nei campioni generati vengono trasformati in interi.

5.6 Realisticità dei dati: Loss

Mentre per un generatore di immagini è sufficiente osservare le immagini prodotte per stabilire se il generatore riesce a produrre campioni realistici, nel caso di generatori di dati apparentemente astratti come questo, è impossibile osservare i campioni e determinarne visivamente la realisticità.

Quindi è stata considerata l'analisi di alcune condizioni necessarie per stabilire se il generatore avesse imparato correttamente e se dunque i campioni generabili post-addestramento fossero davvero realistici.

La prima condizione necessaria è quella riguardante l'osservazione della **loss** durante il processo di addestramento del generatore e del discriminatore. In particolare alcune ricerche hanno evidenziato un forte legame fra l'andamento della loss del generatore e la capacità di quest'ultimo di generatore campioni realistici. Questo perché se la perdita del generatore durante il processo di addestramento non decresce, è altamente probabile che il generatore non abbia appreso con successo come generare dati realistici e che dunque i feedback ricevuti dal discriminatore lo abbiano confuso su come generare nuovi campioni.

Questo accade poiché l'addestramento di una GAN è un processo molto delicato in quanto il generatore e il discriminatore si sviluppano l'uno rispetto all'altro e basta dunque una casualità come ad esempio vedere un batch di campioni piuttosto che un altro, per far si che il generatore o il discriminatore non riesca a svilupparsi adeguatamente. Questi fattori perdono importanza quando il dataset di apprendimento è molto ampio, vario ed equilibrato ma nel caso del dataset KDD99 questo non è detto.

Per garantire il soddisfacimento di questa condizione, è stato modificato l'addestramento di ogni GAN per ogni porzione del dataset introducendo un controllo post-addestramento volto a verificare che alcuni valori della loss del generatore fossero l'uno più basso dell'altro (evidenziando quindi un trend decrescente). Qualora questa condizione non fosse verificata, l'addestramento viene automaticamente ripetuto.

