

SISTEMI DIGITALI M

RICONOSCIMENTO DI IMMAGINI MODIFICATE

A.A. 2020/2021

Andrea Mengascini
Antonio Spina

Indice

1	Introduzione	4
2	Tecniche utilizzate	5
2.1	Analisi RGB	5
2.2	SRM	6
2.3	Noiseprint	7
2.3.1	Architettura di estrazione del Noiseprint	9
3	Dataset	11
4	Rete Neurale	12
4.1	Residual Network	12
4.2	ResNet50	12
4.3	Parametri	13
5	Implementazione	14
5.1	SRM	14
5.2	Noiseprint	15
6	Risultati	16
6.1	Tempistiche	17
7	Portabilità su Android	18
7.1	Conversione modelli in TfLite	18
7.1.1	Conversione dei modelli per l'estrazione del noiseprint	19
7.2	RGB	19
7.3	SRM	19
7.4	Noiseprint	20
7.4.1	Chaquopy	20
7.4.2	Implementazione	21
7.5	Interfaccia	23
7.6	Test	23
7.6.1	Risultati	24

8 Attacchi	25
8.0.1 Resistenza modello RGB	30
8.0.2 Resistenza modello SRM	31
8.0.3 Resistenza modello Noiseprint	32
9 Conclusione	33

Capitolo 1

Introduzione

Al giorno d'oggi validare file digitali e documenti è d'importanza critica per molte realtà. Negli ultimi 20 anni, con l'incremento dei servizi presenti su internet, siamo diventati sempre più dipendenti delle copie digitali di documenti cartacei, il più delle volte nella forma di foto. Ma possiamo davvero fidarci di loro?

Metodi per modificare le immagini diventano sempre più sofisticati e facili da usare; è possibile infatti ottenere ottimi risultati con tecniche difficili da individuare (per l'occhio umano). In questo progetto esploriamo vari metodi per identificare immagini modificate, per poi implementarli su ambiente Android. Infine viene valutata la robustezza dei metodi analizzati rispetto ad alcune possibili tecniche che un attaccante potrebbe usare per nascondere la modifica.

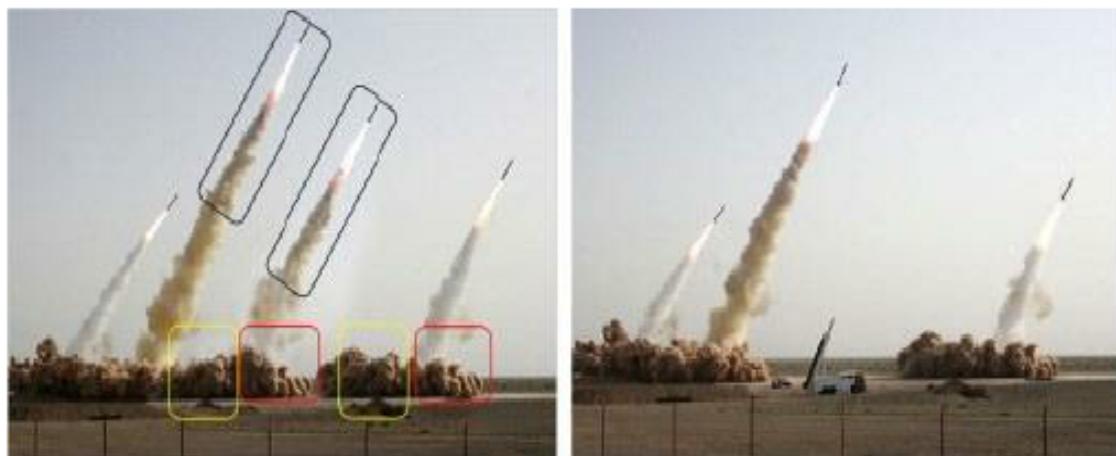


Figura 1.1: Un esempio di immagine modificata: a destra l'originale, con solo 3 missili, mentre a sinistra l'immagine a cui è stato aggiunto un missile con la tecnica del *copy-move*.

Capitolo 2

Tecniche utilizzate

In base al tipo di immagine e al tipo di attacco, differenti tipi di informazione possono essere utili. Per esempio pattern e correlazione tra i pixel dell'immagine potrebbero nascondere informazioni sul tipo di algoritmo di compressione usato. Queste informazioni sono molto utili in quanto è possibile sfruttare queste *features* invisibili (all'occhio umano) per identificare possibili modifiche.

Ad esempio l'area "modificata" dell'immagine che analizziamo ha una più alta possibilità di avere un livello di compressione diverso dal resto dell'immagine. In un modo simile possiamo sfruttare il fatto che due immagini usate per ottenere un'immagine modificata siano state prodotte da due differenti sensori di telecamere. Ogni sensore infatti ha piccole imperfezioni che si trasferiscono nelle immagini che cattura. Così con le informazioni della compressione possiamo sfruttare queste imperfezioni (che sono chiamate "Photo Response Non-Uniformity" o PRNU) per identificare in modo affidabile se l'immagine analizzata è stata modificata o meno.

Sono state utilizzate quindi tre tecniche che vanno ad analizzare tre aspetti diversi dell'immagine.

2.1 Analisi RGB

RGB data: I semplici valori RGB dell'immagine sono stati utilizzati per allenare la rete. Questa tra le tecniche utilizzate è la più naiva, utilizzata principalmente come base di confronto per analizzare le altre due.



(a) Immagine non modificata.

(b) Immagine modificata.

Figura 2.1: Immagini in RGB.

2.2 SRM

SRM (Steganalysis Rich Model) noise [4]: Mappa delle caratteristiche del rumore ottenuta da pixel adiacenti, la quale cattura le inconsistenze tra regioni modificate e regioni autentiche. I canali RGB non sono sufficienti per identificare tutti i casi di manipolazione. In particolare, immagini modificate con attenzione al nascondere i contorni e le differenze di contrasto sono difficili da identificare per il modello RGB.

Contrariamente all’analisi RGB, lo stream di rumore che estraiamo è pensato per prestare più’ attenzione al rumore più’ che al contenuto semantico dell’immagine.

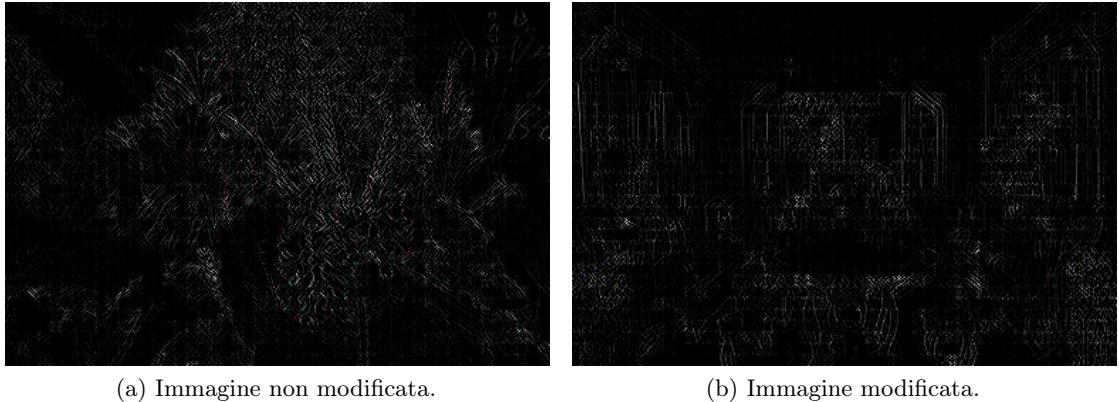
Questo è diverso da molti modelli che funzionano molto bene nel rappresentare caratteristiche gerarchiche dal contenuto RGB dell’immagine in quanto prova ad imparare dalla distribuzione del rumore. Come visto in alcuni paper di analisi forense [8] usiamo i filtri SRM per estrarre le features locali del rumore dalle immagini RGB che sono l’input.

