

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea Magistrale in Fisica

Sviluppo ed ottimizzazione di algoritmi per super-risoluzione ed object detection mediante deep neural network

Relatore:
Prof. Gastone Castellani

Presentata da:
Alex Baroncini

Correlatore:
Dott. Nico Curti

Abstract

Con il progredire delle tecniche di machine learning e deep learning sono stati sviluppati numerosi algoritmi per la costruzione di reti neurali atte a svolgere i più svariati scopi. La super-risoluzione sta ottenendo risultati molto promettenti proprio attraverso l'applicazione di questi metodi. In questo lavoro è proposto un nuovo framework di sviluppo di reti neurali in C++ (Byron, *Build YouR Own Neural Network*) con particolare attenzione all'implementazione di architetture a super-risoluzione ed object detection. Il software sviluppato è interamente parallelizzato su architetture a CPU andando a massimizzarne l'efficienza di calcolo su strutture a server con elevato numero di core. Questa caratteristica lo pone come un nuovo framework di sviluppo applicabile anche a settori di ricerca finora lasciati fuori dal mondo del deep learning poichè privi di acceleratori GPU, come quello della bioinformatica e della bio-medica. Le performance di calcolo di Byron sono state testate e confrontate con le più comuni librerie ottenendo performance superiori nel campo dell'object detection e risultati comparabili nel campo della super-risoluzione.

Indice

Abstract	i
Sigle e abbreviazioni	
Introduzione	1
1 Immagini digitali	5
1.1 Super-risoluzione	6
1.2 Preprocessing	7
1.2.1 Ricampionamento bicubico	8
1.3 Qualità delle immagini	9
1.3.1 PSNR	9
1.3.2 SSIM	10
1.4 Dataset DIV2K	11
2 Reti neurali	13
2.1 Layer	13
2.1.1 Convoluzione	13
2.1.1.1 Implementazione dell'algoritmo	15
2.1.2 Blocchi residui e concatenazioni	18
2.1.3 Attivazioni ReLU	18
2.1.4 Pixel-shuffle	20
2.1.5 Batch normalization	21
2.1.6 Layer YOLO	22
2.2 Modelli utilizzati	23
2.2.1 EDSR	23

2.2.2	WDSR	26
2.2.3	YOLO	29
3	Framework	35
3.1	Darknet	35
3.2	PyTorch e Keras	36
3.3	Byron	37
4	Risultati	40
4.1	Tempi e performance	40
4.1.1	EDSR vs WDSR	40
4.1.2	Numero di core	41
4.1.3	Byron vs Darknet	42
4.1.4	PSNR e SSIM	44
4.1.5	Confronto visuale	44
4.2	Super-risoluzione e object detection	48
4.3	Conclusioni	49
	Ringraziamenti	52
	Bibliografia	53

Sigle e abbreviazioni

ANN	Artificial Neural Network
BYRON	Build Y ou R Own Neural network
DNN	Deep N
EDSR	Enhanced Deep Super Resolution
GEMM	GEneral Matrix Multiply
NTIRE	New Trends in Image Restoration and Enhancement
PSNR	Peak Signal to Noise Ratio
ReLU	Rectified Linear Unit
RGB	Red Green Blue
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SR	Super Resolution
SSIM	Structural SIMilarity
WDSR	Wide Deep Super Resolution
YOLO	You Only Look Once

Introduzione

Le reti neurali artificiali (ANN) sono modelli di calcolo informatico-matematici composti da neuroni artificiali che si ispirano al funzionamento biologico del cervello umano e basati sull’interconnessione delle informazioni.

L’idea di poter replicare artificialmente il cervello, simulandone il funzionamento attraverso delle unità di calcolo, ha una storia che inizia dai primi anni Quaranta del secolo scorso (il primo neurone artificiale fu proposto da W.S. McCulloch e W. Pitts¹). Da allora le ANN si sono sviluppate diventando un fenomeno emergente della realtà odierna ed evolvendosi nelle cosiddette Deep Neural Network (DNN). Esse consistono semplicemente in reti neurali che però hanno degli strati nascosti (comunemente chiamati hidden layer) tra il livello degli input e quello degli output della rete. Questi layer possono essere di vario tipo, a seconda delle funzioni che svolgono.

Il campo di applicazione principale delle DNN è il machine learning, ovvero l’apprendimento di informazioni dall’esperienza guidato da algoritmi matematici adattivi e automatici. In parole poche, le reti neurali possono imparare a risolvere problemi molto complessi, se strutturate e addestrate nel modo giusto. Per fare ciò ovviamente devono prima apprendere: ciò avviene nella cosiddetta fase di training, che può essere di vario tipo:

- **Supervisionato:** all’algoritmo vengono forniti sia i dati in input che i dati in output attesi, in modo che la rete aggiusti i suoi parametri per avvicinarsi sempre di più ad ottenere il risultato desiderato, imparando una o più regole o in generale funzioni molto complesse che collegano una classe di input simili a quello dato con i rispettivi output.

- **Non supervisionato:** al sistema vengono forniti solamente i dati in input, lasciando che la rete stessa trovi qualche connessione logica o schema sottostante alla struttura dei dati. In questo caso gli output della rete non sono sempre di facile interpretazione ed è anche difficile capirne la validità.
- **Semi-supervisionato:** questo è un modello ibrido in cui alcuni dati di input hanno i corrispettivi dati di output attesi mentre altri non sono etichettati. L'obiettivo è sempre quello di identificare le regole per trasformare gli input in modo da ottenere qualcosa il più simile possibile agli output. Si noti che il concetto di “similarità” dipende dalla rete e viene scelto da chi crea il suo modello. Alcuni esempi molto utilizzati sono le GANN (General Adversarial Neural Network).
- **Per rinforzo:** il sistema in questo caso interagisce con un ambiente dinamico e, una volta elaborati i dati in input, deve raggiungere un obiettivo. A seconda del risultato ottenuto verrà fornita una “ricompensa” o una “punizione”, per far capire alla rete in quale direzione sta procedendo. Come anche in tutti gli altri casi, le routine di addestramento vengono ripetute moltissime volte finché la rete non svolge le funzioni desiderate o raggiunge la saturazione di tuning dei pesi.

Ovviamente nel caso in cui la rete smetta di apprendere prima di raggiungere il suo funzionamento previsto, potrebbe essere il caso di cambiare metodo e parametri scelti durante l'addestramento oppure di rivedere la struttura della rete stessa.

Per addestrare (in modo supervisionato) una rete neurale solitamente viene usata una procedura nota come *backpropagation*. Ciò permette di partire da una funzione di errore scelta a priori che valuti quanto siano differenti l'output atteso e quello ottenuto, e propagare all'indietro nella rete questo errore, correggendo i pesi di un layer alla volta. Per fare ciò ovviamente i layer hanno bisogno di funzioni che dicano alla rete come aggiustare i pesi in base all'errore. Solitamente queste funzioni, chiamate appunto backward, sono simili alle rispettive forward utilizzate quando usiamo una rete per inferenza, ma più complesse. Infatti per aggiustare i pesi viene seguita la regola di discesa del gradiente: viene calcolata la derivata della funzione di errore rispetto ai pesi e in questo modo si calcola la funzione di errore per il layer precedente. Questo processo viene ripetuto fino a raggiungere il primo layer della rete, permettendo di aggiustare tutti i pesi per seguire

appunto la direzione decrescente del gradiente per cercare di arrivare a un minimo della funzione di errore.

L’addestramento delle reti neurali esula dallo scopo di questa trattazione e quindi non approfondirò l’argomento. Tuttavia per capire il motivo per cui i layer di cui parlerò servano effettivamente alla rete, è importante parlare del problema della scomparsa del gradiente. Questo fenomeno si presenta durante l’addestramento di reti neurali con molti layer in cui l’errore viene propagato seguendo la regola della discesa del gradiente. In tale metodo, ogni parametro del modello riceve ad ogni iterazione un aggiornamento proporzionale alla derivata parziale della funzione di errore sull’output rispetto al parametro stesso. Solitamente durante il forward, dopo aver calcolato quella che è l’effettiva funzione del layer, viene applicata al risultato una funzione di attivazione (che rappresenta in biologia il *firing* dei neuroni). Le funzioni comunemente usate nelle ANN sono la tangente iperbolica e la funzione logistica, che hanno un gradiente nell’intervallo di valori [0,1]. Ciò significa che durante la *backpropagation* i vari gradienti che vengono moltiplicati per determinare la correzione dei parametri dei primi layer della rete, il cui numero dipende appunto da quanti layer è profonda la rete stessa, tendono a zero. Di conseguenza i layer più vicini agli input sono molto più difficili da addestrare di quelli vicini agli output e ciò può bloccare l’avanzamento dell’apprendimento della rete.

Le soluzioni più comunemente impiegate per ovviare a questo problema comprendono la sostituzione delle attivazioni lineari con attivazioni ReLU², l’utilizzo di blocchi residui³ e l’addestramento con l’algoritmo di Stochastic Gradient Descent (SGD)⁴. Quest’ultimo consiste praticamente nell’aggiustare i pesi della rete solo dopo aver fatto il forward di un gruppo di dati (chiamato *batch*).