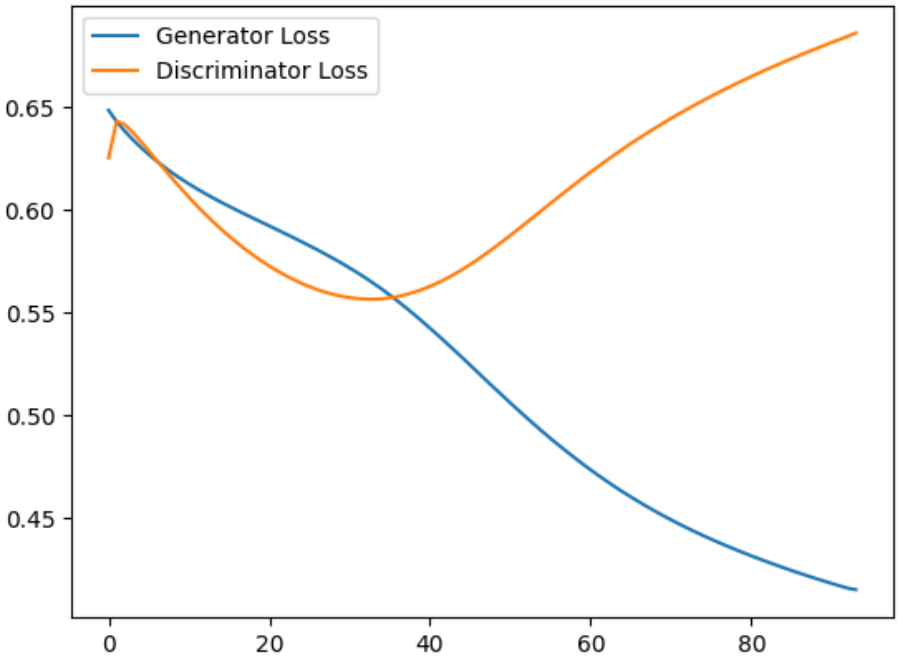

Figure 5.2. Condizione di loss del generatore valida

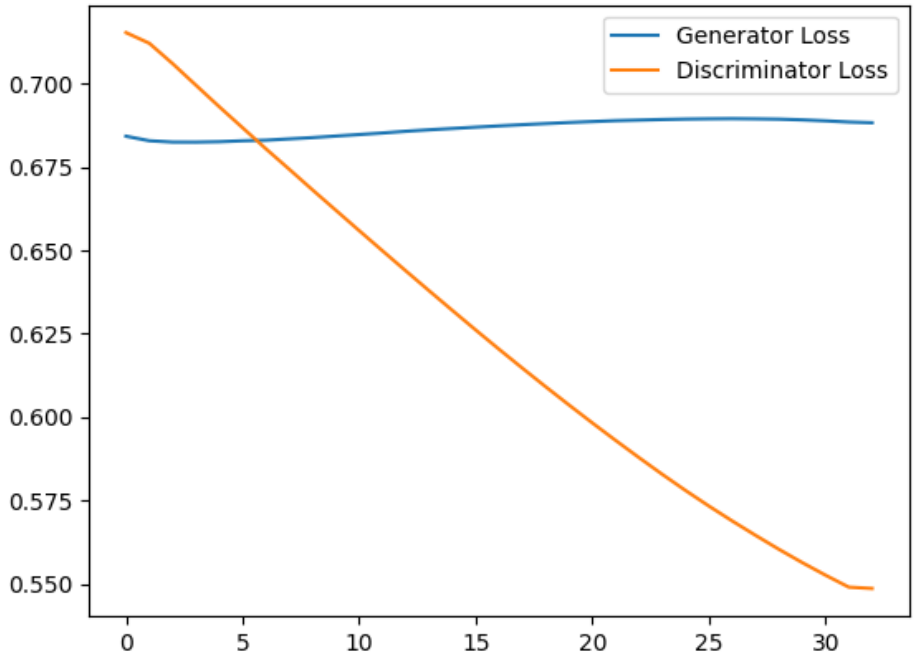

Figure 5.3. Condizione di loss del generatore non valida

5.7 Realisticità dei dati: Matrice di correlazione

La seconda condizione utile da verificare per stabilire se i campioni generati possano essere realistici o meno, consiste nel controllare che le correlazioni fra le caratteristiche dei dati generati siano il più simile possibile rispetto alle correlazioni fra le caratteristiche dei dati originali.

Questa tecnica consiste nel prendere tutte le possibili coppie di caratteristiche osservando come ogni caratteristica varia rispetto ad un'altra nel dataset.

Esistono diverse metriche con cui si possono calcolare le correlazioni fra le caratteristiche, nel caso specifico è stato usato l'**indice di correlazione di Pearson** che mira ad esprimere una relazione di linearità fra le caratteristiche.

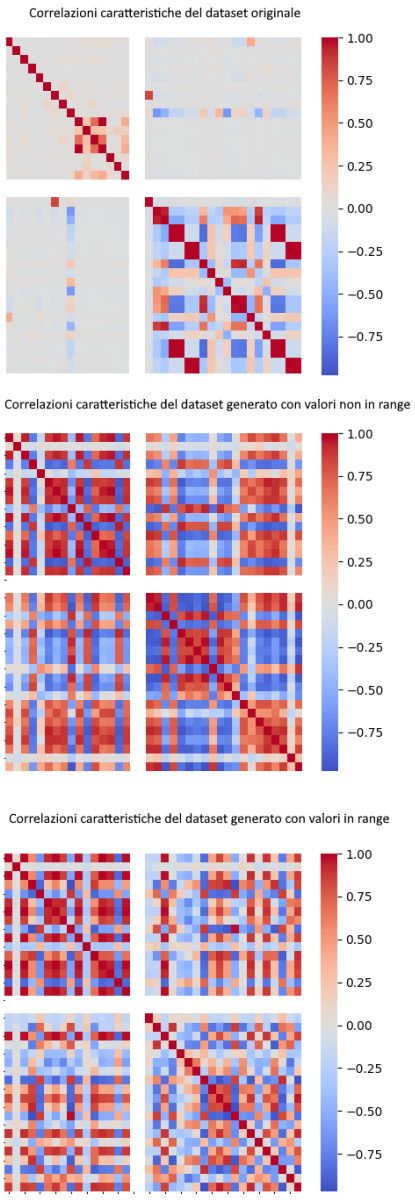

Figure 5.4. Correlazioni fra le caratteristiche

Dalle figure mostrate, in cui ogni punto della matrice corrisponde ad una coppia di caratteristiche, si può osservare che nel dataset KDD99 molte coppie di caratteristiche sono poco o per nulla correlate tra loro e che quando invece i campioni generati hanno le caratteristiche in range, le correlazioni di Pearson risultano più simili a quelle del dataset originale KDD99.

Nel caso invece in cui i campioni generati non hanno le caratteristiche nei rispettivi range rispetto al dataset KDD99, allora le correlazioni di Pearson presentano più differenze rispetto a quelle del dataset originale, infatti si può notare dall'immagine mostrata come ci siano molte più coppie di caratteristiche fortemente correlate (in positivo o in negativo).

Per cercare di ridurre le differenze di correlazione fra le caratteristiche del dataset generato e quelle del dataset originale, è stata creato uno script per ripetere automaticamente nuovi addestramenti di generatori GAN in grado di produrre dataset con meno differenze di correlazioni possibili.

A tale scopo è stato anche necessario scegliere una metrica per calcolare la differenza di correlazione ed esprimerla con un numero. Tale metrica scelta è la **Distanza di Frobenius**.

Questa distanza è calcolata come la radice quadrata della somma dei quadrati di tutti i valori della matrice differenza A-B, dove in questo caso la matrice A è la matrice di correlazione del dataset originale e la matrice B è la matrice di correlazione del dataset generato.

Più bassa risulta tale distanza, minori sono le differenze globali di correlazione fra le due matrici e ciò significa che i dati generati sono più realistici.

Inizialmente utilizzando un'architettura di partenza scelta per la GAN, la distanza di frobenius è risultata pari a 25.85, ciò indica una bassa realisticità dal punto di vista delle correlazioni fra caratteristiche. Nell'immagine seguente si può osservare la differenza fra le matrici di correlazione.