Il rumore è modellato dalla differenza del valore di un pixel e la stima di quel pixel prodotta dall’interpolazione dei valori dei suoi pixel vicini.

Come consigliato da Zhou et al. [4] abbiamo scelto tre *kernel* con i pesi riportati nella tabella sottostante. Come si può vedere dalla mappa estratta nella figura 2.3, viene esaltato il rumore locale invece del contesto dell’immagine, rendendo possibile il rilevamento di artefatti dovuti alla modifica dell’immagine non visibili tramite i soli canali RGB.

$$\frac{1}{4} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 2 & -4 & 2 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \frac{1}{12} \begin{bmatrix} -1 & 2 & -2 & 2 & -1 \\ 2 & -6 & 8 & -6 & 2 \\ -2 & 8 & -12 & 8 & -2 \\ 2 & -6 & 8 & -6 & 2 \\ -1 & 2 & -2 & 2 & -1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (2.1)$$

Figura 2.2: I tre kernel usati per estrarre la mappa SRM



(a) Immagine non modificata.

(b) Immagine modificata.

Figura 2.3: Mappa SRM delle immagini.

2.3 Noiseprint

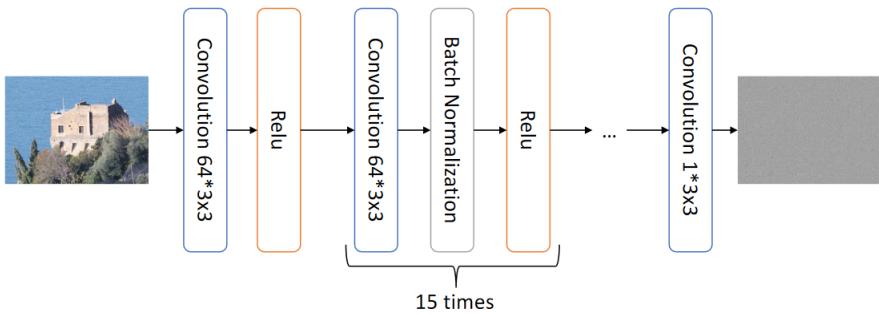


Figura 2.4: Architettura della rete per l'estrazione del noiseprint

La terza ed ultima tecnica analizzata è stata proposta ed implementata da Cozzolino et al. [3]. In questo progetto viene ripresa l'implementazione per ottenere il *noiseprint* che sarà utilizzato per allenare la rete.

La maggior parte dei metodi data-driven per l'analisi di immagini manomesse si basano sul "noise residual" ovvero un segnale simile ad un rumore che rimane una volta rimossi dall'immagine i contenuti semanticici di alto livello. Tra i vari metodi basati sul rumore residuo, quelli che si affidano al rumore "photo-response non-uniformity" (PRNU) meritano un'attenzione speciale per la loro popolarità ed efficacia.

Ogni dispositivo infatti lascia dei segni specifici su tutte le immagini che acquisisce (il pattern PRNU) a causa delle imperfezioni nel processo di fabbricazione del dispositivo stesso. Grazie a questa singolarità, e stabilità nel tempo, il pattern PRNU può essere equiparato ad un *fingerprint* del dispositivo ed usato in diversi lavori forensi con delle

buone prestazioni.

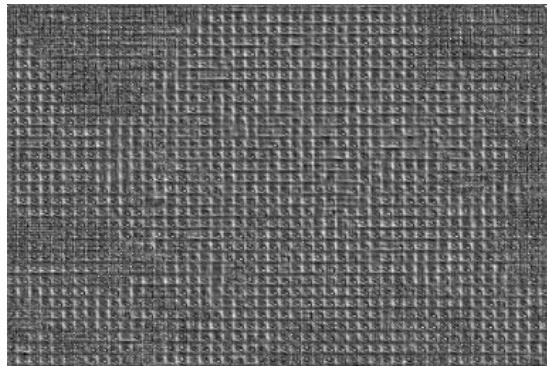
Gli svantaggi dei metodi basati su PRNU sono:

- La necessità di un grande numero di immagini catturate dalla fotocamera per ottenere una buona stima
- La bassa potenza del segnale che ci interessa rispetto al rumore, che sfortunatamente, incide sulle prestazioni. In particolare la fonte principale di rumore è il contenuto di alto livello dell' immagine che finisce nel PRNU. In situazioni di immagini scura, saturata o con *texture* può capitare che questo rumore predomini sul PRNU.

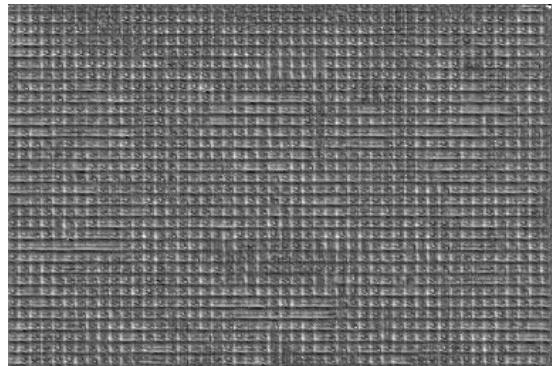
L'approccio considerato in questo progetto è quello utilizzato da Cozzolino et al. [3] in cui viene estratto il **Noiseprint**, ovvero il rumore residuo dell'immagine lasciato dai vari modelli di fotocamere. Questa tecnica permette di superare le problematiche che affliggono i metodi basati su PRNU sopra descritte.

Si noti che, nonostante il noiseprint estragga l'impronta lasciata dal modello di fotocamera, anche una modifica che utilizzi porzioni di immagine create dalla stessa fotocamera verrà rilevata, in quanto il noiseprint estratto è sì relativo al modello, ma dipende anche dalla posizione, quindi pezzi di immagine presi da posizioni diverse, seppur scattate dallo stesso modello, avranno un noiseprint diverso.

Nelle immagini sottostanti si può notare come questa tecnica sia poco utile per un'analisi fatta ad occhio nudo, ma come vedremo in seguito risulta molto performante se utilizzata da un computer.



(a) Noiseprint di un'immagine non modificata.



(b) Noiseprint di un'immagine modificata.

Figura 2.5: Mappa del Noiseprint delle immagini.

2.3.1 Architettura di estrazione del Noiseprint

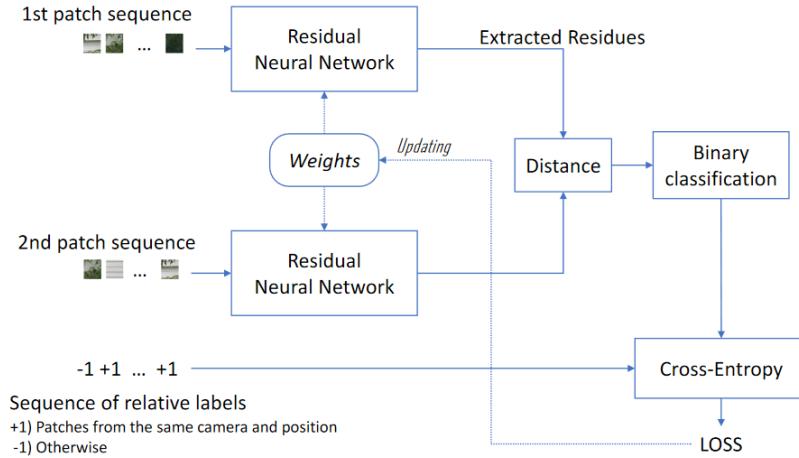


Figura 2.6: Architettura siamese per l'estrazione del noiseprint.