In questo lavoro è proposto un nuovo framework di sviluppo di reti neurali in C++ (Byron, *Build YouR Own Neural Network*) con particolare attenzione all’implementazione di architetture a super-risoluzione ed object detection. Il software sviluppato è interamente parallelizzato su architetture a CPU andando a massimizzarne l’efficienza di calcolo su strutture a server con elevato numero di core, solitamente impiegate in settori di ricerca finora lasciati fuori dal mondo del deep learning poichè privi di acceleratori GPU, come quello della bioinformatica e della bio-medica. Le performance di calcolo di Byron sono

state testate e confrontate con le più comuni librerie quali Darknet⁵, PyTorch⁶ e Keras⁷ ottenendo performance superiori nel campo dell'object detection e risultati comparabili nel campo della super-risoluzione.

Capitolo 1

Immagini digitali

Nel mondo del processing di immagini digitali, queste vengono solitamente rappresentate come tensori tridimensionali dove le tre dimensioni rappresentano altezza, larghezza e numero di canali dell’immagine. Ogni elemento del tensore ha un valore numerico che a seconda del formato dell’immagine può essere compreso nell’intervallo [0,255] o nell’intervallo [0,1]. Altri formati meno utilizzati hanno range diversi, ma la scelta di come rappresentare l’immagine digitale è solitamente lasciata all’utente. Byron si appoggia alla libreria OpenCV⁸ per caricare e salvare in memoria velocemente le immagini, ma ha un oggetto proprio per la loro elaborazione una volta che sono state importate, dotato di tutte le funzioni più comuni di elaborazione immagini. La necessità di utilizzo di un oggetto separato da OpenCV è dovuta principalmente alla parallelizzazione: in Byron una sessione parallela viene aperta all’inizio del programma principale e viene chiusa alla fine. Appoggiarsi a funzioni esterne per il calcolo parallelo come per esempio quelle implementate nella libreria OpenCV creerebbe ulteriori sessioni parallele che verrebbero aperte e chiuse ad ogni elaborazione delle immagini. Ciò è da evitare in quanto l’apertura e la chiusura di una sessione parallela richiede un tempo finito in cui bisogna creare i thread e distribuirli sui core o raggrupparli ed è quindi meglio rimanere all’interno di una singola sessione parallela per tutta la durata di esecuzione del programma.

1.1 Super-risoluzione

La super-risoluzione è una tecnica in generale utilizzata per migliorare la risoluzione spaziale di un’immagine. Può essere applicata nella sua forma base solo a immagini che contengono aliasing, ovvero che sono state sotto-campionate o ridotte in dimensione. In queste immagini il contenuto ad alta frequenza dell’immagine ad alta risoluzione (HR) desiderata è nascosto nel contenuto a bassa frequenza dell’immagine a bassa risoluzione (LR) di input. Di conseguenza applicare algoritmi per ripristinare questo contenuto (eventualmente dopo aver ingrandito l’immagine alle giuste dimensioni) consente di riprodurre immagini abbastanza simili all’output desiderato.

I primi metodi di super-risoluzione di immagini digitali consistevano nella stima del contenuto ad alta frequenza tramite associazione di patch di immagine LR con la corrispettiva immagine HR. Queste patch (ovvero riquadri dell’immagine di dimensioni piccole, solitamente inferiori a 50x50), venivano prese dopo aver filtrato l’immagine con un filtro di edge detection (che trova i bordi degli oggetti) o dalla trasformata di Fourier dell’immagine per avere direttamente il contenuto in frequenza. In generale un filtro di elaborazione immagini è un operatore spaziale che esegue delle operazioni su un intorno di un pixel su cui viene applicato. Una volta imparato un “dizionario” di queste associazioni, da un insieme di coppie di immagini HR e LR note, era possibile applicarlo su un’immagine a LR e ottenerne una versione ad alta risoluzione. Si noti che in questo caso le immagini di input e output non avevano necessariamente dimensioni diverse: la risoluzione spaziale infatti dipende non solo dalle dimensioni dell’immagine ma anche dal passo di campionamento.

Nel 2014, grazie al lavoro del Department of Information Engineering dell’Università di Hong Kong, è nata l’idea di utilizzare i layer di convoluzione delle ormai sempre più popolari reti neurali per imparare in maniera automatica un analogo del dizionario delle patch, che in questo caso sarebbe rappresentato da un insieme di moltissime feature. Nacque così la prima ANN per super-risoluzione, chiamata SRCNN⁹ (Super-Resolution Convolutional Neural Network), che consisteva semplicemente in tre layer di convoluzione. Il primo estraeva le patch a LR dall’immagine, il secondo collegava queste patch organizzate in un vettore a molte dimensioni a un altro vettore che idealmente rappresentava le patch HR.

Infine l'ultimo layer riorganizzava le patch ad alta risoluzione e le combinava per ottenere le immagini HR di output.

Da allora sono stati fatti molti progressi nella ricerca in questo campo, ma le idee fondamentali non sono cambiate. Semplicemente i modelli hanno molti più layer, grazie alla sempre crescente potenza di calcolo dei computer odierni, e utilizzano quindi contromisure per riuscire a gestire l'addestramento con un numero enorme di parametri.

1.2 Preprocessing

Le reti per super-risoluzione seguono un metodo di addestramento supervisionato ed hanno quindi bisogno di un dataset di immagini di input e di output attesi. Nel caso della super-risoluzione l'immagine di input consiste di una versione ridimensionata dell'immagine di output, a bassa risoluzione. Questo introduce un fattore di aliasing che la rete dovrà quindi imparare a nullificare quando ingrandisce l'immagine, ripristinando i contenuti ad alta frequenza e quindi la risoluzione dell'output ottenuto. Queste reti sono molto grandi e hanno un numero di parametri dell'ordine 10^7 : di conseguenza se l'input fosse modificato con dei filtri per simulare i problemi realistici delle immagini reali come rumore e sfocature, la rete imparerebbe in parte anche a risolvere questi problemi.

Per le analisi svolte in questa tesi ho considerato solamente un fattore di scala di 4 e un metodo di riduzione dell'immagine bicubico. Inoltre seguendo la procedura standard viene rimossa la media (RGB) del dataset utilizzato durante l'addestramento prima di processare un'immagine con la rete, e dopo aver ottenuto l'immagine di output viene sommata nuovamente. Altre accortezze come riscalare tutti i valori dell'immagine per normalizzarli, che dipendono dal modello della rete e dai pesi usati, sono state tenute in considerazione per avere risultati ottimali.

1.2.1 Ricampionamento bicubico

Il ricampionamento bicubico, che solitamente viene diviso in *upsampling* e *downsampling* o analogamente chiamati *upscaleing* e *downscaleing*, consiste in un metodo di interpolazione per determinare i valori dei pixel di un’immagine dopo che questa è stata cambiata di dimensioni (rispettivamente aumentata o diminuita). Il nome deriva dalla complessità massima dell’algoritmo di interpolazione usato, in cui l’operazione più complicata eseguita in questo caso è appunto il cubo del valore di un pixel. L’interpolazione viene eseguita in un intorno di 4 pixel. In generale i filtri scelti per il ricampionamento bicubico appartengono a una famiglia con la seguente forma:

$$k(x) = \frac{1}{6} \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + \\ \quad +(6 - 2B) & \text{se } |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + \\ \quad (-12B - 48C)|x| + (8B + 24C) & \text{se } 1 \leq |x| < 2 \\ 0 & \text{altrimenti} \end{cases}$$

Tra i più popolari cito le opzioni $B=0$, $C=0.75$ usata dalla libreria OpenCV⁸ o $B=0$, $C=0.5$ solitamente chiamato filtro di Catmull-Rom usato dalle librerie di Matlab¹⁰. Byron utilizza di default l’opzione $(0, 0.75)$ ma può essere agevolmente cambiata secondo necessità. Inoltre in Byron ho anche implementato il filtro di Lanczos, che garantisce risultati migliori (soprattutto quando si parla di upsampling dove si vede effettivamente il miglioramento di qualità rispetto a un upsample lineare) e ha un intorno di 8 pixel.

Per applicare questi filtri bisogna ridimensionarli in modo che siano larghi quanto 4 pixel dell’immagine più piccola (quindi quella di partenza nel caso dell’upsampling e quella ridimensionata in caso di downsampling). Poi per ogni pixel dell’immagine obiettivo, bisogna calcolare tutti i contributi dei pixel che rientrano nel range del filtro per il pixel obiettivo e pesarli per il rispettivo valore del filtro a quella distanza dall’origine. Bisogna solitamente anche normalizzare i pesi del filtro in modo da non avere un aumento o una diminuzione della intensità complessiva dell’immagine.

1.3 Qualità delle immagini

Per valutare la bontà delle immagini SR (super-resolution) in output dalla rete bisogna confrontarle con le originali HR prima che esse venissero ridimensionate. Ci sono varie misure della somiglianza tra immagini, tra cui PSNR e SSIM.

1.3.1 PSNR

Il peak signal to noise ratio (PSNR) è una misura che solitamente viene adottata per misurare la bontà di compressione di un'immagine rispetto all'originale. Matematicamente viene definito come:

$$PSNR = 20 \cdot \log_{10} \left(\frac{\max(I)}{\sqrt{MSE}} \right)$$

dove $\max(I)$ è il massimo valore assumibile dai pixel dell'immagine, solitamente 1 per immagini con valori decimali e 255 per immagini con valori interi. MSE è il Mean Square Error e indica la discrepanza quadratica media fra i valori dell'immagine super-risoluta ed i valori dell'immagine originale, viene definito come:

$$MSE = \frac{1}{WH} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \|I(i, j) - K(i, j)\|^2$$

dove W , H sono rispettivamente larghezza e altezza dell'immagine e I , K sono rispettivamente immagine originale e immagine super-risoluta.

