POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Miglioramento delle prestazioni dei moderni codificatori video con reti neurali

Relatore
prof. Maurizio Martina

Candidato
Tiziano Carnevale

Anno Accademico 2022-2023

A mia Madre
A mio Padre
A mio Fratello

Sommario

In questo trattato si analizza principalmente come la tecnologia delle reti neurali stia ormai da qualche anno entrando nel mondo dei codificatori video, per renderli più efficienti. Un codificatore video è sostanzialmente un insieme di script che elaborano dati di immagini per restituire, in output, i frames codificati. Alcuni ricercatori in giro per il mondo stanno sperimentando nel sostituire parti di questi codici con sistemi di Deep-Learning, facendo sì che determinate fasi di codifica vengano operate da una rete neurale invece che dal codificatore stesso.

Gli utilizzi sono dei più vari. Esistono reti impiegate per codifiche *intra* o *inter*. Nella presente tesi, ad esempio, si tratterà di reti neurali impiegate invece come filtri ricostruttori dei frames. Infatti, i codificatori comprimono i dati per ridurne il peso e ciò può causare problemi di *blocking* ed artefatti nelle immagini: urge un filtro che, post-codifica, riaumenti la qualità del fotogramma da reinserire poi nel flusso di codifica. Tale filtro, detto di *deblocking*, può essere quindi implementato con una rete neurale, per avere prestazioni maggiori.

Un approccio del genere, visti i numerosi vantaggi del mondo del Machine-Learning, farà sì che, in uscita, ci siano dati relativi a frames di maggiore qualità con bit-rate inferiore. Anche se, almeno per ora, questo implica numerose volte un tempo di elaborazione maggiore.

Indice

Abstract

Come è ormai noto, l'avvento del digitale ha permesso di rappresentare segnali ed informazioni sotto forma di codice binario. Anche per i contenuti multimediali, ovvero dati video ed audio, questo si traduce nel "codificare" i file: analizzarne le informazioni per riscrivere il tutto come stringa di bit. Tali operazioni vengono normalmente svolte dai codificatori che, inoltre, comprimono le informazioni eliminando i dati in eccesso e ottenendo così contenuti a più bassa memoria. Oggi, le reti neurali stanno piano piano entrando in questo mondo per svolgere alcune fasi di codifica sostituendosi ai codificatori stessi. Il tutto è ovviamente mirato ad ottenere codifiche più prestanti: meno dati e qualità d'immagine più alta.

Questo progetto di tesi vuole esplorare le soluzioni ad oggi proposte riguardo alcune reti neurali introdotte come supporto dei moderni codificatori. Se ne analizzeranno le prestazioni per poi sperimentare le stesse idee là dove non sono ancora state applicate.

Dopo un leggero approfondimento riguardo reti già esistenti e utilizzate per migliorare le prestazioni del codec HEVC, il trattato in questione si sofferma sul funzionamento del MIF-Net: sistema di reti che sostituisce il filtro di *deblocking* in HEVC. La grande intuizione alla base del MIF-Net consiste nell'approccio multi-frame: per ogni fotogramma da filtrare, la rete utilizza due o più frames passati come riferimenti, per aumentare l'efficienza del filtro [**7**]. Verranno mostrati i risultati di diverse codifiche con MIF-Net mirate a verificarne le prestazioni, sperimentando con le diverse modalità di approccio. Tutto questo lavoro è stato svolto a fronte del tirocinio di 300 ore presso il gruppo di ricerca Video del centro sperimentale di Rai – Radiotelevisione italiana.

Terminata la fase da tirocinante, il presente elaborato si impegna a mostrare infine quali sono i primi passi compiuti per cominciare a riportare un sistema come il MIF-Net su un altro codec: EVC.

Questa è la fase di *porting* ed è fondamentale perché, ad oggi, non vi è ancora una rete neurale che sostituisca il DBF (Filtro di *deblocking*) in EVC. L'idea nasce dal gruppo di progetto MPAI che, con ambizione su scala mondiale, ha lo scopo di migliorare le prestazioni di EVC con l'utilizzo di reti neurali. Il presente elaborato punta quindi a collaborare su quanto verrà poi integrato dal gruppo MPAI, il quale collegamento è stato possibile grazie al tutor aziendale del tirocinio, Roberto Iacoviello, che ne fa parte. Si ha l'obiettivo di mostrare i momenti chiave del processo di codifica e dare il giusto spazio alla fase di filtraggio, analizzandone gli aspetti cruciali e descrivendo come essa possa essere potenziata grazie al Machine-Learning. In futuro ci immaginiamo una situazione nella quale le codifiche verranno realizzate per gran parte da reti neurali, cosi da fornire in poco tempo output su numerosi frames ad alta qualità e risoluzione. Tutto questo permetterà di rispettare le moderne specifiche video come 4K, HDR e WCG, senza inciampare in una enorme complessità di calcolo.

L'intero progetto e stato svolto dall'autore della tesi, su direzione ed assistenza del tutor aziendale Roberto Iacoviello, con aiuto e supervisione di colleghi MPAI Alessandra Mosca e Attilio Fiandrotti e grazie alla disponibilità e gentile concessione del professore Martina Maurizio.

Introduzione

Uno dei più strabilianti e rivoluzionari effetti del recente, ma non troppo, progresso tecnologico è stato il passaggio dal mondo analogico a quello digitale. In particolare, questo cambiamento ha impattato pesantemente su aspetti come la trasmissione di segnali ed elaborazione di contenuti multimediali. In musica e cinema, ad esempio, il nuovo mondo digitale ha permesso di riscrivere sotto forma di bit i dati multimediali, rendendo l'elaborazione più veloce, semplice e soprattutto flessibile. Infatti, una volta che un audio o un video viene rappresentato come stringa di bit, ne risulta più immediato poterlo modificare e gestire in qualsiasi momento. Inoltre, sempre per ciò che concerne il multimedia, vi sono i codec: algoritmi pensati per elaborare queste informazioni binarie, riuscendo ad eliminare le meno importanti e ridurre quindi il peso dei file senza impattare in grande misura sulla qualità. Ad esempio, se sappiamo che l'uomo non percepisce determinate frequenze sonore, ci penserà l'algoritmo del MP3 a scartare tutte le informazioni che riguardano tali frequenze. Discorso simile vale poi per l'occhio umano e i dati dei file video: gli algoritmi elimineranno quelle variazioni di colore non percepibili dall'uomo.

Il codificatore (*encoder*) è il lato del codec che codifica: ne esistono già diversi per audio e video, ognuno pensato per uno specifico utilizzo e per essere più efficiente di quello creato in precedenza. Di base i codificatori sono dei codici programmati che ricevono in input i dati multimediali per poi elaborarli, riducendo le ridondanze e le informazioni in eccesso. In output avremo un *bitstream* che racchiude tutti i dati già codificati.

Proponiamo come esempio lo schema a blocchi di uno dei primi codificatori video: MPEG2, del '94.

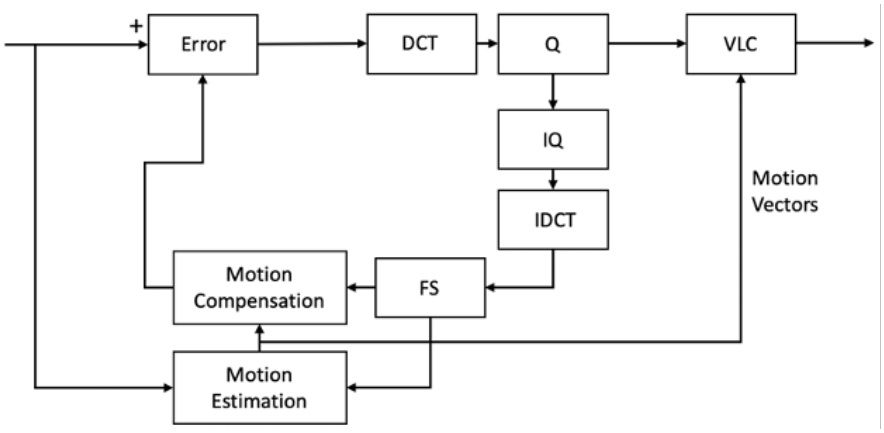

Figura 1: Schema MPEG2

Dall'altro lato, per decodificare, si avrà un *decoder* che riceve in input il *bitstream* al quale applica le operazioni inverse del codificatore, ricavandosi così il video vero e proprio. Questo output non sarà uguale alla sequenza originale in quanto vi sono operazioni non invertibili e alcune informazioni vengono perse per sempre. Di conseguenza, la qualità risulterà inferiore.

Nel mondo video, dopo i primi codificatori MPEG, quali MPEG 1 e 2, che sfruttavano le somiglianze tra frames per codificarle una volta sola e risparmiare quindi spazio in memoria, sono stati fatti enormi progressi, fino ad arrivare inizialmente ad AVC/H.264. Questo codec, nato dalla collaborazione tra la MPEG e la ITU-T, è stato per anni uno standard di riferimento utilizzato ovunque, soprattutto in Internet e nei contenuti *Blu-ray*. Gli sviluppi in informatica ed elettronica hanno poi spinto i due enti di standardizzazione sopra citati a collaborare ancora dando vita ad HEVC/H265. Rispetto al suo predecessore, questo codec sperimenta nuovi tool e algoritmi come il filtro SAO e la dimensione del blocco di codifica fino a 64x64, con una riduzione del bit-rate di circa il 50%. HEVC fu rilasciato nel 2013 e supporta risoluzioni maggiori fino a 8K. Di seguito ne mostriamo lo schema:

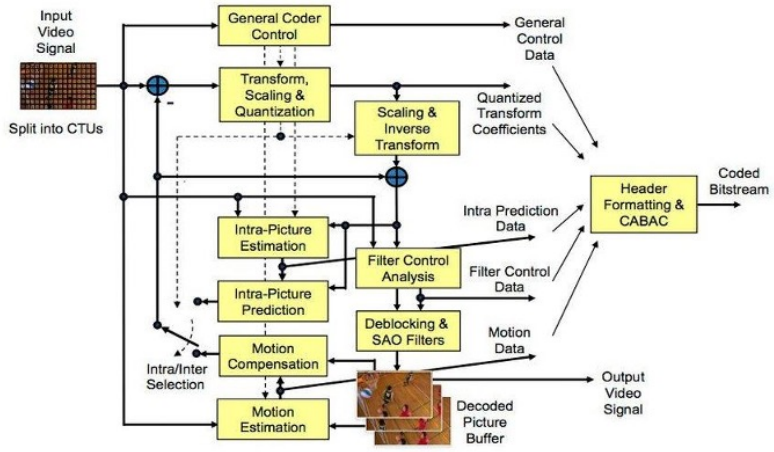

Figura 2: Schema HEVC [**9**]

Sfortunatamente, anche se H265 è ancora uno dei più utilizzati oggi, MPEG e ITU-T hanno avuto non pochi problemi burocratici con esso dato che, nel rilasciarlo, non sono stati specificati in maniera chiara quali brevetti tecnologici furono sfruttati per realizzarlo. Ne deriva che non è facile capire quali licenze pagare, e quanto pagare, per usarlo o per renderlo disponibile a clienti.

Ecco quindi, che MPEG decise di creare un altro codec con un *baselayer* formato da tecnologie sviluppate più di 20 anni prima e quindi teoricamente *royalty free*: EVC (Essential Video Coding). Nel frattempo, le grandi potenze tecnologiche, quali Netflix, Google, Amazon ecc. si unirono per crearne uno di loro proprietà: AV1, ad ora già in uso.

Come detto, i codec sono sostanzialmente algoritmi (C, C++, Python...) ed i risultati ottenuti, seppur ottimi, cominciano a non essere più all'altezza delle specifiche moderne con un mondo multimediale alla continua ricerca di risoluzioni più alte (4K, 8K...), gamme di colore e luminosità più vaste (WCG e HDR) e tempi di calcolo più brevi. Ed è per questa ragione che, in diverse parti del mondo, studiosi e ricercatori stanno già sostituendo alcuni blocchi degli schemi, che corrisponderebbero a parti

di codice, con reti neurali. Ovvero: bypassare sezioni di codice introducendo delle funzionalità che, arrivati a quel punto, inviano i dati a reti neurali già allenate e che svolgeranno le operazioni al posto del codificatore. Al giorno d'oggi, esperimenti del genere sono già stati eseguiti per blocchi che si occupano di codifica *intra* o *inter*, per la super-risoluzione e, più recentemente, per la fase di filtro di *deblocking* (DBF). A questo punto c'è da precisare che, siccome gli algoritmi di codifica ragionano a blocchi di pixel, ovvero codificano una sezione del frame alla volta, si vanno a creare artefatti di codifica proprio là dove vi è il confine tra un blocco e l'altro, e quindi dove finisce una codifica e ne inizia un'altra.

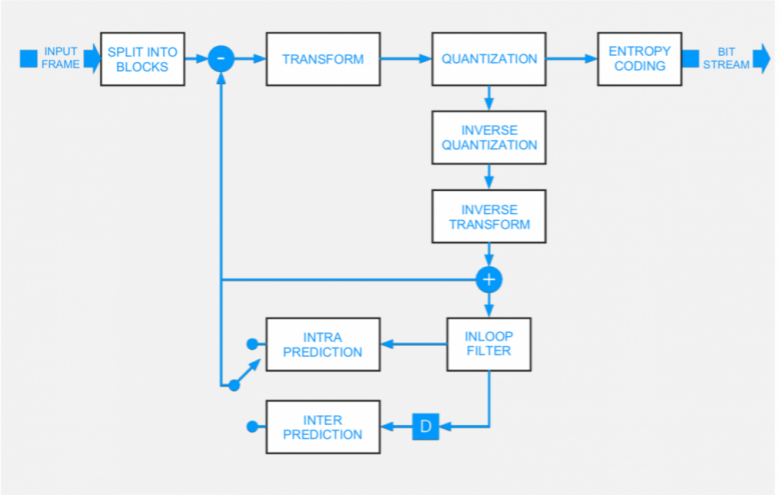

Figura 3: Schema EVC - [**10**]

In particolare, in questa maniera il frame codificato e poi ricostruito appare composto da blocchi che abbassano la qualità dell'immagine dando vita al problema del *blocking artifact*. È per questo che nei codec vi è una sezione dedicata ad analizzare il frame codificato per poi filtrarlo e migliorarlo riducendo la formazione dei blocchi. Da qui, il filtro prende il nome di *deblocking*. In figura 3, dove abbiamo lo schema a blocchi di EVC, il *deblocking* rientra nella casella con il nome di In-Loop-Filter.

Ora, tra i progressi raggiunti nel campo, abbiamo già introdotto le ambizioni di MPAI nell'Abstract del presente testo. MPAI ha inserito a pieno regime in EVC una rete neurale per le fasi di predizione *intra* e per la super-risoluzione. Si è ora quindi interessati ad ottenere risultati simili nelle altre fasi di codifica come, appunto, quella del filtro *in-loop*. Da qui, ribadiamo ancora una volta, nasce l'idea di tesi.

Prima di capire come un sistema di reti neurali venga chiamato da HEVC per svolgere il ruolo di DBF, in questa tesi analizziamo innanzitutto cos'è una CNN (Convolutional Neural Network), per poi fare una panoramica generale delle reti attualmente utilizzate da codificatori moderni per aumentare e migliorare le loro prestazioni, in termini di maggiore qualità (PSNR) e un bit-rate inferiore. Infine, mostreremo come ci si approccia per far sì, appunto, che una simile rete neurale faccia da filtro anche per EVC

Accenni di CNN

Le reti neurali rientrano nel campo del Deep-learning e, tra esse, una particolare configurazione risiede nelle CNN (Convolutional Neural Network). Sono reti neurali artificiali (ANN) utilizzate per analizzare dati di immagini. Condividono modelli più semplici con meno connessioni e complessità limitata, applicando la convoluzione in almeno uno dei livelli (da qui, il loro nome). Da [**2**], scopriamo che LeNet-5 è conosciuta come una classica architettura CNN:

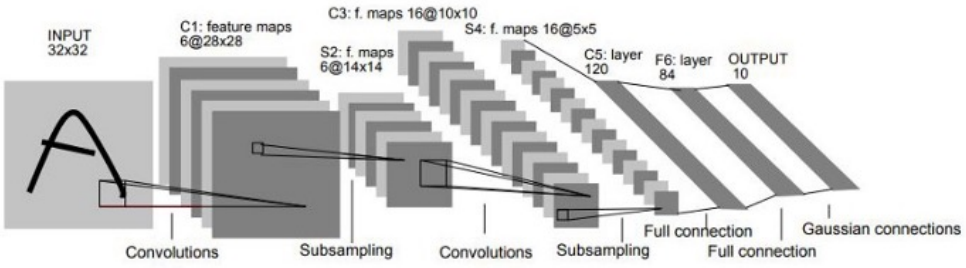

Figura 4: LeNet-5 (struttura) - datasciencecentral.com

Funzionano a livelli di nodi (neuroni). Vi è infatti un primo livello di input (numerico) rappresentato dall'immagine che si vuol analizzare. Poi livelli nascosti intermedi, tra i quali alcuni svolgono l'operazione di convoluzione che restituisce, in output, una mappa delle caratteristiche dell'immagine che sarà poi input per lo strato successivo (come succede nei neuroni del cervello).

Abbiamo anche livelli di *Pooling*, di normalizzazione e un livello completamente connesso che di solito è posto alla fine per rielaborare insieme le *features*, generando così l'output finale [**1**].

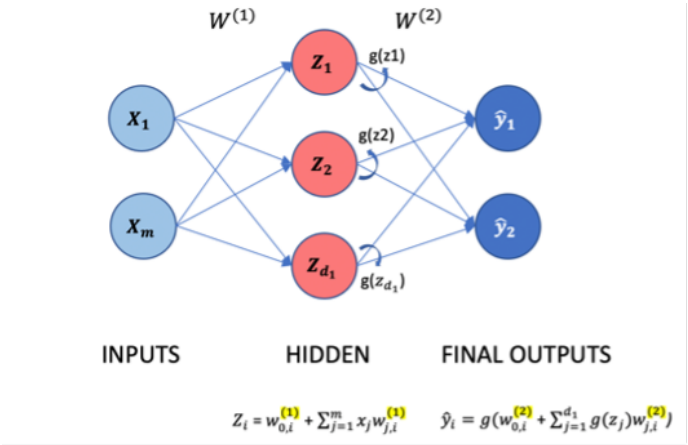

Figura 5: CNN a singolo strato nascosto

Nelle videolezioni di introtodeeplearning.com ([**3**]), registrate al MIT di Boston, si spiega che le CNN, come le reti neurali in generale, possiedono un numero x di input, moltiplicati ognuno per i rispettivi pesi w (appresi in fase di training della rete), che vengono sommati nel nodo stesso e poi sottoposti ad una funzione utile a rendere il sistema non lineare [1]. Come in figura 6:

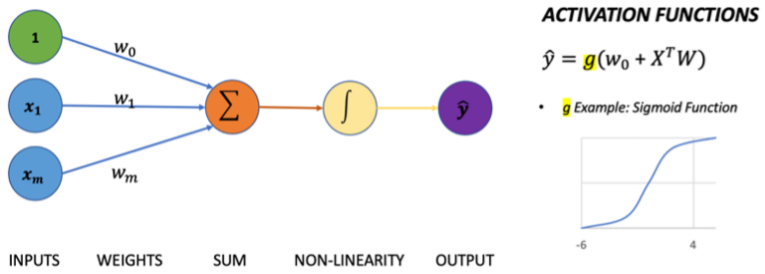

Figura 6: Esempio di nodo (neurone)

[1] La funzione non lineare è spesso attivata da un'unità nota come ReLU (unità lineare rettificata) che porta a zero tutti i valori negativi che escono dopo ogni strato convoluzionale. Va fatto perché le immagini di solito non sono lineari. [**3**]

Esempio di operazione nei nodi in una CNN a 4 strati [**4**]:

$$F_1(Y) = g(W_1 Y + B_1) \tag{1}$$

$$F_i(Y) = g(W_i F_{(i-1)}(Y) + B_i), i = 2,3 \tag{2}$$

$$F(Y) = W_4 F_3(Y) + B_4 \tag{3}$$

Dove Y rappresenta il vettore di input, F_i l'output dello strato i-esimo, g è la funzione non lineare mentre W_i è il vettore dei pesi sullo strato i-esimo. È interessante capire anche il significato di B_i: vettore dei valori di Bias dello strato i-esimo, spiegato bene in [**6**]. Il bias va inteso come una soglia (*threshold*) che determina se l'informazione elaborata dal neurone possa o meno procedere in avanti per poi trasformarsi in input nel livello successivo.

Livelli Convoluzionali

Ogni livello convoluzionale ha lo scopo di estrarre, dall'immagine, le caratteristiche principali per passarle allo strato successivo. Questo avviene attraverso un Kernel, filtro quadrato, generalmente 3x3 [**2**], che applica il prodotto scalare tra i suoi pesi e i valori dei pixel della porzione d'immagine sulla quale viene posizionato (che prende il nome di campo recettivo, figura 7). Il risultato diventa poi l'input per il neurone dello strato seguente [**2**]. Successivamente, il Kernel si sposta di un passo (n pixel) ricominciando l'operazione con i pixel della nuova porzione d'immagine sulla quale si è mosso. E così via scorrendo tra destra e sinistra e dall'alto verso il basso.

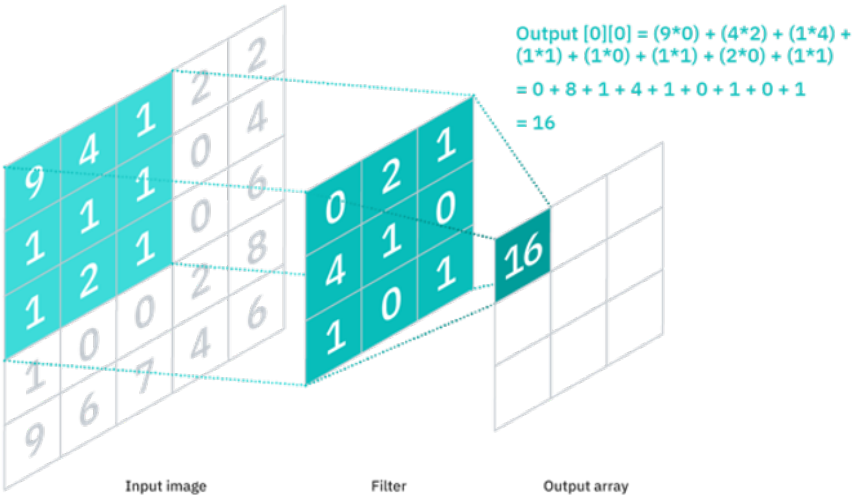

Figura 7: Esempio di Filtro applicato ad una regione di pixel [**2**]

L'output finale dell'intero livello, che entra totalmente nello strato successivo, è detto mappa delle caratteristiche. Man mano che i dati dell'immagine avanzano nella CNN, essa inizia a riconoscere elementi o forme dell'oggetto fino a identificare finalmente le *features* previste.

Nella pratica possono esserci più filtri: ognuno per una caratteristica da rilevare. Più è alto il risultato della convoluzione e più è probabile che quella caratteristica sia stata trovata nel campo recettivo. Quindi la mappa totale delle caratteristiche uscente da un determinato strato ha una dimensione pari al valore espresso in equazione 4:

$$D = F * Canali(RGB) * NumeroFiltri \tag{4}$$

Dove F sta per il numero di volte che un Kernel è applicato, scorrendo, sull'immagine. A patto poi che i filtri abbiano tutti la stessa dimensione (nxn).

Livelli di Pooling

Come concordato tra [**1**] e [**2**], dopo una convoluzione, è sempre bene proporre uno strato di *Pooling*. Qui si riducono i dati e si perde informazione per limitare la complessità della rete, la memoria e la potenza di calcolo richiesta. Tale operazione è possibile grazie ad un Kernel vuoto che si poggia su una fetta di input (output dello strato precedente) per ridurne i dati, applicando un *pooling* massimo o medio. Inoltre, può essere globale (su tutti i dati) o locale (solo a zone) [**1**].

Massimo: *pooling* che sceglie il massimo tra i valori che raggruppa (scartando il resto)

Medio: *pooling* che raggruppa più valori in uno solo, uguale alla loro media

Un processo del genere è anche noto come *down-sampling*, con il quale non solo si riescono a classificare meglio gli elementi nell'immagine, ma si fa in modo che il filtro dello strato successivo abbia una visione su una zona più ampia dell'immagine di input (fare riferimento a figura 7).

Strati completamente connessi

Nelle CNN, la catena procede così alternandosi tra livelli di convoluzione e di *Pooling*. Poi di solito, alla fine di essa, si pone uno strato completamente connesso che, di base, consiste in un livello in cui tutti i neuroni sono collegati a qualsiasi nodo dei livelli adiacenti. Fare questo all'ultimo strato di una CNN vuole dire rielaborare insieme le caratteristiche e far svolgere alla rete neurale l'operazione per la quale è pensata [**1**]. Otteniamo quindi la classificazione finale.

Al di là dei tipi di livello descritti finora, in una CNN è importante conoscere alcuni termini e parametri come: *overfitting, depth, stride e zero-padding.*

Overfitting: per le ANN tradizionali, elaborare immagini di una certa dimensione, richiederebbe un numero elevato di neuroni e livelli. Questo aumenterebbe la complessità computazionale richiesta portando la rete all'*overfitting*. In una situazione del genere, una rete neurale mostra riduzione di capacità nell'individuare le funzionalità [**2**]. Per quanto riguarda una CNN invece, i metodi sono stati sviluppati in modo da ridurre notevolmente i parametri dello strato e, quindi, la complessità. Per tale ragione le reti di questo tipo sono più adatta ad elaborare immagini: ad esempio, con una bassa risoluzione a 3 canali RGB, ogni nodo di un determinato strato potrebbe essere soggetto a centinaia di pesi da gestire, contro le migliaia che invece si conterebbero in una comune ANN.

La profondità (depth) del volume di output è data dal numero di filtri in uno strato che si connettono alla stessa regione dei valori di input. Ad esempio, si dice in [**2**], se il primo strato convoluzionale prende l'immagine originale come input, allora diversi filtri possono attivarsi in presenza di varie caratteristiche interne. Ognuno di loro genera un proprio output e, mettendoli insieme, si avrà un output generale più profondo. Così si produce un numero di mappe pari a quanti sono i filtri.

Stride (passo): numero di pixel di cui si sposta il Kernel dopo ogni prodotto scalare. Un piccolo passo si traduce in campi recettivi sovrapposti e grande dimensione di output. Al contrario, un grande passo limita la sovrapposizione diminuendo la dimensione del risultato a fine strato.

Zero-padding: solitamente utilizzato quando i filtri non si adattano all'immagine di input. Vengono impostati a zero tutti gli elementi che non rientrano nella matrice di input, producendo un output di dimensioni maggiori o uguali rispetto all'input [**2**].