Per estrarre il noiseprint, in [3] viene utilizzata una architettura **Siamese** (rete neurale costituita da due sottoreti identiche, le cui uscite sono unite) formata da due reti neurali convoluzionali **identiche**. Due *patch* differenti d'immagine vengono date in input alle due reti, ed i rispettivi output vengono utilizzati come output desiderati reciproci per calcolare l'errore ed aggiornare i pesi. Nel caso in cui le *patch* vengano dallo stesso modello di fotocamera, e dalla stessa posizione, i pesi sono aggiornati in modo da ridurre la distanza tra gli output, altrimenti sono aggiornati in modo da aumentarne la distanza.
Di seguito vengono descritti i dettagli dell'implementazione effettuata in [3] il cui output è stato utilizzato nel nostro progetto per ottenere il noiseprint dalle immagini analizzate.

Dettagli implementativi

La rete in figura 2.4 è stata allenata utilizzando *minibatch* di 200 *patch* di 48x48 pixel, ma essendo completamente convoluzionale, funziona su immagini di qualsiasi dimensione.

Loss function

Per quanto riguarda la *loss function*, è stata utilizzata la *distance based logistic* (DBL):

Siano $\{x_1, \dots, x_n\}$ un *minibatch* di input, e $\{r_1, \dots, r_n\}$ l'output residuo corrispondente. Sia quindi $d_{ij} = \|r_i - r_j\|^2$ la distanza euclidea al quadrato tra i residui i e j , è necessario che la distanza sia piccola quando i e j devono avere il noiseprint simile e grande in caso contrario.

Si può costruire ora per ogni *patch* una distribuzione di probabilità:

$$p_i(j) = \frac{e^{-d_{ij}}}{\sum_{j \neq i} e^{-d_{ij}}}$$

La definizione precedente di distanza si trasforma ora nella necessità che $p_i(j)$ sia grande quando le due *patch* appartengono allo stesso gruppo, quindi $l_{ij}=+1$, e piccolo altrimenti. Definiamo quindi la *loss* della *patch* i-esima come:

$$\mathcal{L}_i = -\log \sum_{j:l_{ij}=+1} p_i(j)$$

Di conseguenza, la *loss* per i *minibatch* è definita come:

$$\mathcal{L}_0 = \sum_i [-\log \sum_{j:l_{ij}=+1} p_i(j)]$$

Optimizer

Come ottimizzatore è stato utilizzato *Adam*

Risultato

Considerando il forte impatto della compressione JPEG, sono state utilizzate reti differenti per ogni *JPEG quality factor*. In questo progetto vengono quindi utilizzate queste reti per ottenere il noiseprint.

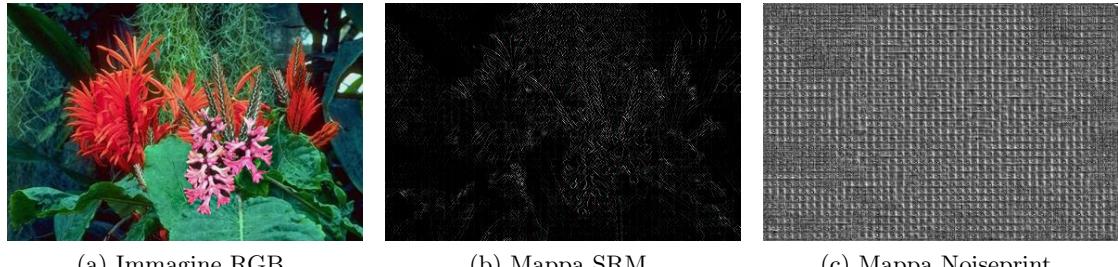


Figura 2.7: Mappe estratte dall'immagine.

Capitolo 3

Dataset

Una decisione importante è stata quella del dataset da utilizzare.

Sfortunatamente non ci sono molti candidati su internet. Molti dataset per le immagini modificate sono o a pagamento, o troppo piccoli per l'allenamento di modelli di deep learning. Alla fine la scelta è ricaduta sul dataset CASIA2 [1].

Questo dataset contiene 7492 immagini autentiche e 5125 immagini modificate di diverse dimensioni sia in formato .jpeg che .tiff .

Un altro aspetto a cui portare attenzione è il tipo di modifica effettuata sull'immagine. Nel dataset CASIA2 solo due tipi di modifiche sono state utilizzate:

- **Image splicing:** Copia di una o più sezioni di un'immagine in un'altra immagine.
- **Copy-Move Forgery:** Copia di una sezione di un'immagine in un'altra posizione della stessa immagine.



(a) Image Splicing.



(b) Copy-Move.

Figura 3.1: Tipologia di immagini modificate presenti nel dataset.

Capitolo 4

Rete Neurale

Come base per il modello abbiamo optato per ResNet50 [9].

4.1 Residual Network

Comunemente, una rete neurale convoluzionale è composta da vari livelli che estraggono caratteristiche dell'immagine di astrazione crescente. Nel *residual learning* invece di voler imparare delle *feature*, si imparano dei "residui", ovvero una sottrazione delle *feature* imparate dall'input di un layer. **ResNet** fa questo usando delle connessioni "scorciatoia" che connettono un layer in posizione n ad un altro in posizione n+x.

4.2 ResNet50

ResNet50 [9] è una rete pre-allenata composta da 50 livelli, utilizzata per la classificazione di oggetti. Nello specifico, è composta da 48 livelli Convolutionali, con 1 MaxPool ed 1 Average Pool. Noi abbiamo scelto di allenare la rete ResNet50 (inizializzata con pesi casuali) con in uscita un livello *Flatten* e tre livelli *Dense*, i primi due con funzione di attivazione *relu* ed in output rispettivamente 128, e 64 nodi, mentre l'ultimo livello (anch'esso un *Dense*) ha un solo nodo di output il quale indicherà se l'immagine in input è stata modificata o meno. (Un valore ≥ 0.5 indicherà che l'immagine è stata modificata), con una *sigmoide* come funzione di attivazione.

```
1 def build_model(self, input_shape, n_classes) -> Model:  
2     resnet = ResNet50(include_top=False, input_shape=input_shape,  
3                         weights=None)  
4  
5     model = Flatten()(resnet.output)  
6     model = Dense(128, activation='relu', kernel_initializer='  
7         he_uniform')(model)  
8     model = Dense(64, activation='relu', kernel_initializer='  
9         he_uniform')(model)  
10    model = Dense(1, activation='sigmoid')(model)
```

```
8  
9     return Model(inputs=resnet.inputs, outputs=model)
```

Listing 4.1: Funzione che definisce la struttura del modello.

Nell'inizializzazione della ResNet50, il parametro weights=None indica che la rete è inizializzata con pesi casuali.