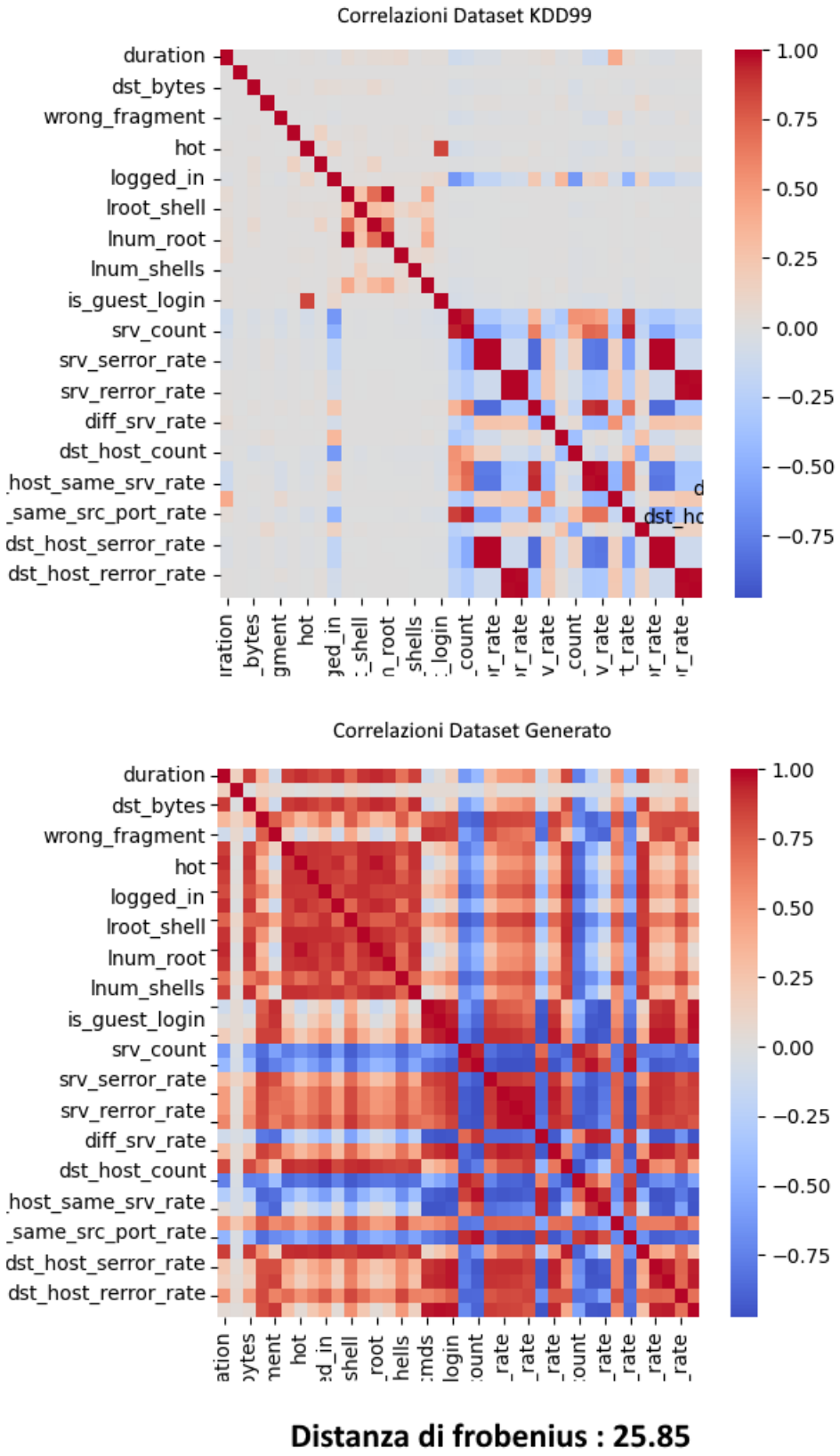

Figure 5.5. Distanza di frobenius

In particolare si osservano forti correlazioni positive (colore rosso) completamente inesistenti nel dataset KDD99. Le correlazioni negative (colore blu) invece sono effettivamente già presenti nel dataset KDD99 ma in quello generato risultano di molto amplificate.

In seguito è stata avviata una nuova ricerca bayesiana per trovare un'architettura tale da minimizzare la distanza di frobenius fra i dati generati dai modelli GAN addestrati e i dati reali del Dataset KDD99.

Dopo molti tentativi è stata trovata una configurazione GAN in grado di generare dei campioni aventi distanza di frobenius rispetto ai campioni reali, pari a 7.3.

In particolare durante gli esperimenti è stato osservato che il learning rate per addestrare le reti GAN, riveste un ruolo fondamentale nel determinare la distanza di frobenius dei dati generabili.

L'uso di learning rate più alti durante l'addestramento dei modelli GAN aiuta il generatore ad imparare meglio come creare i campioni, con particolare attenzione però alla possibilità di overfitting che è stata opportunamente gestita mediante l'introduzione di alcuni **dropout** nella rete del discriminatore (come mostrato in una figura precedente). I dropout in una rete neurale, servono a disporre lo spegnimento casuale, di volta in volta, di una certa quantità di neuroni negli strati della rete in modo tale che gli altri neuroni rimasti attivi possano favorire l'apprendimento in maniera più robusta riducendo quindi di molto la possibilità di overfitting della rete e più in generale per evitare le problematiche legate alle reti neurali.