Per le CNN non ci sono architetture prestabilite ma esistono modelli comuni. Come in figura 8:

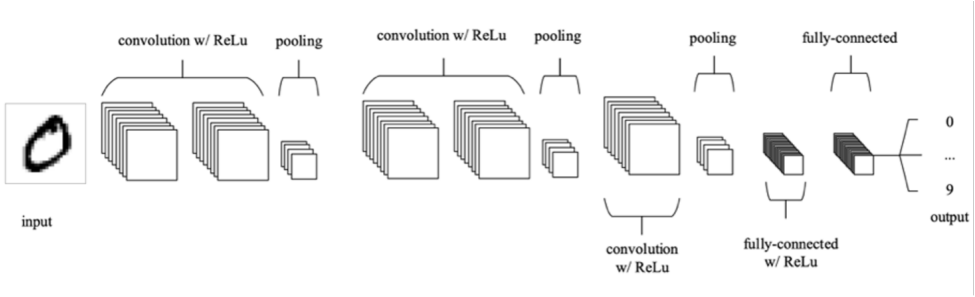

Figura 8: Tipica architettura di una CNN – [**1**]

In conclusione

Come descritto in [**1**], le reti neurali convoluzionali differiscono quindi dalle altre forme di reti neurali in quanto, invece di concentrarsi sull'intero dominio del problema, sfruttano la conoscenza delle caratteristiche specifiche dell'input. Questo consente una architettura più semplice da configurare. È consigliato inoltre, sempre in [**1**], dividere grandi strati convoluzionali in vari strati di dimensioni più piccole (come vedremo nel caso della VRCNN [**4**]), per ridurre la quantità di calcolo computazionale all'interno di essi. È preferibile anche usare Kernel di dimensioni non molto grandi e di farli spostare poi con un passo pari ad 1. Infine, le applicazioni delle CNN sono delle più varie: dalla robotica alla medicina, dalla sicurezza alla segmentazione di immagini...

Capitolo 1

CNN nei moderni codificatori Video

I codec video sono particolari algoritmi che servono a codificare dati e a comprimerli per ridurre le informazioni ridondanti dei frames, abbassando quindi il peso dei singoli file a discapito, chiaramente, della qualità video. In generale, comunque, questi algoritmi funzionano molto bene e sono ampiamente utilizzati: la qualità persa è di solito limitata tanto da non risultare evidente all'occhio umano. Conosciamo AVC (usato per lo più in internet e per risoluzione FullHD), HEVC, EVC e, di ultima generazione, nominiamo VVC. Tuttavia, i codec elaborano i frames dividendoli in porzioni rettangolari codificate singolarmente. I cosiddetti "blocchi". Ciò può generare artefatti di compressione specialmente vicino ai bordi delle suddivisioni, proprio dove avviene il salto tra due diverse codifiche e queste differenze possono risultare evidenti. Inoltre, la quantizzazione scarta le alte frequenze non necessarie dell'immagine. Gli artefatti si traducono così in errori e problemi come discontinuità, sfocature ecc... Insomma, perdite di qualità. Ora, già all'interno dei codici dei codificatori esistono sezioni dedicate a rimuovere questi errori con dei filtri, come il DBF (filtro di *deblocking*) e il SAO in HEVC (per problemi

di *banding*). Ma, nonostante ciò, le prestazioni non sono ottimali e gli artefatti sono spesso ancora visibili. Allora, come succede in molti campi della tecnologia, ultimamente anche nei codec si sta facendo ricorso all'uso di reti neurali per migliorarne i risultati. In particolare, numerose volte, proprio le CNN sono state introdotte all'interno dei codificatori come ad esempio una AR-CNN, per ridurre ulteriormente gli artefatti, e una SRCNN che è invece pensata come sostituto del DBF e del SAO. Tra le due, la prima sembrerebbe essere più performante [**5**]. Adesso analizziamo alcuni casi d'esempio.

1.1 Approccio VRCNN

La maggior parte degli approcci provati in questo ambito, tratta di una rete addestrata con la sequenza stessa da testare e, quindi, una rete non generalizzabile. Come descritto in [**4**], dal 2016, Yuanying Dai, Dong Liu e Feng Wu (University of Science and Technology of China) propongono, invece, una CNN addestrata con immagini naturali e testata con sequenze standard HEVC per certificarne la generalizzabilità. Questa rete, chiamata VRCNN, sostituisce DBF e SAO in HEVC e ha lo scopo di ridurre gli artefatti. Può essere considerata un algoritmo di post-elaborazione [**5**] e le sue prestazioni sono maggiori rispetto alle CNN sopra citate perché, la VRCNN, fa uso di filtri a dimensione variabile. Si articola in 4 strati:

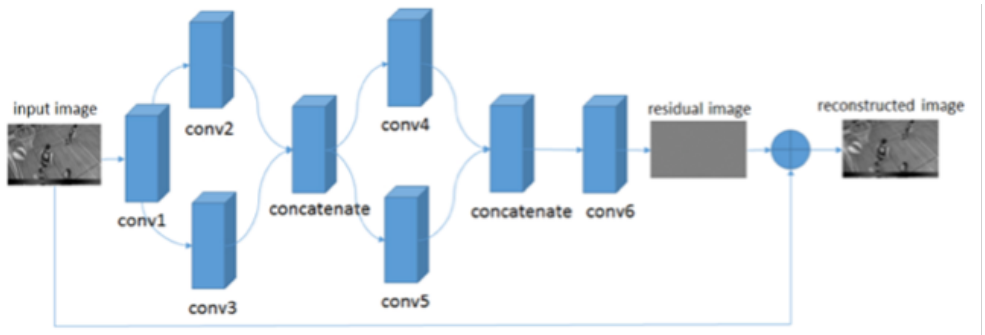

Figura 1.1: Architettura VRCNN – [**4**]

Layer	Layer 1	Layer 2		Layer 3		Layer 4
Conv. module	conv1	conv2	conv3	conv4	conv5	conv6
Filter size	5×5	5×5	3×3	3×3	1×1	3×3
# filters	64	16	32	16	32	1
# parameters	1600	25600	18432	6912	1536	432
Total parameters	54512					

Figura 1.2: Tabella dati VRCNN – [**4**]

Nel secondo *layer*, questa rete utilizza due sotto-strati con filtri a dimensione variabile [**4**], rispettivamente di dimensione 5x5 e 3x3. Lo stesso accade nel terzo strato con filtri 3x3 e 1x1. Un approccio del genere pulisce il rumore generato in codifica. Nel frattempo, i livelli 1 e 4 non adottano questo meccanismo in quanto in essi si compiono rispettivamente l'estrazione delle caratteristiche del frame e la ricostruzione finale. È molto probabile che VRCNN sia la prima rete ad utilizzare filtri a dimensione variabile per combattere gli artefatti di codifica.

In [**4**] è evidenziato che la rete deve i suoi buoni risultati anche soprattutto all'approccio basato sull'apprendimento dei residui. Descritto in equazione 1.1:

$$F(Y) = W4F3(Y) + B4 + Y \tag{1.1}$$

Di base, l'input viene sommato all'output dell'ultimo strato. Così, la rete apprende a misurare la differenza tra input ed output filtrato, cercando poi di minimizzarla in quanto è sostanzialmente un valore di errore. Inoltre, la minimizzazione dovrebbe risultare poi più facile perché abbiamo una differenza tra due grandezze generalmente simili o comunque confrontabili.

La tecnica dell'apprendimento dei residui si traduce in una minore complessità nella VRCNN e quindi un numero minore di parametri rispetto

ad altri approcci. In [4] VRCNN viene confrontata con un'altra rete chiamata AR-CNN. Grazie all'apprendimento residuo e alla sua semplicità, come dicevamo, la VRCNN impiega molto meno tempo della AR-CNN per addestrarsi.

Layer	Layer 1	Layer 2	Layer 3	Layer 4
Filter size	9×9	7×7	1×1	5×5
# filters	64	32	16	1
# parameters	5184	100352	512	400
Total parameters	106448			

Figura 1.3: Tabella dati AR-CNN – [4]

Training VRCNN

Yuanying Dai ed il suo team hanno allenato la rete con una serie di 400 immagini naturali di cui venne preso in considerazione solo il canale Y (luminanza). Immagini poi codificate ma senza aver apportato alcuna operazione di miglioramento (quindi con DBF e SAO disattivati).

L'allenamento è stato avviato e gestito dal software dedicato "Caffe" [4] che prende in input il set di immagini cercando di migliorarne la qualità. Durante questa fase, un algoritmo di minimizzazione viene sfruttato per ridurre l'errore della rete. Il tutto ripetuto 4 volte a 4 valori QP diversi (22, 27, 32, 37) contando fino a 160 epoche di training.

Una volta fatto, la rete è stata integrata in HEVC al posto dei filtri *in-loop* per il test validativo. Esso è stato eseguito su 20 sequenze video HEVC standard, lavorando anche sui canali U e V, nonostante l'addestramento sia stato svolto esclusivamente con i dati di luminanza.

Class	Sequence	BD-rate		
		Y (%)	U (%)	V (%)
Class A	Traffic	-5.6	-3.5	-4.1
	PeopleOnStreet	-5.4	-5.9	-5.7
	Nebuta	-0.9	-4.9	-4.1
	SteamLocomotive	-1.9	-0.5	-0.3
Class B	Kimono	-2.5	-1.5	-1.4
	ParkScene	-4.4	-3.3	-2.5
	Cactus	-4.6	-3.9	-6.3
	BasketballDrive	-2.5	-3.7	-5.3
	BQTerrace	-2.6	-3.3	-3.0
Class C	BasketballDrill	-6.9	-5.8	-6.8
	BQMall	-5.1	-5.3	-5.3
	PartyScene	-3.6	-4.4	-4.4
	RaceHorses	-4.2	-6.7	-11.0
Class D	BasketballPass	-5.3	-4.4	-6.5
	BQSquare	-3.8	-4.2	-6.4
	BlowingBubbles	-4.9	-8.4	-7.9
	RaceHorses	-7.6	-8.5	-11.5
Class E	FourPeople	-7.0	-5.3	-5.2
	Johnny	-5.9	-5.0	-5.5
	KristenAndSara	-6.7	-6.1	-6.2
Class Summary	Class A	-3.5	-3.7	-3.6
	Class B	-3.3	-3.2	-3.7
	Class C	-5.0	-5.5	-6.9
	Class D	-5.4	-6.4	-8.1
	Class E	-6.5	-5.5	-5.6
Overall	**All**	**-4.6**	**-4.7**	**-5.5**

Figura 1.4: Risultati test VRCNN – [**4**]

La tabella parla chiaro: in [**4**] è dimostrato come VRCNN risparmi bit-rate (scala BD) sia sul canale Y (in media -4,6%), che sulla crominanza (fino a -11,5% per V e -4,7% di media per U). La rete vince anche in termini di PSNR contro AR-CNN e l'HEVC *baseline*. Questo sarebbe in realtà già visibile ad occhio nudo ed è raggiungibile, in particolare, grazie ai filtri di dimensione variabile. Gli autori di [**4**] ci tengono a precisare che le sequenze di addestramento sono diverse da quelle dei test e ciò rende la rete più generalizzata e capace di raggiungere ottimi risultati anche con altre diverse sequenze o immagini.

Complessità computazionale

Gli esperimenti mostrano come, rispetto a AR-CNN, la rete di cui parliamo sia leggermente più lenta a causa del fatto che gli strati 2 e 3 sono divisi in sotto-strati per poter fare uso di filtri a diverse dimensioni [**4**]. Essi causano problemi per il calcolo parallelo. E comunque, i risultati sui valori di tempo sembrano non essere soddisfacenti per i requisiti dei moderni PC. Quindi c'è ancora molto lavoro da svolgere per ottimizzare e velocizzare le CNN.

Ricapitolando, infine, rispetto al profilo base di HEVC, VRCNN risparmia in media il 4,6% di BD-rate (per la luminanza) sulle sequenze di test standard. Il PSNR è maggiore, ma queste reti impiegano ancora diverso tempo nello svolgere le loro operazioni. Aggiungiamo che ciò non risulterebbe un grande problema per eventuali codifiche offline (come per il Video-On-Demand), ma di certo le tempistiche descritte non rendono, per ora, VRCNN del tutto adeguata a codifiche live.

Di recente, Yuanying Dai, Dong Liu e Feng Wu hanno lavorato per rendere la rete più semplice e per estendere il suo operato anche all'*inter*-codifica di HEVC.

1.2 Approccio RHCNN

Fino ad ora, dopo aver chiarito i principi fondamentali delle CNN, abbiamo accennato ad approcci quali VRCNN e AR-CNN che, tuttavia, sono applicabili solo ad immagini *jpeg* o frames compressi in modalità *intra* (frames I), trascurando i casi di previsione tra frames. C'è bisogno di un metodo che migliori le prestazioni anche per frames di tipo P e B. In tal merito, Yongbing Zhang, Tao Shen, Xiangyang Ji, Yun Zhang, Ruiqin Xiong e Qionghai Dai mostrano in [**5**], nel 2018 sotto IEEE, una rete neurale RHCNN da loro addestrata, composta da diverse unità "*Highway*" a cascata con strati convoluzionali per migliorare la qualità dei frames *inter*, riducendo il bit-rate. In [**5**] gli autori sostengono che la RHCNN sia il primo caso di apprendimento residuo nelle reti ad unità *Highway* per risolvere il problema del filtraggio *in-loop* nei codec video. Per la sua realizzazione hanno sfruttato strumenti open source, per poi riscrivere la rete in C++ [**5**]. Questo gli ha permesso allora di poter inserire RHCNN dentro HEVC, senza entrare in conflitto con filtri DBF e SAO, bensì identificandosi come filtro successivo ad essi. Vediamone il funzionamento.

Unità Highway

Le suddette unità si articolano in strati convoluzionali 3x3 con un ramo "scorciatoia" in parallelo. In figura 1.5 è mostrata la tipica configurazione che queste unità assumono nella RHCNN. Notiamo degli strati convoluzionali con Kernel 3x3 ed il ramo "scorciatoia" che somma l'input all'output finale. Il gruppo che ci ha lavorato ha addestrato le unità per ricostruire le perdite nelle immagini codificate in HEVC con un QP pari a 40. Una volta addestrate e testate, sono messe in cascata su più livelli (13) per formare la rete RHCNN (quindi l'output di un'unità diventa l'input di quella successiva). Come dichiarato in [**5**], l'architettura *Highway* di figura 1.5 è la migliore possibile in mezzo ad altre configurazioni che, ad esempio, presentano ulteriori parametri o fattori di scala.

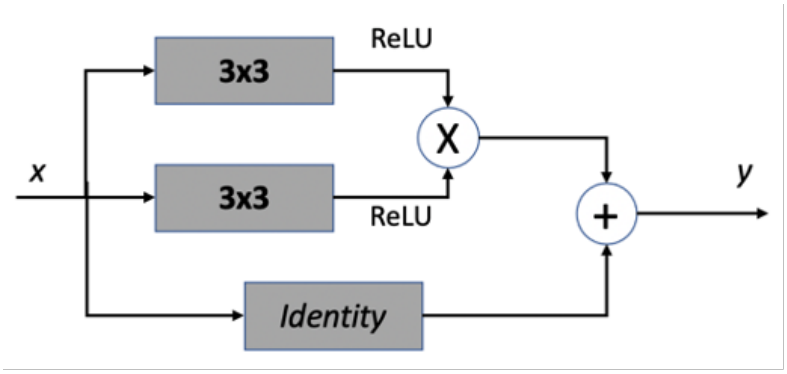

Figura 1.5: Unità Highway

Scorciatoia

La scorciatoia è fondamentale per raggiungere buone prestazioni. In questo caso non è nient'altro che un collegamento di "identità" che semplicemente riporta l'input alla fine dell'unità, per poi sommarlo all'output finale. Di fatto, queste scorciatoie non aggiungono parametri e complessità alla rete, però il ramo può essere percorso a ritroso, favorendo così il processo di apprendimento dei residui [**5**] (come in [**4**]). Inoltre, viene usato non solo nelle singole unità ma anche a livello globale portando l'immagine pulita direttamente alla fine dell'architettura generale.

Addestramento

Stando all'articolo del 2018, più RHCNN sono state addestrate con frames estratti da 15 sequenze video standard. Per il test i ricercatori usarono invece 11 sequenze, diverse da quelle sfruttate per l'addestramento, dalle quali estrapolarono solo 50 frames ciascuna. La procedura è sempre la stessa: addestrare la rete per ridurre l'errore quadratico medio, e quindi la differenza, tra il frame originale e quello codificato. Entrambi vengono mandati in input e confrontati dalla rete. In questo modo, con l'opportuno algoritmo di minimizzazione, si fa sì che in output ci

sia un'immagine che assomigli il più possibile al frame originale. In [**5**] è chiarito che tutto ciò è stato ripetuto a diverse bande di QP e per entrambi i tipi di frames P e B.

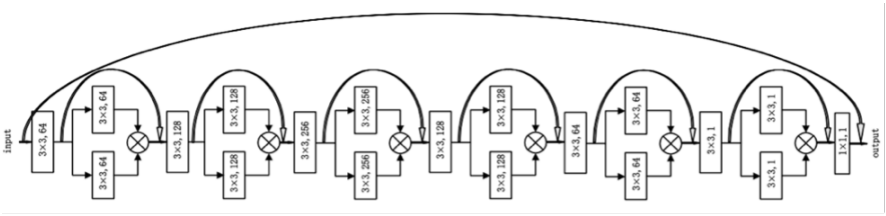

Figura 1.6: Architettura RHCNN – [**5**]

Nello specifico si parla di addestramento progressivo: i pesi di un addestramento a determinata banda QP vengono riciclati come pesi iniziali per l'addestramento con banda QP successiva. Successivamente la rete viene fatta lavorare e se ne valutano le prestazioni. Nel confronto con altri approcci, trattandosi di reti neurali che emulano il lavoro dei filtri post codifica, si intende implicitamente di inserire questi sistemi all'interno di un flusso di codifica HEVC (HM-12).

Prestazioni

Gli autori hanno lavorato nel mettere a confronto le prestazioni della RHCNN con altre due reti: una convoluzionale piana ("normale") e una che fa uso dell'apprendimento dei residui. Queste reti sono state fatte lavorare sulle operazioni per le quali sono addestrate, prendendo in input immagini di test. Alla fine, calcolando il PSNR dell'output, la RHCNN vince su tutte [**5**]:

Networks	Plain Network	Residual Network	RHCNN
Average	0.23	0.27	**0.32**
Maximum	0.28	0.33	**0.39**

Figura 1.7: Confronto PSNR (dB) sui 3 tipi di reti neurali – [**5**]

I risultati sperimentali proposti in [**5**], in realtà, dimostrano che il metodo studiato è in grado non solo di aumentare il PSNR del frame ricostruito, ma riduce anche il bit-rate. Infatti, il team di lavoro riporta uno studio della sopra citata AR-CNN confrontandola con la loro RHCNN. Essendo poco profonda, la AR-CNN ha prestazioni limitate e, seppur migliorando la qualità, rischia di aumentare il bit-rate di codifica rispetto all'approccio standard. Invece la RHCNN è più profonda e, grazie alle unità *Highway*, mostra ottimi risultati sia per PSNR che per riduzione bit-rate in output.

L'approccio HM-12+RHCNN viene quindi descritto come superiore rispetto a HM-12+AR-CNN e rispetto anche al solo codificatore standard HEVC (HM-12 con soli DBF e SAO). Questo risultato vale in tutte le configurazioni di codifica e sembrerebbe più evidente all'aumentare del QP [**5**].

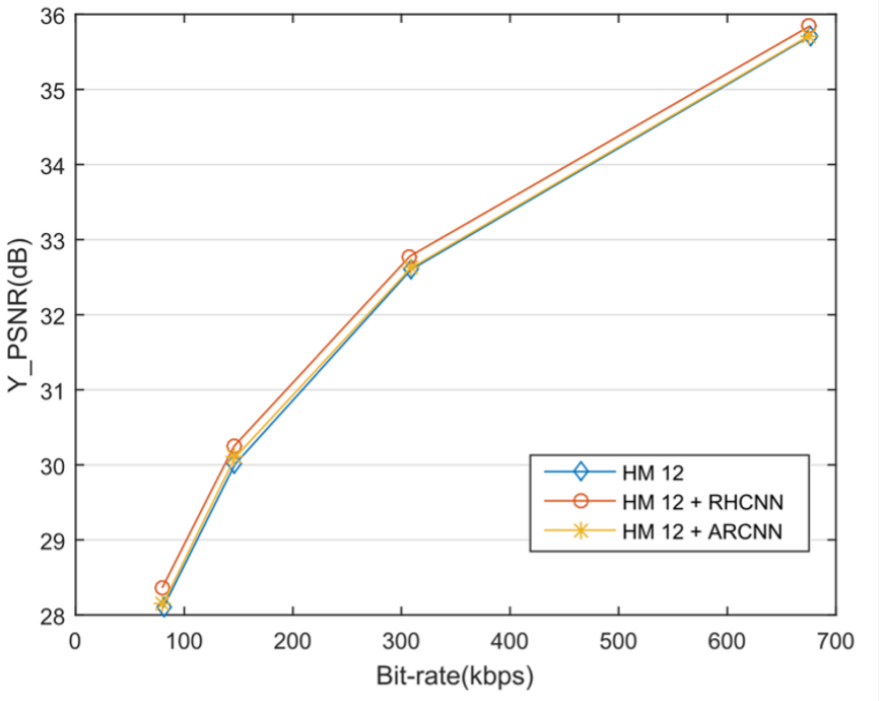

Figura 1.8: Grafico RD – [**5**]

Complessità e conclusioni

Chiaramente questa RHCNN deve risultare complessa data la profondità (13 livelli) e la sua struttura in unità. I tempi di calcolo misurati in [**5**], con sistema HEVC+RHCNN, mostrano infatti una maggiore lentezza rispetto all'HM-12 standard. Tuttavia, se si parallelizza il calcolo con una GPU, il tempo di codifica viene ridotto ad un valore sopportabile, anche se comunque più elevato dell'approccio classico. La rete è tutt'ora in fase di miglioramento per renderla ancora più veloce e sensibile ad aspetti particolari del video come movimenti di oggetti, complessità delle texture e altro ancora [**5**]. Ciò nonostante, nell'articolo è ampiamente ribadito e dimostrato come la RHCNN abbia ottime prestazioni, migliori del codificatore classico, di altre reti come la AR-CNN e anche di filtri standard del mondo delle codifiche, come l'ALF di HEVC.

Riassumendo, RHCNN è caratterizzata da unità *Highway* concatenate su 13 livelli, addestrate in maniera progressiva per recuperare le perdite accumulate in codifica. Si tratta di una rete neurale progettata per lavorare come filtro *in-loop* in aggiunta a filtri classici quali DBF e SAO. La sua risposta è ottima per PSNR e bit-rate, migliore di altri metodi basati su apprendimento e ha come unico *trade-off* un tempo di esecuzione maggiore. Tuttavia, così come i vari approcci descritti finora, si tratta di una rete che lavora su ogni frame singolarmente, senza far riferimento a quelli passati come possibili predittori. Risulterà necessario, allora, testare anche un approccio del genere che verrà propriamente chiamato "multi-frame".

1.3 Approccio MIF-Net

Come anticipato nella sezione precedente, nel nostro trattato abbiamo per ora scritto riguardo sistemi che, con CNN più o meno performanti, aiutano a recuperare la qualità del frame dopo la codifica senza però sfruttare i frames adiacenti come predittori. Abbiamo per ora parlato quindi di approcci a frame singolo. Da [**7**], invece, riportiamo lo studio di reti neurali per filtro *in-loop* multi-frame. Si tratta della rete MIF, che si riduce a IF nel caso in cui non vi siano abbastanza predittori.

Vale la pena ricordare che l'intero discorso nasce dalla presenza di "blocchi" ed artefatti di post-codifica che si vuole ridurre con filtri *in-loop*, i quali a loro volta posso essere implementati e migliorati con reti neurali (CNN). In confronto alle CNN descritte sopra, ci si aspetta che l'approccio multi-frame (che sfrutta i dati dei frames precedenti) dia vita ad una rete con risultati superiori. In [**7**] ne viene fornita un'analisi esaustiva, ispirandosi a lavori preesistenti nello stesso campo.

Idea: approccio multi-frame

Sappiamo che gli stessi elementi di un video possono essere presenti in più fotogrammi. Ne deriva che i frames di bassa qualità possono essere migliorati sfruttando gli stessi dati dai frames vicini, con qualità superiore. Se queste immagini migliori, dopo la codifica e l'uso del filtro, le tenessimo da parte, potremmo utilizzarle come riferimento quando dovrò poi migliorare un frame successivo. Come ribadito in [**7**], un tale metodo è vantaggioso rispetto ad un algoritmo che, senza aiuti esterni, prende il fotogramma e cerca di filtrarlo solo sulla base del frame stesso e di come l'algoritmo è stato addestrato.

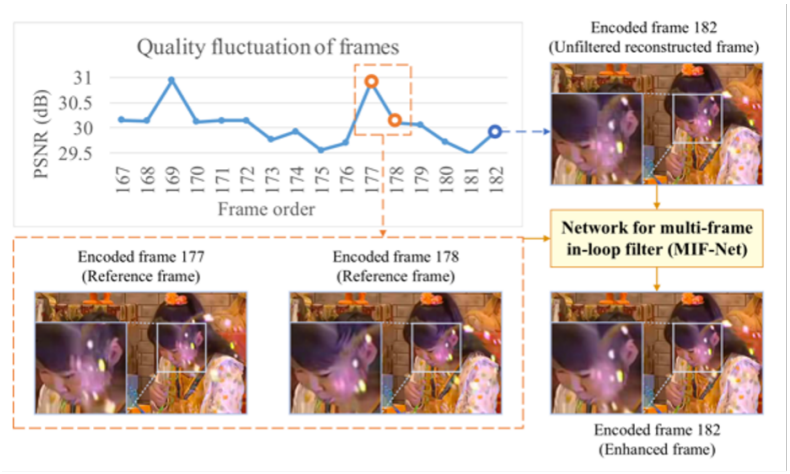

Figura 1.9: Schema generale – [**7**]

Ciò che dovrebbe succedere, spiegano Tianyi Li, Mai Xu, Ce Zhu ed altri in [**7**], pubblicato da IEEE nel 2019, è l'attivazione di un selettore che, "spulciando" tra i frames precedenti, individui quelli migliori e maggiormente simili al frame corrente per usarli come *references*. Tutte le immagini entrerebbero poi nella rete neurale (MIF-Net) allenata per aumentare il PSNR del frame attuale, riducendo il bit-rate generale rispetto ad un approccio standard. Qualora non vi siano abbastanza predittori, si attiverebbe la classica rete a singolo frame, qui chiamata IF, che lavora sfruttando soltanto l'immagine corrente.

RFS

I dati riportati nelle prime sezioni di [**7**] sono interessanti: la deviazione standard del PSNR tra frames codificati di una singola sequenza risulta spesso notevole, comunque più alta del *gap* di qualità aggiunto da filtri DBF e SAO. Ha quindi senso procedere con un sistema multi-frame (MIF). Questa rete neurale è stata implementata come filtro *in-loop* nel codificatore HEVC, con il quale sono state fatte le codifiche. Man mano che una sequenza viene elaborata, si spiega in [**7**], i frames migliorati dal

filtro vengono messi in un database che contiene i riferimenti, i dati sulle partizioni CU e TU (divisione a blocchi), il frame corrente codificato ma non ancora filtrato (URF) ed il suo rispettivo frame *raw*.