4.3 Parametri

Per allenare la rete abbiamo provato diversi parametri e configurazioni, ed alla fine quelli che hanno dato risultati migliori sono i seguenti:

- **Learning Rate:** 0,0001
- **Optimizer:** Adam
- **Batch Size:** 16

Capitolo 5

Implementazione

5.1 SRM

```
1 class SRMExtractor(MapExtractor):
2
3     def __init__(self):
4
5         filter1 = [[0, 0, 0, 0, 0],
6                    [0, -1, 2, -1, 0],
7                    [0, 2, -4, 2, 0],
8                    [0, -1, 2, -1, 0],
9                    [0, 0, 0, 0, 0]]
10
11        filter2 = [[-1, 2, -2, 2, -1],
12                   [2, -6, 8, -6, 2],
13                   [-2, 8, -12, 8, -2],
14                   [2, -6, 8, -6, 2],
15                   [-1, 2, -2, 2, -1]]
16
17        filter3 = [[0, 0, 0, 0, 0],
18                   [0, 0, 0, 0, 0],
19                   [0, 1, -2, 1, 0],
20                   [0, 0, 0, 0, 0],
21                   [0, 0, 0, 0, 0]]
22
23        filter1 = np.asarray(filter1, dtype=float) / 4
24        filter2 = np.asarray(filter2, dtype=float) / 12
25        filter3 = np.asarray(filter3, dtype=float) / 2
26
27
28    def extract(self, base):
29        return cv2.filter2D(base, -1, self.filter)
```

Listing 5.1: Estrazione SRM.

Per estrarre l'SRM dalle immagini è stata utilizzata la funzione **filter2D** della libreria **OpenCV**, con i seguenti parametri:

- **src** = base (Immagine di input.)
- **ddepth** = -1 (Profondità dell'immagine di destinazione, ovvero il tipo di dato di ogni pixel. Il valore **-1** indica che la destinazione avrà la stessa profondità della sorgente.)
- **kernel** = self.filter (kernel da applicare all'immagine per estrarre l'SRM. Presi da [4])

5.2 Noiseprint

Per l'estrazione del noiseprint ci siamo affidati ad una rete pre-allenata da Cozzolino et al. [3]. Nello specifico abbiamo a disposizione 51 modelli allenati per estrarre la mappa del noiseprint. In base alla qualità dell'immagine viene scelto il modello da utilizzare. Il codice seguente che si occupa di estrarre il noiseprint è anch'esso ripreso e adattato a partire da quello usato nell'articolo citato precedentemente.

```

1 class NoiseprintExtractor(MapExtractor):
2
3     def extract(self, image):
4
5         quality = 200
6         try:
7             quality = jpeg_qtableinv(image)
8         except:
9             quality = 200
10            image = np.asarray(Image.fromarray(image).convert("YCbCr"))[...,
11            0].astype(np.float32) / 256.0
12
13            return NoiseprintEngineV2(quality).predict(image)[..., np.newaxis
14        ]

```

Listing 5.2: Estrazione SRM.

La funzione *extract* della classe *NoiseprintExtractor()* si occupa di ottenere la qualità dell'immagine, e di chiamare il modulo che estrarrà il noiseprint.

La funzione *jpeg_qtableinv(image)* estrae la qualità dell'immagine, in base alla quale verrà scelto il modello che si occuperà di estrarre il noiseprint.

In base alla qualità dell'immagine viene quindi istanziato un oggetto *NoiseprintEngineV2* il quale si occupa di estrarre il noiseprint e normalizzarlo.

Capitolo 6

Risultati

Le precisioni dei modelli allenati sul dataset di test sono i seguenti:

Tabella 6.1: Accuratezza dei modelli

Set	Modelli		
	<i>RGB</i>	<i>SRM</i>	<i>Noiseprint</i>
test set	69%	72%	83%

Come si può vedere, i risultati sull'immagine RGB sono i peggiori, con solo il 69% di accuratezza. Abbiamo ottenuto risultati un po' migliori con il modello SRM il quale predice correttamente il 72% delle immagini, ma siamo ancora lontani da un valore ottimale. Il noiseprint, come previsto, è la tecnica che ci ha dato i risultati più soddisfacenti con l'83% di accuracy nel dataset di test.

A nostro avviso, utilizzando un'altra tipologia di rete, ottimizzata per questo compito, ed avendo un dataset migliore, si dovrebbero poter raggiungere risultati migliori. Durante la fase di allenamento, infatti, ci siamo resi conto che la rete soffre di *overfitting*, e nonostante una serie di perfezionamenti che abbiamo effettuato, non siamo riusciti a migliorare la situazione. L'utilizzo quindi di una rete ad hoc, associata ad un dataset di qualità, dovrebbero migliorare i risultati.

6.1 Tempistiche

Di seguito il tempo necessario a predire se un'immagine è modificata in base alla tecnica utilizzata.

Tabella 6.2: Tempistiche dei modelli.

Set	Modelli		
	<i>RGB</i>	<i>SRM</i>	<i>Noiseprint</i>
Tempo	0.0559s	0.0535s	0.0602s

Capitolo 7

Portabilità su Android

Il progetto sopra descritto è stato poi adattato per poter eseguire su ambiente Android. L'applicazione risultante sarà in grado di analizzare le immagini della galleria dando un responso per ognuna delle tre tecniche utilizzate.

7.1 Conversione modelli in TfLite

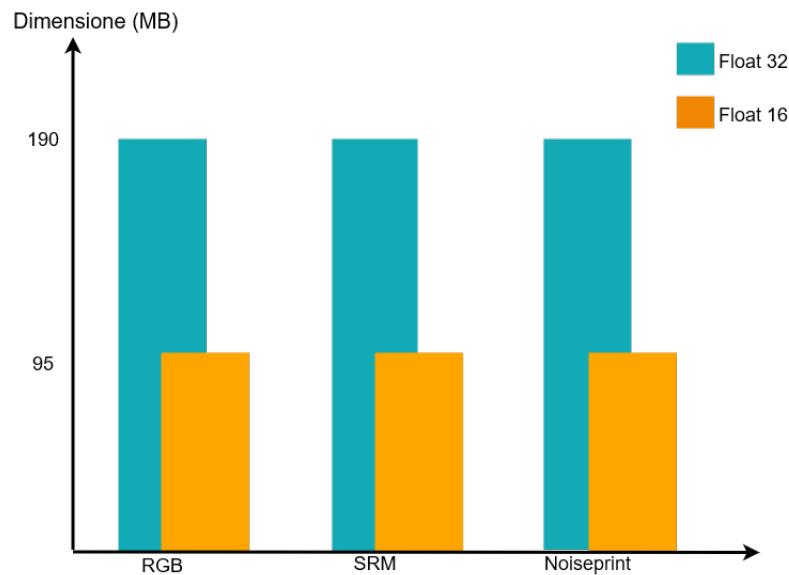


Figura 7.1: Dimensione dei modelli in TfLite

La prima operazione da eseguire per poter operare in ambiente Android è la conversione dei modelli Tensorflow ottenuti in modelli Tensorflow Lite. Per tutti e tre i modelli sono state considerate sia la conversione in float32, che la quantizzazione con pesi in float16. Dai risultati ottenuti l'accuratezza è rimasta pressochè invariata mentre, come si può

vedere in figura 7.1, la dimensione dei modelli è passata da 190MB a 95MB per tutti e tre, ottenendo una buona riduzione della dimensione dell'app. La dimensione finale dell'apk è di 1GB, la maggior parte dei quali dovuti alle dimensioni dei modelli *.tflite* ed all'utilizzo di *Chaquopy*.