Il PSNR è quindi il rapporto tra la massima potenza del segnale e il rumore di fondo. Viene solitamente espresso in decibel (dB) perchè le immagini hanno un intervallo dinamico ampio e quindi avere una scala logaritmica rende i numeri più gestibili. Il PSNR è uno degli indici di qualità più popolari tra le immagini¹¹, anche se non sempre ha un collegamento diretto con una qualità visuale percettibile dall'occhio umano. Faccio notare che sia EDSR che WDSR sono state addestrate per massimizzare questo valore (e quindi la verosimiglianza all'immagine originale). Una funzione diversa per l'ottimizzazione, per

esempio la cosiddetta *visual loss* che dovrebbe essere una misura della qualità visuale percepita dall'occhio umano, può dare risultati visivamente migliori anche se con PSNR peggiori. La scelta del criterio per misurare la qualità dell'immagine e di conseguenza la funzione da ottimizzare per la rete rimane al giorno d'oggi un dibattito aperto e con varie opzioni valide.

1.3.2 SSIM

Un altro indice di qualità delle immagini molto usato è il *Structural SIMilarity index* (SSIM), una funzione molto complessa che cerca di valutare la somiglianza strutturale tra due immagini e che tiene anche conto del miglioramento visivo valutabile dall'occhio umano. Nella figura 1.1 è illustrato il diagramma dei calcoli necessari per ottenere il SSIM tra due immagini. Matematicamente può essere espresso come:

$$SSIM(I, K) = \frac{1}{N} \sum_{i=1}^N SSIM(x_i, y_i)$$

dove abbiamo N box dell'immagine di dimensioni arbitrarie, solitamente 11x11 o 8x8. Per ogni box il SSIM è calcolato come:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

dove μ rappresenta la media, σ^2 la varianza, σ_{xy} la covarianza, c_1 e c_2 due parametri fissati per evitare divergenze al denominatore. Per calcolare il SSIM tra le immagini ho usato l'apposita funzione dalla libreria di Python per elaborazione ed analisi immagini *scikit_image*¹².

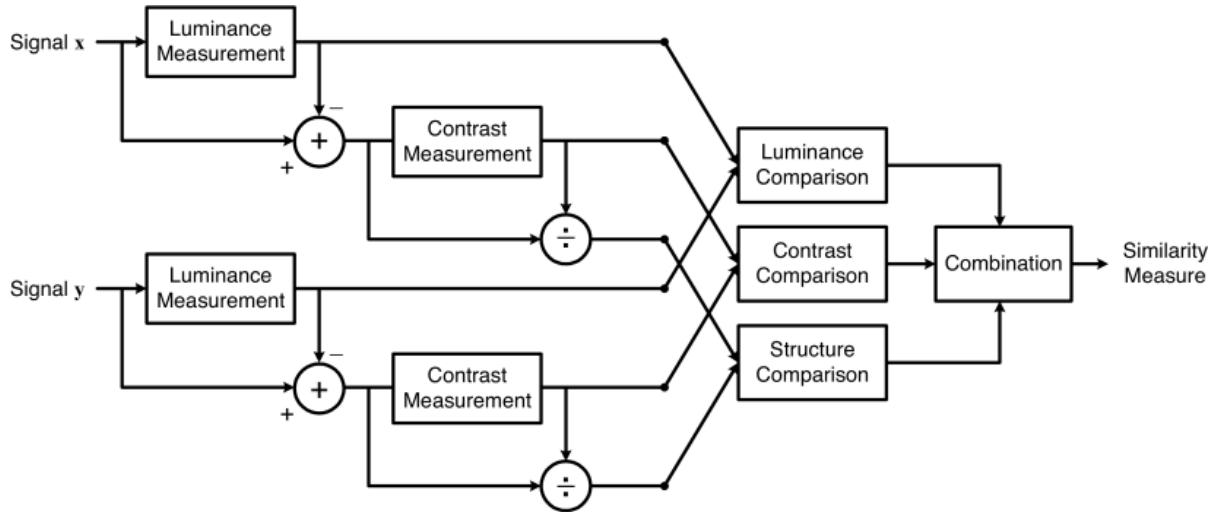


Figura 1.1: Diagramma di flusso delle istruzioni necessarie per il calcolo dell'indice SSIM tra due segnali x e y .

1.4 Dataset DIV2K

Il dataset utilizzato durante l’addestramento e il test delle reti analizzate in questa tesi è il DIV2K¹³ (DIVerse 2K resolution high quality images), composto da 1000 immagini a risoluzione 2K. Questo dataset è stato realizzato appositamente per la NTIRE (New Trends in Image Restoration and Enhancement) challenge del 2017 ed è stato poi riutilizzato anche per le NTIRE degli anni successivi. La NTIRE challenge prende luogo ogni anno durante la conferenza CVPR (Computer Vision and Pattern Recognition) e ha come obiettivo l’avanzamento dello stato dell’arte nel campo dell’elaborazione di immagini digitali per vari compiti, tra cui la super-risoluzione. In questa gara vari team di ricerca propongono modelli di network con l’obiettivo di migliorare la risoluzione di un’immagine ridimensionata in vari modi. La squadra che riesce ad avere il PSNR medio sulle immagini di validazione del dataset DIV2K più alto si aggiudica il primo posto. I metodi di downsampling delle immagini cambiano ad ogni anno, ma comprendono generalmente il ricampionamento bicubico e un ricampionamento che simula l’acquisizione immagini di una fotocamera digitale.

Il dataset DIV2K è diviso in:

- Training: 800 immagini HR per l’addestramento delle reti con rispettive versioni riscalate (LR) con vari fattori di scala (2, 3 e 4) e con metodi di ricampionamento

diversi;

- Validation: 100 immagini HR con rispettive versioni LR che vengono usate per la validazione e il test dai team per provare, controllare e migliorare il loro modello durante la gara;
- Test: 100 immagini LR su cui i team devono fare i test con il loro modello finale. I risultati SR su queste immagini verranno confrontati con le originali HR che non sono a disposizione del pubblico per valutare i concorrenti e stabilire il vincitore.

Capitolo 2

Reti neurali

2.1 Layer

I layer che compongono le reti neurali possono essere di varia natura, a seconda delle funzioni che devono svolgere. Nella libreria Byron ho implementato tutti quelli più comunemente utilizzati al momento, ma per lo scopo di questa tesi mi concentrerò solo su quelli necessari alla super-risoluzione e alla object detection.

2.1.1 Convoluzione

La convoluzione è un'operazione matematica tra due funzioni che consiste nell'integrare la prima funzione con la seconda traslata di un certo valore. Viene indicato con convoluzione sia il risultato dell'operazione che l'operazione stessa. Nel campo dell'elaborazione immagini, la prima funzione è data dall'immagine mentre la seconda da un filtro che scorre su di essa, e viene utilizzata una versione discreta invece che continua. Matematicamente l'operazione di convoluzione discreta in due dimensioni è formulabile come:

$$C = f * I$$
$$C[i, j] = \sum_{u=-k}^k \sum_{v=-k}^k f[u, v] \cdot I[i - u, j - v]$$

dove C è il risultato della convoluzione, f è il filtro di dimensioni $k \times k$ e I è l'immagine. $C[i, j]$ è il valore di un singolo pixel dell'immagine risultante dall'operazione. Nel caso di un'immagine a più canali, anche il filtro deve avere lo stesso numero di canali e il risultato della convoluzione sarà semplicemente la somma dei risultati di tutti i canali. Nel processing di segnali questa operazione è in realtà chiamata correlazione incrociata. L'unica differenza tra le due operazioni è che nella convoluzione il filtro viene rovesciato prima di essere applicato all'immagine, ma visto che i pesi del filtro vengono aggiustati dalla rete durante l'addestramento, questo rovesciamento è superfluo. Di conseguenza nel campo delle ANN si usa il termine convoluzione intercambiandolo con correlazione incrociata.

Di base quindi abbiamo come parametri le dimensioni dell'immagine e quelle del filtro (entrambi rappresentati come tensori numerici tridimensionali). Nel campo del deep learning poi vengono solitamente usati alcuni parametri addizionali, tra cui:

- *Pad*: Definisce se aggiungere dei pixel ai bordi dell'immagine prima di applicare la convoluzione. Solitamente vengono aggiunti pixel neri, dei pixel con valori riflessi rispetto al bordo o dei pixel con i valori circolari come se l'immagine fosse arrotolata su sè stessa. Ciò permette di gestire le dimensioni dell'immagine in output dalla convoluzione e le condizioni al contorno dell'operazione.
- *Stride*: Definisce se applicare il filtro su tutta l'immagine o se saltare alcuni pixel. Per esempio uno stride di 2 equivale a applicare il filtro prendendo solo 1 pixel dell'immagine ogni 2. Questo parametro permette di ridurre le dimensioni dell'immagine in output.
- Numero di filtri: Ogni filtro deve avere tanti canali quante le dimensioni delle immagini in partenza. Il numero di canali dell'output dipenderà dal numero di filtri che scegiamo di applicare.

I parametri della convoluzione vengono scelti prima dell'addestramento e di solito sono invariabili e caratteristici della struttura della rete. Quello che invece la rete impara e modifica durante l'addestramento sono i valori nei filtri, solitamente chiamati pesi:

ciò permette di insegnare alla rete ad applicare trasformazioni anche molto complesse all’immagine in input, a seconda del numero di filtri e di pesi aggiustabili.