A questo punto, gli autori dell'articolo fanno lavorare un semplice selettore (RFS) che, sulla base di 6 metriche, analizza i frames del database decidendo quali tra questi possono essere sfruttati come predittori. Dato che si necessitano riferimenti ad alta qualità e simili al frame corrente, viene da sé che le 6 metriche di decisione si articolano in PSNR e CC (coefficiente di correlazione) nelle 3 componenti YUV. Sono considerate valide tutte quelle immagini che per almeno una componente soddisfano un PSNR minimo e un CC sufficiente rispetto al fotogramma che si vuole migliorare [**7**].

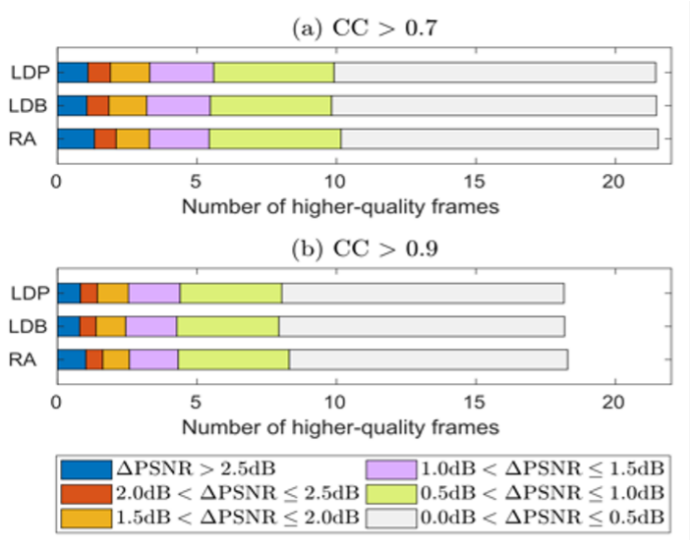

Figura 1.10: Esempio frames validi – [**7**]

In figura 1.10, ad esempio, si capisce come, imponendo ragionevoli e sensate soglie di PSNR e CC, si riesca ad avere un numero di possibili predittori, per ogni URF, che sia maggiore di 7. Lo schema completo del selettore è riportato in figura 1.11:

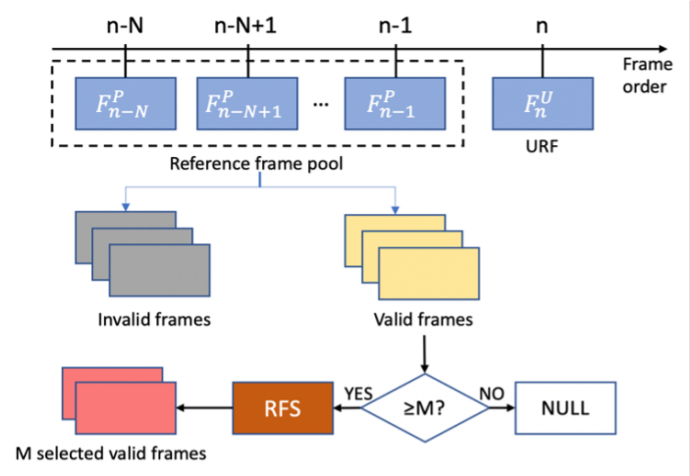

Figura 1.11: Schema RFS

MIF-IF

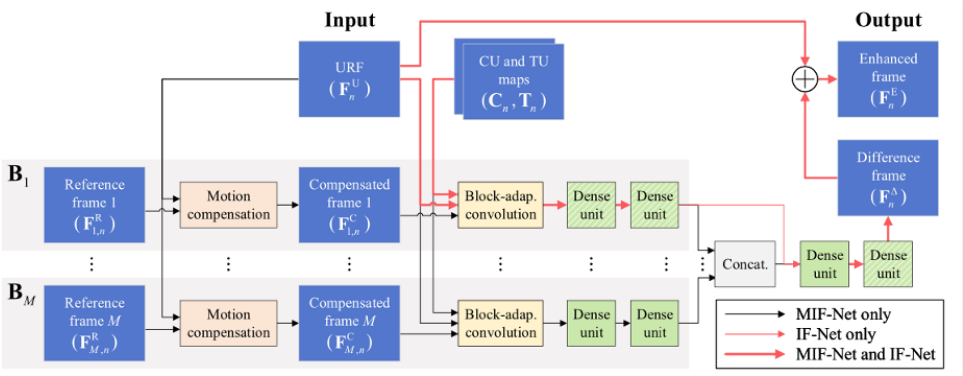

Figura 1.12: Funzionamento MIF-IF – [7]

La struttura della rete neurale MIF è riportata in figura qui sopra, estrapolata da [7]. In input, disposti su più rami paralleli, abbiamo i predittori scelti dal RFS ed il frame corrente URF. Dopo una compensazione del moto volta ad allineare tutte le immagini fra loro, per ogni ramo, i dati passano attraverso dei blocchi adattativi seguiti da Unità Dense.

Block-adaptive convolutional layers: livelli convoluzionali della rete che si adattano alla struttura a blocchi del frame, ricevendo in input anche le informazioni sulle partizioni CU e TU. Qui vengono estratte le mappe delle caratteristiche del fotogramma.

Unità dense: in questa fase avviene il calcolo dei valori da applicare al frame per migliorarlo. Ogni unità ha quattro livelli convoluzionali [**7**].

A seguito di una serie di Unità Dense concatenate poi fra loro, viene prodotto l'output che consiste nel *delta* di qualità in più da sommare all'URF. Questa addizione si svolge quindi nello step finale, completando il lavoro del filtro *in-loop* (figura 1.12).

IF: come detto, nel caso in cui il selettore non trovi abbastanza fotogrammi a cui far svolgere il compito di predittore e quindi non sono rispettate le specifiche su valori minimi di PSNR e CC, la rete neurale riceve in input solamente il frame codificato attuale che segue il percorso individuato dalle linee rosse di figura 1.12. Come se ci fosse un solo ramo del MIF. In questo caso la rete prende il nome di IF e di conseguenza non c'è compensazione del moto (approccio single-frame).

Addestramento

La rete completa proposta in [**7**] fu addestrata in due fasi usando 120 sequenze di training e 40 di convalida. Inizialmente i ricercatori hanno lavorato per insegnare alla rete a ridurre la differenza tra frames compensati e URF. Fase fondamentale per svolgere una corretta compensazione del moto. Poi sono stati mostrati alla rete, in input, i fotogrammi originali e quelli ricostruiti, così da poterne fare un confronto e capire come migliorare l'URF. L'obiettivo, alla fine, rimane comunque quello di ridurre la differenza tra immagine *raw* e output finale. Anche qui, ridurre vuol dire sfruttare un algoritmo di minimizzazione che, in questo caso, risulta essere l'algoritmo di Adam [**7**].

Sottolineiamo che nell'articolo è specificato che solo il canale Y delle sequenze fu utilizzato per l'addestramento di MIF e IF. Inoltre, a proposito di IF, la sua fase di training risulta chiaramente più semplice, in quanto così di natura appare la rete e anche perché non c'è compensazione del moto. I parametri di addestramento vennero poi presi direttamente dal MIF, senza partire da 0. Successivamente, dopo addestramento e convalida, si è proceduti con la fase di test. Le valutazioni riportate in [**7**] si basano su scale BD-rate e BD-PSNR, dove si va ad indicare con un "-" la percentuale di guadagno di bit-rate e, in valore assoluto, il miglioramento del PSNR in dB.

Prestazioni

Class	Sequence	DBF and SAO		[10]		[20]		Proposed MIF	
		BD-BR (%)	BD-PSNR (dB)	BD-BR (%)	BD-PSNR (dB)	BD-BR (%)	BD-PSNR (dB)	BD-BR (%)	BD-PSNR (dB)
A	*PeopleOnStreet*	-8.287	0.368	-12.025	0.540	-12.484	0.568	**-16.824**	**0.777**
	Traffic	-5.348	0.162	-6.169	0.188	-9.816	0.304	**-12.152**	**0.383**
B	*BasketballDrive*	-6.655	0.148	-8.838	0.198	-11.053	0.250	**-14.870**	**0.347**
	BQTerrace	-7.149	0.111	-11.397	0.173	-14.365	0.228	**-17.126**	**0.271**
	Cactus	-7.540	0.157	-8.904	0.189	-12.518	0.272	**-15.829**	**0.349**
	Kimono	-7.536	0.220	-9.261	0.273	-10.482	0.311	**-12.235**	**0.368**
	ParkScene	-3.679	0.112	-4.083	0.124	-5.940	0.182	**-7.994**	**0.249**
C	*BasketballDrill*	-5.017	0.206	-5.393	0.222	-7.818	0.326	**-10.324**	**0.434**
	BQMall	-3.933	0.150	-4.451	0.170	-7.663	0.296	**-9.376**	**0.367**
	PartyScene	-1.054	0.044	-1.224	0.051	-2.417	0.100	**-4.159**	**0.173**
	RaceHorses	-6.154	0.222	-7.082	0.257	-10.403	0.386	**-12.736**	**0.476**
D	*BasketballPass*	-3.852	0.182	-4.324	0.204	-7.700	0.370	**-9.984**	**0.484**
	BlowingBubbles	-0.829	0.034	-0.831	0.034	-3.072	0.126	**-3.980**	**0.164**
	BQSquare	-0.053	0.002	0.010	-0.000	-3.263	0.123	**-4.401**	**0.165**
	RaceHorses	-4.441	0.199	-4.797	0.215	-8.860	0.407	**-10.992**	**0.510**
E	*FourPeople*	-7.018	0.262	-8.489	0.319	-13.937	0.538	**-16.480**	**0.644**
	Johnny	-5.599	0.143	-8.034	0.208	-11.649	0.302	**-14.370**	**0.378**
	KristenAndSara	-6.410	0.203	-8.011	0.254	-12.637	0.406	**-15.340**	**0.501**
Average		-5.031	0.162	-6.295	0.201	-9.227	0.305	**-11.621**	**0.391**

Figura 1.13: Tabella test standard – [**7**]

Nella tabella sono riportati i risultati dei test. Queste prove vennero eseguite in configurazione RA e consistono nel far lavorare il MIF-Net al posto del DBF di HEVC. Appare chiaro, almeno per quanto dichiarato in [**7**], che MIF guadagni abbastanza bit-rate su tutti gli altri approcci di confronto, tra i quali quello standard (DBF e SAO) e poi uno euristico e

uno basato anch'esso sull'apprendimento. Di media, dopo aver analizzato più sequenze, il risultato si riassume in un 11,6% di bit-rate ridotto dal MIF rispetto al codificatore senza filtro *in-loo*p. Una percentuale più alta rispetto a quelle raggiunte con gli altri strumenti.

Concentriamoci ora sulla qualità. Il PSNR sembra essere maggiore nei frames migliorati dal MIF che in quelli di altri metodi. Dalla tabella si evince che, sulle varie sequenze, il miglioramento medio si quantifica in 0,39 dB in più. Risultato più alto tra gli approcci proposti. Insomma, la rete neurale che prende il nome di MIF sembra essere il metodo più efficace nel migliorare la qualità dei frames URF in fase di *in-loop filtering*. Il MIF continua a vincere anche quando le misure sono calcolate su scala SSIM e non PSNR [**7**]: il SSIM indica la qualità in maniera più soggettiva ed è talvolta più affidabile del PSNR. Allora, si può benissimo affermare oramai che tali prestazioni positive sono dovute all'approccio multi-frame. Infatti, gli altri metodi analizzati sono accomunati dal non basarsi su una tale idea e, come ci si aspettava, raggiungono risultati più bassi.

Conclusioni

Basicamente, come ormai sappiamo bene, la partizione in CTU (poi CU e TU) in HEVC, ha impatto sugli artefatti. Questi vanno ridotti e combattuti con filtri *in-loop*. I metodi di apprendimento possono allora aiutarci e sviluppare questi filtri con tecnologie superiori, raggiungendo risultati maggiori. Nel corso di queste pagine sono stati analizzati vari esempi, tra cui VRCNN e RHCNN. Entrambi metodi utili e portatori di buone prestazioni. Come abbiamo visto, in generale le reti neurali apportano sempre un miglioramento rispetto a HEVC classico. Poi invece, in questo ultimo esempio abbiamo studiato le reti MIF-IF ed il relativo approccio multi-frame. Questa caratteristica rende il MIF-Net superiore ad altri schemi esplorati.

Dopo la selezione dei migliori frames predittori, con una seguente dovuta compensazione del moto, gli artefatti sono gestiti dai blocchi convoluzionali adattativi, che svolgono un ruolo essenziale nel miglioramento dell'immagine. Inoltre, MIF è ancora meglio e prestante perché usa Unità Dense. La tabella in figura 1.13 è prova dei risultati di cui abbiamo trattato.

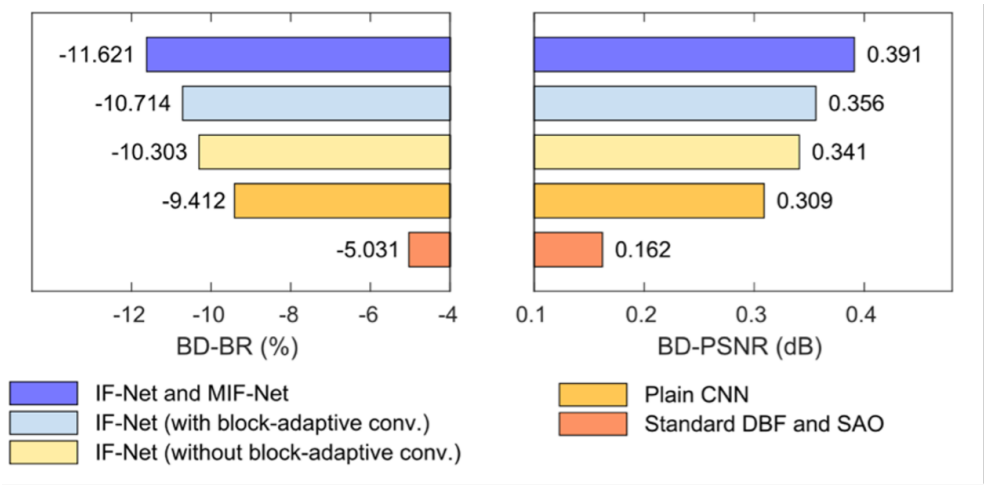

Figura 1.14: Confronto prestazioni vari metodi – [**7**]

Perciò, ricapitolando, le CNN possono essere migliori filtri *in-loop* rispetto ai classici DBF e SAO. Ancora meglio se poi introduciamo blocchi convoluzionali adattativi ed Unità Dense. Infine, i risultati più alti vengono raggiunti nel momento in cui si procede con un approccio multi-frame. Di conseguenza concludiamo affermando che, essendo IF una rete a frames singoli, le sue prestazioni saranno senz'altro ottimali ma inferiori al caso della rete MIF. Più adatte a frames come quelli *intra* [**7**].

Capitolo 2

HM-16.5_MIF-Net: testando la rete neurale

(Tirocinio in Rai-Radiotelevisione italiana)

2.1 Uno sguardo al codice

La presente tesi è frutto di quanto imparato, testato e studiato durante le ore di Tirocinio nel centro sperimentale di "Rai-Radiotelevisione Italiana". Come già visto, il gruppo di lavoro condotto da Tianyi Li, Mai Xu, Ce Zhu (e altri) si dedicò nel 2019 ad implementare un filtro che migliorasse la qualità dei frames codificati in HEVC, facendo uso, appunto, di rete neurale. Ciò che è stato spiegato nell'articolo "*A deep learning approach for multi-frame In-Loop filter of HEVC*" ([**7**]), è visibile in pratica nello script TEncGop.cpp di HEVC (HM-16.5) modificato e rinominato da loro come HM-16.5_MIF-Net (scritto in C++ e fornitomi direttamente dal Centro Sperimentale). L'obiettivo in Rai era quello di capire il funzionamento del MIF-Net, per poi sperimentare e testare le

sue prestazioni operando numerose codifiche. Infine, il lavoro consiste nel provare i primi passi per un *porting* dello stesso sistema verso altri codificatori, come ad esempio EVC. Analizzando le modifiche apportate al codice dagli autori, lo scopo fu dapprima quello di far funzionare il codec, poi di verificarne le prestazioni. Per sfruttare la CNN nel processo di codifica, gli autori hanno infatti messo mano ad alcune sezioni di codice commentate con "added": è stata allora studiata la parte di script che evocasse effettivamente l'utilizzo della CNN alla quale inviare i dati.

Ciò è fondamentale perché queste linee di codice potrebbero poi essere tradotte in altri linguaggi, come C, se si vuole provare a far fare lo stesso gioco ad altri codificatori, ad esempio EVC. Di seguito, sono mostrate le linee di nostro interesse:

```
1937    // 20180105 end
1938    static FILE * fpYuvEnhanced;
1939    static Int iQPFrame0 = pcPic->getSlice(0)->getSliceQp();
1940    if (bUseCNN)
1941    {
1942        // 20180110 added
1943        static FILE * fpParameter, *fpPredStart, *fpPredEnd;
1944
1945        remove(_FileName:"pred_end.sig");
1946
1947        static bool bIsInit = true;
1948
1949        fpParameter = fopen(_FileName:"command.dat", _Mode:"w+");
1950        fprintf(_Stream:fpParameter, _Format:"%d %d %d %d %d %d ",
1951            iEOC,
1952            iPOC,
1953            pcPic->getPicYuvOrg()->getWidth(COMPONENT_Y),
1954            pcPic->getPicYuvOrg()->getHeight(COMPONENT_Y),
1955            iQPFrame0,
1956            bIsInit? 1 : 0
1957        );
1958        for(int i = 0; i < iNumSelect; i++)
1959        {
1960            if(i > 0)
1961                fprintf(_Stream:fpParameter, _Format:"_");
1962            fprintf(_Stream:fpParameter, _Format:"%d", diEOCListSelect[i]);
1963        }
```

Figura 2.1: Screen codice 1

In figura 2.1, ammesso che bUseCNN sia "true" (riga 1940), il codice apre "command.dat" e ci scrive dentro valori essenziali del frame corrente. Tra questi: il numero dell'Encoding Order Count (EOC), il Picture Order Count (POC), il QP, le dimensioni del frame, un booleano inizializzato

a “true” e, riga 1962, due valori presi da una lista che corrisponderebbero agli indici EOC di 2 frames codificati precedentemente e scelti ora come predittori [**7**], basandosi sulle metriche di PSNR e CC. In precedenza però, più in alto nel codice, i dati del frame non ancora migliorato con il filtro, (uhInfoNoF), venivano inseriti in un nuovo file .yuv detto “rec_nof”, dove *nof* sta per *No-Filtered*.

```
1675    // save YUV content without in-loop filters
1676    for (int iCpnt = 0; iCpnt < MAX_NUM_COMPONENT; iCpnt++)
1677    {
1678        iWidth = pcPicYuv->getWidth(ComponentID(iCpnt));
1679        iHeight = pcPicYuv->getHeight(ComponentID(iCpnt));
1680
1681        if (iCpnt == 0)
1682        {
1683            fpYuvFilterLess = fopen("rec_nof.yuv", "wb+");
1684        }
1685        else
1686        {
1687            fpYuvFilterLess = fopen("rec_nof.yuv", "ab+");
1688        }
1689        fwrite(uhInfoNoF[iCpnt], sizeof(unsigned char), iWidth*iHeight, fpYuvFilterLess);
1690        fclose(fpYuvFilterLess);
1691
```

Figura 2.2: Screen codice 2

```
1964            fprintf( _Stream: fpParameter, _Format: " [end]");
1965            fclose( _Stream: fpParameter);
1966            bIsInit = false;
1967
1968            // generate start signal
1969            fpPredStart = fopen( _FileName: "pred_start.sig", _Mode: "w+");
1970            fclose( _Stream: fpPredStart);
1971
1972            // wait for Python predicting enhanced YUV frame
1973            while ((fpPredEnd = fopen( _FileName: "pred_end.sig", _Mode: "r")) == NULL)
1974    #ifdef _MSC_VER
1975                Sleep( dwMilliseconds: 0.1); // Windows
1976    #else
1977                usleep(100000); // Linux
1978    #endif
1979
1980            Int iRemoveResult = -1;
1981            while (iRemoveResult < 0)
1982            {
1983                fclose( _Stream: fpPredEnd);
1984                iRemoveResult = remove( _FileName: "pred_end.sig");
1985            }
```

Figura 2.3: Screen codice 3

Così, ora, la CNN ha le informazioni necessarie da prendere in input e le trova principalmente in "command.dat" e in "rec_nof.yuv" (e in altri file come vedremo più avanti). Dopodiché, tornando alla sezione in cui si richiama la rete, si genera un segnale di "Start" creando e aprendo il file "pred_start.sig". Bisogna ora sottolineare che, le rete neurale, gira su un server che va prima di tutto attivato da terminale per metterlo in ascolto, eseguendo il rispettivo codice Python. Poi si fa partire la *build* del codificatore e, quando a riga 1969 viene creto il segnale di "Start", il server lo rileva. A questo punto la CNN prenderà i suoi input, tra cui i due file sopra citati (tutta la rete è scritta in Python). Lo "Sleep" dentro il *while* di riga 1973, servirà per aspettare che il MIF (o IF) compia ogni volta le sue operazioni. Il ciclo corrente termina solo quando, dopo aver migliorato il frame, la CNN genera anche "pred_end.sig", che viene rilevato da HEVC. Infine, in un altro *while*, "pred_end.sig" viene rimosso per poter poi ricominciare l'intero processo al frame successivo.

Ricapitolando: la CNN prende le informazioni del frame non migliorato e anche i relativi dati scritti in "command.dat". Così, i suoi .py avviano il processo di miglioramento qualità grazie a tutto ciò che la rete ha imparato in fase di addestramento (come descritto in [**7**]).

```
1991
1992    for (int iCpnt = 0; iCpnt < MAX_NUM_COMPONENT; iCpnt++)
1993    {
1994        iWidth = pcPicYuv->getWidth(ComponentID(iCpnt));
1995        iHeight = pcPicYuv->getHeight(ComponentID(iCpnt));
1996
1997        if (iCpnt == 0)
1998        {
1999            fpYuvEnhanced = fopen( _FileName: "rec_enhanced.yuv", _Mode: "rb");
2000        }
2001
2002        assert(fread( _Buffer: uhInfoCNN[iCpnt], _ElementSize: sizeof(unsigned char), _ElementCount: iWidth * iHeight, _Stream: fpYuvEnhanced) > 0);
2003
2004        if (iCpnt == MAX_NUM_COMPONENT - 1)
2005        {
2006            fclose( _Stream: fpYuvEnhanced);
2007        }
2008    }
2009
2010    20180105 end
```

Figura 2.4: Screen codice 4

Il MIF-Net, infine, "sputerà" fuori il risultato sotto forma di file .yuv chiamato "rec_enhanced". Qui, ogni volta, la rete sovrascrive i dati del

frame corrente filtrato eliminando quelli precedenti. In figura 2.4, abbiamo la parte di codice che si occupa di leggere il frame filtrato per reinserirlo in HEVC. Si osserva, infatti, che il file "rec_enhanced.yuv" viene trasportato in "uhInfoCNN" (riga 2002). Il C++ ha ora a disposizione sia i dati sul frame non filtrato che quelli del frame migliorato dalla rete. A questi si aggiunge il fotogramma filtrato in maniera standard, con DBF, in qualche altra parte del codice. Per ogni indice di frame, l'HM-16.5_MIF-Net va a scegliere il migliore tra i tre in termini di PSNR, per inserirlo nell'output finale del video codificato [**7**]. Se tutto funziona, questo risultato deve avere un PSNR maggiore o uguale a quello di uno stesso video i quali frames sono però solo ed esclusivamente filtrati con rete MIF-IF (quindi senza selettore).

2.2 Testare HEVC con approccio MIF-IF

Successivamente siamo passati alle prove volte a verificare il funzionamento del codice, della rete, e soprattutto la generazione dei file appena elencati. Di base, quelli di "End" e "Start" sono segnali vuoti, ma è di nostro interesse verificare la scrittura di "command.dat" e "rec_enhanced.yuv". Inoltre, in *enhance_one_frame.py*, un .py che fa da *Main* per i codici della CNN, ne sono citati di ulteriori, come si vede nella seguente figura:

```
yuv_file_ori = 'ori.yuv'
yuv_file_in = 'rec_nof.yuv'
yuv_file_out = 'rec_enhanced.yuv'
yuv_file_out_total = 'rec_enhanced_total.yuv'
file_CU = 'Info_CUDepth.dat'
file_TU = 'Info_TUDepth.dat'
command_file = 'command.dat'
cfg_file = 'cfg.dat'
start_file = 'pred_start.sig'
end_file = 'pred_end.sig'
```

Figura 2.5: Screen di enhance_one_frame.py

Bisogna solo seguire la guida scritta dal team, che fortunatamente gli autori hanno lasciato nella cartella di file in cui sono presenti i codici, per gestire il processo tra codec ed il server dove gira la rete neurale.

Prima è stato utile installare, da terminale, *Tensorflow v1.13.1* (è consigliato dalla 1.8.0 in poi). È stato fatto in un *enviroment* Anaconda con dentro python3.7, facendolo girare almeno per ora su CPU. Poi la guida dice di lanciare *enhance_one_frame.py* da terminale: questa operazione attiva il server che si mette in ascolto in attesa del segnale di avvio. Nel frattempo, vanno scelti i parametri del video che si intende codificare (titolo e percorso, QP, fps, dimensione frame, bit depth, fbe ecc...) nei due cfg che si chiamano *encoder_yuv_source.cfg* ed *encoder_randomaccess_main.cfg* (della stessa cartella). Infine, su un altro terminale, bisogna chiamare l'eseguible "TAppEncoderStatic"[1] (dopo aver indicato quale configurazione del Gop usare, nel file cfg.dat).

È adesso che il codificatore comincia a girare. Codifica i suoi frames e, quando arriva alla sezione di codice che abbiamo descritto prima, il server rileva la presenza del file di "Start" e va a prendersi le informazioni necessarie nei file in questione. Come ci aspettavamo, tali file vengono generati dal C++ correttamente, insieme anche a "rec_enhanced.yuv" (riempito invece dalla rete). Queste prove ci hanno fatto notare che ci sono in realtà altri file che interagiscono nel processo intero. Infatti, ovviamente vi è un .bin che corrisponde al bitstream totale della codifica. Poi vi sono le informazioni sulle partizioni CU e TU salvate in file .dat, i quali verranno anch'essi presi in input dalla rete. Essi sono fondamentali all'operazione di miglioramento qualità del frame (come si diceva in [**7**]). È presente anche un "frame_period.txt" che indica ogni quanti frames ce ne sarà uno di tipo *intra* da non inviare alla CNN. È lui che regola quindi il valore booleano di "bUseCNN" (impostato per ora a "true" 31 volte ogni 32 frames). Infine, troviamo "rec.yuv" e "rec_enhanced_total.yuv".