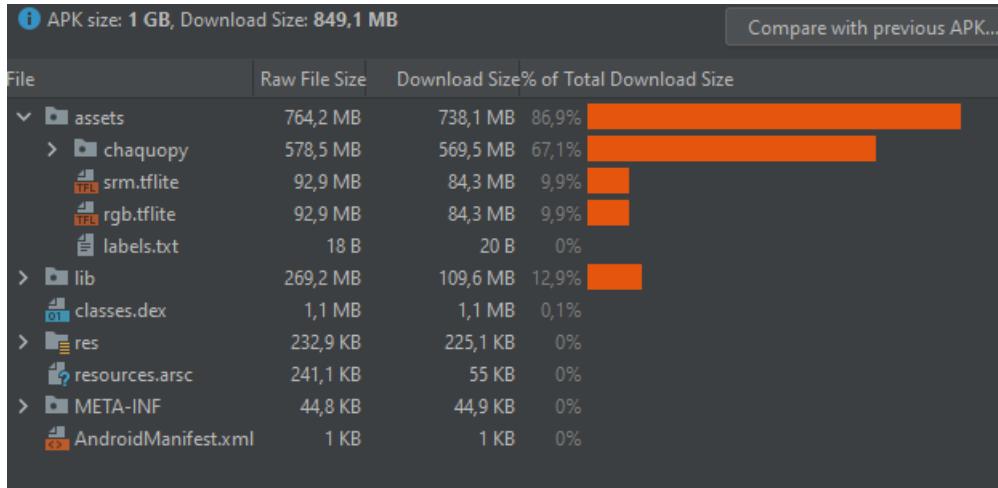


Figura 7.2: Dimensione APK

7.1.1 Conversione dei modelli per l'estrazione del noiseprint

Anche i modelli già allenati forniti da Cozzolino et al. [3] sono stati ovviamente convertiti in Tensorflow Lite per poter estrarre il noiseprint anche sui dispositivi mobili. Si tratta dei 51 modelli di cui, in base ad ogni immagine, uno viene scelto per estrarre il noiseprint. In questo caso abbiamo preferito non applicare quantizzazione per evitare di poter compromettere la qualità del noiseprint in output. I modelli *.tflite* ottenuti pesano ciascuno 2,18MB per un totale di 111MB.

7.2 RGB

Per quanto riguarda la semplice analisi RGB viene utilizzato il modello TfLite risultante per ottenere l'output direttamente dalle immagini in galleria.

7.3 SRM

L'estrazione dell'SRM map è stata effettuata in modo simile al codice python, utilizzando sempre la funzione *filter2D* di OpenCV, ma sfruttando questa volta gli oggetti *Mat* di OpenCV al posto degli array *Numpy*, e le funzioni *bitmapToMat* e *matToBitmap* per convertire le immagini *Bitmap* ad oggetti *Mat* e viceversa.

```

1 public static Bitmap extract(Bitmap img){
2     System.loadLibrary( Core.NATIVE_LIBRARY_NAME );
3
4     Mat src = new Mat();
5     Mat srcRgb = new Mat();
6
7     Utils.bitmapToMat(img, src);
8     Imgproc.cvtColor(src, srcRgb, Imgproc.COLOR_RGBA2RGB);
9
10
11    float[][] filter1 = {{0, 0, 0, 0, 0},{0, -1, 2, -1, 0},{0, 2, -4,
12    0},{0, -1, 2, -1, 0},{0, 0, 0, 0, 0}};
13    float[][] filter2 = {{-1, 2, -2, 2, -1},{2, -6, 8, -6, 2},{-2, 8,
14    -12, 8, -2},{2, -6, 8, -6, 2},{-1, 2, -2, 2, -1}};
15    float[][] filter3 = {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 1, -2,
16    1, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}};
17
18    //Create the kernels
19    Mat kernel = new Mat(5, 5, CvType.CV_32F);
20    for(int i=0; i<5; i++){
21        for(int j=0; j<5; j++) {
22            kernel.put(i, j, (filter1[i][j] / 4) + (filter2[i][j] /
23            12) + (filter3[i][j] / 2));
24        }
25    }
26
27    Mat dst = new Mat();
28    Imgproc.filter2D(srcRgb, dst, -1, kernel);
29
30    Bitmap result = Bitmap.createBitmap(img.getWidth(), img.getHeight
31    (), Bitmap.Config.ARGB_8888);
32    Utils.matToBitmap(dst, result);
33
34    return result;

```

Listing 7.1: Estrazione del noiseprint in Java

Una volta estratto l'SRM, questo è dato in input al modello TfLite che restituisce il risultato.

7.4 Noiseprint

Per quanto riguarda il Noiseprint, avendo riscontrato difficoltà nel convertire il codice python ottenuto da [3] in corrispondente codice Java, in particolare nel trovare un'alternativa valida alla manipolazione di array data da *numpy*, abbiamo preferito utilizzare il plugin *Chaquopy*.

7.4.1 Chaquopy

Chaquopy è un plugin per sistemi Android basati su *Gradle* che permette di utilizzare codice python integrandolo in applicazioni Android.

7.4.2 Implementazione

L'idea di base era quella di manipolare l'immagine ed ottenere il noiseprint in ambiente Chaquopy, per poi restituirlo in ambiente Java che l'avrebbe utilizzato per effettuare l'inferenza. Questa operazione consisteva nel convertire l'immagine *Bitmap* un array di *byte* da dare in input alla funzione di estrazione in python, e restituire il noiseprint sempre in forma di array di byte come consigliato anche nella documentazione di Chaquopy.

Purtroppo dopo svariati tentativi, di fronte anche alla difficoltà di trovare una documentazione completa, non siamo riusciti a risolvere gli errori che si presentavano in ambiente python per convertire l'immagine da array di byte ad array numpy, quindi abbiamo preferito ricorrere ad una tecnica che, seppur meno efficiente, ci ha portato ad una soluzione funzionante e valida, ovvero salvare temporaneamente l'immagine nel file system per utilizzarla in python, ed una volta estratto il noiseprint, ottenere direttamente in python il risultato della rete che sarà poi passato all'ambiente Java:

```
1 def noiseprint_calc():
2
3     #Load the image file
4     path = os.path.join(os.environ['HOME'], 'src.png')
5     src = Image.open(path).convert("RGB")
6     src = np.asarray(src)
7
8     #Load the model based on the quality of the image
9     quality = 200
10    try:
11        quality = jpeg_qtableinv(src)
12    except:
13        quality = 200
14
15    quality = min(max(quality, 51), 101)
16    file_name = 'net.jpg' + str(quality) + '.tflite'
17    model_path = os.path.join(os.path.dirname(__file__), file_name)
18
19    #Setup the interpreter
20    global interpreter
21    interpreter = tf.lite.Interpreter(model_path=str(model_path))
22    interpreter.allocate_tensors()
23    global input_details
24    input_details = interpreter.get_input_details()
25    global output_details
26    output_details = interpreter.get_output_details()
27
28    #Extract the noiseprint
29    src = np.asarray(Image.fromarray(src).convert("YCbCr"))[..., 0].
30    astype(np.float32) / 256.0
31    noiseprint = predict(src)[..., np.newaxis]
32
33    #Normalization
34    margin = 34
35    v_min = np.min(noiseprint[margin:-margin, margin:-margin])
36    v_max = np.max(noiseprint[margin:-margin, margin:-margin])
```

```

36     norm_noiseprint = ((noiseprint - v_min) / (v_max - v_min)).clip(0, 1)
37
38     return java.jfloat(get_result(norm_noiseprint))

```

Listing 7.2: Caricamento dell'immagine - estrazione del noiseprint - normalizzazione - inferenza

La funzione *noiseprint_calc* chiamata dall'ambiente Java carica l'immagine dal file system, sceglie il modello più adatto per estrarre il noiseprint, lo estraе, lo normalizza, poi chiama la funzione *get_result* che restituisce la classificazione sotto forma di *float*, e passa il risultato ottenuto all'ambiente Java.