Nell'immagine seguente è possibile osservare come la differenza fra le matrici di correlazione del dataset KDD99 e del nuovo dataset generato sia stata ridotta applicando queste migliorie:

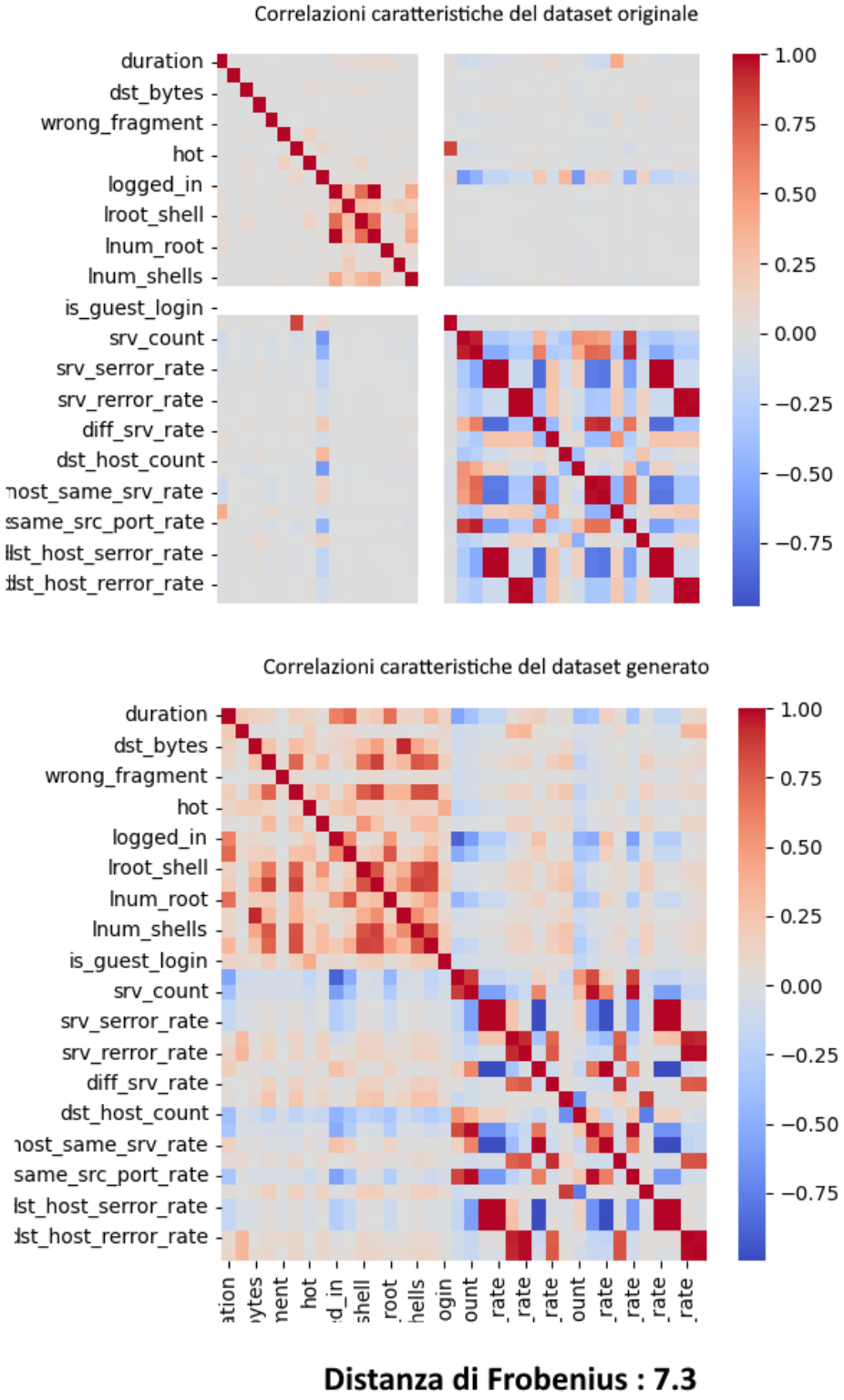

Figure 5.6. Differenze fra matrici di correlazione

5.8 Controllo di ugualianza campioni

Dal momento che l'aumento del learning rate durante il processo di addestramento incrementa la probabilità che il generatore possa creare campioni uguali ad altri già presenti nel dataset KDD99, è stato introdotto un ulteriore controllo, il cui codice è mostrato in figura.

```
def controllo_campioni_diversi():

    originale = pd.read_csv("dataset.csv")
    generato = pd.read_csv("generated.csv")

    # Trova campioni duplicati nel dataset generato
    duplicati = generato.duplicated()


    # Trova i campioni duplicati nel dataset generato rispetto
        al KDD99
    campioni_in_comune = generato.merge(originale, how='inner'
       )

    uguali = duplicati.any() or not(campioni_in_comune.empty):

    return uguali
```

Listing 5.3. Codice Python controllo campioni diversi

Chapter 6

Test del classificatore

Dopo aver generato i nuovi campioni ed averli raccolti in un nuovo dataset, sono stati eseguiti nuovi test di accuracy per il classificatore, unendo questi nuovi campioni al dataset KDD99 per addestrare meglio il classificatore. Inoltre sono state eseguite delle ottimizzazioni sul dataset generato per renderlo più leggero senza perdere efficacia e sono stati eseguiti ulteriori test da diverse angolazioni per stabilire il contributo offerto dal dataset generato per quanto riguarda l'accuracy del classificatore ultimato.