2.1.1.1 Implementazione dell’algoritmo

Applicare direttamente la formula della convoluzione passando il filtro su tutta l’immagine è un processo molto lento, e in genere i layer di convoluzione occupano la maggioranza del tempo di calcolo delle reti di elaborazione immagini. La complessità computazionale per la convoluzione diretta su un’immagine di dimensioni $H \times W \times C$ con un filtro di dimensioni $k \times k$ è un $O(HWCk^2)$. Per migliorare le performance ci sono vari algoritmi di convoluzione che cercano di ottimizzare diversi aspetti dell’operazione. Uno dei più utilizzati, che ho implementato in Byron, consiste in due fasi: Im2col (image to columns) e GEMM (GEneral Matrix Multiply).

L’Im2col è una trasformazione che “appiattisce” l’immagine originale trasformandola in una enorme matrice, dove ogni colonna contiene tutti gli elementi a cui deve essere applicato un singolo filtro in un singolo step. Il numero di colonne di questa matrice dipende quindi da quante volte il filtro deve scorrere sull’immagine per coprirla interamente. Questa fase consiste solo nella copia di dati dell’immagine in una matrice delle giuste dimensioni, dove alcuni elementi sono solitamente ripetuti perché rientrano nelle finestre di applicazione del filtro più volte man mano che si sposta. Di conseguenza non richiede calcoli (escluse eventuali conversioni tra gli indici) ma solo copie di dati in memoria, ed ha una complessità $O(HWC)$. Tuttavia è molto onerosa dal punto di vista della memoria, in quanto l’immagine trasformata è sempre molto più grande dell’originale. Ciò non è un problema nell’ottica di implementazione di Byron, in quanto i server e i computer tipicamente usati in bioinformatica hanno grandi quantità di memoria disponibile.

Una volta eseguito l’Im2col sull’immagine, per ottenere il risultato della convoluzione è necessario moltiplicare tra di loro le matrici dell’immagine appiattita e del filtro. I filtri vengono anche essi appiattiti in modo da essere una matrice in cui ogni riga è la riorganizzazione unidimensionale di un filtro, e avremo quindi tante righe quanti i filtri da applicare. Questo passaggio costituisce il vero vantaggio di questo modo di eseguire

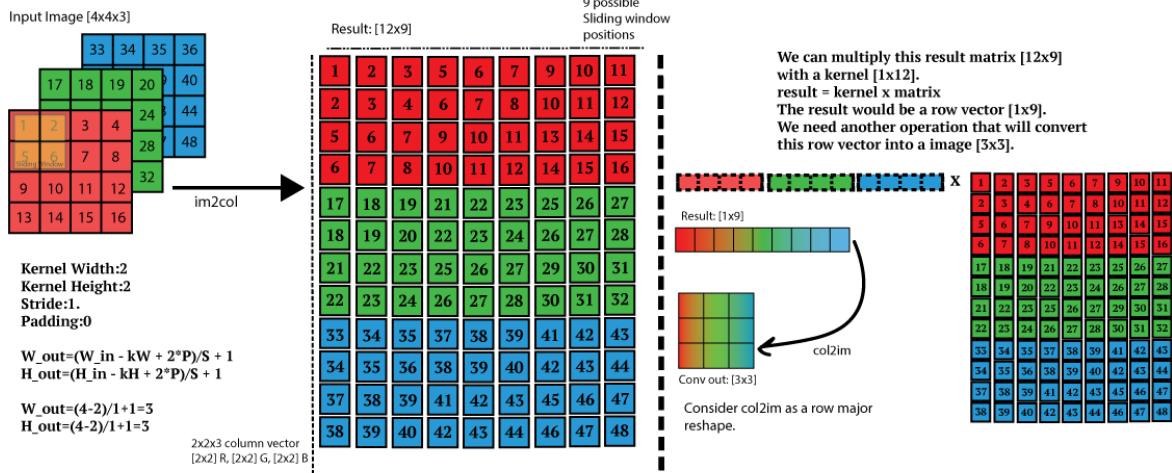
la convoluzione, in quanto per molto tempo anche prima dell'avvento delle reti neurali l'operazione di moltiplicazione tra matrici è stata ottimizzata per essere il più efficiente possibile nell'utilizzo della memoria cache del processore. Per fare ciò è necessario seguire una serie di tecnicismi, tra cui per esempio preservare il Single Instruction Multiple Data stream (SIMD) del processore: ogni processore moderno ha infatti una memoria chiamata cache (solitamente divisa in vari livelli chiamati L1, L2 etc) molto piccola ma molto vicina al processore che permette di risparmiare tempo rispetto all'accesso alla RAM. Su questa memoria è possibile eseguire la stessa istruzione in parallelo su tutti gli elementi, il che permette un certo grado di vettorizzazione (che dipende dal processore) a livello di singolo core. Essendo la memoria cache molto piccola, saper gestire attentamente gli accessi e l'ordine dei calcoli che vengono eseguiti in essa permette di ottenere dei buoni speedup¹⁴ a parità di complessità computazionale dell'operazione di GEMM, che è solitamente un $O(N^3)$ per matrici $N \times N$. Inoltre il tempo di copia dei dati per l'Im2col rispetto al tempo di calcolo necessario per la GEMM diventa trascurabile per matrici grandi, implicando un'alta intensità aritmetica (cioè vengono svolte molte operazioni al secondo).

Ovviamente poi le operazioni necessarie per le moltiplicazioni tra matrici vengono divise tra i vari core, garantendo la parallelizzazione massima possibile e dividendo così il carico di lavoro in parti più piccole: ogni core si occuperà di calcolare il risultato per un singolo filtro. Per questo motivo solitamente il numero di filtri dei layer di convoluzione delle reti neurali è una potenza di 2, come anche solitamente il numero di core su un computer. A questo proposito è facile notare come le GPU siano così superiori alle CPU in termini di performance nel campo delle reti neurali: sebbene abbiano core molto meno potenti, sono immensamente più numerosi e generalmente parlando la struttura hardware delle GPU è molto più specifica e pensata appositamente per favorire il calcolo parallelo. Ne consegue che operazioni come la convoluzione risultano molto più veloci sulle GPU rispetto alle CPU, a parità di algoritmo¹⁴.

In figura 2.1 è rappresentato uno schema dell'operazione di Im2col seguita dalla GEMM.

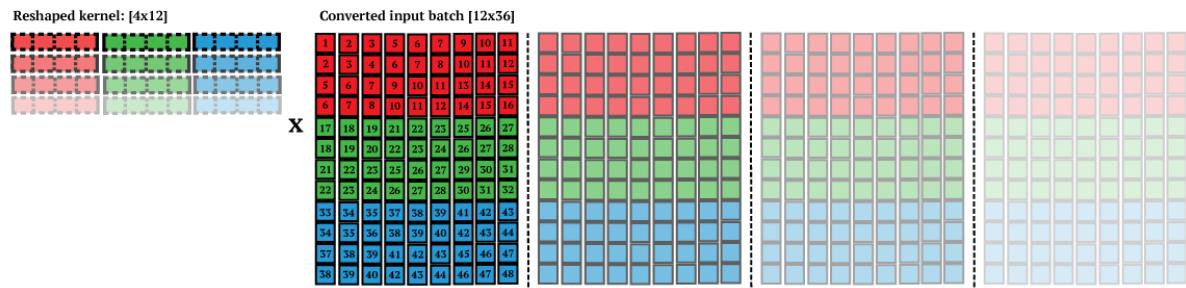
Image to column operation (im2col)

Slide the input image like a convolution but each patch become a column vector.



We get true performance gain

when the kernel has a large number of filters, ie: F=4
and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2].
The only problem with this approach is the amount of memory



2.1.2 Blocchi residui e concatenazioni

Nelle strutture delle reti che presenterò sono presenti dei collegamenti a livelli precedenti della rete, solitamente chiamati blocchi residui. Ciò perchè gli output di alcuni layer vengono passati in avanti nella rete senza essere modificati e vengono poi sommati a layer successivi. Questi layer avranno quindi un residuo da un livello meno profondo della rete. Durante l'addestramento ci saranno quindi alcuni layer che, oltre a imparare a svolgere il compito necessario per ottenere l'output desiderato, si occupano di dare un contributo ai layer precedenti della rete a cui magari il gradiente della funzione di errore non arriva abbastanza grande da essere significativo nell'aggiustamento dei pesi. Infatti nel caso in cui il gradiente sia nullo, verrà semplicemente retro-propagata la funzione identità da questi layer³.

Oltre ai blocchi residui un altro metodo popolare per propagare le feature da livelli meno profondi del network a livelli più profondi (cioè verso l'output) consiste nei layer di *route*, che semplicemente concatenano l'output di due o più layer creando immagini con un numero di canali pari alla somma dei canali delle immagini concatenate.

2.1.3 Attivazioni ReLU

Le funzioni di attivazione ReLU (Rectified Linear Unit) hanno la seguente forma:

$$f(x) = \max(0, x)$$

e sono sempre più utilizzate nelle DNN in quanto permettono al network di ottenere rappresentazioni sparse delle feature. Sperimentalmente² ciò comporta vari benefici:

- *Information disentangling*: uno degli obiettivi delle DNN è riuscire a distinguere i vari fattori che causano le variazioni tra i dati in input e i dati in output. Una rappresentazione densa è altamente *entangled* perchè quasi qualsiasi cambiamento nell'input modifica la maggior parte della rappresentazione dei dati nella rete. Viceversa una rappresentazione sparsa risente meno di cambiamenti nell'input

garantendo la conservazione dell’insieme delle feature.