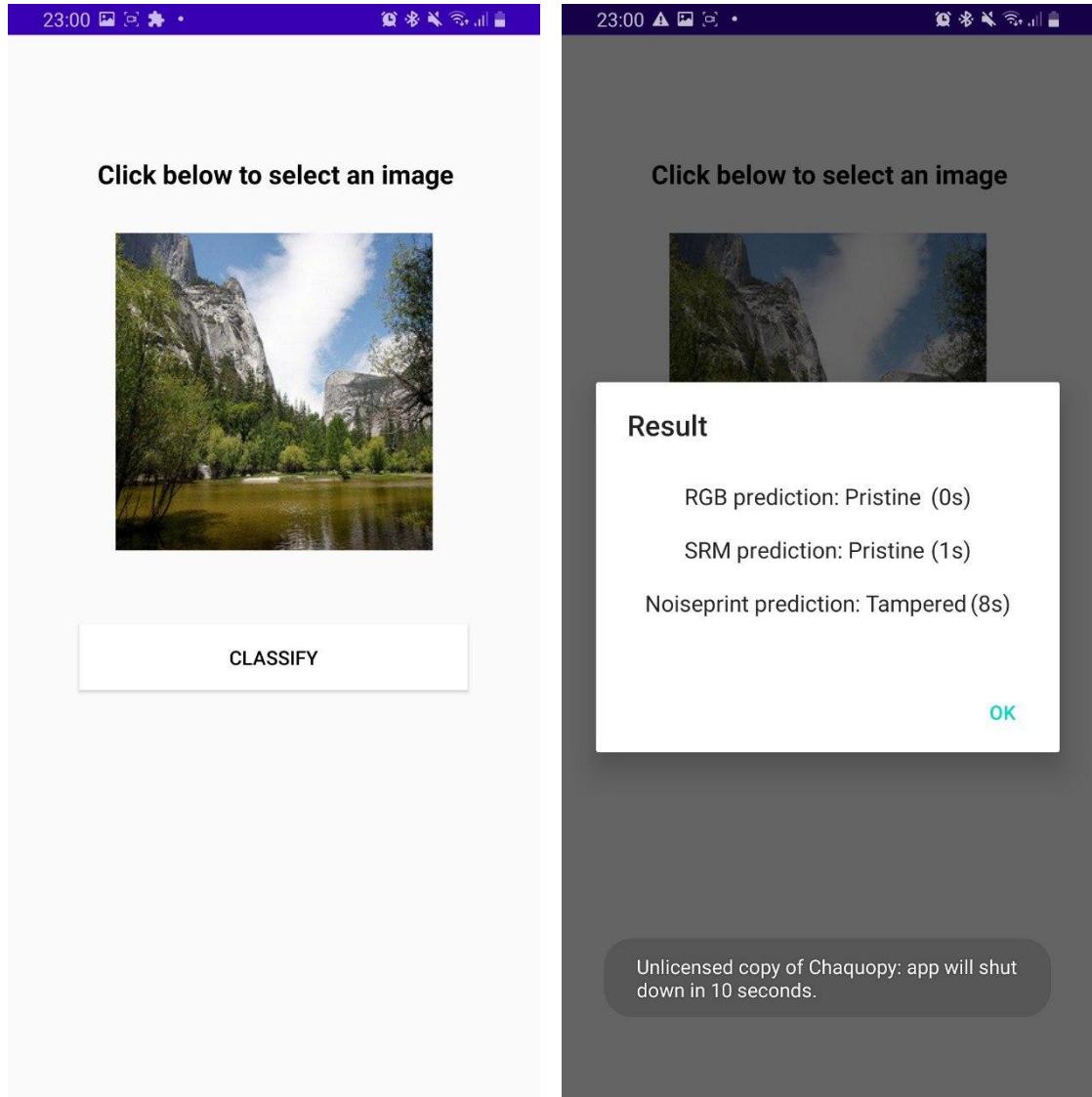
```

1 def get_result(norm_noiseprint):
2
3     noiseprint = np.expand_dims(norm_noiseprint, axis=0)
4
5     model_path = os.path.join(os.path.dirname(__file__), 'noiseprint.
6         tflite')
7     interpreter = tf.lite.Interpreter(model_path=model_path)
8     interpreter.allocate_tensors()
9
10    input_details = interpreter.get_input_details()
11    output_details = interpreter.get_output_details()
12
13    interpreter.set_tensor(input_details[0]['index'], noiseprint)
14    interpreter.invoke()
    return interpreter.get_tensor(output_details[0]['index'])[0][0].item
()
```

Listing 7.3: Inferenza

La funzione *get_result* carica il modello *.tflite* in ambiente python e deduce l'output dal noiseprint calcolato prima.

7.5 Interfaccia



(a) Selezione dell'immagine dalla galleria.

(b) Risultati con tempistiche di calcolo in secondi.

Figura 7.3: Schermate dell'applicazione finale.

7.6 Test

L'applicazione è stata testata su due dispositivi:

Xiaomi Mi A2 Lite

- **CPU:** Cortex-A53 Qualcomm Snapdragon 625, 2 GHz 8 Core
- **GPU:** Adreno 506
- **RAM:** 4 GB
- **OS:** Android 10

Samsung Galaxy S9 Plus

- **CPU:** Exynos 9810
- **GPU:** Mali-G72 MP18
- **RAM:** 6 GB
- **OS:** Android 10

7.6.1 Risultati

Sono state analizzate quindi le tempistiche di esecuzione dell'app sui due dispositivi, e come è evidente dal grafico sottostante, le prestazioni sono **notevolmente** influenzate dall'hardware utilizzato. Nello smartphone *Samsung Galaxy S9 Plus* infatti, sono necessari in totale 8 secondi per ottenere un risultato, circa un quarto del tempo necessario allo smartphone *Xiaomi Mi A2 Lite*.

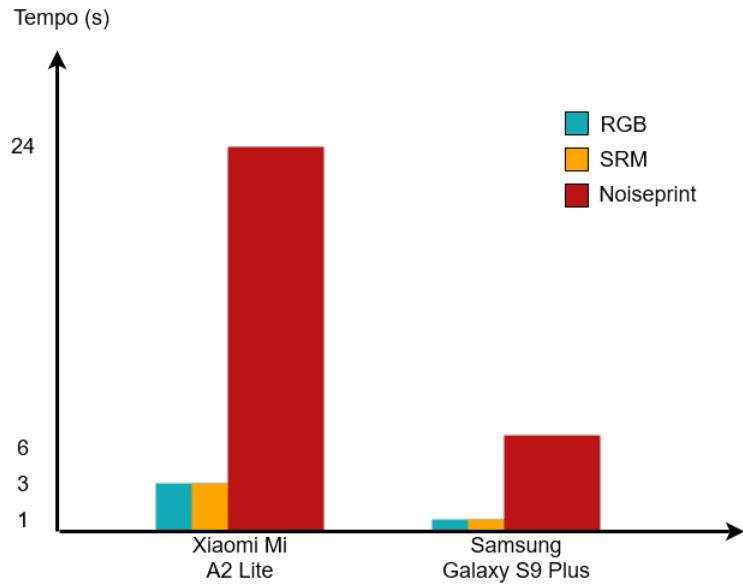


Figura 7.4: Tempo medio necessario per analizzare un'immagine, partendo dal caricamento dell'immagine fino all'ottenimento del risultato.