6.1 Uso del dataset generato

L'addestramento del classificatore di attacchi è stato rieseguito fornendogli l'80% dei campioni presenti nel dataset KDD99 e il 100% dei campioni presenti nel dataset generato. Il rimanente 20% dei campioni nel dataset KDD99 è stato riservato ai test.

Nell'immagine seguente è possibile notare il nuovo schema con il relativo risultato:

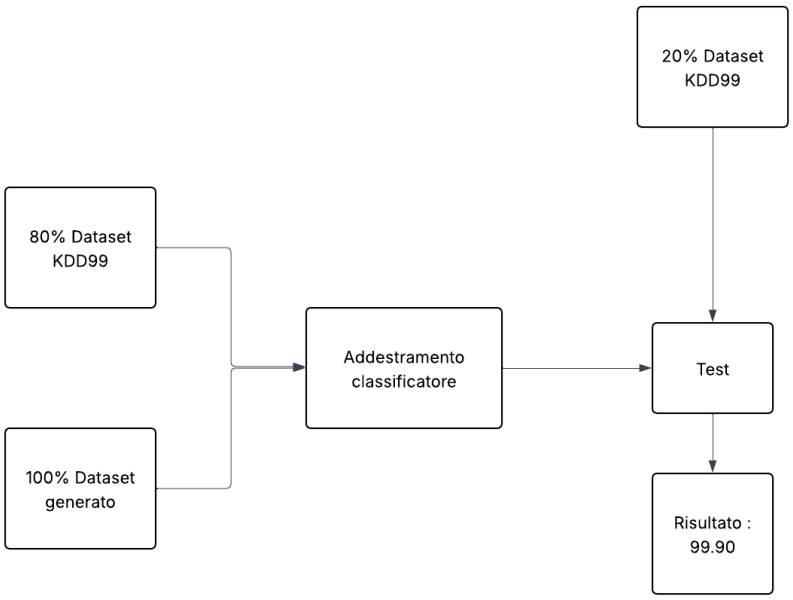

Figure 6.1. Risultati addestramento con dataset generato

Dai test effettuati dunque, la probabilità di riconoscere gli attacchi da parte del classificatore è pari al 99.9% circa, mentre in precedenza, con il solo addestramento con Dataset KDD99 e ottimizzazione dell'architettura, la probabilità di riconoscere gli attacchi si avvicinava al 98.2% circa. Questo significa che il dataset generato offre un contributo notevole nell'addestramento del classificatore, abbassando il tasso di errore nel riconoscere le minacce da 1.8% a 0.1%. Quindi in questo modo, il classificatore fa 1 errore ogni 1000 anziché quasi 2 errori ogni 100.

6.2 Minimizzazione del dataset generato

Dopo aver generato alcuni lotti di campioni, è stata notato un fenomeno comune, ovvero che il dataset generato pesasse molto di più rispetto al dataset KDD99 a parità di numero di campioni.

Analizzando meglio i dati generati rispetto a quelli del dataset KDD99, è stato riscontrato che i valori delle caratteristiche del dataset generato risultano molto precisi, quindi ci sono molte cifre decimali per ogni valore a differenza invece dei valori del dataset KDD99 che sono in molti casi addirittura semplici interi. Questo comporta inevitabilmente un aumento del peso del dataset generato fino a 10 volte.

Per ridurre quindi l'impatto sulla memoria, è stato cercato un compromesso per tagliare le cifre decimali dei valori del dataset generato senza però rischiare di perdere realisticità nei dati e dunque senza ridurre le prestazioni del classificatore

che usa il dataset generato per l'addestramento.

Dopo alcuni tentativi è stato osservato che tagliare i valori decimali dei dati oltre la seconda cifra decimale non cambia il risultato complessivo del classificatore.

In tal modo l'impatto sulla memoria da parte del dataset generato è stato ridotto dell'80% circa senza perdere efficacia.

6.3 Test con diverse proporzioni di training-test

Per verificare meglio l'utilità dei campioni del dataset generato, sono stati eseguiti ulteriori test cambiando le unità di campioni forniti per l'addestramento del classificatore.

Anziché quindi eseguire l'addestramento usando l'80% del dataset KDD99 (e dunque il 20% del dataset KDD99 riservato ai test) ed il 100% del dataset generato, sono state usate diverse combinazioni, come ad esempio il 30% del dataset KDD99 (e dunque il rimanente 70% per i test) e il 40% del dataset generato e sono stati osservati i risultati.

Il processo è stato automatizzato similmente alla precedente ricerca casuale effettuata, in modo da raccogliere risultati diversi per poi poterli analizzare.

Dai risultati raccolti è emerso che il dataset generato non riesce a "sostituire" il dataset KDD99. Ad esempio, in uno degli esperimenti sono stati forniti all'addestramento del classificatore, l'1% dei campioni del dataset KDD99 e il 100% dei campioni del dataset generato ed infine il 99% dei campioni del dataset KDD99 sono stati utilizzati per il testing. Il risultato di correttezza del classificatore nell'etichettare i campioni di test è solo del 40.6% circa.