[1] Per le prime prove di codifica vennero sfruttate sequenze qcif (176x144) disponibili online presso la Video Trace Library.

Questi ultimi due sono istanziati dal codificatore stesso e adesso capiamo meglio di cosa si tratta.

Innanzi tutto, partiamo dal dire che i nostri test si possono dividere in due categorie: a codifica terminata e a processo intermedio interrotto. Per le prove usiamo la sequenza *akiyo_qcif.yuv*, con frames di 176x144 pixels, in configurazione RA e un QP=37. I rispettivi file .yuv generati sono stati poi esplorati con il programma YUViuwer.

1) Processo completo:

Una volta terminati i frames a disposizione nel video originale, il codificatore smette di girare mentre il server rimane in ascolto senza passare però alcun dato alla rete neurale. A questo punto sono stati esplorati i file di interesse, che vengono salvati nella cartella "bin". Tra questi, notiamo "rec_nof.yuv" che è popolato sempre e solo da dati di un solo frame. Lo stesso succede anche per "rec_enhanced.yuv". E tutto ciò è in linea con quanto ci aspettavamo visto che per entrambi, ad ogni giro del codificatore, esiste un'operazione di sovrascrittura che cancella i dati precedenti. Nella stessa cartella appare anche "rec.yuv", contenente l'intero video codificato e ricostruito. Infine vi è "rec_enhanced_total.yuv": anch'esso sequenza .yuv della stessa durata.

2) Processo intermedio:

Se con ctrl-z, durante la codifica, interrompiamo il lavoro del server, si possono esplorare i file che sono stati generati fino a quel momento. Una volta fatto, notiamo che "rec_enhanced_total.yuv" consiste in una sequenza video con tutti i frames codificati fino a quell'istante, che non seguono però il Picture Order Count (quindi la sequenza va a scatti). Invece "rec.yuv" risulta un video fluido, con meno frames di "rec_enhanced_total.yuv", ma disposti in ordine (seguendo il POC). Di nostro interesse, ora, sono proprio questi ultimi due. La loro spiegazione è facilmente reperibile all'interno del C++ di cui ci stiamo occupando:

```
2209        // 20180505 added
2210        // save final decide YUV content
2211        for (int iCpnt = 0; iCpnt < MAX_NUM_COMPONENT; iCpnt++)
2212        {
2213            iWidth = pcPicYuv->getWidth(ComponentID(iCpnt));
2214            iHeight = pcPicYuv->getHeight(ComponentID(iCpnt));
2215
2216            if (iPOC == 0 && iCpnt == 0)
2217            {
2218                fpYuvEnhanced = fopen("rec_enhanced_total.yuv", "wb+");
2219            }
2220            else
2221            {
2222                fpYuvEnhanced = fopen("rec_enhanced_total.yuv", "ab+");
2223            }
2224            fwrite(uhInfoBest[iCpnt], sizeof(unsigned char), iWidth*iHeight, fpYuvEnhanced);
2225            fclose(fpYuvEnhanced);
```

Figura 2.6: Screen codice 5

Di fatto, HEVC modificato dagli autori di [**7**], recupera l'immagine migliorata dal MIF (CNN) e la confronta poi in termini di PSNR con quella Nof e quella migliorata dal DBF (approccio ILF). Il frame vincitore viene inserito nella matrice uhInfoBest. Sottolineiamo che, arrivati fin qui, il codificatore mette a disposizione la possibilità di cambiare e sostituire a livello di blocchi[2] (qui di dimensione 256x256, ma modificabile) il frame scelto tra le 3 opzioni. Ovvero, per ogni componente, per ogni blocco che la costituisce, si fa un confronto di MSE (Errore quadratico medio) rispetto al blocco originale, tra i 3 rispettivi blocchi dei frames Nof, ILF e CNN. Tra questi si sceglie quello con errore minore per poi inserirlo in uhInfoBest. Il tutto ripetuto per le 3 componenti dell'immagine. Quindi, alla fine, il singolo fotogramma di output non corrisponderà completamente al frame di uno dei 3 approcci, bensì ad una nuova immagine ibrida, con il migliore PSNR possibile.

[2]HEVC modificato offre la possibilità di svolgere le sostituzioni a livello di blocchi o anche a livello di componenti del frame. La scelta è libera, basta semplicemente modificare la dimensione dichiarata del blocco.

```
2107    else // block-level mode selection
2108    {
2109        Int iWidthInBlocks = iWidth % iBlockWidth[iCpnt] == 0 ? iWidth / iBlockWidth[iCpnt] : iWidth / iBlockWidth[iCpnt] + 1;
2110        Int iHeightInBlocks = iHeight % iBlockWidth[iCpnt] == 0 ? iHeight / iBlockWidth[iCpnt] : iHeight / iBlockWidth[iCpnt] + 1;
2111        printf("Component %d: Flag = \n", iCpnt);
2112        for (int yBlock = 0; yBlock < iHeightInBlocks; yBlock++)
2113        {
2114            Int yStart = yBlock * iBlockWidth[iCpnt];
2115            Int yEnd = min((yBlock + 1)*iBlockWidth[iCpnt], iHeight);
2116            printf("    ");
2117            for (int xBlock = 0; xBlock < iWidthInBlocks; xBlock++)
2118            {
2119                Int xStart = xBlock * iBlockWidth[iCpnt];
2120                Int xEnd = min((xBlock + 1)*iBlockWidth[iCpnt], iWidth);
2121                ullSquaredErrorNoF = 0;
2122                ullSquaredErrorILF = 0;
2123                ullSquaredErrorCNN = 0;
2124                for (int y = yStart; y < yEnd; y++)
2125                {
2126                    for (int x = xStart; x < xEnd; x++)
2127                    {
2128                        deltaTemp = (int)uhInfoNoF[iCpnt][y*iWidth + x] - (int)uhInfoOri[iCpnt][y*iWidth + x];
2129                        ullSquaredErrorNoF += deltaTemp * deltaTemp;
2130                        deltaTemp = (int)uhInfoILF[iCpnt][y*iWidth + x] - (int)uhInfoOri[iCpnt][y*iWidth + x];
2131                        ullSquaredErrorILF += deltaTemp * deltaTemp;
2132                        deltaTemp = (int)uhInfoCNN[iCpnt][y*iWidth + x] - (int)uhInfoOri[iCpnt][y*iWidth + x];
2133                        ullSquaredErrorCNN += deltaTemp * deltaTemp;
2134                    }
2135                }
```

Figura 2.7: Screen codice 6 - a

```
2136                if (ullSquaredErrorNoF <= ullSquaredErrorILF && ullSquaredErrorNoF <= ullSquaredErrorCNN)
2137                {
2138                    for (int y = yStart; y < yEnd; y++)
2139                        for (int x = xStart; x < xEnd; x++)
2140                            uhInfoBest[iCpnt][y*iWidth + x] = uhInfoNoF[iCpnt][y*iWidth + x];
2141                    printf(" 0");
2142                }
2143                else if(ullSquaredErrorILF <= ullSquaredErrorCNN)
2144                {
2145                    for (int y = yStart; y < yEnd; y++)
2146                        for (int x = xStart; x < xEnd; x++)
2147                            uhInfoBest[iCpnt][y*iWidth + x] = uhInfoILF[iCpnt][y*iWidth + x];
2148                    printf(" 1");
2149                }
2150                else
2151                {
2152                    for (int y = yStart; y < yEnd; y++)
2153                        for (int x = xStart; x < xEnd; x++)
2154                            uhInfoBest[iCpnt][y*iWidth + x] = uhInfoCNN[iCpnt][y*iWidth + x];
2155                    printf(" 2");
2156                }
2157            }
2158            printf("\n");
2159        }
2160    }
```

Figura 2.8: Screen codice 6 - b

Gli stessi dati sono passati poi al file “rec_enhanced_total.yuv” in modalità append (Figura 2.6, riga 2224). Allora è chiaro che tale file non è nient’altro che il video, codificato e ritrasformato in .yuv, dove, per ogni frame, vi è la migliore combinazione possibile tra i dati di Nof, ILF e CNN. Il fatto che il codice segua l’Encoding Order Count (EOC), e non POC, chiarisce perché “rec_enhanced_total.yuv” sia, a processo intermedio, composto da frames in disordine. A fine processo però, i dati

vengono riordinati dalla lettura del *decoder buffer* per generare così la versione finale di "rec_enhanced_total.yuv". Questa volta con frames in ordine.

Per quanto riguarda "rec.yuv", bisogna ricordare che solitamente, qualsiasi codificatore MPEG, possiede in codifica una parte di codice che si occupa di ricostruire il file originale con i frames già codificati. Il video viene incrementato solo quando è disponibile il frame con il valore di POC giusto per disporli tutti in ordine, come se si stesse svolgendo, contemporaneamente, l'operazione di decodifica. Questo avviene dopo aver fatto passare i frames attraverso i filtri utili a renderli dei predittori migliori. Ovvero, avviene *post-loop*. Nel nostro caso però, per i frames si sceglie sempre l'approccio a più alta qualità, scartando in maniera definitiva le altre opzioni. Di conseguenza, "rec.yuv", a fine processo coinciderà proprio con "rec_enhanced_total.yuv". A confermare l'ipotesi, il PSNR dei due file dovrebbe essere uguale se confrontati con il video originale.

A tal proposito mostriamo test di calcolo di PSNR: usando il software Psnr.video si evince in tutte le prove fatte che il PSNR delle sequenze, confrontate con *akiyo_qcif.yuv* (video originale) nei 300 frames di codifica, è lo stesso per entrambi (Figura 2.9). Molto più semplicemente basta anche vedere che, confrontando i rec tra loro, il risultato è "inf" (infinito – Figura 2.10). "rec.yuv" e "rec_enhanced_total.yuv" sono cioè la stessa sequenza video.

```
(base) tiziano@tiziano-VirtualBox:~/Scaricati/HM-16.5_MIF-Net/bin$ ./psnr_video
akiyo_qcif.yuv rec.yuv 176 144 300
34.270290
(base) tiziano@tiziano-VirtualBox:~/Scaricati/HM-16.5_MIF-Net/bin$ ./psnr_video
akiyo_qcif.yuv rec_enhanced_total.yuv 176 144 300
34.270290
```

Figura 2.9: Risultato confronto PSNR - a

```
(base) tiziano@tiziano-VirtualBox:~/Scaricati/HM-16.5_MIF-Net/bin$ ./psnr_video
 rec.yuv rec_enhanced_total.yuv 176 144 300
inf
(base) tiziano@tiziano-VirtualBox:~/Scaricati/HM-16.5_MIF-Net/bin$
```

Figura 2.10: Risultato confronto PSNR - b

A quanto pare, i sistemi di Deep-Learning, riescono ad apportare grandi benefici soprattutto nelle prime fasi di lavoro, cioè nei primi frames nel caso delle codifiche. Dopodiché però, più si va avanti e maggiori sono i problemi creati dalla rete stessa. Ovvero, nel nostro caso, con una minore cura il MIF-Net riuscirà a migliorare i singoli blocchi dell'immagine. Questo perché, raggiunto un elevato numero di blocchi da processare, si alzerà per la rete la probabilità di un errore che può in seguito propagarsi in maniera critica per le grandi moli di lavoro. Ne deriva che, confrontando il PSNR con gli altri approcci, più si va avanti con la codifica e più alto sarà il numero di blocchi scelti dagli approcci ILF o Nof. Diminuendo così il numero di pixel migliorati dalla CNN all'interno di un singolo frame del file di output.

Ne mostriamo un esempio su una codifica con HM-16.5_MIF-Net, fatta su 25 frames di una sequenza in FullHD.

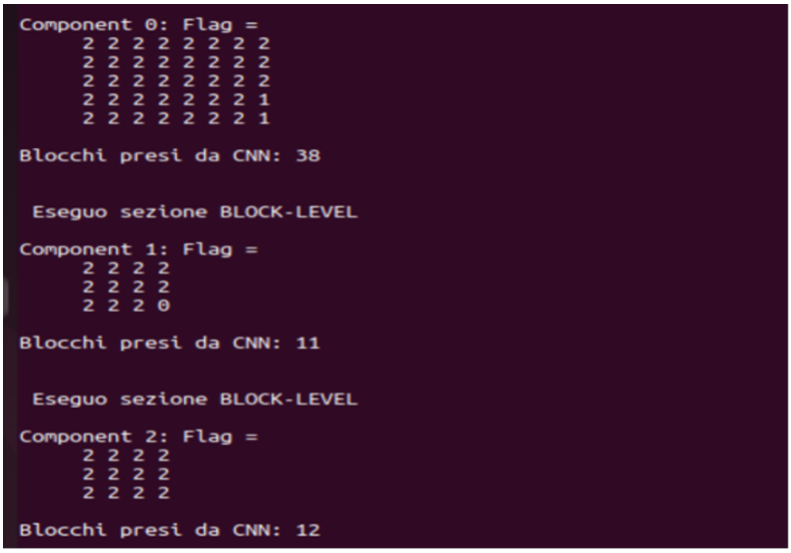

Figura 2.11: Sostituzione blocchi frame: 1

In figura 2.11 è mostrata la sostituzione a blocchi delle 3 componenti sul primo frame della sequenza. Il numero 2 sta a simboleggiare che il corrispondente blocco 256x256 è stato migliorato dalla CNN.

È chiaro dalla figura che, nel frame in questione, questo accade per quasi tutti i blocchi. La disposizione di tutti quei numeretti va poi a riformare l'intera risoluzione FullHD dell'immagine (con dimensione più piccola su U e V per via del sottocampionamento 4:2:0).

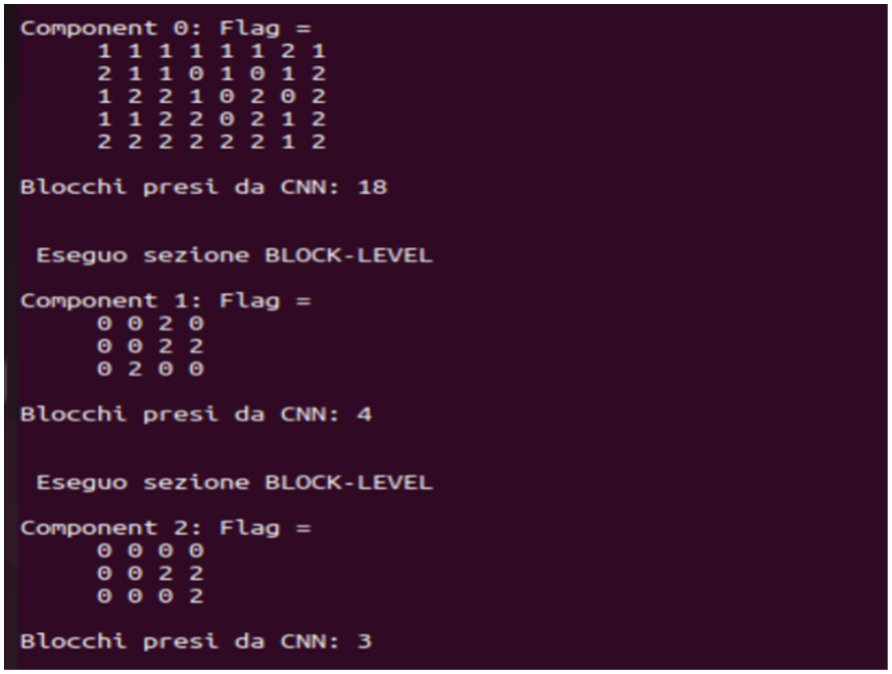

Figura 2.12: Sostituzione blocchi frame: 24

Invece, in figura 2.12, osserviamo lo stesso fenomeno applicato però al frame codificato numero 24. È evidente ciò che abbiamo appena anticipato: il numero di blocchi prelevati dal metodo MIF-Net cala drasticamente, mostrando una preferenza verso i metodi ILF (1) e Nof (0). A testimoniare quindi come la rete neurale perda efficacia col procedere della codifica, frame dopo frame. Dopo tutto, è proprio per tale ragione che fu inserita la sostituzione a blocchi in HM-16.5_MIF-Net. Questo fenomeno dovremmo tenerlo bene a mente e ricordarcelo perché sarà sempre presente nel resto delle prove e sperimentazioni con CNN.

2.3 Prestazioni MIF-Net

La suddivisione a blocchi, impostata in maniera standard sulla dimensione quadrata di 256, risulta allora leggermente anomala data la grandezza del blocco stesso. Non è chiaro, tra l'latro, come e quando la scelta di un blocco, piuttosto che un altro, sia segnalata nel bitstream per informare il decodificatore a riguardo. Durante le ore di Tirocinio abbiamo provato ad entrare in contatto con gli autori di [**7**] per avere spiegazioni, però senza successo. Di conseguenza, abbiamo voluto verificare le effettive prestazioni dell'approccio MIF-IF e della sostituzione a blocchi, confrontadole con quanto dichiarato.

Nel realizzare ciò, è bastato confrontare i valori di PSNR e bit-rate su 25 frames di sequenze FullHD e HDTV, codificandole con 3 metodi differenti: HM-16.5 standard, HM-16.5 CNN only (nel quale, per ogni blocco di ogni frame, si opta solamente per il filtro della rete neurale) e HM-16.5_MIF-Net (ovvero approccio ibrido). Il tutto a valori QP di 27, 32, 37, 45, 50, in configurazione RA, per poi tracciarne infine i relativi grafici Rate-Distortion (PSNR in dB e bit-rate in kbps). Le sequenze sono state reperite dalla libreria *xiph.org*.

Nelle figure 2.13 e 2.14 abbiamo grafici RD di due sequenze 1280x720. Come si evince, l'approccio con solo CNN è migliore di HM-16.5 solo a fasi alterne, variando in base al bit-rate (QP) e al tipo di sequenza. Invece, in entrambi i casi, l'approccio ibrido è sempre il migliore, infatti la sua curva si poggia al di sopra delle altra due. È ciò che ci si poteva aspettare in quanto, per come è impostato il C++, questo approccio sceglierebbe sempre il filtro migliore in ogni blocco 256x256. Di conseguenza la curva non può trovarsi al di sotto di una delle altre due.

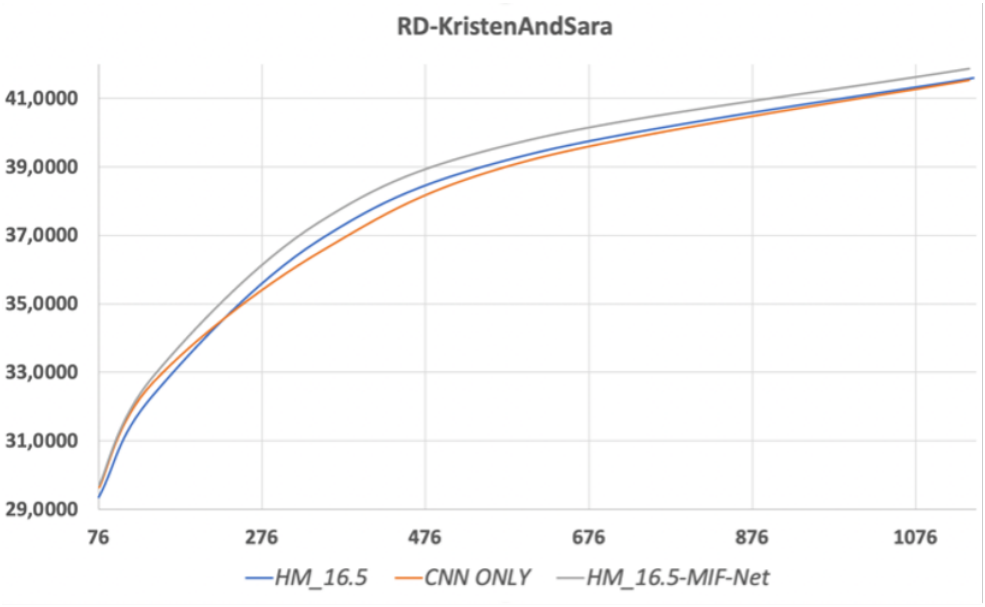

Figura 2.13: Grafico RD – a

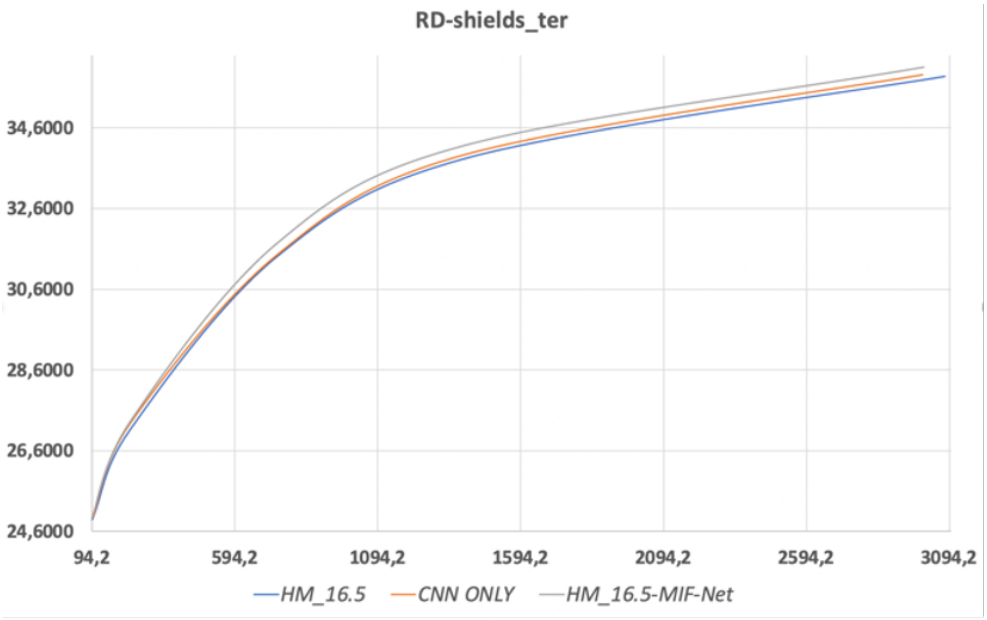

Figura 2.14: Grafico RD - b

Risultato analogo si ottiene per una sequenza FullHD:

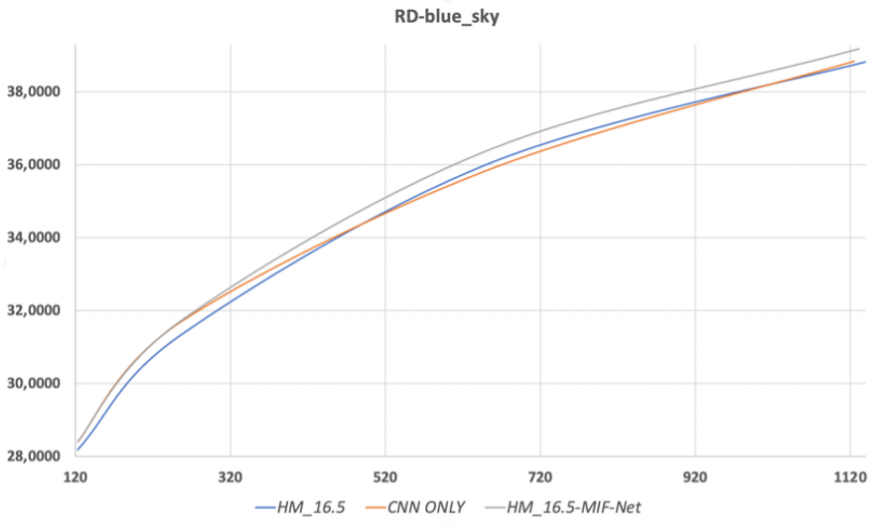

Figura 2.15: Grafico RD sequenza FullHD

In figura 2.16, le curve RD di city_4cif. I grafici sono stati tracciati su Excel prendendo i valori di PSNR (y) e bit-rate (x) ottenuti dalle codifiche e precedentemente riportati sulla tabella di figura 2.17.

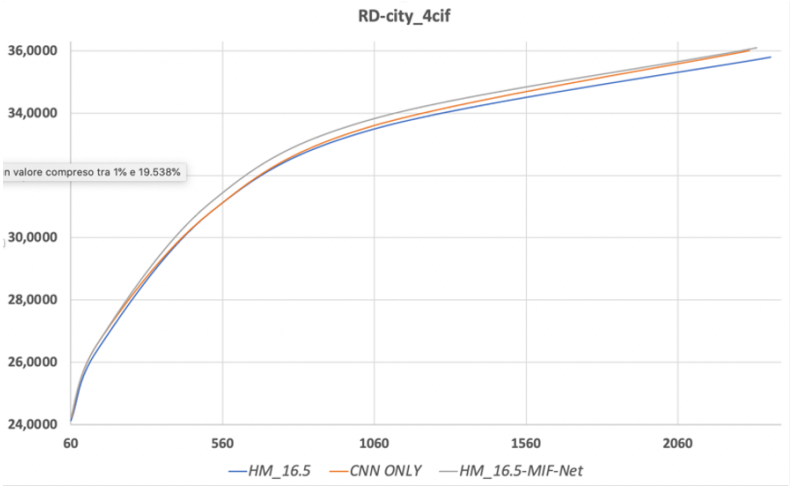

Figura 2.16: Grafico RD sequenza 4cif

Nella tabella qui sotto, per gli approcci con Deep-learning, è riportata la differenza tra PSNR della sequenza codificata corrispondente e il PSNR ottenuto dal metodo standard, calcolandolo però non sullo stesso bit-rate, bensì sui relativi valori dei metodi in questione. Da qui non si evincerebbe quindi in maniera chiara quale dei 3 è l'approccio migliore ed è per questo che sono stati mostrati i grafici RD, molto più immediati e comprensibili.