- Rappresentazione di informazione in maniera efficiente: diversi input potrebbero contenere diverse quantità di informazione utile allo scopo della rete e sarebbero perciò più adeguatamente rappresentati con una struttura di dati di dimensione variabile. Variare il numero di neuroni attivi nella rete (i neuroni il cui output è 0 vengono considerati “spenti”) consente alla rete di gestire le dimensioni della rappresentazione a seconda dell’informazione nell’input.
- Distribuite ma sparse: le rappresentazioni sparse sono esponenzialmente più efficienti delle rappresentazioni dense distribuite, che a loro volta sono esponenzialmente più efficienti delle rappresentazioni puramente locali², dove la potenza dell’esponente dipende dal numero di feature non nulle.
- Riduzione del problema di scomparsa del gradiente: nel caso in cui l’attivazione sia positiva, il valore della derivata non è limitato a un range finito come nel caso delle attivazioni logistiche o tangenti iperboliche. Nel caso in cui l’attivazione sia nulla, introdurre un grado di non-linearietà nella rete non è un problema nella propagazione del gradiente finché ci sono neuroni in cui il flusso di informazione riesce a retro-propagarsi.

Per questi motivi nei modelli analizzati molti layer hanno attivazioni ReLU mentre altri avranno semplici attivazioni lineari (ovvero il risultato dell’attivazione è identico all’output del layer).

Una versione leggermente diversa dell’attivazione ReLU, che in qualsiasi caso trasmette l’informazione anche se con attivazione molto ridotta, si chiama Leaky ReLU ed ha la seguente forma:

$$f(x) = \begin{cases} x & \text{se } x > 0 \\ 0.01 \cdot x & \text{altrimenti} \end{cases}$$

2.1.4 Pixel-shuffle

Molte delle prime reti neurali per super-risoluzione pre-processavano l’immagine a bassa risoluzione di input con un upsample bicubico. In seguito l’immagine già delle dimensioni uguali a quella dell’output atteso veniva passata alla rete che cercava di migliorarne appunto la risoluzione. Questo rendeva l’addestramento più semplice per la rete ma molto più lento in quanto l’immagine di input aveva già dimensioni notevoli, aumentando di un fattore pari alla scala al quadrato i calcoli richiesti durante i forward dei vari layer. Per ovviare a questo problema è stato introdotto qualche anno fa un layer chiamato pixel-shuffle¹⁵, anche noto come layer di convoluzione sub-pixel. Questo layer mescola appunto i canali di un’immagine a bassa risoluzione per generarne una con meno canali ma con dimensioni maggiori. Praticamente consiste nel riorganizzare le dimensioni del tensore dell’immagine, ma anche nel mescolare tra loro i vari pixel durante l’operazione. Matematicamente la funzione applicata è la seguente:

$$PS(I[x, y, c]) = I[x // r, y // r, C \cdot r \cdot x \% r + C \cdot y \% r + c]$$

e trasforma un’immagine $[H \times W \times C^2]$ in una immagine $[rH \times rW \times C]$. Nella formula il simbolo ” // ” rappresenta il quoziente della divisione intera mentre ” % ” rappresenta il resto. Se l’immagine invece è in formato channel-first, ovvero ordinata come $[C, H, W]$, la funzione di pixel-shuffle è leggermente diversa ma analoga. In Byron sono state implementate entrambe le versioni, in modo da garantire la massima versatilità possibile. In figura 2.2 riporto uno schema esplicativo dell’operazione di shuffle applicata in seguito ad una convoluzione.

Faccio notare che la funzione `PixelShuffle` della libreria PyTorch opera su immagini $[C, H, W]$ mentre la funzione `depth_to_space` della libreria Tensorflow esegue il pixel-shuffle su immagini $[H, W, C]$.

Il vantaggio principale nell’utilizzo di questo layer è l’incremento di velocità della rete¹⁵: infatti utilizzando questo layer alla fine (o comunque in uno dei layer finali) della rete, è possibile estrarre tutte le feature necessarie per la super-risoluzione (che in questo caso

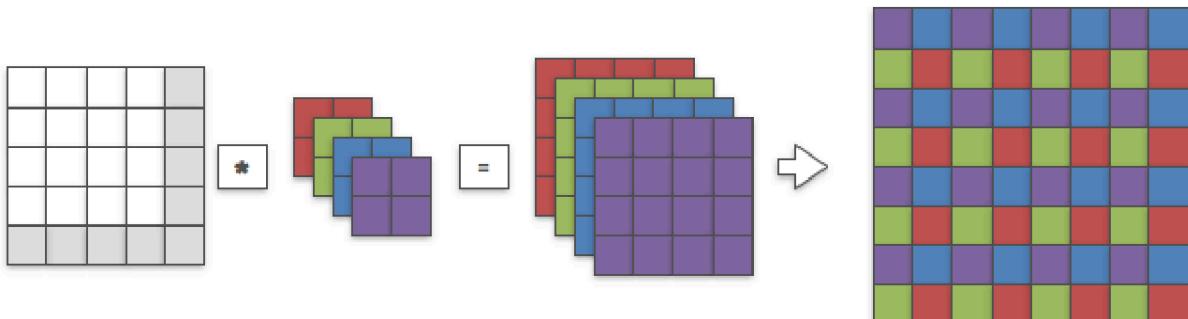


Figura 2.2: Schema esplicativo dell'operazione di riorganizzazione tensoriale effettuata dal layer di pixel shuffle applicato dopo un layer di convoluzione.

sono i vari filtri applicati) direttamente dall'immagine a bassa risoluzione di input, applicando vari layer di convoluzione, per poi riorganizzarle nell'immagine finale di dimensioni volute.

2.1.5 Batch normalization

Con l'aumento del numero di layer delle DNN permesso dalla sempre crescente potenza di calcolo dei computer odierni sono sorti vari problemi soprattutto durante l'addestramento, tra cui l'esplosione e la scomparsa del gradiente. Una delle possibili soluzioni a questi problemi consiste nel layer di batch normalization¹⁶. Come già detto precedentemente, solitamente l'input di una rete viene normalizzato (cioè tutti gli input vengono riscalati per avere valori compresi in un range scelto, solitamente [0,1]) o standardizzato (cioè i valori degli input vengono divisi per la media del dataset e in seguito gli viene sottratta la deviazione standard, per avere dei dati distribuiti con media 0 e deviazione standard 1). Ciò aiuta l'addestramento in quanto riduce il range dinamico dei dati in input a un range fisso, permettendo alla rete di estrarre feature più robuste e più velocemente¹⁷. Tuttavia se la rete ha un elevato numero di layer, a seconda dei valori dei pesi l'output dei layer potrebbe tornare ad avere range dinamici ampi. Per ovviare a questo problema, si interpone un layer di batch normalization dopo il layer della rete da normalizzare. In questo modo non solo l'input della rete ma anche l'output dei vari layer viene standardizzato. Ogni layer di batch normalization ha due pesi (per ogni batch), un fattore di scala e un bias, che modificano l'output standardizzato permettendo di cambiarne media e deviazione standard. Questi pesi possono venire aggiustati durante l'addestramento. Il termine batch

nel nome deriva dal gruppo di dati su cui viene effettuata la normalizzazione nel layer, che in questo caso è appunto un batch utilizzato durante l’addestramento con Stochastic Gradient Descent (SGD)⁴.

2.1.6 Layer YOLO

Il layer YOLO è il punto di forza della rete YOLOv3 e permette l’estrazione delle feature necessarie alla object detection in un unico passaggio dell’immagine attraverso la rete. L’immagine di input della rete viene divisa in una griglia $S \times S$. Se il centro di un oggetto ricade all’interno di una cella della griglia, questa sarà responsabile di rilevare quell’oggetto. Ogni cella prevede B box e i rispettivi punteggi di confidenza, definiti come $Pr(\text{object}) \times IOU(\text{prevision}/\text{groundtruth})$. Pr rappresenta la probabilità che il box contenga un oggetto, mentre IOU (intersection over union) rappresenta la precisione del box rispetto all’oggetto effettivo da rilevare. Ogni box ha 5 previsioni t_x , t_y , t_w , t_h e t_o , che poi permettono di calcolare:

$$b_x = \sigma(t_x) + c_x$$

$$b_y = \sigma(t_y) + c_y$$

$$b_w = p_w e^{t_w}$$

$$b_h = p_h e^{t_h}$$

$$\text{Confidenza} = \sigma(t_o)$$

Le coordinate b_x e b_y rappresentano il centro del box rispettivamente ai bordi della cella (di posizione c_x , c_y), quindi assumono valori compresi tra 0 e 1. Anche la confidenza, essendo una misura di probabilità, è compresa in questo intervallo. Per garantire che durante l’addestramento questi valori rimangano effettivamente in questo range, viene applicata una funzione di attivazione logistica σ a queste previsioni, con la seguente forma:

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

Le dimensioni del box invece vengono determinate da b_w e b_h a partire da p_w e p_h , ovvero le dimensioni dei box a priori. Si sceglie di utilizzare dei box a priori di dimensione fissa invece di prevedere direttamente da zero le dimensioni dei box perchè è stato dimostrato in altre reti come la Faster R-CNN¹⁸ che usare dei box a priori e determinare l'offset e la confidenza dei box per la detection a partire da essi migliora le performance e rende l'addestramento più stabile. Nella Faster R-CNN le dimensioni dei box fissi sono decise a priori e a mano. In YOLOv3 sono decise a priori ma sono state scelte dopo aver effettuato un k-mean clustering¹⁹ sui dataset VOC e COCO per avere un buon rapporto efficienza / precisione. Si è optato per avere 9 box fissi diversi, da cui la rete parte per trovare i box finali.