Vengono di seguito riportati altri esperimenti simili e i relativi risultati:

% KDD99 training	% generato training	% KDD99 testing	risultato
3	35	97	41.03168210217907
5	34	95	41.25829126883847
18	38	82	41.36666140052925
1	100	99	40.6101313704442
12	77	88	41.36066282685903

Table 6.1. Esiti di alcuni esperimenti

Questi risultati dimostrano che il dataset generato non può sostituire quello originale, ovvero il KDD99, tuttavia come visto in precedenza, il suo impiego

nell'addestramento del classificatore (congiunto all'impiego del KDD99) garantisce risultati nettamente superiori rispetto al solo impiego del KDD99.

6.4 Test particolari

Per concludere la parte relativa ai test riguardanti il dataset generato, è stato pensato di provare alcune combinazioni training-test al solo scopo di valutare l'impatto del dataset generato da altre angolazioni, senza quindi avere l'obiettivo di migliorare il classificatore.

Questi test hanno riguardato l'impiego del dataset generato come sostituto del dataset KDD99 solo per quanto concerne i test. Quindi anziché fornire per il testing una parte del dataset KDD99, è stato fornita esclusivamente una parte del dataset generato.

I primi test sono stati eseguiti fornendo il 100% del dataset KDD99 per il training e il 20% del dataset generato per il testing senza però equilibrare i dati del testing, ovvero senza estrarre il 20% da ogni porzione del dataset generato ma semplicemente prendendo casualmente il 20% fra tutti i campioni.

I risultati sono molto diversi fra loro nonostante le percentuali utilizzate siano sempre le stesse, questo perché nei casi in cui i campioni selezionati per il test sono meno equilibrati (ad esempio appartenenti a poche porzioni o porzioni su cui il classificatore fa più fatica a riconoscere i campioni) il risultato è più basso.

% KDD99 training	% generato testing	suddivisione equilibrata test	risultato
100	20	No	94.8463624915
100	20	No	80.0929112181
100	20	No	96.6145095515
100	20	No	96.8986255891
100	20	No	97.0648961505
100	20	No	96.4039917412
100	20	No	95.9627140601
100	20	No	80.1111291048
100	20	Si	97.147107
100	20	Si	96.891832
100	20	Si	96.299032
100	20	Si	97.666125
100	20	Si	97.967234
100	20	Si	96.982445
100	20	Si	97.682598
100	20	Si	97.149855

Table 6.2. Esiti di alcuni esperimenti

Dai risultati raccolti, il classificatore sembra etichettare abbastanza bene i dati generati, pur avendo imparato solo dal dataset KDD99.

Inoltre come si può notare, quando la suddivisione del dataset usato per i test è equilibrata per ogni etichetta, i risultati oltre ad essere più stabili, sono anche più alti, questo perché il classificatore ha evidentemente imparato meglio a riconoscere gli attacchi.

Per quanto riguarda invece i test effettuati dopo aver addestrato il classificatore con il 100% del dataset KDD99 ed aver eseguito il testing sul 100% dei campioni del dataset generato, i risultati sono i seguenti:

% KDD99 training	% generato testing	suddivisione equilibrata test	risultato
100	100	Si	94.12263637228565
100	100	Si	85.7205610560388
100	100	Si	94.27041054905192
100	100	Si	92.73193966764983
100	100	Si	86.64020226843482
100	100	Si	98.02266005664
100	100	Si	94.57871201646972

Table 6.3. Esiti di alcuni esperimenti

I risultati, nonostante la suddivisione equilibrata, sono molto instabili e questo evidenzia che l'intero dataset KDD99 non è sufficiente per addestrare con successo il classificatore a riconoscere tutti i campioni del dataset generato, questo perché alcuni campioni di quest'ultimo risultano più rumorosi di altri in relazione a quelli del dataset KDD99 e pertanto il classificatore commette più errori.

Chapter 7

Conclusioni

7.1 Risultati

Nella presente relazione sono stati visti i metodi con cui ricercare un dataset per costruire uno specifico classificatore basato su reti neurali per risolvere uno specifico problema di sicurezza informatica, come costruire un classificatore dato un certo problema, come tentare di migliorare l'accuracy di classificazione utilizzando due diversi metodi di ricerca di architetture migliori, come realizzare un modello generativo basato su reti neurali per generare nuovi campioni a partire da un dataset per migliorare l'accuracy del classificatore, come valutare la realisticità dei campioni generati, come analizzare i risultati degli esperimenti e produrre statistiche ed infine come intepretare i risultati ottenuti.

Il procedimento utilizzato può essere applicato con la dovuta astrazione anche a dati di diversa natura e di diverso peso rispetto a quelli trattati, rendendolo dunque flessibile e scalabile.

7.2 Possibili miglioramenti

Come visto in precedenza, i campioni sintetici sono stati generati mediante la realizzazione di un modello GAN ma esistono anche altri metodi per generare campioni che potrebbero essere considerati per migliorare ulteriormente i risultati del classificatore.

Fra i metodi di generazione basati sulle reti neurali (come GAN), i più utilizzati sono:

- **VAE** che basa la sua capacità di generazione sulla compressione dei dati che costituiscono i campioni di un dataset. Tali dati compressi vengono poi

decompressi, consentendo al modello di generarne di nuovi.