Sequenze	Risoluzioni	Frames	*QP*	*HM_16.5*		*HEVC CNN Only*		*HM_16.5_MIF-NET*	
				PSNR (dB)	*Bit-rate*	*PSNR (dB)*	*Bit-rate*	*PSNR (dB)*	*Bit-rate*
					kbps	Rispetto a HM_16.5	kbps	Rispetto a HM_16.5	kbps
Kristen (60)	1280x720	25	27	41,5973	1146,547	-0,0592	1140,768	+0,27758	1141,402
			32	39,3900	606,778	-0,1716	606,163	**+0,40020**	602,707
			37	36,8389	344,525	-0,2939	348,614	**+0,53040**	344,429
			45	32,3194	142,387	+0,3912	143,002	**+0,55418**	142,368
			50	29,3585	76,57	+0,2881	77,203	**+0,36913**	76,896
ducks_take (50)	1280x720	25	27	34,6469	11150,544	+0,19584	11032,224	+0,19914	11047,296
			32	31,9482	5180,96	+0,19326	5132,304	+0,20730	5134,768
			37	29,3575	2559,248	+0,24086	2539,904	+0,30590	2545,024
			45	25,5847	746,528	+0,22258	743,824	**+0,22703**	740,624
			50	23,4676	290,672	+0,15231	292,192	**+0,16180**	292,128
shields (50)	1280x720	25	27	35,8775	3078,592	+0,03860	3000,576	+0,22990	3004,720
			32	33,8665	1413,28	+0,06435	1395,856	**+0,28240**	1388,848
			37	31,4220	743,872	+0,04400	743,728	**+0,31050**	740,288
			45	27,1044	226,752	+0,18868	229,088	**+0,21883**	227,888
			50	24,8880	95,344	+0,05330	96,192	**+0,06954**	94,848
blueSky (25)	1920x1080	25	27	41,3349	2076,064	+0,02570	2040,128	+0,34710	2053,448
			32	38,8269	1139,808	+0,00650	1124,8	+0,34741	1131,232
			37	36,0647	656,608	-0,21676	651,168	+0,32350	651,608
			45	31,3370	255,832	+0,33780	254,528	**+0,36020**	254,456
			50	28,1910	122,608	+0,23460	123,464	+0,21757	122,688
city_4cif (60)	704x576	25	27	35,7978	2364,691	+0,20909	2294,016	+0,30241	2318,746
			32	33,5159	1068,019	+0,02964	1039,373	+0,28562	1049,453
			37	30,8238	521,011	-0,07380	511,411	+0,28340	517,306
			45	26,3941	147,418	+0,23686	149,683	+0,21590	149,088
			50	24,1276	60,365	+0,09713	61,133	**+0,10620**	60,787

Figura 2.17: Tabella di confronto PSNR e bit-rate

Nelle seguenti figure[3] notiamo una porzione di pixel di un frame di *KristenAndSara.yuv*, prima migliorata dal metodo classico ILF (a) e poi dal MIF-Net (b), con QP=37. La differenza si nota osservando soprattuto i capelli, gli occhi e la bocca della ragazza.

[3]Gli screen delle immagini codificate che riportiamo sono estrapolati dal visualizzatore YUViuwer.

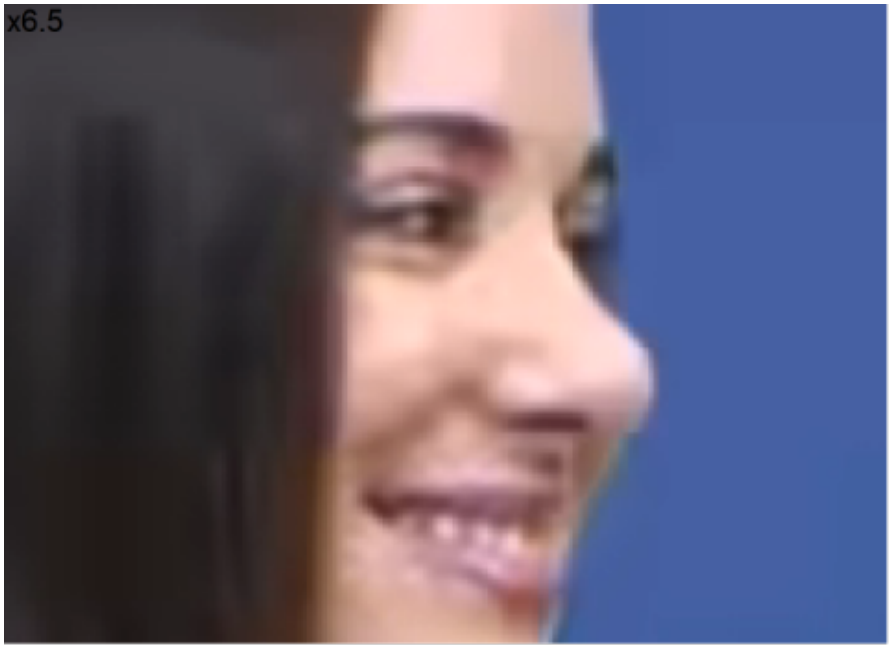

Figura 2.18: KristenAndSara – a

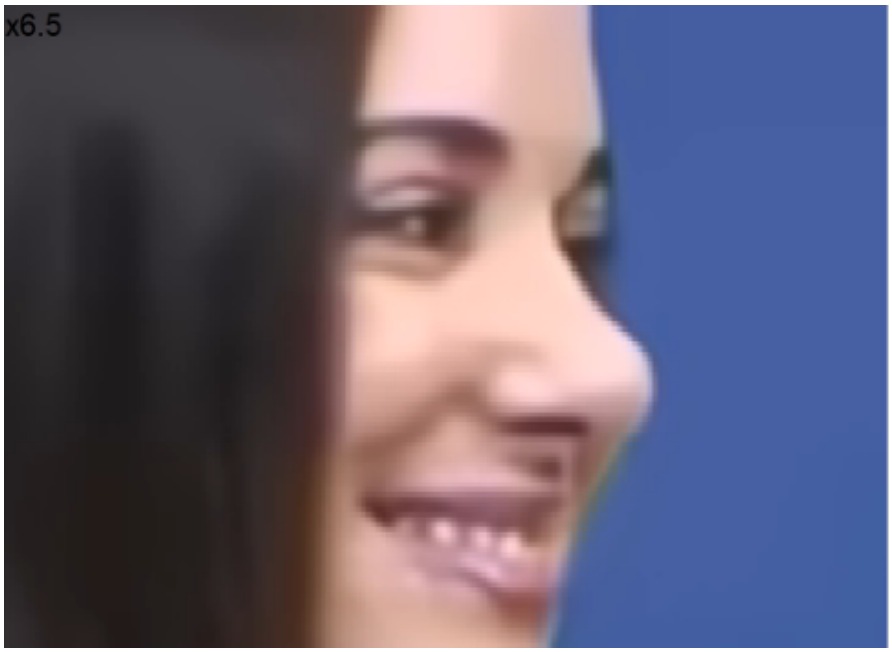

Figura 2.19: KristenAndSara – b

Bjontegaard Metric

Calcoliamo adesso le misure di BD-rate e BD-PSNR. Queste ci mostrano il guadagno di PSNR a parità di bit-rate ed il guadagno di bit-rate, a parità di PSNR, di un metodo rispetto all'altro [**8**]. Tali misure sono state ottenute questa volta su QP più usati a livello standard: 22, 27, 32, 37 e 42, sui 25 frames di sequenze 720p (*KristenAndSara.yuv*, *shields_ter.yuv*, *ducks_take_off.yuv*), FullHD (*blue_sky.yuv*) e 4cif (*city_4cif.yuv*), in configurazione RA.

		BD-rate (%)	BD-PSNR dB
4cif	city_4cif	-10,18	0,33
1280x720	KristenAndSara	-10,29	0,39
	shields_ter	-10,2	0,29
	ducks_take_off	-7,28	0,26
1920x1080	blue_sky	-8,39	0,387
	AVERAGE	**-9,2**	**0,331**

Figura 2.20: Tabella dati metriche Bjontegaard

Le misure di Bjontegaard si articolano in una media indicativa, sui vari frames e diversi QP, del guadagno che in codifica un approccio ha nei confronti di un metodo meno performante [**8**]. Tale guadagno è strettamente legato all'area compresa tra i due grafici RD della medesima sequenza. Maggiore è la distanza tra i due grafici, come abbiamo visto poco fa, maggiore è il guadagno di un metodo rispetto a quello di confronto e più significativi saranno i valori di Bjontegaard. In figura 2.20 vediamo i risultati BD nel confronto tra HM-16.5 e HM-16.5_MIF-Net, ottenuti grazie al software apposito: per il bit-rate, un valore negativo indica un guadagno, espresso in percentuale, con la stessa qualità d'immagine mentre, per il PSNR, il valore assoluto rappresenta i dB in più

raggiunti a parità di bit-rate. Si può osservare che per entrambi le misure il metodo MIF-Net è più efficiente. Inoltre, i risultati sono in linea con le tabelle riportate in [**7**], ed è un'ulteriore conferma di come il metodo funzioni e di quanto sia veritiero ciò che viene descritto nel relativo articolo. L'eseguibile che ci ha prodotto questi numeri mette a disposizione la possibilità di tracciare i grafici RD delle stesse sequenze analizzate con le metriche di Bjontegaard:

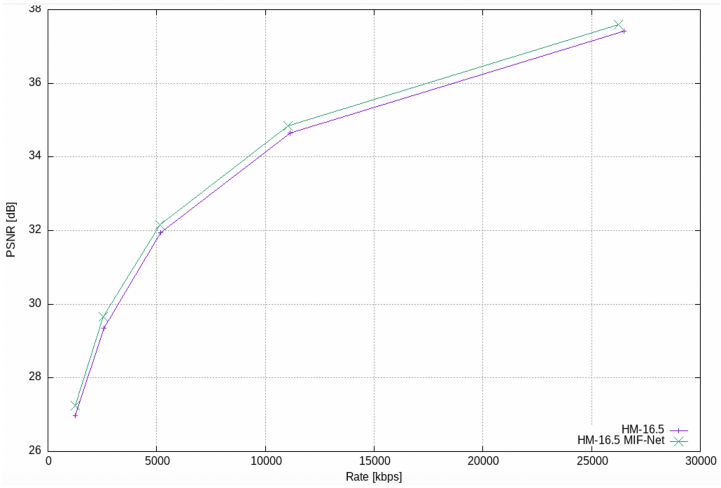

Figura 2.21: Grafico RD (ducks_take_off)

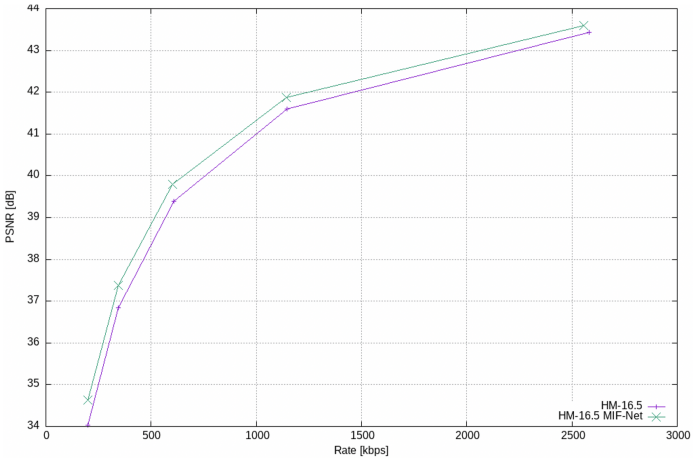

Figura 2.22: Grafico RD (KristenAndSara)

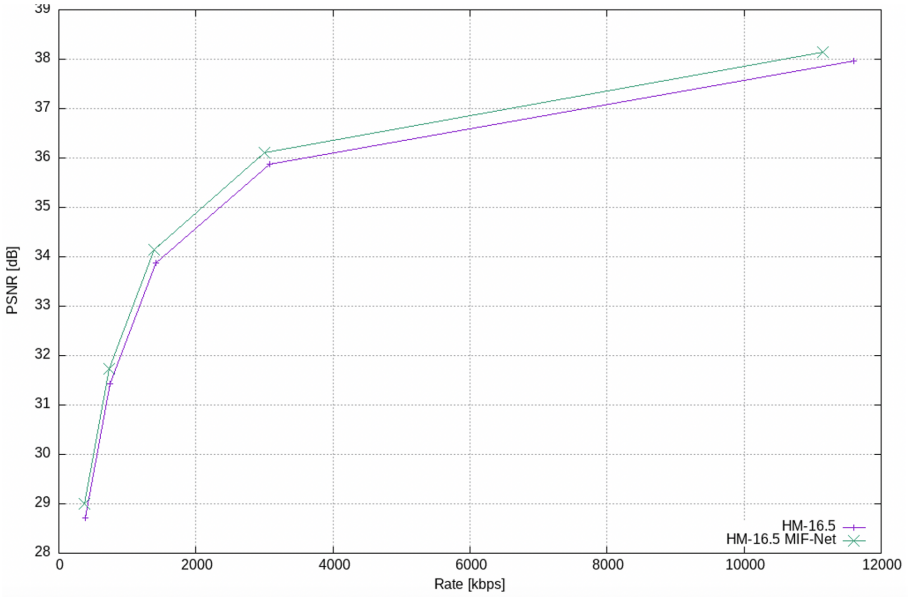

Figura 2.23: Grafico RD (shields_ter)

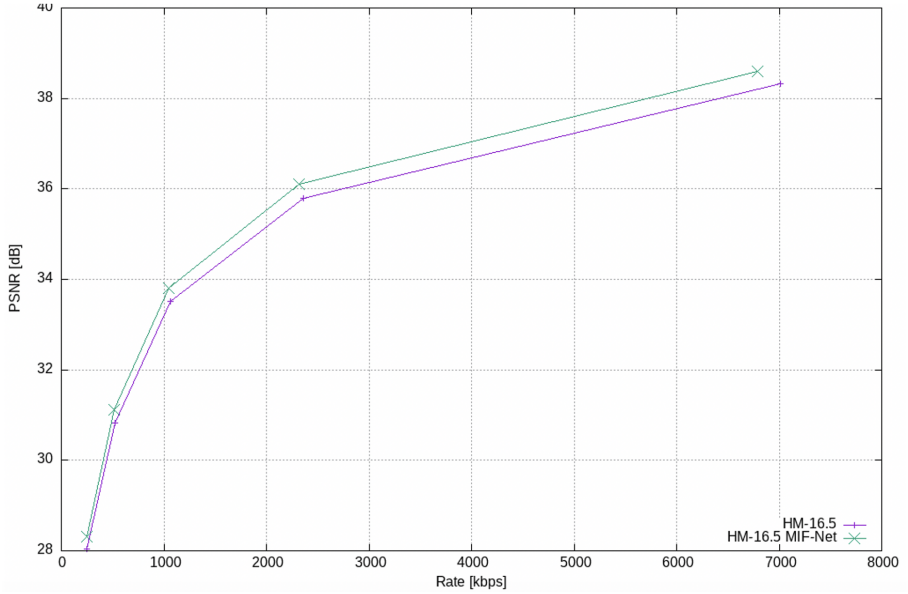

Figura 2.24: Grafico RD (city_4cif)

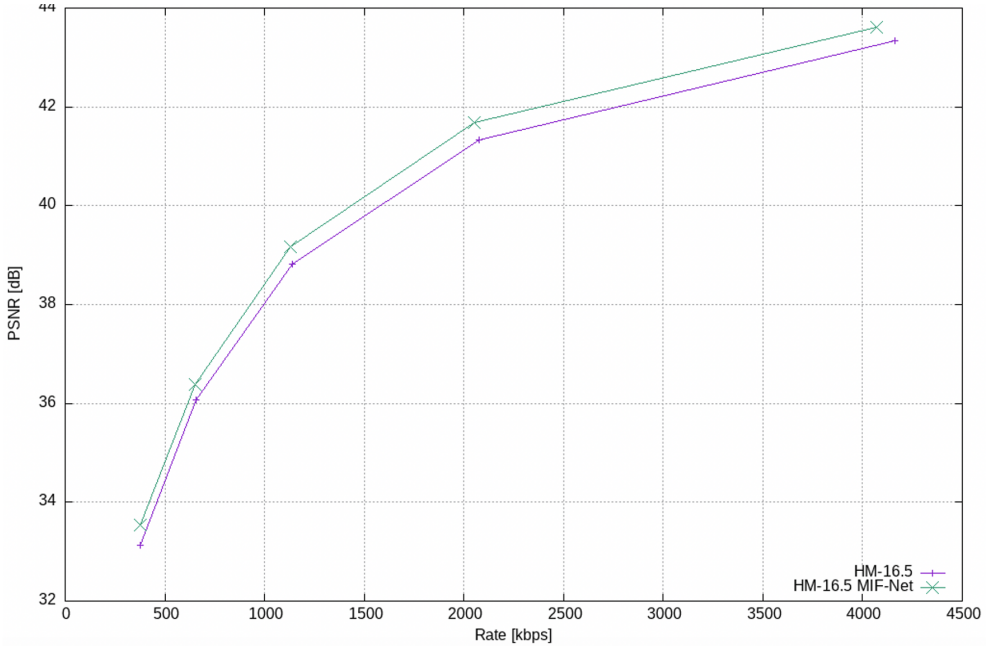

Figura 2.25: Grafico RD (blue_sky)

Le presenti curve ripercorrono gli stessi andamenti ottenuti con i grafici Excel nelle figure da 2.13 a 2.16. Tutte le prove di cui abbiamo trattato sono state svolte su un computer con CPU Intel(R) Core(TM) i7-9750H @2,6 GHz, 16GB di RAM e Ubuntu 22.04 (64 bit).

Commenti

Di conseguenza, per quanto mostrato fino ad ora, HM-16.5_MIF-Net si identifica come il metodo di filtraggio migliore, rispetto a HEVC classico e l'approccio con blocchi migliorati solo dalla rete. Quanto sperimentato rispecchia ciò che è dichiarato in [**7**]. Dai grafici RD si osserva ancora una volta come l'approccio ibrido abbia senso dato che con l'aumentare del numero di blocchi da elaborare la CNN può creare danni, che vengono quindi soppressi scegliendo singoli blocchi da altri approcci, come appunto quello standard o, talvolta, dal frame Nof.

Un sistema così, quindi, deve essere necessariamente il migliore tra i vari proposti. Ne risulta inoltre che la curva del MIF-Net si posiziona sempre nella parte superiore del grafico grazie anche all'approccio multi-frame che la caratterizza. Confermandosi vincente. Su queste note, sono poi terminate le ore di Tirocinio a disposizione dando spazio e tempo al lavoro supplementare. Esso è volto prima di tutto a fare ulteriori codifiche, modificando questa volta la dimensione dei blocchi di sostituzione con valori minori, come 128 e 64, per capire se vi sia un guadagno in qualità e bit-rate rispetto al caso 256x256, con o meno un aumento del tempo di calcolo. Il progetto proseguirà poi nell'approcciare i primi step da realizzare per far funzionare il MIF-Net anche su EVC.

Concludiamo questa sezione mostrando quanto sperimentato su una porzione di pixel di un frame della sequenza FullHD *ducks_take_off.yuv*, con QP=42.

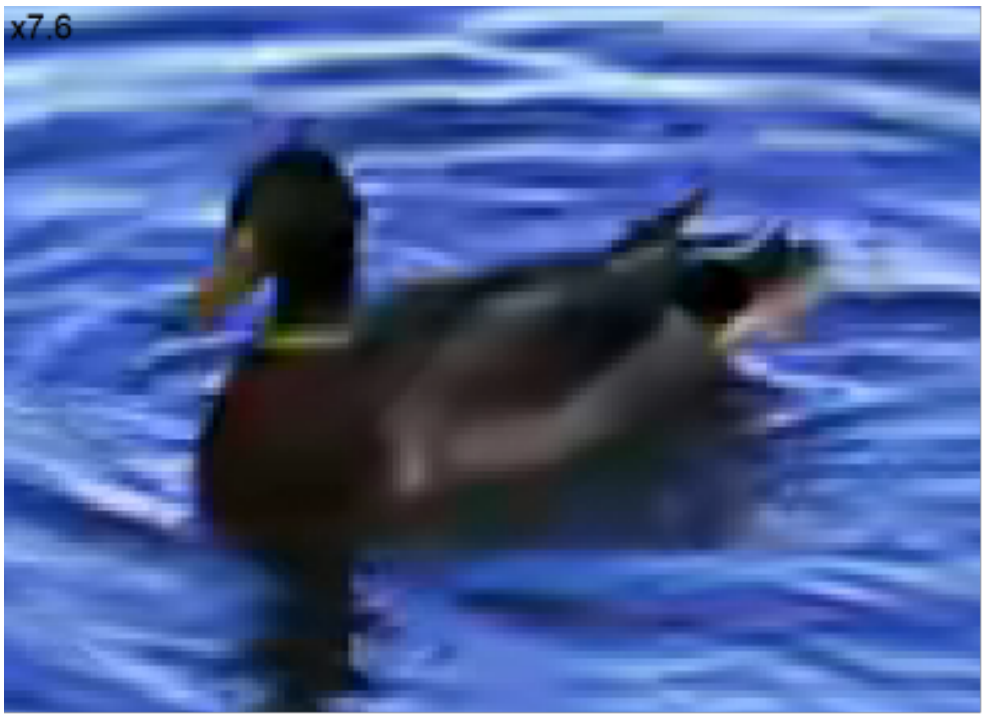

Figura 2.26: ducks_take_off – ILF

Figura 2.27: ducks_take_off – MIF-Net

2.4 MIF-Net e sostituzioni a blocchi di 256, 128 e 64

Nel presente capitolo mostriamo come cambia il rapporto PSNR/bit-rate, attraverso le curve di Rate-Distortion, se procediamo con una sostituzione a blocchi di 128x128 e 64x64 pixel, invece che 256x256. Di norma, ci aspettiamo che la qualità dei frames aumenti quando si lavora su blocchi più piccoli. Questo perché, andando più in profondo nei dettagli, permettiamo alla sostituzione di scegliere il blocco adeguato nelle zone di pixel di dimensioni minori e, quindi, di operare con maggiore accuratezza. Ovviamente però, il centro del discorso è trovare un *trade-off* tra la dimensione del blocco e il tempo complessivo di calcolo.

Per fare questo, basta cambiare il valore della variabile che indica la dimensione del blocco nel codice TEncGop.cpp di HEVC con MIF-Net. Inserendo 128 o 64 invece che 256. Nei nostri test, abbiamo codificato centinaia di frames su sequenze di diverse risoluzioni, con il MIF-Net seguito da una sostituzione a blocchi 256x256 per poi ripetere le stesse codifiche con sostituzioni a 128x128 e 64x64. In tutti e 3 i casi, abbiamo scelto 4 valori diversi di QP standard per ogni sequenza: 22, 27, 32, 37 (configurazione RA). Infine, avendo raccolto i dati su bit-rate e PSNR, rispetto ai video originali, sono stati tracciati i grafici RD e calcolate poi le metriche di Bjontegaard.

Già da subito possiamo anticipare che vi è certamente un guadagno, anche se limitato. Nei seguenti grafici, mostriamo le curve RD per alcune delle sequenze di test utilizzate:

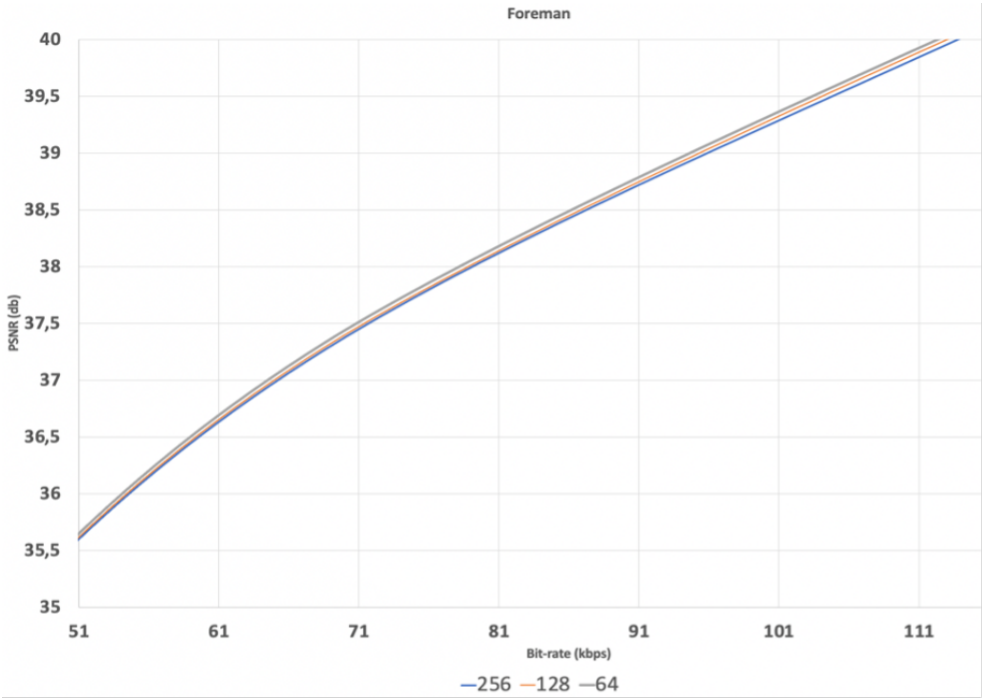

Figura 2.28: Grafico RD – Foreman - qcif

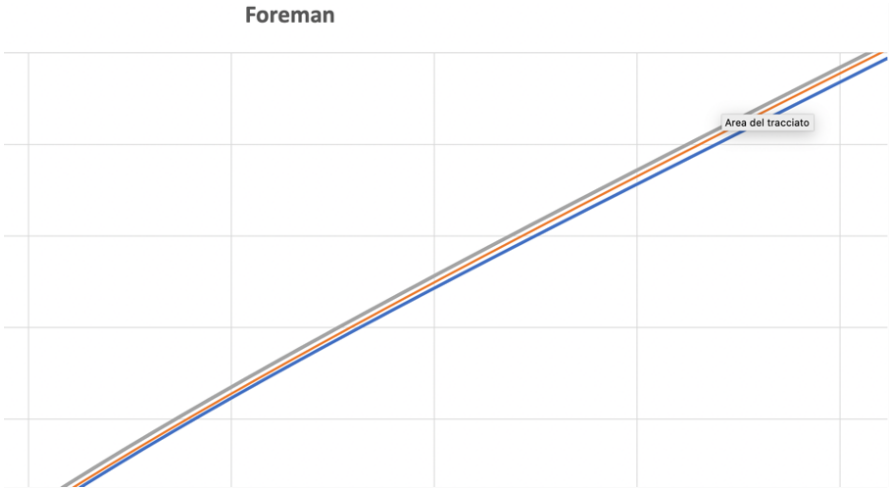

Figura 2.29: Zoom Grafico RD – Foreman – qcif

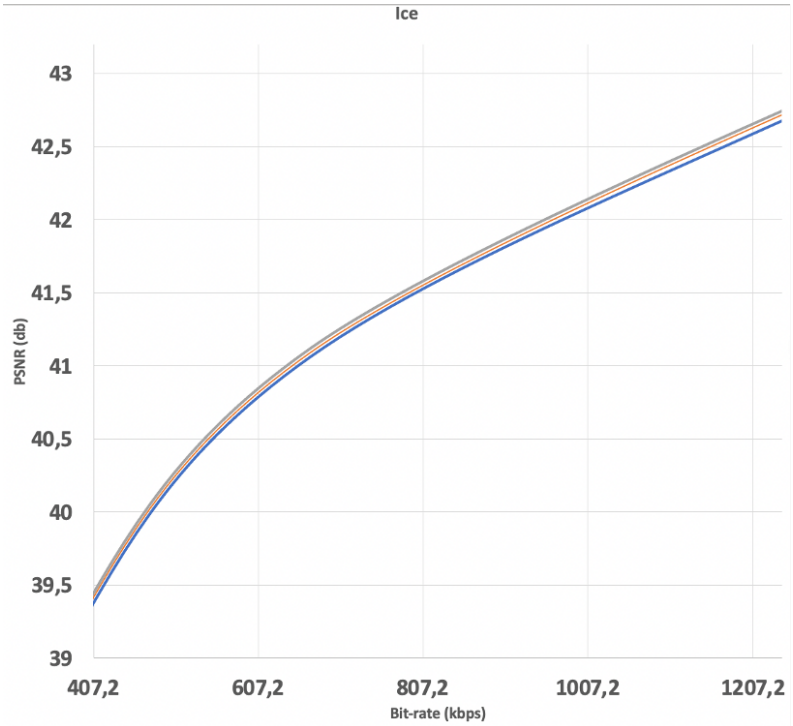

Figura 2.30: Grafico RD – Ice – 4cif

La linea grigia corrisponde alle performance nel caso di blocchi 64x64 mentre, a seguire, l'arancione indica una sostituzione 128x128 e la blu il caso 256x256. Come si nota, soprattutto in figura 2.29, la linea grigia è sempre al disopra delle altre, anche se solo leggermente. Ciò indica comunque un guadagno, per ogni tipo di valore in kbps. Lo stesso identico fenomeno si ripete per le altre sequenze di test, in tutto 8. Esponiamo i grafici per i casi in cui il guadagno risulta più netto.