Capitolo 8

Attacchi

Come fase finale del progetto sono stati implementati una serie di attacchi per testare la robustezza di ciascun modello.

Sono stati scelti degli attacchi di facile riproducibilità, ma che abbiano comunque un forte impatto sull'affidabilità dei modelli testati in precedenza. Le tecniche considerate modificano solamente l'input, e non hanno conoscenze sul modello classificatore.

Qui una lista degli attacchi scelti:

- **Sfocatura**

Un classico attacco con un filtro gaussiano. È prevista una grande efficacia in quanto questo attacco dovrebbe nascondere facilmente il rumore che i modelli SRM e Noiseprint utilizzano. L'effetto di sfocatura è stato applicato con diversi livelli di intensità.

Tabella 8.1: Intensità della sfocatura

	<i>Intensità 1</i>	<i>Intensità 2</i>	<i>Intensità 3</i>
Sigma	2	5	7



(a) Sigma 2.



(b) Sigma 5.



(c) Sigma 7.

Figura 8.1: Intensità di sfocatura.

```

1 from scipy import ndimage
2
3 class BlurModification:
4
5     def __init__(self, sigma):
6         self.sigma = sigma
7
8     @abstractmethod
9     def apply(self, sample):
10        return ndimage.uniform_filter(sample, size=(self.sigma, self
11 .sigma, 1))

```

Listing 8.1: Implementazione della sfocatura

Il filtro gaussiano utilizza la tecnica della convoluzione attraverso un *kernel* i cui valori seguono una distribuzione gaussiana. *Sigma* è un parametro che rappresenta la dimensione del filtro. All'aumentare di sigma aumenta la sfocatura.

- **Luminosità**

Un attacco che dovrebbe ingannare facilmente i modelli che utilizzano i confini segnati dai colori per trovare eventuali modifiche. Il modello noiseprint dovrebbe essere resistente a questo tipo di attacco ma ci si aspetta che un successo contro il modello RGB. La luminosità delle immagini è modificata incrementalmente con i seguenti valori:

Tabella 8.2: Aumento di luminosità

	<i>Intensità 1</i>	<i>Intensità 2</i>	<i>Intensità 3</i>
Incremento	20	50	70



(a) Incremento della luminosità - (b) Incremento della luminosità - (c) Incremento della luminosità -
Intensità 1. Intensità 2. Intensità 3.

Figura 8.2: Immagini con luminosità aumentata.

```

1 import cv2
2
3 class BrightnessModification:
4
5     def __init__(self, value):
6         self.value = value
7
8     @abstractmethod
9     def apply(self, sample):
10
11         hsv = cv2.cvtColor(sample, cv2.COLOR_BGR2HSV)
12         h, s, v = cv2.split(hsv)
13
14         lim = 255 - self.value
15         v[v > lim] = 255
16         v[v <= lim] += self.value
17
18         final_hsv = cv2.merge((h, s, v))
19         img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
20
21         return img

```

Listing 8.2: Implementazione dell'aumento della luminosità

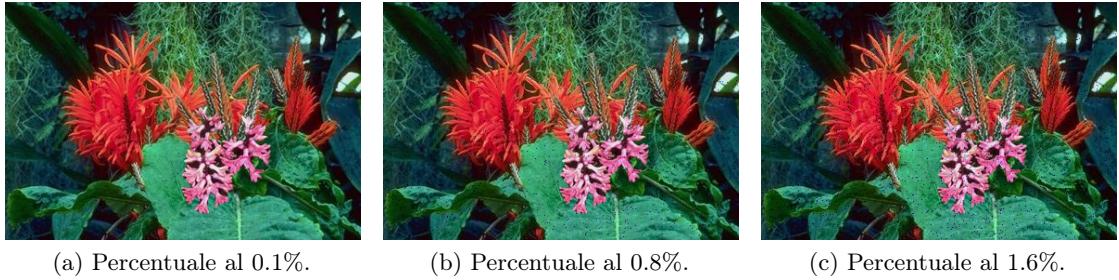
L'immagine da modificare viene prima convertita in rappresentazione *HSV* (H: hue, S: saturation, V: value). V è il parametro che ci interessa, in quanto rappresenta la luminosità dell'immagine.

- **Salt & Pepper**

Con del rumore *salt & pepper* viene aggiunto del disturbo nell'immagine nella forma di pixel sparsi modificati. Quest'attacco dovrebbe avere abbastanza successo contro i classificatori che usano piccole variazione del rumore per individuare immagini modificate. [6] Ogni livello di attacco è denotato dalla quantità di pixel modificati, descritti nella seguente tabella:

Tabella 8.3: Intensità del Salt & Pepper

	<i>Intensità 1</i>	<i>Intensità 2</i>	<i>Intensità 3</i>
Percentuale di pixel modificati	0.1%	0.8%	1.6%



(a) Percentuale al 0.1%.

(b) Percentuale al 0.8%.

(c) Percentuale al 1.6%.

Figura 8.3: Rumore Salt & Pepper.

```

1  class SaltAndPepperModification:
2
3      def __init__(self, s_vs_p, amount):
4          self.s_vs_p = s_vs_p
5          self.amount = amount
6
7      @abstractmethod
8      def apply(self, sample):
9          row, col, ch = sample.shape
10         out = np.copy(sample)
11         # Salt mode
12         num_salt = np.ceil(self.amount * sample.size * self.s_vs_p)
13         coords = [np.random.randint(0, i - 1, int(num_salt))
14                   for i in sample.shape]
15         out[coords] = 1
16         # Pepper mode
17         num_pepper = np.ceil(self.amount * sample.size * (1. - self.
18             s_vs_p))
19         coords = [np.random.randint(0, i - 1, int(num_pepper))
20                   for i in sample.shape]
21         out[coords] = 0
22
23     return out

```

Listing 8.3: Implementazione del rumore salt & Pepper

Il parametro s_vs_p indica la percentuale di *sale* (pixel con valore 1) rispetto al totale di *sale e pepe* (pixel con valore 0) da utilizzare. Per s_vs_p è stato utilizzato il valore 0.5. Il parametro *amount* invece è la quantità di pixel da modificare (i valori sono quelli della tabella sopra).

- **Compressione JPEG**

Questo attacco ricrea la perdita di informazione nell'immagine dovuta all'uso di algoritmi di compressione. Ci si aspetta che sia efficace contro il modello noiseprint in quanto dovrebbe nascondere il rumore sulla quale il modello si basa. La compressione è stata applicata con diverse intensità.

Tabella 8.4: Intensità della compressione

	<i>Intensità 1</i>	<i>Intensità 2</i>	<i>Intensità 3</i>
Qualità rispetto all' originale	75%	50%	25%



(a) Qualità 75%.

(b) Qualità 50%.

(c) Qualità 20%.