Ogni cella prevede anche C probabilità condizionali di classe $Pr(Classe_i/oggetto)$. A test time queste probabilità vengono moltiplicate per i punteggi di confidenza per ottenere punteggi specifici di classe per ogni oggetto. L'insieme di tutte le previsioni, che costituisce l'output del layer, viene codificato in un tensore $S \times S \times (B \times 5 + C)$. Il 5 è dato dalle 4 coordinate e della confidenza.

2.2 Modelli utilizzati

2.2.1 EDSR

Il primo modello che ho analizzato e riprodotto in Byron si chiama Enhanced Deep Super Resolution (EDSR)²⁰ ed è appunto una DNN per la super-risoluzione che si è classificata al primo posto nella NTIRE challenge del 2017²¹.

La struttura di base dell'EDSR è la SRResNet, una modifica della ResNet²² (famosa rete neurale a blocchi residui nel campo dell'elaborazione immagini) pensata per la super-risoluzione, con ulteriori modifiche pensate per velocizzare l'addestramento e aumentare la qualità dell'immagine ottenuta. In particolare vengono rimossi i layer di batch normaliza-

tion che risultano non solo poco efficaci per velocizzare l’addestramento ma anzi richiedono molto tempo in più per effettuare i calcoli di normalizzazione necessari. Infatti è stato dimostrato²⁰ che per task di cosiddetta low-level vision come la super-risoluzione, dove non è necessario svolgere compiti difficili come l’object detection, mantenere un ampio range dinamico di output è benefico per i risultati e non ha ripercussioni sull’addestramento.

La struttura della rete EDSR è illustrata nell’immagine 2.3. Essa consiste in:

- Un layer di convoluzione che prende l’immagine LR come input, con 256 filtri.
- Un gruppo di 32 blocchi residui, ognuno a sua volta composto da:
 - Un layer di convoluzione con 256 filtri.
 - Un layer di attivazione ReLU.
 - Un altro layer di convoluzione con 256 filtri.
 - Una moltiplicazione del risultato ottenuto per il fattore di scala, in questo caso equivalente a 0.1 , prima di sommare l’output del blocco residuo al suo input e continuare l’elaborazione nella rete.
- Un layer di convoluzione con 256 filtri, a cui viene sommato l’output del primo layer di convoluzione della rete.
- Un blocco per l’upsample dell’immagine, che nel caso del fattore di scala (x4) utilizzato è composto da:
 - Un layer di convoluzione con 1024 filtri.
 - Un layer di pixel-shuffle con scala $r = 2$.
 - Un layer di convoluzione con 1024 filtri.
 - Un layer di pixel shuffle con scala $r = 2$.
- Un layer finale di convoluzione che ha come output l’immagine super-risoluta, con 3 filtri.

I vari blocchi residui con i rispettivi layer di convoluzione hanno la funzione di trovare le feature ed il contenuto ad alta frequenza nell’immagine a bassa risoluzione di input, mentre il primo layer di convoluzione crea una versione con il contenuto a bassa frequenza dell’immagine, che poi viene sommata alla componente ad alta frequenza estratta dai

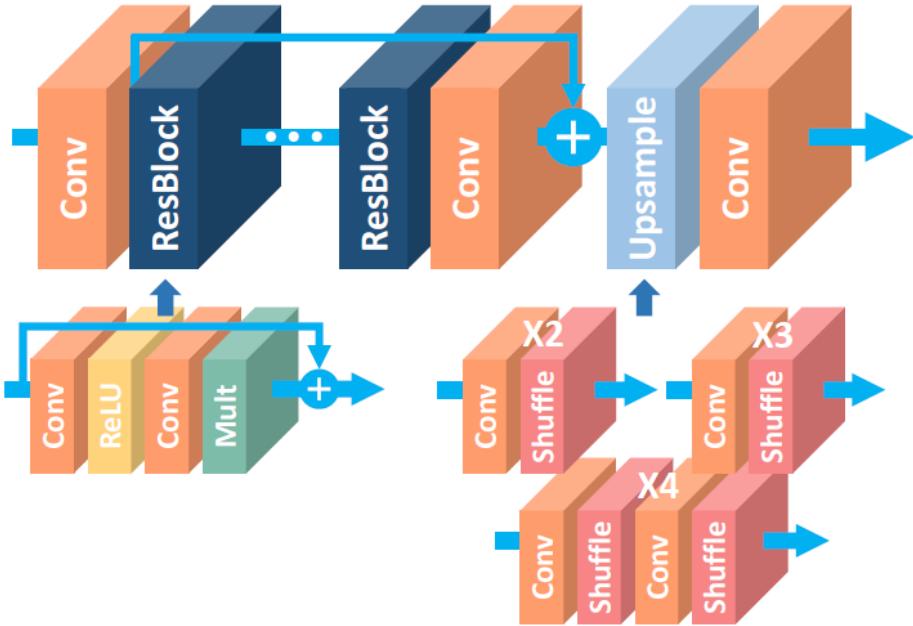


Figura 2.3: Struttura della rete EDSR per super-risoluzione. I vari blocchi indicano la tipologia dei layer, tra cui convoluzione, blocchi residui, pixel-shuffle, e attivazioni ReLU.

blocchi residui. Infine l'immagine così elaborata attraversa ulteriori layer di convoluzione e pixel-shuffle per venire ridimensionata a dimensioni 4 volte superiori a quelle di partenza. Il blocco di upsample è l'unica parte della rete che cambia a seconda del fattore di scala scelto per l'upscale. Con un fattore 2 questo blocco è formato solamente da un layer di convoluzione con 1024 filtri e da un layer di pixel-shuffle con scala 2; con un fattore 3 è formato da un layer di convoluzione con 2304 filtri e da un layer di pixel-shuffle con scala 3; con un fattore di 4 come nel caso in analisi è formato da due parti che applicano entrambe un fattore di scala 2. A causa del grande numero di filtri e delle dimensioni delle immagini in input nei layer di convoluzione del blocco di upsample, questi ultimi risultano di gran lunga i più lenti della rete e occupano buona parte del tempo di calcolo. Nella tabella 2.1 riporto il numero di parametri utilizzati nei layer del modello. In totale i pesi della rete sono oltre 43 milioni.

Tabella 2.1: Parametri per ogni tipo di layer della rete EDSR, in funzione del numero di canali in input e output del layer e delle dimensioni del filtro di convoluzione utilizzato.

Layer	Canali		
	input/ouput	Dim. filtri	Parametri
Convoluzione input	3 / 256	3x3	6912

Layer	Canali		
	input/ouput	Dim. filtri	Parametri
Conv. (blocco residuo)	256 / 256	3x3	589824
Convoluzione (pre-shuffle)	256 / 256	3x3	589824
Conv. (blocco upsample)	256 / 1024	3x3	2359296
Convoluzione output	256 / 3	3x3	6912

2.2.2 WDSR

Il secondo modello che ho analizzato e riprodotto in Byron si chiama Wide Deep Super Resolution (WDSR)²³ ed è un miglioramento della EDSR. Questa rete si è classificata al primo posto nella NTIRE challenge del 2018, nelle track con metodo di downsample sconosciuto, e ha performato molto bene anche nella track con metodo di downsample bicubico. Rispetto alla EDSR, la WDSR modifica principalmente due aspetti:

- **La struttura del network:** Come mostrato in figura 2.5, la struttura della rete WDSR è leggermente più semplice della rete EDSR. Essa infatti non ha i layer di convoluzione dopo il pixel-shuffle e inoltre nel caso del fattore di scala x4 laddove il blocco di upsample della EDSR consiste in multipli layer di pixel-shuffle con $r = 2$ e layer di convoluzione, nella WDSR l'upsample è formato unicamente da un layer di pixel-shuffle con $r = 4$. Ciò permette un notevole risparmio di tempo in quanto i layer di convoluzione nel blocco di upsample della EDSR sono i più pesanti in termini di tempi di calcolo e parametri. Inoltre a differenza della EDSR dove c'è un blocco residuo tra il primo layer di convoluzione e il blocco di upsample per aggiungere il contenuto a bassa frequenza all'immagine di output, nella WDSR il contenuto a bassa frequenza viene processato in un ramo completamente separato della rete che viene sommato solamente alla fine al contenuto ad alta frequenza. Un esempio di ciò può essere visto nella figura 2.4.
- **I blocchi residui:** Aumentare la profondità e i parametri delle reti neurali generalmente migliora le performance a discapito dei tempi di calcolo. Per migliorare



Figura 2.4: Output dei due rami della rete WDSR. A sinistra il contenuto ad alta frequenza, ottenuto dopo layer di convoluzione, blocchi residui e pixel-shuffle. A destra il contenuto a bassa frequenza, ottenuto semplicemente con un layer di convoluzione e un pixel-shuffle.

effettivamente le performance senza cambiare complessità computazionale (e quindi senza aggiungere parametri), la WDSR propone dei blocchi residui leggermente diversi basati sulla congettura²⁴ che i layer di attivazione ReLU, sebbene garantiscono la non-linearietà della rete e la stabilità durante l’addestramento, impediscono in parte il flusso di informazione dai layer meno profondi, che nel caso delle reti per super-risoluzione sono quelli con il contenuto a bassa frequenza da cui deve venire estrapolato quello ad alta frequenza. Per ovviare a questo problema senza aumentare il numero di parametri, nella WDSR c’è il cosiddetto “allargamento del passaggio”. Esso consiste semplicemente nel ridurre il numero di canali in input ed aumentare il numero di canali in output (quest’ultimo dato dal numero di filtri) ai layer di convoluzione prima dei layer ReLU. Ciò consente di avere un maggior numero di canali su cui viene applicata l’attivazione, consentendo un migliore passaggio dell’informazione lungo la rete ma mantenendo la non-linearietà necessaria. Per compensare a questo aumento di parametri pre-attivazione ovviamente i layer di convoluzione dopo i layer ReLU hanno un aumento dei canali in input e una riduzione dei canali in output, in modo da conservare il numero totale dei parametri (che in un layer di convoluzione è dato dal prodotto tra numero di canali in input e in output e le dimensioni del filtro).