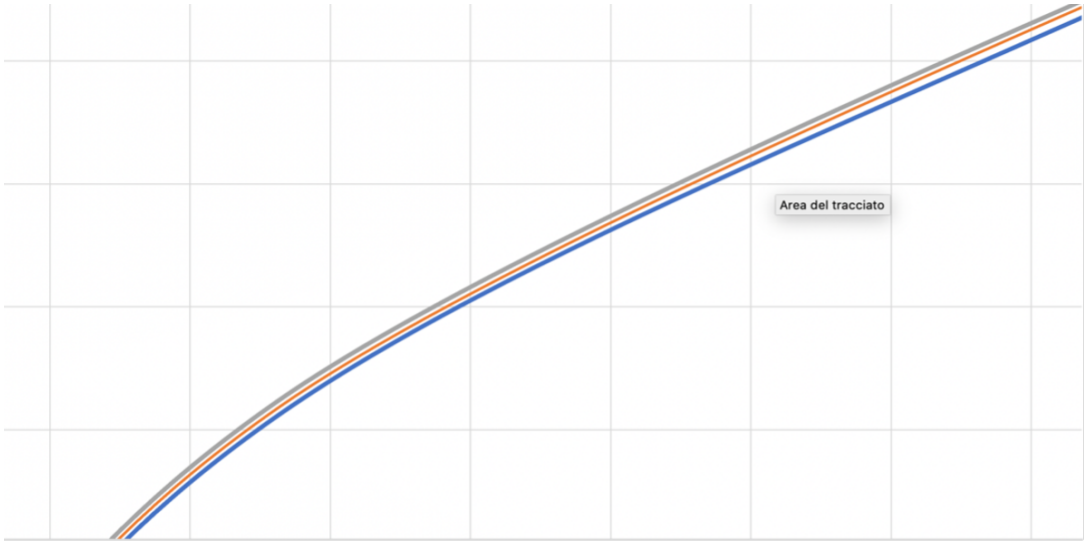

Figura 2.31: Zoom Grafico RD – Ice – 4cif

Anche qui, la linea grigia, che si trova più in alto tra tutte, è relativa ad una sostituzione a blocchi di dimensione 64x64 pixel. Ora, dato che come si osserva le curve non sono nettamente distanti fra loro, i risultati complessivi per guadagni in rate e qualità sono meglio esplicitati nelle metriche di Bjontegaard, di cui abbiamo già parlato in un paragrafo precedente. Come visibile in tabella di figura 2.32, di media sulle 8 sequenze, abbiamo un guadagno di 0,41% sul bit-rate e di 0,2 dB in PSNR, nel caso di blocchi grandi 128x128 pixel rispetto al caso 256x256.

128x128	Sequences	Frames	BD-rate (%)	BD-PSNR dB
qcif	Carphone	250	-0,33	0,02
	Foreman	300	-0,39	0,02
	Akiyo	300	-0,13	0,01
4cif	City	250	-0,24	0,01
	Ice	250	-1,04	0,04
1280x720	KristenAndSara	200	-0,74	0,02
	Parkrun	220	-0,22	0,01
FullHD	blue_sky	150	-0,18	0,01
	AVERAGE		**-0,41**	**0,02**

Figura 2.32: Tabella di Bjontegaard – caso 128x128

64x64	Sequences	Frames	BD-rate (%)	BD-PSNR dB
qcif	Carphone	250	-0,7	0,03
	Foreman	300	-0,99	0,06
	Akiyo	300	-0,4	0,03
4cif	City	250	-0,81	0,02
	Ice	250	-1,91	0,07
1280x720	KristenAndSara	200	-1,55	0,05
	Parkrun	220	-0,43	0,02
FullHD	blue_sky	150	-0,55	0,02
	AVERAGE		**-0,92**	**0,04**

Figura 2.33: Tabella di Bjontegaard – caso 64x64

Dimezzando poi la dimensione del blocco, è naturale che i guadagni tendano più o meno a raddoppiare. Infatti, la figura 2.33 ci dichiara un 0,92% di bit-rate in meno e fino a 0,4 dB in più per il PSNR. Da sottolineare il caso del video 4cif *Ice.yuv*, per il quale si arriva fino a quasi un 2% in meno di bit-rate con una qualità più alta di 0,07 dB, quando i blocchi sono grandi 64x64 pixel. A questo punto non ci resta che verificare

quanto tempo in più ha impiegato l'intero sistema a svolgere tali sostituzioni per capire quindi quanto convenga o meno diminuire la dimensione dei blocchi ed eventualmente fino a quali valori. Partiamo dal presupposto che, come spiegato nel capitolo dedicato allo studio del MIF-Net, il processo di sostituzione avviene solo dopo che la rete neurale ha migliorato il frame e solo dopo aver inserito i dati in "rec_enhanced.yuv", introdotti poi da HEVC nel flusso di codifica. Questo vuol dire che le sostituzioni avvengono dentro il .cpp (TEncGop.cpp) e non hanno a che vedere con la rete neurale. Di conseguenza le operazioni influiscono solo sul tempo di calcolo del codificatore e non su quello della rete. Possiamo ipotizzare quindi che, almeno per quanto testato in questa fase, le differenze di tempo impiegato non dovrebbero essere notevoli. HEVC fornisce in automatico il risultato sulle tempistiche di codifica stampandolo a schermo. È bastato quindi collezionare i tempi e confrontarli poi tra loro in punti percentuale. Chiaramente tutto dipende dalla sequenza, con i suoi elementi, movimenti e dalla lunghezza. Però il risultato finale ha del curioso: in totale le codifiche con blocchi grandi 128x128 pixel hanno addirittura impiegato lo 0,513% di tempo in meno rispetto al caso con blocchi più grandi. Mentre, per blocchi 64x64, si sale allo 0,061% di tempo in più. Questi valori sono stati calcolati dapprima ottenendo la differenza di tempo in percentuale per ciascuna codifica rispetto al caso 256x256, facendo poi una media sui 4 tempi a diversi QP. Ne risultano 8 differenze di tempo in percentuale, essendo 8 le sequenze, sulle quali è stata poi calcolata la media totale. Il tutto ripetuto sia per il caso 128x128 che per quello 64x64. Va sottolineato che tutto dipende anche dal sovraccaricamento della CPU o GPU in un particolare momento e da piccoli rallentamenti vari che possono o meno incorrere durante la codifica. In generale quindi, con numeri così vicini ed essendo le percentuali così basse, la conclusione che traiamo è che in nessun caso vi è un particolare aumento dei tempi di codifica. Come se, un più alto numero di blocchi da sostituire, per lo meno alle risoluzioni testate, non fosse un grande problema per la macchina sul quale gira l'algoritmo di codifica.

Conclusioni

I test appena descritti sono stati realizzati su un computer con CPU Intel(R) Core(TM) i7-9750H @2,6 GHz, 16GB di RAM e Ubuntu 22.04 (64 bit). Il diminuire delle dimensioni dei blocchi di codifica, come visto, non corrisponde ad un drastico aumento dei tempi, ma ci fa comunque guadagnare piccole dosi di PSNR e bit-rate. In conclusione, allora, si afferma che gli autori di [**7**] avrebbero potuto istanziare la dimensione dei blocchi a valori più piccoli di 256, nel codice sorgente. Questo non avrebbe influito negativamente sulle prestazioni ma, anzi, porterebbe a bitstreams ancora più efficienti. Si può in definitiva pensare di ripetere i test con dimensioni di 32x32 pixel o 16x16. In tal caso, se oltre alla qualità dovessero davvero aumentare i tempi di calcolo, si potrebbe provare a codificare usando macchine più performanti e magari facendo girare il codec su GPU, invece che su CPU. Risolvendo così il discorso legato al carico computazionale.

Terminiamo la sezione mostrando un esempio di sostituzione a blocchi di 64x64 su un frame in FullHD:

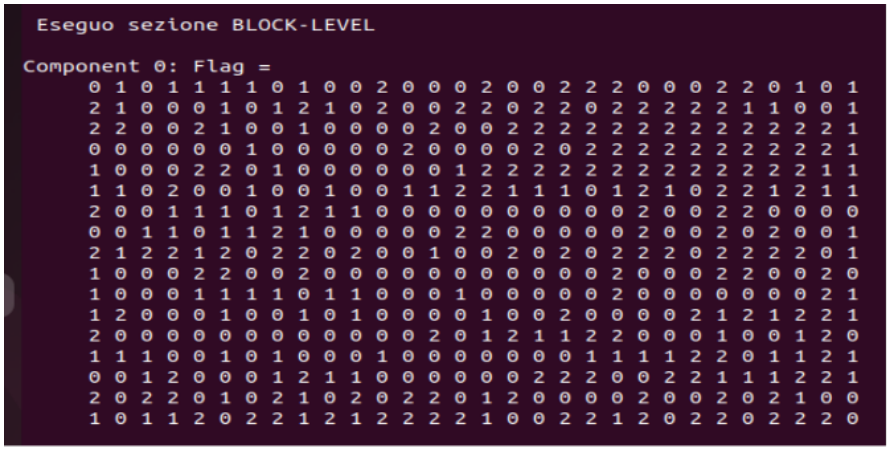

Figura 2.34: Esempio sostituzione e blocchi di 64x64 pixel (FullHD)

2.5 Prendere controllo del server

Arrivati qua, se si volesse cercare di trasportare l'intero lavoro su codec come EVC, in codice C, un ulteriore step da compiere è quello di provare a far lavorare il server da terminale facendogli processare un frame senza interpellare HEVC. In questo modo possiamo essere sicuri di prendere controllo del server facendo lavorare la rete neurale in maniera indipendente.

Allora, si è proceduti inizialmente con una classica codifica di prova, attivando anche HEVC, per generare i file di input. La rete necessita, infatti, di "command.dat", "rec_nof.yuv", "cfg.dat" (dove è indicata la configurazione del Gop), "Info_CuDepth.dat" e "Info_TuDepth.dat", per le partizioni del frame. Se ha questi file a disposizione, la rete saprà che deve migliorare "rec_nof.yuv" e anche con quali parametri.

Nel test in questione si vuole provare a far migliorare un solo frame senza predittori, quindi in "command.dat" sono stati inseriti valori di -1 al posto dei due indici EOC dei fotogrammi di riferimento (tutto funziona anche con una coppia di 0, o con un -1 accompagnato da un numero maggiore o uguale a 0). Dopodiché si attiva il server da terminale lanciando *enhance_one_frame.py*. Esso si accende rimanendo in attesa e, ovviamente, non opera su nulla perché non c'è nessun codice a generare un segnale di "Start". Aprendo un altro terminale allora, basta creare sul momento un nuovo file "pred_start.sig" (nella stessa directory del .py) che viene rilevato dal server avviando cosi la rete per migliorare "rec_nof.yuv" (con i dati del frame di prova).

L'esperimento è stato fatto su un frame della sequenza *carphone_qcif.yuv*, sempre reperibile presso la Video Trace Library, con risoluzione Qcif e QP=37. Come si vede in figura 2.35, dopo essere partita, la rete cattura subito i dati di "command.dat" e "cfg.dat" [4]. Poi va a sfruttare gli altri input nella funzione principale "enhance_one_420yuv_frame".

```
print('Python: Tensorflow initialized.')

while True:
    time.sleep(0.01)
    if os.path.isfile(start_file):
        i_frame_enc, i_frame_disp, frame_width, frame_height, qp_seq_temp, is_init, PQF_EOC_list = get_command(command_file)
        cfg_temp = get_cfg(cfg_file)
        if i_frame_enc >= 0:
            cfg_last = cfg
            cfg = cfg_temp
            qp_seq_last = qp_seq
            qp_seq = qp_seq_temp
            os.remove(start_file)

            if qp_seq != qp_seq_last or cfg != cfg_last:
                if qp_seq < 25:
                    qp_center = 22
                elif qp_seq < 30:
                    qp_center = 27
                elif qp_seq < 35:
                    qp_center = 32
                else:
                    qp_center = 37
                model_file_PQF = 'model_%s_PQF_qp%d.dat' % (cfg, qp_center)
                model_file_non_PQF = 'model_%s_Non-PQF_qp%d.dat' % (cfg, qp_center)
                saver_PQF.restore(sess_PQF, model_file_PQF)
                saver_non_PQF.restore(sess_non_PQF, model_file_non_PQF)
                print('Model of QP %d loaded.' % qp_seq)

            enhance_one_YUV420_frame(sess_PQF, sess_non_PQF, PQF_EOC_list, i_frame_disp, yuv_file_in, yuv_file_out, yuv_file_out_total, file_CU, file_TU, frame_width, frame_height)

            fid_end = open(end_file,'wb') # generate end signal
            fid_end.close()

            n_frame_total += 1
            print('%d frames predicted.'% n_frame_total)
```

Figura 2.35: Codice del Main di enhance_one_frame.py

L'output è salvato in "rec_enhanced.yuv". Ribadiamo che cercare di far funzionare il MIF vorrebbe dire dare alla rete anche altri due frames di riferimento che però non abbiamo (infatti, i valori di -1 o 0 che mettiamo nella lista EOC di "command.dat", indicano l'assenza di fotogrammi predittori). In questo caso, quindi, non avendo frames di predizione, la rete che lavora in realtà non è il MIF bensì l'IF di cui si parla in [**7**]. Il processo funziona, "rec_enhanced.yuv" viene generato e, per conferma, verifichiamo che input ed output siano effettivamente lo stesso frame, uno di qualità maggiore dell'altro. Dalle figure 2.36 e 2.37 è chiaramente visibile la differenza di qualità a favore di "rec_enhanced.yuv": possiamo affermare che su tale argomento è veritiero quanto riportato nell'articolo.

[4]Nella funzione *get_command* di *enhance_one_frame.py*, la seconda parte della condizione "if len(str_arr) == 8 and str_arr[-1] == '[end]':" impedisce la corretta lettura dei valori di "command.dat", nel caso in questione. È bene bypassarla.

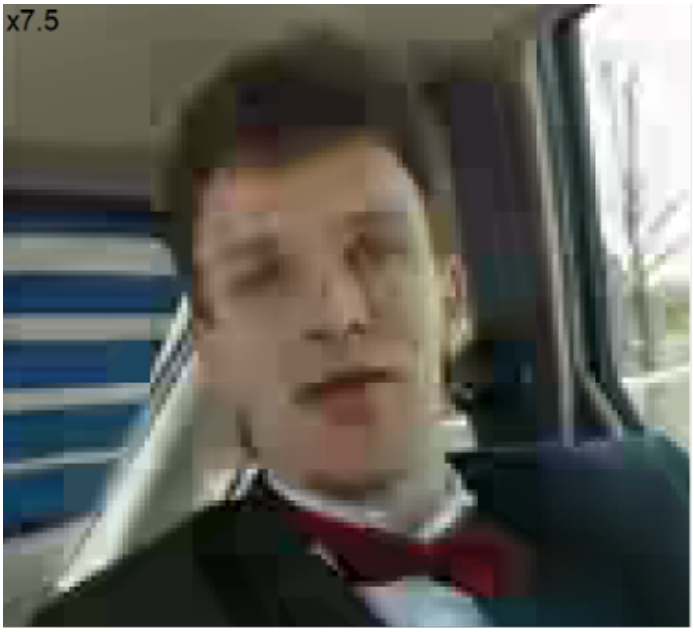

Figura 2.36: rec_nof

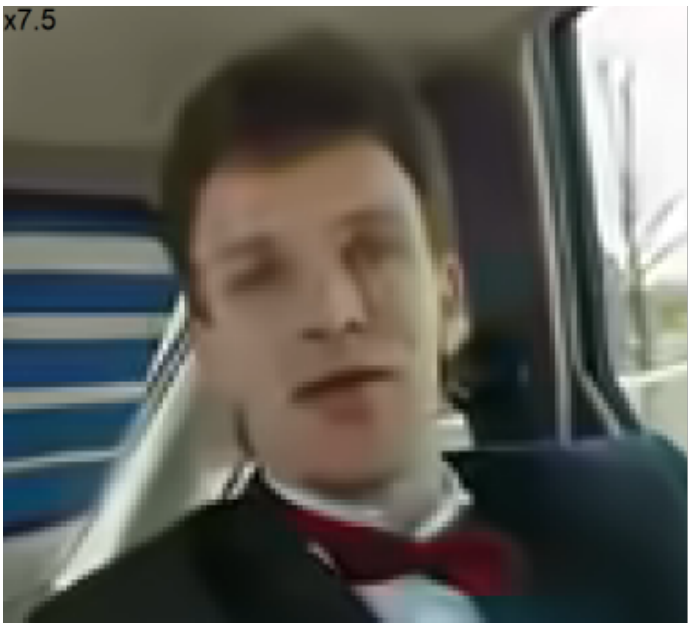

Figura 2.37: rec_enhanced

D'accordo che la rete dia risultati superiori rispetto al metodo Nof in questi piccoli casi di prova, come ovviamente ci si aspetta, ma è anche più prestante dell'approccio standard ILF? Controllando sempre il server da terminale si può far lavorare la rete IF su frames a risoluzioni semplici (dalla Video Trace Library), per poi confrontare la loro qualità con gli stessi frames ricostruiti però dal codificatore standard (HM-16.5 con DBF). Ne è risultato che, per singole immagini a risoluzioni qcif, cif e 4cif, anche se di poco, IF ci restituisce un PSNR maggiore rispetto ai frames codificati e ricostruiti con approccio ILF:

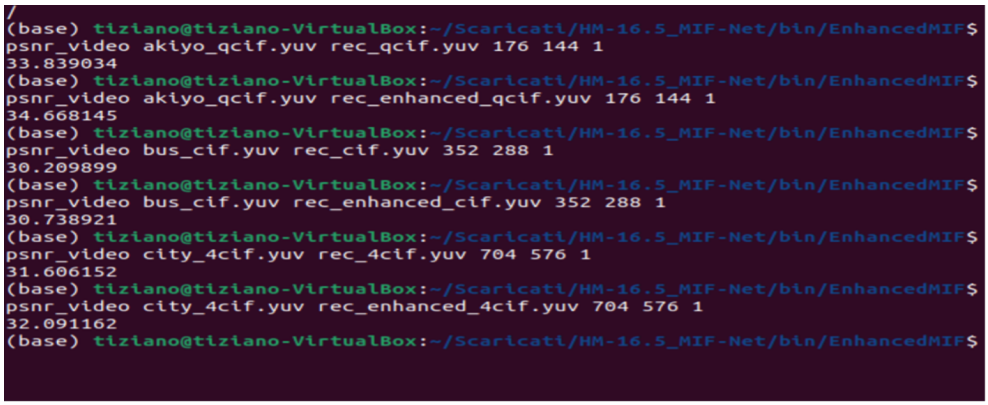

Figura 2.38: Confronti PSNR tra metodo CNN e ILF

Di base, i sistemi di apprendimento migliorano il frame non filtrato e talvolta lo fanno meglio dei filtri standard. Questo, come accennato prima, non avverrà per tutta la sequenza, ragion per cui nel codice è stata inserita la sostituzione a blocchi di cui sopra. Comunque le prestazioni dichiarate in [**7**] sembrano essere rispettate. Per completezza, segue un esempio di confronto tra frame Nof e gli output degli approcci ILF e CNN, "zoomando" su una particolare porzione di pixel presa da un frame di *BasketballDrive.yuv*. Risoluzione FullHD e QP=37:

Figura 2.39: BasketballDrive - Nof (1920x1080)

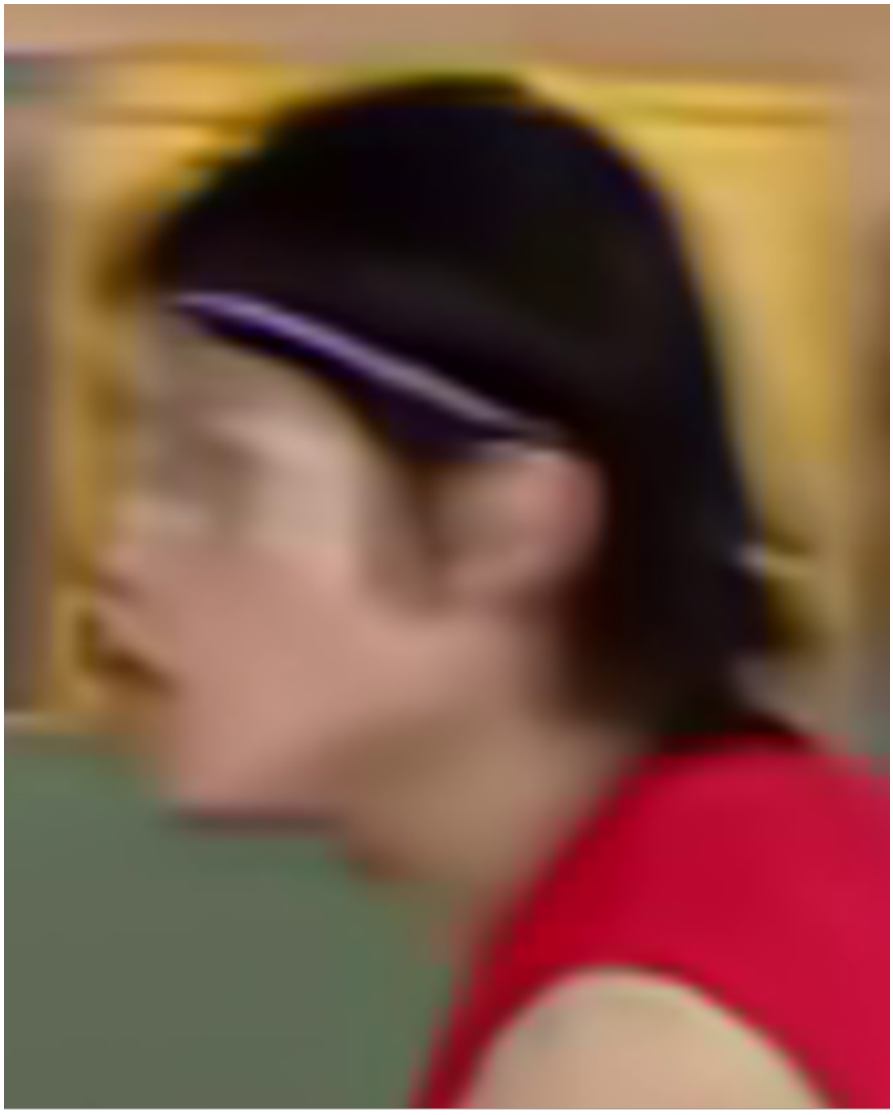

Figura 2.40: BasketballDrive - CNN (1920x1080)

Figura 2.41: BasketballDrive - ILF (1920x1080)

Anche qui la differenza si evince a livello visivo. Soprattutto nel paragone tra frame CNN e frame Nof, nel quale si nota bene il problema degli artefatti. La CNN guadagna anche sul filtro standard ma di poco. Il risultato è comunque più rilevante trattandosi di risoluzione più importante e diffusa a livello broadcast.

Ora sappiamo controllare il server, e quindi le reti neurali, da terminale in maniera indipendente senza dover far ricorso ai codici di HM-16.5. Questo è un ottimo risultato perché cosi è come se "staccassimo" l'approccio da HEVC per poterlo poi quindi riciclare su un nuovo codificatore, come ad esempio EVC.

Capitolo 3

EVC e MIF-Net: primi passi del porting

Nel capitolo conclusivo del presente trattato, ci occupiamo dei primi passi svolti per far sì che il sistema MIF-Net si presti anche a frames codificati da EVC. Noi sappiamo come comunicano server e codificatore e sappiamo controllare il server perché conosciamo quali sono i file che richiede in input. Ci basta allora replicare la stessa quantità di dati facendo sì, però, che il frame ricostruito, non ancora filtrato, ed altri file ad esso associati, provengano da una codifica fatta con EVC.

3.1 Primi problemi riscontrati

Riassumendo, per applicare il *deblocking*, la rete vuole principalmente il frame di tipo Nof, le informazioni sulle partizioni CU e TU ed il "command.dat". Prima di provare a replicare l'intero lavoro, per adesso procediamo a piccoli passi e ci concentriamo semplicemente sul mandare un solo frame EVC al server, che ora sappiamo controllare, ed osservare poi se esso venga o meno migliorato dalla rete. Di conseguenza, in questo caso si parla della semplice rete IF che, preso in ingresso il Nof EVC, lo

migliora senza l'utilizzo di riferimenti passati. Solo dopo, ad avvenuto successo, si può pensare di intervenire sul codice EVC per sfruttare un approccio multi-frame, attivando anche la più grande rete MIF. Per tutto ciò, risultano necessari i file elencati poco fa.

Da qui, appare chiaro che un primo problema riguarda Info_CUDepth.dat e Info_TUDepht.dat, in quanto essi non vengono certamente generati in automatico da EVC. Su HEVC, infatti, erano state aggiunte righe di codice dagli autori di [**7**] per riportare quei dati su file .dat. Tale procedimento andrebbe allora replicato su EVC.

Info_CUDepth.dat e Info_TUDepht.dat contengono al loro interno una serie di valori espressi in esadecimale riferiti ad un particolare "Address", che stanno ad indicare di quanto in profondità va applicato il filtro in una specifica regione di pixel del frame. Ne mostriamo uno *screenshot*:

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000010	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000020	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000030	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000040	01	01	01	01	03	03	02	02	04	04	03	03	03	03	03	03	.....
00000050	03	03	04	03	03	03	03	03	03	03	03	03	03	03	03	03	.....
00000060	03	03	02	02	03	03	03	03	03	03	03	03	02	02	02	02	.....
00000070	02	02	03	03	01	01	01	01	01	01	01	01	02	02	03	03	.....
00000080	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000090	01	01	01	01	01	01	01	01	01	01	01	01	02	02	02	02	.....
000000a0	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
000000b0	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
000000c0	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
000000d0	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
000000e0	01	01	01	01	04	03	02	02	04	04	04	04	04	04	04	04	.....
000000f0	04	04	04	03	03	04	04	04	04	04	03	04	04	04	03	04	.....
00000100	03	04	02	02	03	04	04	04	03	04	04	03	02	02	02	02	.....
00000110	02	02	04	04	01	01	01	01	01	01	01	01	02	02	03	03	.....
00000120	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000130	01	01	01	01	01	01	01	01	01	01	01	01	02	02	02	02	.....
00000140	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....
00000150	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	.....