- **Modelli di diffusione** che basano la loro capacità di generazione sull'aggiunta di rumore ai campioni per poi apprendere come eliminare il rumore aggiunto, così da ottenere campioni simili a quelli originali conservandone i pattern fra i dati.
- **Modelli autoregressivi** che basano la loro capacità di generazione sulla predizione dei dati che costituiscono i campioni, a partire dai dati visti in precedenza.

Inoltre per arricchire i dataset, esiste un altro metodo completamente diverso chiamato **Data Augmentation**, indicato soprattutto quando generare campioni completamente nuovi non è possibile.

Questo metodo non consiste come i precedenti nel generare nuovi campioni completamente nuovi, bensì nell'ottenere dei nuovi campioni direttamente da quelli già esistenti applicando delle semplici funzioni matematiche sui valori costituenti i campioni.

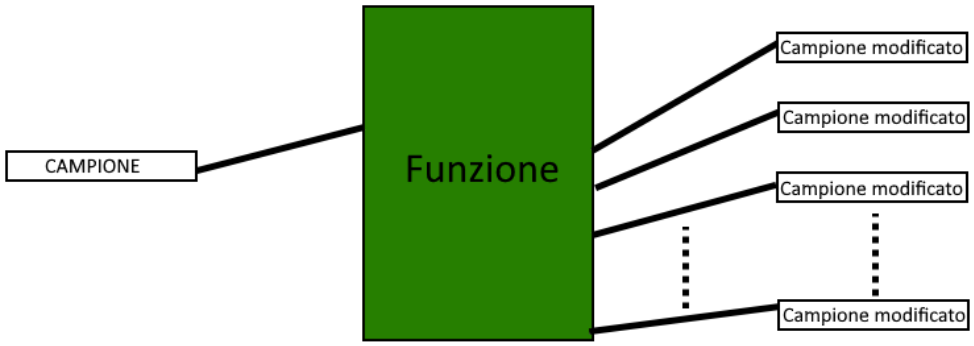

Figure 7.1. Schema generale idea Data augmentation

Esistono molte tecniche con cui applicare la modifica ai campioni, fra le più

comunemente usate si hanno:

- **Random cropping** che consiste nel ritagliare solo una parte dei dati che costituiscono i campioni.
- **Random erasing** che consiste nel cancellare una parte dei dati che costituiscono i campioni
- **Multiple scaling** che consiste nel moltiplicare i dati dei campioni per un certo numero di fattori.
- **Overlay** che consiste nel sovrapporre o mischiare due o più campioni.

L'applicazione della tecnica più adatta dipende da come sono costituiti i campioni e da cosa rappresentano.

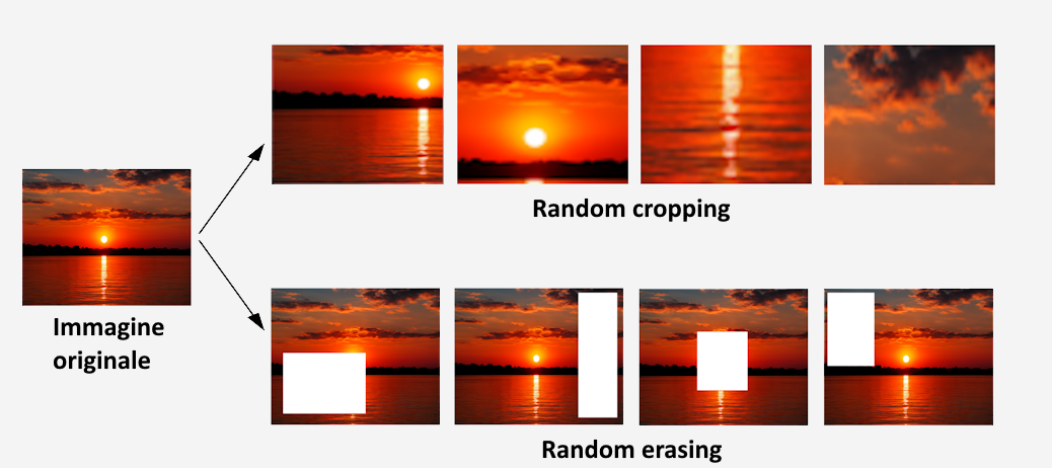

Figure 7.2. Data augmentation

Nel caso esposto in figura, i campioni sono immagini. Prendendo come esempio il random erasing, viene cancellata casualmente un'area dell'immagine, producendone quindi una nuova che può essere usata per arricchire il dataset di addestramento. L'operazione può essere ripetuta un certo numero di volte in base a quanto è grande l'area cancellata rispetto all'immagine e quindi è possibile produrre un certo numero di immagini che sono variazioni "cancellate" di quella originale.

Applicando questa tecnica al problema affrontato con il Dataset KDD99, sarebbe possibile eseguire il random erasing sui campioni andando ad azzerare i valori di alcune caratteristiche.