In questa tesi ho utilizzato, come precedente specificato, dei modelli già addestrati e quindi non mi è stato possibile modificare la struttura delle reti. Al momento sono reperibili

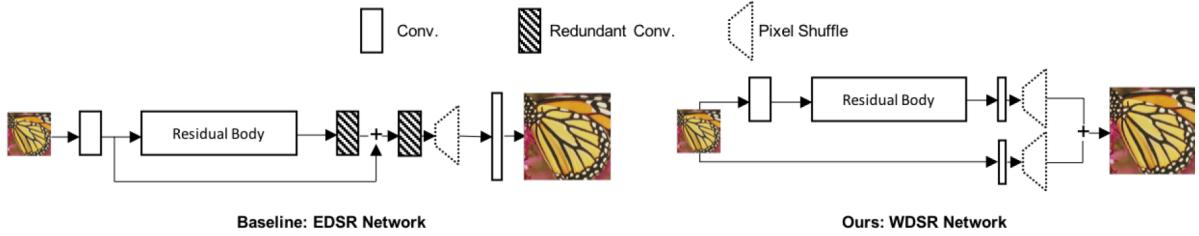


Figura 2.5: Confronto delle strutture delle reti EDSR (a sinistra) e WDSR (a destra). Si noti che anche la struttura interna del residual body è differente tra le due strutture.

soltanto versioni ridotte della rete WDSR, e quella utilizzata nel mio caso ha un numero di parametri notevolmente inferiore rispetto alla rete EDSR: in totale sono poco più di 3,5 milioni di pesi, meno di 1/10 della precedente rete utilizzata. Nella tabella 2.2 ho riportato il numero di parametri per ogni layer del modello della rete WDSR utilizzato. Come si può inoltre notare dal numero di filtri, è stato utilizzato un fattore di allargamento del passaggio pari a 6. Le performance che ho riscontrato dalla rete ovviamente risentono molto di questo numero ridotto di parametri, ma è stato verificato²³ che in caso di parità di parametri la struttura della rete WDSR genera immagini di output di qualità nettamente superiore alla rete EDSR ed è anche più efficiente in termini di tempi di calcolo.

Tabella 2.2: Parametri per ogni layer della rete WDSR, in funzione del numero di canali in input e output del layer e delle dimensioni del filtro di convoluzione utilizzato.

Layer	Canali			
	input/ouput	Dim. filtri	Parametri	
Convoluzione input 1	3 / 32	3x3	864	
Conv. 1 (blocco residuo)	32 / 192	3x3	55296	
Conv. 2 (blocco residuo)	192 / 32	3x3	55296	
Convoluzione (pre-shuffle)	32 / 48	3x3	13824	
Conv. input 2 (pre-shuffle)	3 / 48	5x5	3600	

2.2.3 YOLO

Le prime reti neurali per object detection erano formate da un insieme di classificatori applicati a varie scale e posizioni dell’immagine, per riconoscere vari tipi di oggetti. Altri metodi popolari usavano prima un sistema di segmentazione, per trovare nell’immagine regioni con oggetti da classificare, e in seguito utilizzavano classificatori su queste regioni per determinare a quale classe appartenesse l’oggetto. Con la rete YOLO²⁵ (You Only Look Once) invece l’object detection diventa un problema di regressione dai pixel dell’immagine direttamente alle coordinate dei box contenenti gli oggetti e alle probabilità delle rispettive classi. Ciò comporta due vantaggi principali: YOLO è molto veloce, tanto da poter essere eseguito in real-time anche su registrazioni live da videocamere¹⁹, e inoltre commette molti meno errori di falsi positivi sul background perchè ha uno sguardo d’insieme sull’immagine invece di dividerla in zone. Il problema principale di YOLO è la localizzazione: sebbene sia un ottimo classificatore, le posizioni dei box attorno agli oggetti non sono sempre molto precise²⁵. La versione attuale di YOLO è la v3²⁶. La rete è strutturata nel seguente modo:

- Una parte della rete si occupa dell’estrazione delle feature map, ed è in realtà una modifica della rete per detection (ma non classificazione) chiamata Darknet53, il cui modello è raffigurato nell’immagine 2.6. Rispetto alla versione in figura, nel caso della rete YOLOv3 le dimensioni dell’output sono diverse (il primo layer ha dimensioni dell’immagine 608x608 e gli altri scalano di conseguenza) e inoltre non vengono usati i layer (Avgpool, Connected, Softmax) dopo l’ultimo gruppo di blocchi residui.
- A partire dall’output dell’ultimo blocco residuo viene aggiunto un blocco di detection, composto da:
 - Un layer di convoluzione con 512 filtri di dimensione 1x1.
 - Un layer di convoluzione con 1024 filtri di dimensione 3x3.
 - Altre due coppie di layer con stesse dimensioni e filtri dei precedenti.
 - Un layer di convoluzione con 255 filtri di dimensione 1x1. L’output di questo layer ha dimensioni 19x19 e rappresenta la feature map a scala fine della rete.

	Type	Filters	Size	Output
	Convolutional	32	3×3	256×256
	Convolutional	64	$3 \times 3 / 2$	128×128
1x	Convolutional	32	1×1	
	Convolutional	64	3×3	
	Residual			128×128
	Convolutional	128	$3 \times 3 / 2$	64×64
2x	Convolutional	64	1×1	
	Convolutional	128	3×3	
	Residual			64×64
	Convolutional	256	$3 \times 3 / 2$	32×32
8x	Convolutional	128	1×1	
	Convolutional	256	3×3	
	Residual			32×32
	Convolutional	512	$3 \times 3 / 2$	16×16
8x	Convolutional	256	1×1	
	Convolutional	512	3×3	
	Residual			16×16
	Convolutional	1024	$3 \times 3 / 2$	8×8
4x	Convolutional	512	1×1	
	Convolutional	1024	3×3	
	Residual			8×8
	Avgpool			Global
	Connected			1000
	Softmax			

Figura 2.6: Modello della rete neurale Darknet53, in cui vengono riportati tipo di layer, numero di filtri, dimensione del filtro e dimensione dell'output di tutti i layer componenti la rete. Il $/2$ nell'immagine vicino alla dimensione del filtro indica che è stato utilizzato uno *stride* di 2.

- Un layer YOLO che calcola le previsioni a scala fine.
- A partire dal terzultimo layer di convoluzione precedente viene aggiunto un blocco di upsample e detection, composto da:
 - Un layer di convoluzione con 256 filtri di dimensione 1x1.
 - Un layer di upsample lineare che raddoppia le dimensioni dell’immagine senza interpolazione.
 - Un layer di *route* che concatena l’output del layer precedente con quello dell’ultimo layer del penultimo gruppo di blocchi residui. Ciò permette di prendere la feature map di dimensioni 38x38 calcolata precedentemente nel network ed aggiungervi alcuni dei dettagli estrapolati dalla scala fine.
 - Un layer di convoluzione con 256 filtri di dimensione 1x1.
 - Un layer di convoluzione con 512 filtri di dimensione 3x3.
 - Altre due coppie di layer con stesse dimensioni e filtri dei precedenti.
 - Un layer di convoluzione con 255 filtri di dimensione 1x1. L’output di questo layer ha dimensioni 38x38 e rappresenta la feature map a scala media della rete.
 - Un layer YOLO che calcola le previsioni a scala media.
- Come nel caso precedente a partire dal terzultimo layer di convoluzione precedente viene aggiunto un ulteriore blocco di upsample e detection, che però ha i layer di convoluzione con la metà dei filtri (quindi 128 e 256 rispettivamente), tranne l’ultimo in quanto il numero di filtri del layer di convoluzione prima del layer YOLO dipende dalla forma del tensore delle previsioni. Questo blocco concatena la feature map di dimensioni 76x76 calcolata nel terzultimo gruppo di blocchi residui a quella precedente a scala media dopo che questa è stata raddoppiata in dimensioni, in maniera analoga al passaggio da scala fine a scala media. In questo modo avremo anche un layer YOLO che effettua object detection su una feature map di scala grande che tiene anche conto delle precedenti feature map elaborate.

Ogni layer YOLO prevede 3 box, quindi i 9 box fissi a priori vengono distribuiti sulle 3 scale. Tutti i layer di convoluzione del modello hanno anche un layer di attivazione Leaky

ReLU e un layer di batch normalization dopo di loro, tranne i layer subito prima dei layer YOLO, che hanno un’attivazione lineare e non hanno batch normalization. In tabella 2.3 è riportato il numero di parametri per ogni layer della rete. In totale sono quasi 62 milioni.