Figura 3.1: Sezione di esempio di Info_TUDepth.dat – (notepad ++)

Di base, un numero più alto indica quanto più a fondo agirà il filtro in quella regione di pixel.

Si parte da “00” che, in EVC, corrisponderebbe ad una CU grande 64x64 pixel. Mentre “01” indica 32x32, “02” corrisponde a 16x16 e così via. Fino a “04”: CU grande 4x4 pixel. I seguenti *screenshots* mostrano le linee di codice aggiunte in EVC (nel CTU *loop* dentro il metodo “eveye_enc_pic()” di *eveye.c*) utili ad intercettare i dati sulle dimensioni delle singole CU per poi scriverli su file, seguendo le specifiche ed impostazioni del codec EVC:

```
1497        // added
1498        FILE *CuDepth;
1499
1500        int poc = ctx->poc.poc_val;
1501        int addr = core->ctu_num;
1502        int w = ctx->param.w;
1503        int h = ctx->param.h;
1504
1505        int WidthCtu = w % 64 == 0 ? w/64 : w / 64 + 1;
1506        int HeightCtu = h % 64 == 0 ? h/64 : h / 64 + 1;
1507
1508        int x16;
1509        int y16;
1510        for(int x=0; x<4; x++)
1511        {
1512                for(int y=0; y<4; y++)
1513                {
1514                        x16 = 4*(addr%WidthCtu) + x;
1515                        y16 = 4*(addr / WidthCtu) + y;
1516                        if(x16*16 + 16 <= w && y16*16 + 16 <= h)
1517                        {
1518                                int cuw = 1 << core->log2_cuw;
1519                                if (cuw == ctx->min_cu_size) {
1520                                        CuList[y16*(w / 16) + x16] = ctx-
    >min_cu_size;
```

Figura 3.2: Codice C per compilare Info_CUDepth.dat e Info_TUDepth.dat - a

```
1521                                }
1522                                else if (cuw == 8) {
1523                                        CuList[y16*(w / 16) + x16] = 3;
1524                                }
1525                                else if (cuw == 16) {
1526                                        CuList[y16*(w / 16) + x16] = 2;
1527                                }
1528                                else if (cuw == 32) {
1529                                        CuList[y16*(w / 16) + x16] = 1;
1530                                }
1531                                else {
1532                                        CuList[y16*(w / 16) + x16] = 0;
1533                                }
1534
1535                        }
1536
1537                }
1538        }
```

Figura 3.3: Codice C per compilare Info_CUDepth.dat e Info_TUDepth.dat - b

```
1546        if(addr == WidthCtu*HeightCtu - 1)
1547        {
1548                CuDepth = fopen("Info_CUDepth.dat", "w+");
1549                fwrite(CuList, 1, (w/16)*(h/16), CuDepth);
1550                fclose(CuDepth);
1551
1552                CuDepth = fopen("Info_TUDepth.dat", "w+");
1553                fwrite(CuList, 1, (w/16)*(h/16), CuDepth);
1554                fclose(CuDepth);
1555                printf("\nPartizione fatta per 1 frame\n");
1556                free(CuList);
1557        }
1558        // end
```

Figura 3.4: Codice C per compilare Info_CUDepth.dat e Info_TUDepth.dat - c

In pratica, data una dimensione delle CU, stiamo dicendo al codec di scrivere il corrispondente numero dentro i file in esame.

```
370 def read_boundary_frame(fid, width, height, mode, location_mat):
371     # read information of CU/TU partition
372     assert(width % 8 == 0 and height % 8 == 0)
373     if mode == 'CU':
374         unit_width = 16
375     elif mode == 'TU':
376         unit_width = 16 # modified
377     num_line_in_unit = height // unit_width
378     num_column_in_unit = width // unit_width
379     info_buf = fid.read(num_line_in_unit * num_column_in_unit)
380     info = np.reshape(np.frombuffer(info_buf, dtype = np.uint8),
    [num_line_in_unit, num_column_in_unit])
381     # modified
382     info = info.repeat(2, axis = 0).repeat(2, axis = 1)
383     if height % 16 == 8:
384         info = np.concatenate([info, 3 * np.ones([1, np.shape(info)[1]])], axis
    = 0)
385     if width % 16 == 8:
386         info = np.concatenate([info, 3 * np.ones([np.shape(info)[0], 1])], axis
    = 1)
387     info = info.repeat(8, axis = 0).repeat(8, axis = 1)
388     # end
```

Figura 3.5: Modifiche Python della rete neurale per partizioni CU e TU

Inoltre, Info_CUDepth.dat e Info_TUDepth.dat in HM-16.5_MIF-Net hanno dimensioni diverse: ogni CU può essere divisa in ulteriori TU. Mentre dovrebbero apparire uguali secondo la logica di codifica di EVC perché qui non c'è differenza tra le due partizioni. Ragion per cui, a riga 1553, vediamo che, quando i due file vengono generati, in essi inseriamo le stesse informazioni. Quindi alcune righe del codice Python della rete neurale sono state soggette a modifiche per fare in modo che essa riesca

ad accettare ed elaborare dei .dat di dimensione uguale tra loro, non più quindi come quanto succedeva con HEVC. Altrimenti, se non si inserisse "16" al posto di "8", a riga 376 in figura 3.5, e se non si eliminasse la condizione che prima era presente nella riga 381, il sistema restituirebbe "errore" arrestandosi.

Sistemati i primi due, per i restanti file non ci sarebbe molto da dire: "command.dat" può essere scritto e compilato da EVC introducendo le adeguate righe di codice, mentre il Nof lo otteniamo da una codifica in EVC sempre sul primo frame di *KristenAndSara.yuv* (spegnendo il filtro DBF).

```
1536
1537    // command dat
1538    static FILE * fpCommand;
1539    fpCommand = fopen("command.dat", "w+");
1540    fprintf(fpCommand, "%d %d %d %d %d 0 -1_-1 [end]", EOC, ctx->poc.poc_val,
    w, h, ctx->param.qp);
1541    EOC++;
1542    fclose(fpCommand);
1543
```

Figura 3.6: Codice per scrivere su command.dat

In figura le righe di codice aggiunte in EVC per compilare "command.dat", scrivendoci l'EOC, il POC, QP, risoluzione ed il booleano di inizializzazione [**7**]. Usando per ora solo la rete IF, sarà necessario concludere la riga di scrittura ripetendo due volte il numero "-1", che sta ad indicare l'assenza di frames passati come predittori.

Avevamo a disposizione il codificatore "eveye", che replica il funzionamento di EVC di MPEG. Per codificare bisogna lanciare l'eseguibile da terminale passando, nella riga di comando, il nome della sorgente e le specifiche di codifica quali fps, QP, fbe, risoluzione, bit-depth, configurazione RA...

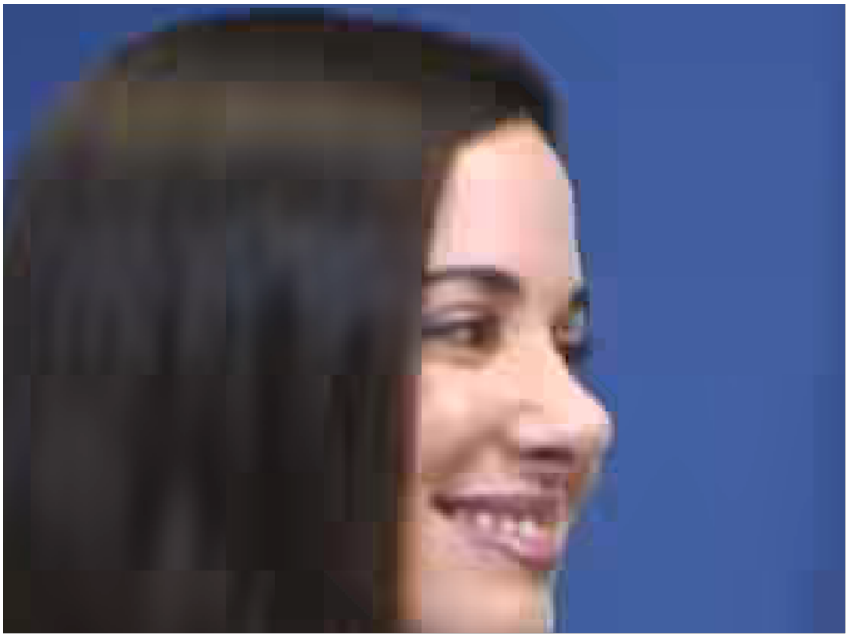

Figura 3.7: Porzione di KristenAndSara, frame Nof

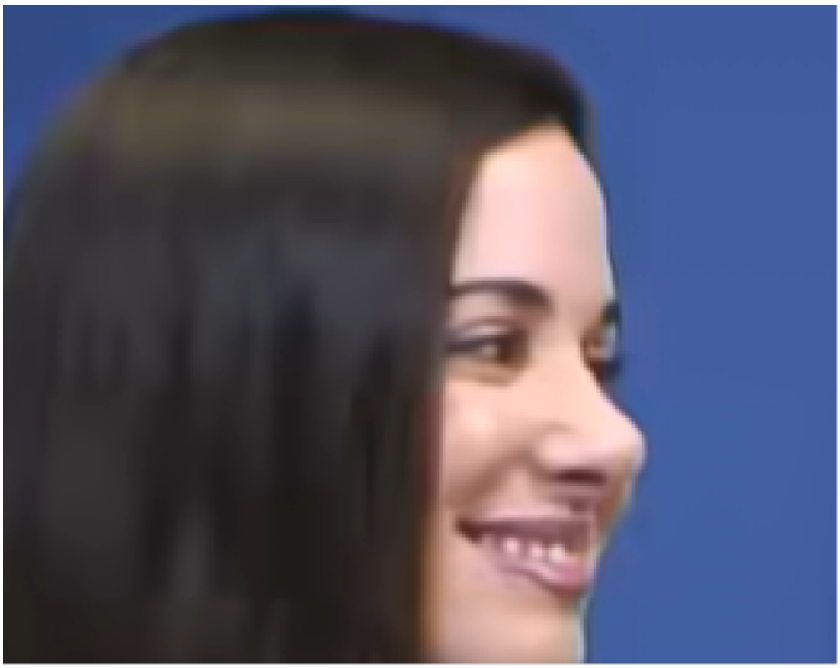

Figura 3.8: Porzione di KristenAndSara, frame migliorato

Una volta ottenuto l'output non filtrato ed i file citati che ora sarà EVC a produrre, abbiamo a disposizione tutto ciò che ci serve per attivare il server e far sì che esso prenda gli input che necessita. Allora, abbiamo proceduto con l'istanziare il segnale di "Start" aspettando poi che l'IF lo rilevasse e facesse il suo lavoro di filtro. La figura 3.7 rappresenta uno zoom su una parte del fotogramma non ancora migliorato della sequenza *KristenAndSara.yuv*. Dalla figura 3.8, invece, si vede chiaramente come il sistema funzioni rimuovendo i blocchi. Identico fenomeno accade anche per un frame di *shields_ter.yuv*.

Figura 3.9: Porzione di shields_ter, frame Nof

Figura 3.10: Porzione di shields_ter, frame migliorato

Ricapitolando: il frame ricostruito ma non migliorato lo otteniamo con codifica EVC spegnendo il DBF, “command.dat” viene scritto dal codificatore stesso, siamo anche in grado estrapolare le partizioni CU del frame e così abbiamo tutto a disposizione per far migliorare l‘immagine dalla rete (IF in questo caso) che si attiva quando creiamo “pred_start.sig”. Dalle immagini appena mostrate appare evidente come vi sia un miglioramento di qualità. Per i dati che riguardano le dimensioni delle CU, potrebbe sembrare che con un numero più alto di profondità il risultato sia necessariamente migliore, andando la rete a lavorare più nello specifico. In realtà non è così: dipendendo tutto dal tipo di frame, dalla sequenza e quindi dagli elementi e movimenti presenti in essa, si avranno risultati diversi all’aumentare o diminuire del valore di profondità. Anche perché, minori sono le dimensioni delle CU, maggiore è il numero delle stesse e, di conseguenza, più codifiche da svolgere e bit-rate più elevato.

3.2 Conversione 10-8 bit

Un altro incidente di percorso fu interpretato dal fatto che la rete lavora con file .yuv scritti su 8 bit, mentre EVC produce un output a 10 bit (LE). Prima di tutto, se si prova ad indicare, nelle specifiche di codifica, di voler generare un Nof a 8 bit, quello che l'eseguibile "eveye" restituisce è un frame errato, dove una metà dell'immagine si ripete due volte sovrapponendosi all'altra. Quindi questa non è la maniera di agire... Navigando nel codice Main Python della rete, scopriamo che alla fine quello che il MF-Net fa è semplicemente leggere valori reali compresi tra 0 e 255 perché, appunto, sono scritti su 8 bit. Ora, l'unica differenza con il caso a 10 bit è che i valori di luminosità e crominanza sono compresi in un range che va da 0 e 1023. Allora, se riuscissimo a far leggere il Nof a 10 bit, prodotto da EVC, alla rete, per poi normalizzare i suoi numeri tra 0 e 255, il risultato dovrebbe essere corretto. All'uscita del MIF-Net poi, si avrà un "rec_enhanced.yuv" a 8 bit.

Questo è ciò che è effettivamente stato fatto prima di procedere con le codifiche EVC ed i test di miglioramento frames del paragrafo precedente. La figura 3.11 mostra le modifiche apportate al .py della rete per ottenere quanto appena descritto.

```
def read_YUV420_frame(fid, height, width):
    d00 = height // 2
    d01 = width // 2
    #added
    yuv_frame = yuvio.imread(fid, width, height, "yuv420p10le")
    #Y_buf = fid.read(width * height)
    #Y = np.reshape(np.frombuffer(Y_buf, dtype=np.uint8), [height, width])

    #U_buf = fid.read(d01 * d00)
    #U = np.reshape(np.frombuffer(U_buf, dtype=np.uint8), [d00, d01])

    #V_buf = fid.read(d01 * d00)
    #V = np.reshape(np.frombuffer(V_buf, dtype=np.uint8), [d00, d01])
    Y = yuv_frame.y
    U = yuv_frame.u
    V = yuv_frame.v
    Y = (Y/1023)*255
    U = (U/1023)*255
    V = (V/1023)*255
    #end
    return FrameYUV(Y, U, V)
```

Figura 3.11: Modifiche al Python della rete

Le righe commentate erano quelle scritte in precedenza. Con esse la funzione leggeva un file .yuv su 8 bit, inserendo i valori nelle matrici Y, U e V. Ora, invece, grazie alla libreria *yuvio*, la funzione legge l'immagine anche se scritta su 10 bit. I numeri dei pixel vengono copiati in Y, U e V e dopo, ogni singolo valore inteso come *float*, viene diviso per 1023 e moltiplicato per 255. Questa normalizzazione permette di passare da 10 a 8 bit senza errori. Così il MIF-Net lavora indisturbato.

3.3 EVC con rete IF

L'obiettivo successivo era quello di far lavorare la rete neurale IF su un normale flusso di codifica EVC (quindi con vere sequenze e non più singoli frames). Vuole dire permettere al codificatore di scrivere il frame corrente su file, che la rete rileva, preleva e migliora, per poi far sì che EVC stesso lo reintroduca nel flusso di codifica, come accade in HEVC con MIF-Net [**7**]. Questo sistema si pensa come sostituto del classico DBF. Il primo passo è stato quello di scrivere le righe di codice che, in EVC, si occupano di prendere i dati Nof e trascriverli su file .yuv.

```
1544    // rec_nof
1545    for (int i=0; i<3; i++)
1546    {
1547        if (i==0)
1548        {
1549            NN_savePredictor("rec_nof.yuv", (*buf_pic)->y, ctx->pic->w_l,
    ctx->pic->h_l, ctx->pic->s_l, 0);
1550        }
1551        if (i==1)
1552        {
1553            NN_savePredictor("rec_nof.yuv", (*buf_pic)->u, ctx->pic->w_c,
    ctx->pic->h_c, ctx->pic->s_c, 1);
1554        }
1555        if (i==2)
1556        {
1557            NN_savePredictor("rec_nof.yuv", (*buf_pic)->v, ctx->pic->w_c,
    ctx->pic->h_c, ctx->pic->s_c, 1);
1558        }
1559
1560    }
```

Figura 3.12: Scrittura del frame Nof su file

La funzione "*NN_savePredictor*" prende il frame (10 bit-LE) per depositarlo nel .yuv di uscita e ciò avviene lungo le 3 componenti. Nelle righe successive vediamo il sistema a "semafori" con il quale EVC crea "pred_start.sig" che serve alla rete, in ascolto su un altro terminale, a

capire che ora può prelevare i diversi file (.yuv e .dat). Come sappiamo già, saranno le nostre precedenti modifiche a far sì che, nel Python della rete, i valori su 10 bit vengano normalizzati tra 0 e 255. Dopo averlo elaborato, IF lo scriverà sull'ormai ben noto file "rec_enhanced.yuv" (8 bit).

```
1561    static FILE* fpParameter, * fpPredStart, * fpPredEnd;
1562    // generate start signal
1563    fpPredStart = fopen("pred_start.sig", "w+");
1564    fclose(fpPredStart);
1565
1566    // wait for Python predicting enhanced YUV frame
1567    while ((fpPredEnd = fopen("pred_end.sig", "r")) == NULL)
1568    {
1569        sleep(0.1); // Linux
1570    }
1571
1572    remove("pred_start.sig");
1573    int iRemoveResult = -1;
1574
1575    while (iRemoveResult < 0)
1576    {
1577        fclose(fpPredEnd);
1578        iRemoveResult = remove("pred_end.sig");
1579    }
```

Figura 3.13: Sistema a semafori

Ora, come si nota nelle righe di codice aggiunte in EVC, lo script in C va a captare la presenza di "pred_end.sig" (istanziato dalla rete neurale a fine processo) capendo quindi che il frame migliorato è presente nella directory e può quindi prelevarlo, come avveniva in HEVC, e reintrodurlo al posto dei dati Nof. Quest'ultima operazione avviene nelle linee mostrate in figura seguente:

```
1677    // y
1678    unsigned char *bufy = (unsigned char *) malloc(w*h*sizeof(char));
1679    int r = fread(bufy, sizeof(unsigned char), w*h, fpYuvEnhanced);
1680
1681    for (int y=0; y<h; y++)
1682    {
1683
1684        for (int x=0; x<w; x++)
1685        {
1686
1687                ctx->pic->y[(y*ctx->pic->s_l) + x] = (pel)
    ((1023/255)*(bufy[(y*ctx->pic->w_l) +x]));
1688
1689        }
1690
1691    }
1692    free(bufy);
```

Figura 3.14: Componente Y di rec_enhanced reintrodotta in EVC

```
1694     // u
1695     unsigned char *bufu = (unsigned char *) malloc(ctx->pic->w_c*ctx->pic-
     >h_c*sizeof(char));
1696     r = fread(bufu, sizeof(unsigned char), ctx->pic->w_c*ctx->pic->h_c,
     fpYuvEnhanced);
1697
1698     for (int y=0; y < ctx->pic->h_c; y++)
1699     {
1700
1701         for (int x=0; x < ctx->pic->w_c; x++)
1702         {
1703
1704                 ctx->pic->u[(y*ctx->pic->s_c) + x] = (pel)
     ((1023/255)*(bufu[(y*ctx->pic->w_c) +x]));
1705
1706         }
1707
1708     }
1709     free(bufu);
1710
```

Figura 3.15: Componente U di rec_enhanced reintrodotta in EVC

```
1711     // v
1712     unsigned char *bufv = (unsigned char *) malloc(ctx->pic->w_c*ctx->pic-
     >h_c*sizeof(char));
1713     r = fread(bufv, sizeof(unsigned char), ctx->pic->w_c*ctx->pic->h_c,
     fpYuvEnhanced);
1714
1715     for (int y=0; y < ctx->pic->h_c; y++)
1716     {
1717
1718         for (int x=0; x < ctx->pic->w_c; x++)
1719         {
1720
1721
1722                 ctx->pic->v[(y*ctx->pic->s_c) + x] = (pel)
     ((1023/255)*(bufv[(y*ctx->pic->w_c) +x]));
1723
1724         }
1725
1726     }
1727
1728     free(bufv);
1729     fclose(fpYuvEnhanced);
1730
```

Figura 3.16: Componente V di rec_enhanced reintrodotta in EVC

In tale fase leggiamo i dati di ogni singola componente inserendoli in un buffer. I cicli *for* a seguire, invece, servono a riapplicare la dovuta conversione inversa così che i valori, disposti su 8 bit in "rec_enhanced.yuv", vengano propriamente riscritti su 10 bit in EVC (passandoli dal buffer intermedio alle strutture "pel" di *ctx->pic*). Per ottenere questo è infatti necessario anche moltiplicare per circa 4 ogni singolo elemento introdotto nei buffer. In pratica la normalizzazione inversa rispetto a quella applicata nel MIF-Net. Il sistema si impegna ad aumentare le prestazioni di codifica, facendo in modo che, reintroducendo il frame migliorato in EVC, esso potrà essere un predittore di più alta qualità per frames successivi.

Spegnendo il DBF, sono state codificate centinaia di frames su 6 sequenze con diverse risoluzioni sfruttando la rete IF come filtro di *deblocking*, per poi ripeterle utilizzando invece il sistema classico (DBF acceso). Le prestazioni sono state messe a confronto tracciando i grafici RD. I risultati che mostriamo da ora in avanti riguardano codifiche svolte sempre in configurazione RA.

3.4 Prestazioni

Ora, la rete che stiamo utilizzando come filtro *in-loop*, si basa come sappiamo su un approccio single-frame. Il che non la rende particolarmente adatta ad elaborare frames codificati *inter*. Infatti, quello che scopriamo sui risultati ottenuti, confrontando il DBF con il metodo IF, si sbilancia a favore del filtro di *deblocking* standard.

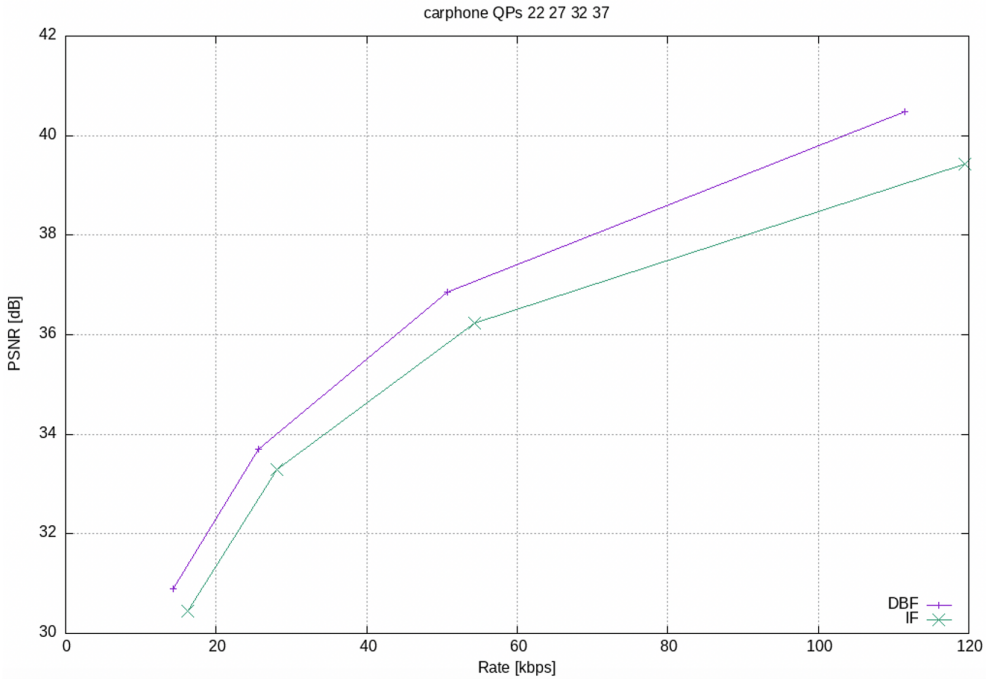

Figura 3.17: Grafico RD – carphone.yuv

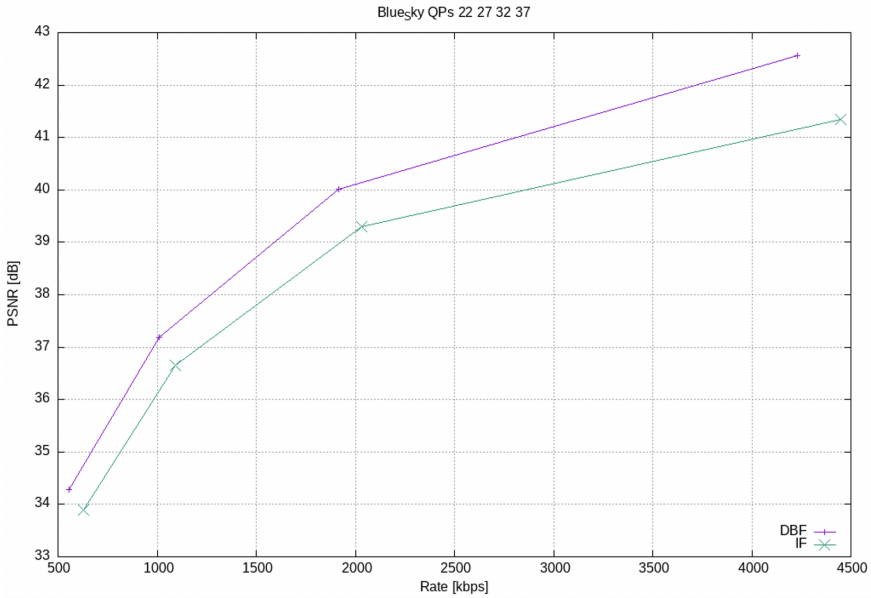

Figura 3.18: Grafico RD – Blue_Sky.yuv

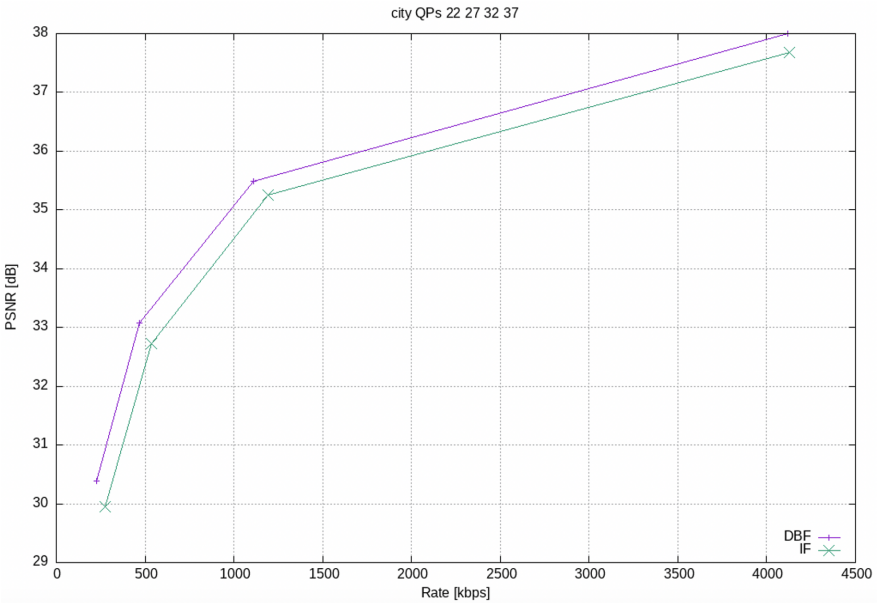

Figura 3.19: Grafico RD – city.yuv

Come si può evincere dai grafici RD, nei quali è indicato il PSNR della componente Y al variare del bit-rate, la curva relativa al metodo con DBF di EVC si trova sempre al di sopra di quella che rappresenta la rete neurale IF. Prima di tutto questo ci dimostra comunque la alta performance del filtro *in-loop* standard di EVC, ormai all'avanguardia. Aggiungiamo però, che la rete IF, come spesso ribadito, non è basicamente adatta a filtrare ogni singolo frame di una sequenza, ma solo quelli privi di buoni riferimenti passati. In particolare, l'atto di reintrodurre il frame filtrato dentro EVC, al posto dei dati non filtrati, potrebbe creare problemi di previsione per il frame successivo e tale dilemma può propagarsi lungo tutta la sequenza. In effetti poi, l'output finale, quando si utilizza IF come filtro *in-loop*, è rappresentato da una sequenza comunque con alta qualità visiva. Ma più semplicemente il bit-rate che ne risulta è molto più elevato rispetto a quello con metodo standard e di conseguenza la curva RD, nel caso di utilizzo di rete neurale, si posiziona più in basso.