Figura 8.4: Immagini con compressione JPEG.

```
1 import imageio
2
3 class CompressionModification:
4
5     def __init__(self, quality):
6         self.quality = quality
7
8     @abstractmethod
9     def apply(self, sample):
10        buf = io.BytesIO()
11        imageio.imwrite(buf, sample, format='jpg', quality=self.
quality)
12        return imageio.imread(buf.getvalue(), format='jpg')
13
```

Listing 8.4: Implementazione della compressione JPEG

8.0.1 Resistenza modello RGB

Tabella 8.5: Accuratezza del modello RGB

Attacchi	Intensità		
	<i>Debole</i>	<i>Media</i>	<i>Forte</i>
Sfocatura	0.64	0.61	0.60
Salt & Pepper	0.60	0.63	0.63
Luminosità	0.59	0.64	0.55
Compressione	0.60	0.66	0.62

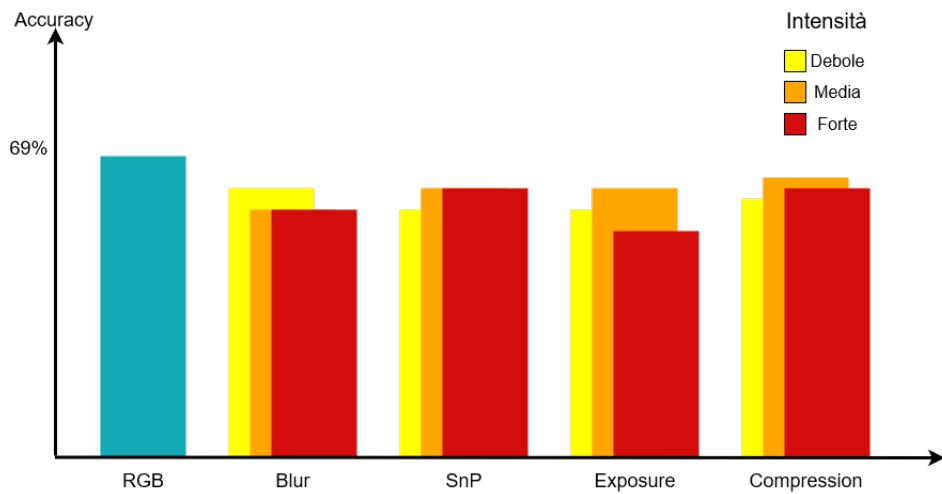


Figura 8.5: Resistenza del modello RGB ai vari attacchi.

8.0.2 Resistenza modello SRM

Tabella 8.6: Accuratezza del modello SRM

Attacchi	Intensità		
	Debole	Media	Forte
Sfocatura	0.59	0.50	0.49
Salt & Pepper	0.70	0.58	0.58
Luminosità	0.72	0.70	0.64
Compressione	0.65	0.63	0.60

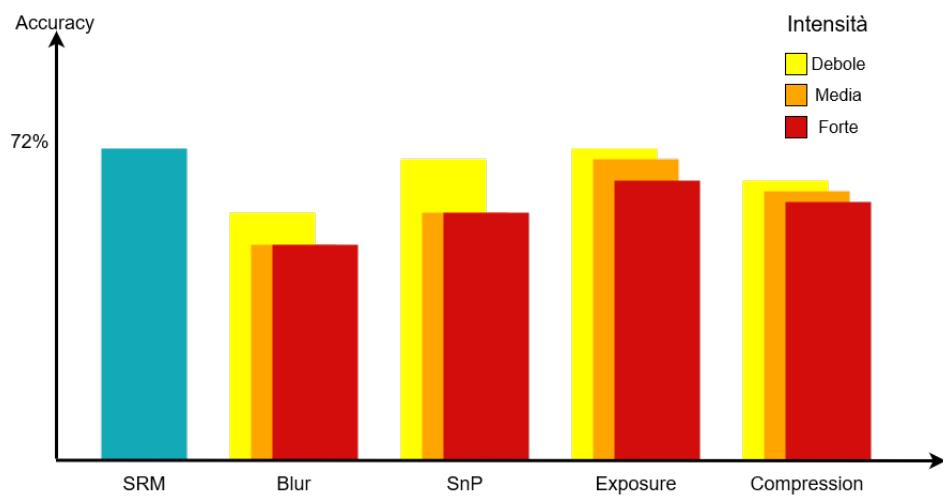


Figura 8.6: Resistenza del modello SRM ai vari attacchi.

8.0.3 Resistenza modello Noiseprint

Tabella 8.7: Accuratezza del modello Noiseprint

Attacchi	Intensità		
	Debole	Media	Forte
Sfocatura	0.54	0.50	0.50
Salt & Pepper	0.78	0.61	0.51
Luminosità	0.75	0.65	0.50
Compressione	0.59	0.55	0.52

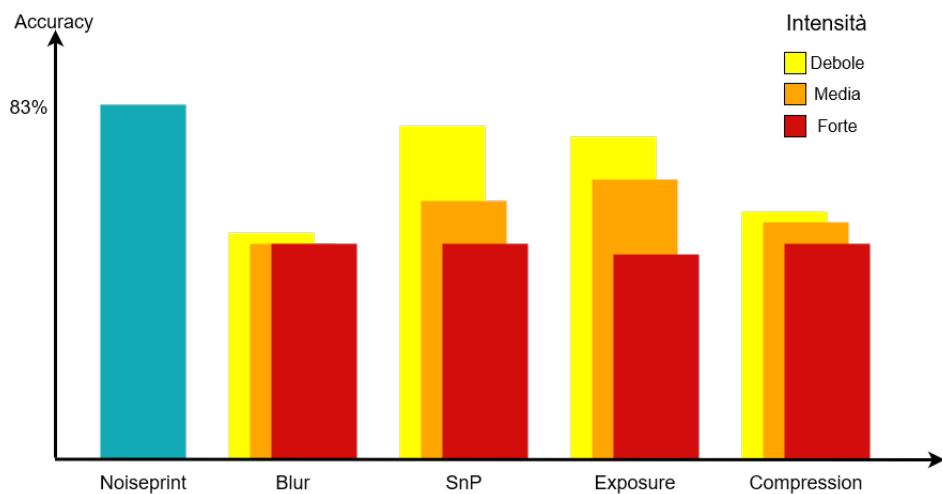


Figura 8.7: Resistenza del modello Noiseprint ai vari attacchi.

Capitolo 9

Conclusione

In conclusione possiamo dire che il noiseprint, anche se ha ottenuto la miglior accuratezza, è allo stesso tempo estremamente vulnerabile a tentativi da parte di un attaccante di nascondere la modifica.

Tecniche facili da implementare, come la sfocatura, funzionano decisamente bene anche se applicate con la minima intensità.

La tecnica SRM ottiene un accuratezza inferiore, è molto suscettibile alla sfocatura, ma allo stesso tempo è più resiliente agli attacchi che aumentano la luminosità ed alla compressione.

Il modello RGB, anche se è il meno performante è quello che è risultato il più difficile da attaccare.

Bibliografia

- [1] Pham, Nam Thanh, et al. "Hybrid image-retrieval method for image-splicing validation." *Symmetry* 11.1 (2019): 83.
- [2] Ahmed, Chuadhry Mujeeb, et al. "Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems." Proceedings of the 2018 on Asia Conference on Computer and Communications Security. 2018.
- [3] Cozzolino, Davide, and Luisa Verdoliva. "Noiseprint: A CNN-based camera model fingerprint." arXiv preprint arXiv:1808.08396 (2018).
- [4] Zhou, Peng, et al. "Learning rich features for image manipulation detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.
- [5] Vasiljevic, Igor, Ayan Chakrabarti, and Gregory Shakhnarovich. "Examining the impact of blur on recognition by convolutional networks." arXiv preprint arXiv:1611.05760 (2016).
- [6] Khalid, Faiq, et al. "Security for machine learning-based systems: Attacks and challenges during training and inference." 2018 International Conference on Frontiers of Information Technology (FIT). IEEE, 2018.
- [7] Yang, Bo, et al. "Random Transformation of Image Brightness for Adversarial Attack." arXiv preprint arXiv:2101.04321 (2021).
- [8] J. Fridrich and J. Kodovsky, "Rich Models for Steganalysis of Digital Images," in *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 868-882, June 2012, doi: 10.1109/TIFS.2012.2190402.
- [9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.