Ad esempio prendendo il seguente campione dal dataset KDD99 (riportato sottoforma di matrice per essere più leggibile):

0,215,2659,0,0,0,0,0,1,0,0,0,0,0,0,0,0.0,
0.0,0,15,15,0.07,0.07,0.0,0.0,1.0,0.0,0.0,
120,255,1.0,0.0,0.01,0.02,0.01,0.0,0.0,0.0,
tcp,http,SF,normal

Dopo aver applicato il random erasing 2 volte si potrebbero ottenere i seguenti campioni:

Cancellando i valori in posizione 1 e 2:

0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0.0,
0.0,0,15,15,0.07,0.07,0.0,0.0,1.0,0.0,0.0,120,255,
1.0,0.0,0.01,0.02,0.01,0.0,0.0,0.0,
tcp,http,SF,normal

Mentre cancellando i valori in posizione 20,21,22,23 si otterrebbe:

0,215,2659,0,0,0,0,0,1,0,0,0,0,0,0,0,0.0,
0.0,0,0,0,0,0,0.0,0.0,1.0,0.0,0.0,120,255,
1.0,0.0,0.01,0.02,0.01,0.0,0.0,0.0,
tcp,http,SF,normal

Inoltre bisogna considerare che alcuni campioni del dataset KDD99 hanno pochissime caratteristiche con valore diverso da zero, quindi in questi casi il numero di caratteristiche da azzerare dovrebbe essere ridotto. Per concludere, le caratteristiche categoriali non possono essere azzerate.

7.3 Applicazione in contesti reali

Il classificatore realizzato può essere impiegato in qualsiasi macchina in grado di supportare il linguaggio Python e le relative librerie utilizzate per il progetto.

Non è necessario addestrare il modello del classificatore su ogni macchina in quanto questo può essere addestrato e poi salvato in formato h5 poiché con l'aiuto della libreria Keras è possibile caricare il modello addestrato e utilizzarlo in qualunque

momento.

Per poter rilevare le intrusioni, è sufficiente creare uno script in grado di caricare il modello addestrato del classificatore e disporre una periodica lettura (ad esempio da un file) dell'attuale stato del sistema riportato nel formato delle caratteristiche del dataset.

A tale proposito, sarebbe necessario collegare le informazioni del sistema al file che viene letto.

Una possibile e semplice implementazione che impiega il modello è mostrato nella figura seguente:

```
def prevedi_su_nuovi_dati(percorso_csv,model,scaler,encoder,
    labels):



    # Carica i nuovi dati
    df_nuovo = pd.read_csv(percorso_csv)

    # Per pulire eventuali spazi nei nomi delle colonne
    df_nuovo.columns = [col.strip() for col in df_nuovo.
        columns]

    # elenco delle colonne categoriali
    categorical_cols = [col for col in df_nuovo.columns if not
         pd.api.types.is_numeric_dtype(df[col])]
    if 'label' in categorical_cols:
         categorical_cols.remove('label')

    # encoding dei dati categoriali tramite one-hot encoding
    encoded = encoder.transform(df_nuovo[categorical_cols])
    encoded_df = pd.DataFrame(encoded, columns=encoder.
        get_feature_names_out(categorical_cols))

    # Unione delle colonne numeriche con le colonne
        categoriche codificate
    X = pd.concat([
         df_nuovo.drop(columns=categorical_cols + ['label'],
             errors='ignore').reset_index(drop=True),
         encoded_df.reset_index(drop=True)
    ], axis=1)
```

```
    # Scaling dei valori
    X_scaled = scaler.transform(X)

    # Previsione
    probabilita = model.predict(X_scaled)
    indici_predetti = probabilita.argmax(axis=1)
    etichette_predette = [labels[i] for i in indici_predetti]

    return etichette_predette[0] if len(etichette_predette) ==
        1 else "mix di attacchi"+str(etichette_predette)


# Carica il modello
model = keras.models.load_model('modello_addestrato.h5')
scaler = joblib.load('scaler.joblib')
encoder = joblib.load('encoder.joblib')
labels = joblib.load('labels.joblib')


while True:

    etichetta = prevedi_su_nuovi_dati("nuovi_dati.csv",model,
       scaler,encoder,labels)

    if (etichetta) != "normal":
        print("   in corso un attacco di tipo : "+ etichetta)

    time.sleep(10)
```

Listing 7.1. Impiego del modello in contesti reali

Bibliography

[1] https://cybersecurity-magazine.com/10-small-business-cyber-security-statistics-that-you-should-know-and-how-to-improve-them/

[2] Elyes Manai, Mohamed Mejri, Jaouhar Fattahi "*Impact of Feature Encoding on Malware Classification Explainability*",2023

[3] Preeti Mishra, Vijay Varadharajan, Uday Tupakula, Emmanuel S. Pilli, "*A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection*", 2018

[4] Jasper Snoek, Hugo Larochelle, Ryan P. Adams, "*Practical Bayesian Optimization of Machine Learning Algorithms*", 2012

[5] James Halvorsen, Clemente Izurieta, Haipeng Cai, Assefaw Gebremedhin, "*Applying Generative Machine Learning to Intrusion Detection A Systematic Mapping Study and Review*", 2024

[6] Sangwoo Mo, Minsu Cho, Jinwoo Shin, *"Freeze the Discriminator: a Simple Baseline for Fine-Tuning GANs"*, 2020

[7] Dataset KDD99 https://www.kaggle.com/datasets/toobajamal/kdd99-dataset

[8] Dataset UNSW-NB15 https://research.unsw.edu.au/projects/unsw-nb15-dataset