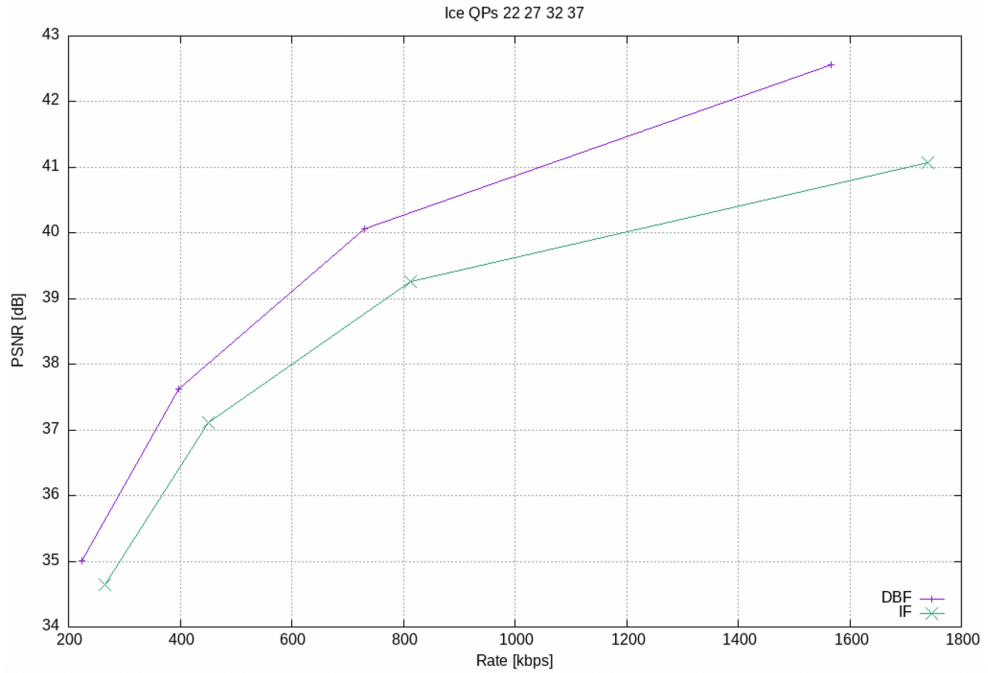

Figura 3.20: Grafico RD – Ice.yuv

Stiamo praticamente dicendo che, per vincere sul DBF, dovremmo implementare un filtro *in-loop* basato su un approccio multi-frame e una seguente sostituzione a blocchi (come in [**7**]) che di certo non farebbe male. Stiamo praticamente ribadendo quanto esposto in [**7**]. Ovvero, si deve fare il *porting* completo dell'intero sistema MIF-Net. Traiamo la conclusione che la rete IF di [**7**] non è un sufficiente approccio single-frame per avere in EVC prestazioni migliori rispetto al suo classico filtro *in-loop* (DBF), con intere sequenze.

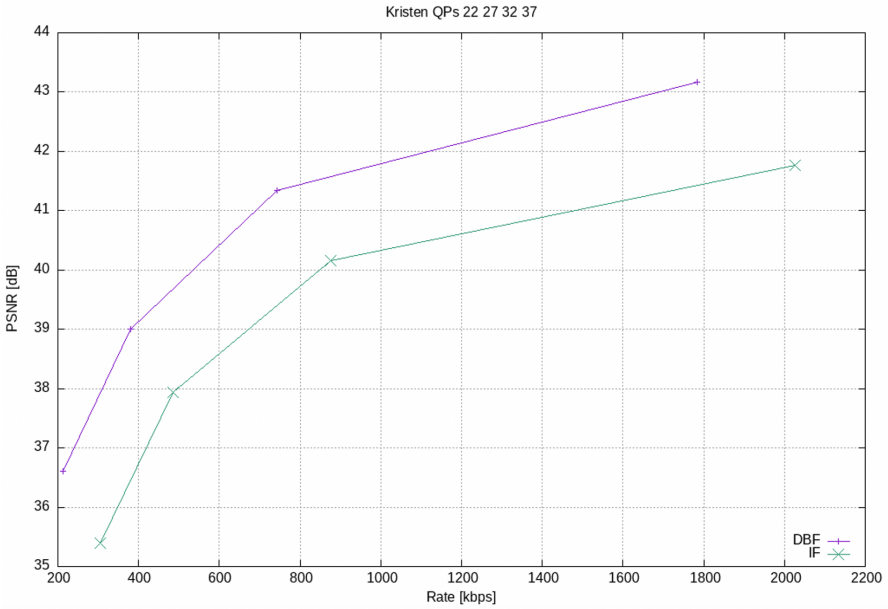

Figura 3.21: Grafico RD – KritenAndSara.yuv

Tuttavia, la nostra rete neurale raggiunge ottimi risultati nel campo per il quale è stata pensata: migliorare frames che non hanno abbastanza riferimenti passati di alta qualità, come ad esempio frames *intra*. Se inviamo ad essa un singolo frame Nof ricostruito, il risultato in uscita è più che positivo, come abbiamo mostrato nella sezione precedente. Anche se confrontiamo tale output con lo stesso frame migliorato però dal DBF, IF mantiene i suoi guadagni. Un ulteriore esempio è mostrato qui di seguito.

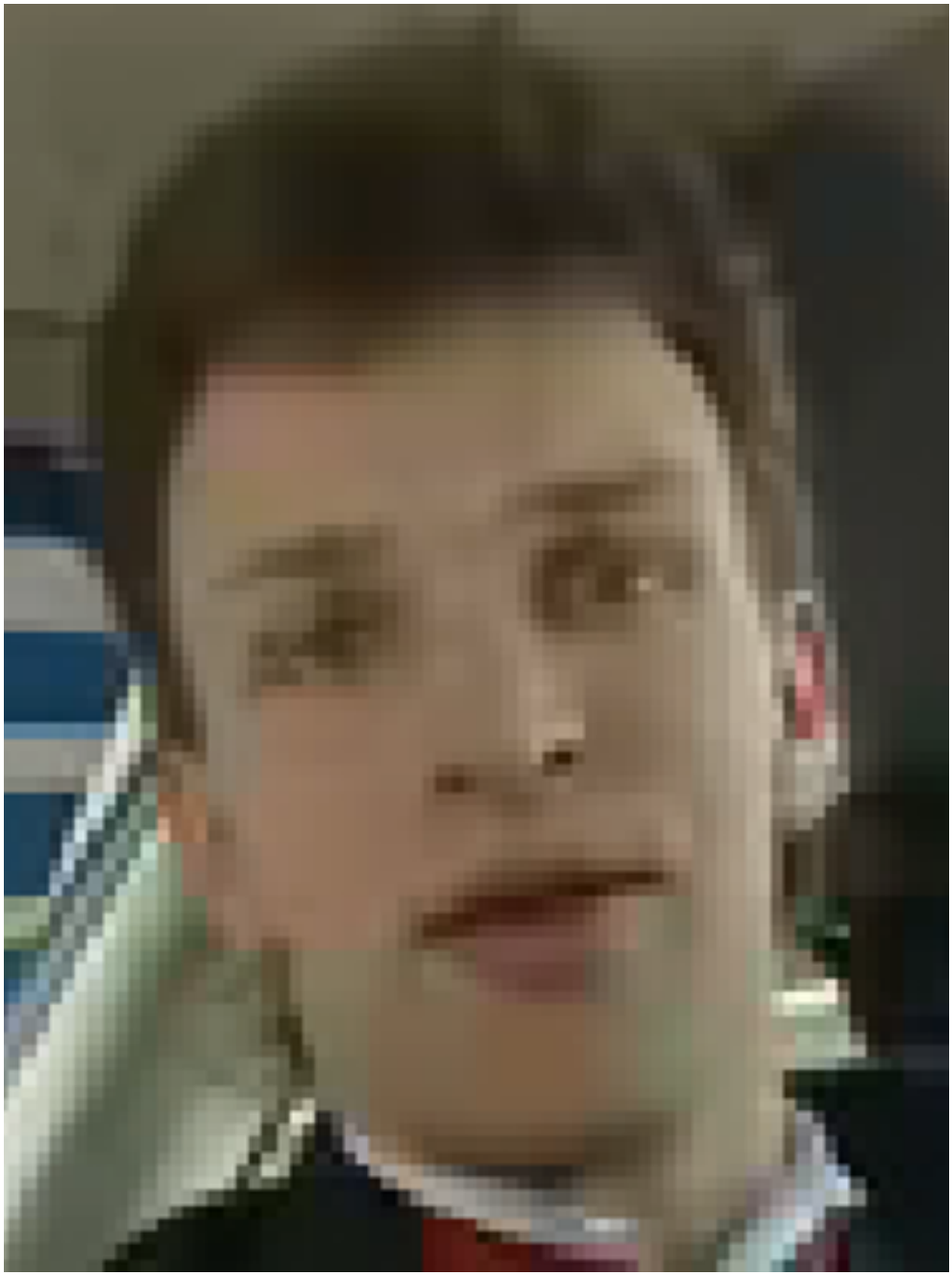

Figura 3.22: Frame ricostruito di carphone.yuv migliorato da rete IF

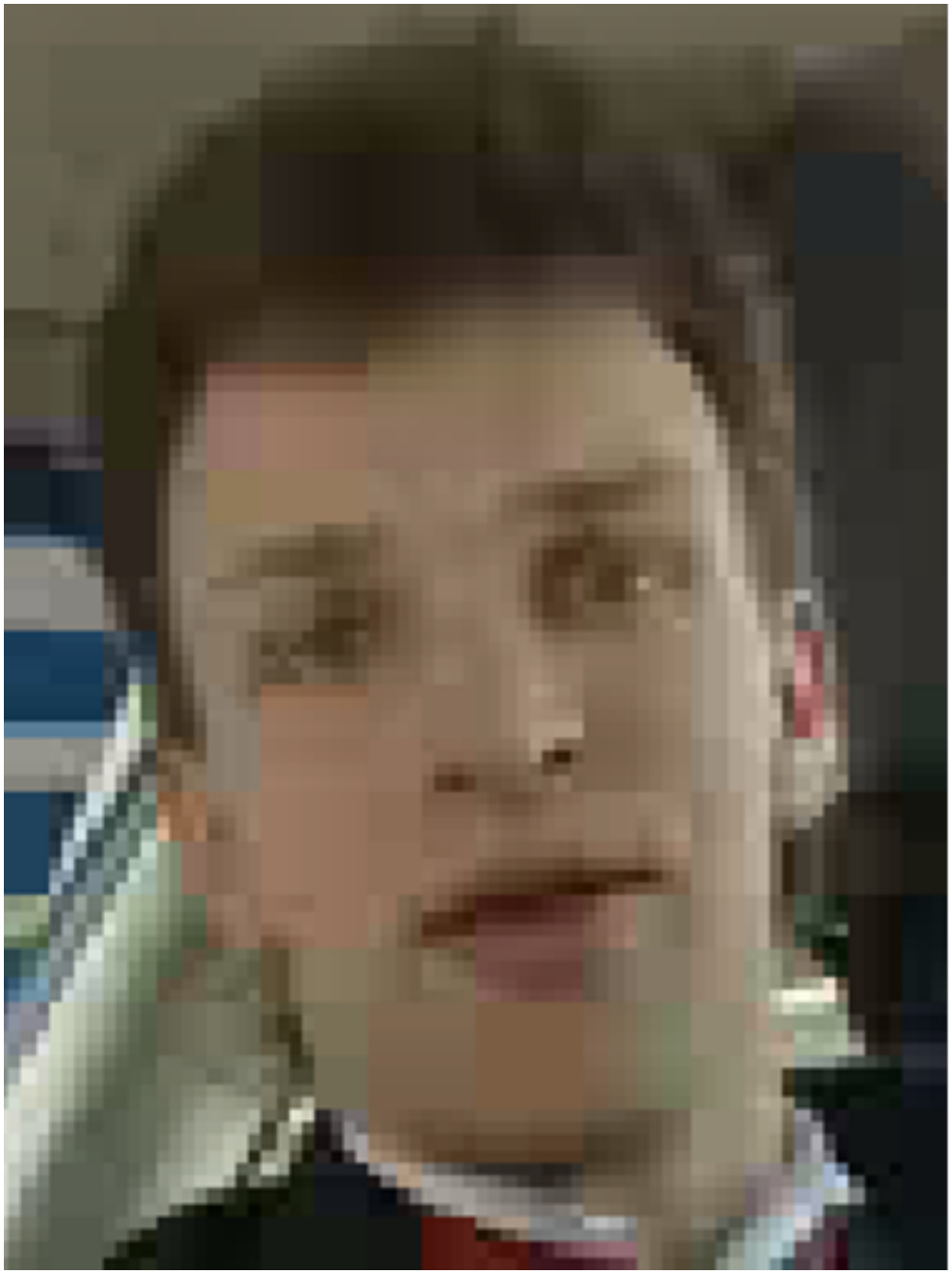

Figura 3.23: Frame Nof di carphone.yuv

Sottolineiamo anche un altro importante risultato ottenuto: la chiusura del *loop* del *porting*. Se volessimo completare il *porting* sfruttando anche la rete multi-frame, di fatto, dovremmo comunque utilizzare il processo in import ed export dati, "da" ed "in" EVC, che già è stato sviluppato. Ricordiamo infatti che, con il metodo "*NN_savePredictor*" scriviamo le informazioni del Nof su file .yuv. Poi istanziamo "pred_start.sig" grazie al quale la rete neurale capisce di poter prelevare il frame da migliorare. Essa produrrà "pred_end.sig" che viene captato da EVC e, a questo punto, il codificatore può leggere i dati in "rec_enhanced.yuv" riscrivendoli nelle sue strutture dati su 10 bit, là dove prima vi era il fotogramma non ancora filtrato. Lo stesso processo si ripete per il frame successivo e così via... Lo chiamiamo sistema a "semafori" e rappresenta la chiusura di un *loop* che collega EVC ed il server dove vive il MIF-Net.

3.5 Prossimi steps

Qui termina la descrizione del lavoro svolto a fronte della presente Tesi. Tuttavia, sappiamo che per un completo *porting* del MIF-Net in EVC manca ancora qualche via da percorrere. In questa sezione elenchiamo quelli che sarebbero i prossimi steps qualora, in studi futuri, si volesse far in modo che anche la rete MIF venga utilizzata da EVC come filtro *in-loop* per tutti quei frame di tipo non *intra*.

- Segnalazione del "Intra period"

Come si accennava nel capitolo "HM-16.5_MIF-Net: testando la rete neurale", tra i vari file in gioco nell'intero processo vi è anche un "frame_period.txt" all'interno del quale è indicato l'*intra period* e, di conseguenza, ogni quanti frames ce n'è uno (*intra*) da non inviare alla rete. Esso verrà migliorato dal DBF dopo la codifica. Per replicare questo stesso ragionamento, si può scegliere un periodo *intra* per ogni codifica EVC e riportare questi numeri in un file di testo con lo stesso nome. Dopodiché, nel metodo "eveye_enc_pic()" di *eveye.c*, potremmo aggiungere linee di codice con le quali, leggendo i numeri scritti in "frame_period.txt", diremo al resto dello script di eseguire il sistema a semafori, e quindi comunicare con la rete neurale, solo per tutti quei frames che non sono di tipo *intra*. Quindi, per quei frames che non sono all'inizio di un *intra period*. Un approccio del genere permette al codificatore di ripartire da zero nei discorsi di codifica e previsione tra fotogrammi ogni volta che inizia un periodo *intra*. Facendo così si riusciranno ad evitare eventuali artefatti d'immagine che spesso vengono causati ed inseriti dalle reti neurali stesse.

- MIF: Multi-frame In-Loop Filter

Ormai è chiaro che, la sola rete IF, non basta per sostituire il DBF di EVC per codifiche di intere sequenze video. Ci aspettiamo che le prestazioni aumentino solo nel momento in cui i frames vengano fatti migliorare dal

MIF-Net nella sua completezza. Per ottenere ciò, sarà necessario aggiungere in *eveye.c* tutte le altre righe di codice che imitano le operazioni di [**7**], introdotte nello script TEncGop.cpp di HM-16.5. In particolar modo mancano ancora le linee che si occupano di verificare quali, tra i frames passati, possiedono il miglior PSNR e miglior CC, così da poter inserire i loro EOC nell'apposita lista e scriverli in "command.dat", per segnalare questa informazione alle reti. Quelle saranno le immagini utilizzate come riferimenti per filtrare il frame attuale. In tale maniera ci si assicura di far funzionare la rete MIF, quando possibile. Qualora fotogrammi passati di alta qualità dovessero mancare, nella lista verranno inseriti valori pari a "-1": il MIF-Net saprà, in tal caso, che è arrivato il momento di utilizzare la semplice rete IF. Se immaginiamo poi di ripetere questi passaggi ad ogni frame, quello che ci aspettiamo è di ottenere prestazioni migliori di quelle ottenute con la sola rete IF e, probabilmente, anche superiori o comunque simili alle performance del DBF standard di EVC.

- Sostituzione a blocchi

Per emulare il lavoro descritto in [**7**] al 100%, a questo punto non manca nient'altro che introdurre la sostituzione a blocchi. È necessario fare in modo che, il frame non filtrato, venga non solo inviato alle reti ma, parallelamente, anche salvato in una struttura dati e poi filtrato dal DBF. Così facendo, dopo aver importato anche i dati migliorati dal MIF-Net nel flusso di codifica, avremmo a disposizione tutti e 3 i tipi di frames: Nof, ILF e CNN. Si dovranno poi aggiungere (anche con un semplice copia ed incolla) tutte quelle righe di codice che in HM-16.5_MIF-Net si occupano di analizzare ogni singolo blocco di pixel, nei 3 approcci, e di scegliere quello con MSE minore. Il blocco vincitore verrà poi inserito nella relativa posizione dentro il frame finale ricostruito che costituirà l'output. Come abbiamo visto nel secondo capitolo, un tale metodo ci assicura di avere in uscita il frame con il miglior PSNR possibile perché costantemente, in ogni blocco di ogni componente di ogni frame, si mettono a confronto sia i dati Nof che quelli ILF e CNN. Quindi anche qualora il frame filtrato dal DBF fosse migliore degli altri in ogni zona di pixel, tutti i suoi blocchi

verranno scelti ed inseriti nell'immagine finale. Perciò, in questa ottica si potrà solo migliorare rispetto al DBF standard e mai fare di peggio. In conclusione, ricordiamo quanto dimostrato nella sezione "MIF-Net e sostituzioni a blocchi di 256, 128 e 64": una dimensione dei blocchi pari a 256x256 sicuramente è efficace ma, visto che non sembra esserci un forte incremento dei tempi di calcolo, varrebbe la pena provare ad introdurre in EVC un sistema di sostituzione a blocchi grandi 64x64 (o anche più piccoli fino a 32x32, 16x16...).

3.6 Conclusioni finali

Giunti al termine delle nostre prove e sperimentazioni, possiamo provare a trarre le conclusioni su quanto testato e calcolato e sugli eventuali studi futuri che sono già stati trattati nel paragrafo precedente. Sicuramente i nostri tentativi di portare il lavoro di una rete neurale all'interno di EVC, per sostituire il filtro di *deblocking*, ci dimostrano quanto già esplicitato dagli autori di [**7**]: per incrementare le prestazioni di un codec, l'approccio migliore consiste in un filtro *in-loop* che sia di stampo multi-frame nel mondo del Deep-Learning. Nei vari capitoli della Tesi, più volte abbiamo avuto occasione di constatare che una rete come il MIF è di gran lunga più efficace di una più semplice coma IF, semplicemente perché i frames di una sequenza non sono quasi mai indipendenti tra loro. Ne consegue che, per migliorarne uno, ha molto senso basarsi su riferimenti passati. Questo permetterà all'intero sistema di filtrare anche i fotogrammi di tipo *inter* con grande efficacia, senza causare problemi di previsione tra frames al codec stesso. Ma soprattutto, grazie proprio all'utilizzo di predittori, una rete multi-frame è capace di migliorare un'immagine limitando il bit-rate, abbassando quindi la relativa curva nei grafici RD.

In definitiva, è la sostituzione a blocchi che ci solleva da qualsiasi dubbio o paura di eventuali cali di prestazione rispetto al filtro standard. Infatti, ripetiamo, qualora una rete neurale dovesse creare artefatti in ogni zona del frame, risultando quindi meno performante del classico DBF, non si avranno particolari problemi grazie alla sostituzione. Il peggio che può succedere è che ogni blocco inserito in output provenga dal fotogramma filtrato dal DBF (ILF) eguagliando quindi la qualità del metodo classico senza reti neurali. Cioè, al limite, non si otterranno miglioramenti ma mai peggioramenti. Questa sostituzione si identifica allora come un'arma di salvataggio nel momento in cui il MIF-Net non dovesse risultare efficace.

Concludiamo il testo nella stessa maniera con la quale lo abbiamo introdotto: lo stato attuale ci fa capire come ormai le reti neurali stiano sempre più prendendo piede all'interno del mondo delle codifiche. Alcune volte con grandi benefici, altre volte meno, ma comunque è un campo ancora tutto da esplorare con diversi eccitanti tentativi da portare avanti. Sappiamo di reti che migliorano la codifica *intra* o *inter* dei codec, reti che incrementano le prestazioni dei filtri *in-loop* e altre che si sostituiscono alle fasi di super-risoluzione. Molte hanno già raggiunto i risultati sperati mentre per altre c'è ancora tanto lavoro da svolgere, soprattutto in termini di tempo di calcolo. Infatti, spesso i moderni supporti non permettono di svolgere carichi di lavoro del genere in maniera veloce. Detto questo, ci aspettiamo comunque un futuro nel quale, la maggior parte delle fasi descritte negli schemi a blocchi dei codificatori, verranno eseguite o affiancate da reti neurali ed intelligenza artificiale. Uno scenario del genere viene continuamente ricercato e favorito anche dal gruppo MPAI, ormai da tempo alle prese col sostituire ogni parte di EVC con una relativa rete neurale che svolga le stesse elaborazioni ma con maggiore qualità.

Bibliografia

[1] Keiron O'Shea and Ryan Nash, "An Introduction to Convolutional Neural Networks", 2 Dec. 2015, https://arxiv.org/pdf/1511.08458.pdf

[2] IBM, "Convolutional Neural Networks", https://www.ibm.com/cloud/learn/convolutional-neural-networks

[3] MIT, Video-Corso MIT 6.S191, "Introduction to Deep Learning", http://introtodeeplearning.com/

[4] Y. Dai, D. Liu and F. Wu, "A convolutional neural network approach for post-processing in HEVC intra coding", in Proc. MMM, Reykjavik, Iceland, Jan. 2017.

[5] Y. Zhang, T. Shen, X. Ji, Y. Zhang, R. Xiong, and Q. Dai, "Residual highway convolutional neural networks for in-loop filtering in HEVC", IEEE Trans. Image Process., vol. 27, no. 8, pp. 3827–3841, Aug. 2018.

[6] Andrea Provino, "Cos'è il bias in una rete neurale artificiale | Towards Machine Learning", 27 Feb. 2019, https://andreaprovino.it/bias-rete-neurale-artificiale-towards-machine-learning/

[7] Tianyi Li, Mai Xu, Ce Zhu, Ren Yang, Zulin Wang and Zhenyu Guan, "A Deep Learning Approach for Multi-Frame In-Loop Filter of HEVC", IEEE Trans. Image Process., VOL. 28, NO. 11, Nov. 2019.

[8] G. Bjontegaard, "Calculation of average PSNR differences between RD curves," document VCEG-M33, ITU-T SG16/Q6, Austin, TX, USA, Apr. 2001

[9] Figura "2" - introduzione: researchgate.net, https://www.researchgate.net/figure/HEVC-H265-Encoder-Block-diagram-6_fig1_322647449

[10] Figura "3" - introduzione: sisvel.com, https://www.sisvel.com/blog/audio-video-coding-decoding/av1-is-a-copy-cat-codec

Ringraziamenti

Con la speranza di esprimere i giusti riconoscimenti senza risultare banale, io dico grazie:

Ai miei genitori, che mi hanno cresciuto come io avrei scelto di crescere, che sempre mi hanno accontentato ma mai viziato e che a me hanno donato l'arte dell'educazione, del rispetto e del pensiero critico. A mio fratello, compagno di stanza, senza il quale mai avrei avuto la maggior parte dei magnifici ricordi immagazzinati fino ad ora.

Ad Andrea, fedele compagno, eletto fra i molti perché in me vede luce anche quando tutto è spento. Ad Eleonora, che continua ad illuminare i miei pensieri semplicemente esistendo. Grazie a Leonardo, tacita certezza, presente se di presenza c'era bisogno. Ad Alessia, ai suoi principi, che non hanno mai smesso di tradursi in ispirazione.

Grazie a Daniele e Luca, da sempre e per sempre, per dimostrarmi il significato di vera amicizia. A Valerio, che in me vede sia un amico che un collega e senza il quale non avrei avuto forza e coraggio di continuare a coltivare i miei sogni. Grazie a Niccolò e Giuseppe, fantastici nel condividere con me l'esperienza Erasmus.

Dico grazie al Professore Maurizio Martina, per aver accettato la mia richiesta di tesi, e a Roberto Iacoviello per essersi incaricato di seguirmi come tutor aziendale.

A Anthony Gonzalez, ormai da anni eterna fonte di spiritualità, per aver prodotto la musica che siede accanto a me ogni giorno, ogni respiro, nel mio presente.

Ai miei cugini, ovunque essi siano.

Ed infine, grazie a Lei, che oggi non è qui. O forse c'è. Ma che comunque un giorno, spero, arriverà.

Ancora grazie, Tiziano.