Il network viene addestrato su immagini di dimensione variabile da 320x320 a 608x608 (andando per multipli di 32, che è il fattore di downscale della rete). Questo rende i filtri più sensibili alle informazioni dettagliate date dalla risoluzione maggiore (rispetto ai 224x224 di YOLOv1) e inoltre rende il network più robusto perchè i filtri si adattano a trovare oggetti a varie scale. Per la detection il network sarà facilmente riscalabile scegliendo le dimensioni di input offrendo un tradeoff tra precisione e velocità. Durante i test in questa tesi ho sempre utilizzato YOLOv3 con dimensioni dell’immagine di input 608x608 per avere i risultati di qualità migliore possibile. La funzione di errore usata dal network durante l’addestramento è molto complessa²⁵, ma è per minimizzare la somma dei quadrati residui, con alcune modifiche:

- l’errore viene pesato 1/10 per i box in cui non ci sono oggetti rilevati, per dare più importanza all’errore sulle coordinate;
- vengono previste le versioni logaritmiche delle dimensioni del box invece di prevedere direttamente altezza e larghezza (come visto nel layer YOLO) per dare più importanza agli errori sui box piccoli rispetto a quelli sui box grandi;
- durante l’addestramento solo il box con l’IOU migliore verrà assegnato ad ogni oggetto, in questo modo ci sarà più specializzazione nelle previsioni migliorando l’accuratezza.

Tabella 2.3: Parametri per ogni layer della rete YOLOv3, in funzione del numero di canali in input e output del layer e delle dimensioni del filtro di convoluzione utilizzato.

Layer	Canali		
	input/ouput	Dim. filtri	Parametri
Convoluzione input	3 / 32	3x3	864
Convoluzione 1	32 / 64	3x3	18432
Conv. 1 (blocco residuo 1)	64 / 32	1x1	2048

Layer	Canali		
	input/ouput	Dim. filtri	Parametri
Conv. 2 (blocco residuo 1)	32 / 64	3x3	18432
Convoluzione 2	64 / 128	3x3	73728
Conv. 1 (blocco residuo 2)	128 / 64	1x1	8192
Conv. 2 (blocco residuo 2)	64 / 128	3x3	73728
Convoluzione 3	128 / 256	3x3	294912
Conv. 1 (blocco residuo 3)	256 / 128	1x1	32768
Conv. 2 (blocco residuo 3)	128 / 256	3x3	294912
Convoluzione 4	256 / 512	3x3	1179648
Conv. 1 (blocco residuo 4)	512 / 256	1x1	131072
Conv. 2 (blocco residuo 4)	256 / 512	3x3	1179648
Convoluzione 5	512 / 1024	3x3	4718592
Conv. 1 (blocco residuo 5)	1024 / 512	1x1	524288
Conv. 2 (blocco residuo 5)	512 / 1024	3x3	4718592
Conv. 1 (scala fine)	1024 / 512	1x1	524288
Conv. 2 (scala fine)	512 / 1024	3x3	4718592
Conv. 3 (scala fine)	1024 / 512	1x1	524288
Conv. 4 (scala fine)	512 / 1024	3x3	4718592
Conv. 5 (scala fine)	1024 / 512	1x1	524288
Conv. 6 (scala fine)	512 / 1024	3x3	4718592
Conv. pre-yolo (scala fine)	1024 / 255	1x1	261120
Conv. pre-upsample (scala media)	512 / 256	1x1	131072
Conv. 1 (scala media)	768 / 256	1x1	196608
Conv. 2 (scala media)	256 / 512	3x3	1179648
Conv. 3 (scala media)	512 / 256	1x1	131072
Conv. 4 (scala media)	256 / 512	3x3	1179648
Conv. 5 (scala media)	512 / 256	1x1	131072
Conv. 6 (scala media)	256 / 512	3x3	1179648
Conv. pre-yolo (scala media)	512 / 255	1x1	130560

Layer	Canali		
	input/ouput	Dim. filtri	Parametri
Conv. pre-upsampole (scala grande)	256 / 128	1x1	32768
Conv. 1 (scala grande)	384 / 128	1x1	49152
Conv. 2 (scala grande)	128 / 256	3x3	294912
Conv. 3 (scala grande)	256 / 128	1x1	32768
Conv. 4 (scala grande)	128 / 256	3x3	294912
Conv. 5 (scala grande)	256 / 128	1x1	32768
Conv. 6 (scala grande)	128 / 256	3x3	294912
Conv. pre-yolo (scala grande)	256 / 255	1x1	65280

Capitolo 3

Framework

Durante gli anni sono state sviluppate molte librerie per l’implementazione delle reti neurali, che si differenziano tra loro per performance, semplicità di uso, linguaggio di programmazione usato e hardware supportato. Di seguito elencherò i principali framework utilizzati durante questo lavoro di tesi.

3.1 Darknet

Darknet⁵ è un framework per reti neurali scritto in Ansi C da Joseph Redmon, dottorando all’Università di Washington in computer vision, con supporto nativo solo per sistemi operativi Linux (anche se sono stati effettuati vari porting per altre piattaforme tra cui Windows) e ottimizzato per GPU (solo CUDA²⁷, quindi solo schede grafiche NVidia). Risulta una delle migliori librerie per reti neurali applicate al campo dell’object detection attualmente disponibili e open source, in termini di performance e risultati²⁶, grazie all’implementazione nativa in questo framework della rete YOLOv3. Tuttavia ha alcuni aspetti che possono essere migliorati, come la compatibilità tra piattaforme diverse e una migliore gestione del calcolo parallelo su CPU. Per questo motivo è nata l’idea di Byron, un porting in C++ di Darknet che per ora si concentra su questi punti e sull’estensione del framework con nuove funzioni. Un porting è una “traduzione” del codice da una piattaforma o linguaggio a un altro, che solitamente viene fatto per motivi di compatibilità o

per migliorare le performance (come nel mio caso).

3.2 PyTorch e Keras

Altri framework per reti neurali molto popolari al momento sono PyTorch⁶ e Keras⁷. Entrambi sono scritti in Python, e di conseguenza sono pensati per essere di facile uso per l'utente e consentono di scrivere e impostare velocemente anche modelli complicati. PyTorch è un porting in Python della libreria Torch, scritta in Lua. Essendo scritto quasi completamente in Python, questo framework offre un tradeoff tra performance e semplicità d'uso, in quanto Python è un linguaggio di alto livello e di conseguenza generalmente meno performante in quanto gestisce alcune variabili, tra cui la memoria allocata, in maniera automatica e non sempre nel modo ottimale. Keras invece è un wrapping di un'altra libreria scritta in Python chiamata Tensorflow, che a sua volta è un wrapping della versione in C++ della stessa libreria. Un wrapping è una interfaccia di codice che permette di usare codice sorgente scritto in un altro linguaggio o in generale più complicato e complesso da utilizzare. Ciò solitamente permette all'utente di scrivere codice più facilmente e più velocemente, tuttavia in questo caso c'è un tempo di overhead poichè il computer deve “tradurre” le istruzioni dal livello più alto (in questo caso Keras) al livello più basso (in questo caso Tensorflow in C++). Faccio notare che comunque questo tempo di overhead è molto inferiore solitamente rispetto ai tempi di calcolo effettivamente necessari nei vari layer della rete, e quindi sia Keras che Tensorflow sono entrambe librerie molto performanti e molto generiche, in grado di consentire l'implementazione di modelli di reti neurali di vario tipo. Inoltre le considerazioni che ho fatto finora valgono per quanto riguarda l'utilizzo su CPU di queste librerie, in quanto per l'utilizzo su GPU tutti questi framework si appoggiano a librerie esterne come cuDNN¹⁴.

Parlo di queste librerie perchè sono state in parte utilizzate durante il mio lavoro di tesi. Visti i lunghi tempi richiesti per la scrittura di una libreria così vasta e per il debugging necessario ad assicurarsi che funzionasse correttamente, ho scelto di non addestrare di persona le reti di cui parlerò più avanti. Questo avrebbe richiesto molti altri test oltre che ovviamente il tempo di addestramento, che per queste reti solitamente è superiore a

una settimana sulle GPU più performanti del momento²⁰. Di conseguenza ho preso i pesi delle reti pre-addestrate, che però erano disponibili solamente per l'implementazione in PyTorch (per la rete EDSR) e per quella in Keras (per la WDSR). Ciò ha reso necessaria ovviamente la scrittura di ulteriore codice per la conversione dei pesi tra i vari modelli. Ho inoltre riscontrato che la versione dell'EDSR messa a disposizione nella repository ufficiale non compila su CPU. Questo problema è noto ai programmatore che hanno fatto il porting della rete da Torch ma non è stato ancora risolto. Di conseguenza l'implementazione su Byron dell'EDSR è per ora l'unica (a mia conoscenza) funzionante su CPU.

3.3 Byron

Come detto sopra, Byron è un framework in C++ (standard 2017) basato per la maggior parte sul codice sorgente di Darknet. Tuttavia essendo stata riscritta da zero, questa libreria ha numerose migliorie e inoltre per alcuni aspetti critici, come la gestione dei core, adotta strategie nuove permettendo delle performance nettamente superiori a Darknet. Sia in Byron che in Darknet la gestione dei core e dei thread del processore viene effettuata tramite la libreria OpenMP²⁸. Tuttavia questa libreria ha varie direttive per gestire la divisione dei compiti da svolgere durante il codice tra i vari thread. Per esempio, in Darknet la GEMM è implementata con il seguente codice:

```
void gemm_nn(int M, int N, int K, float ALPHA,
             float *A, int lda,
             float *B, int ldb,
             float *C, int ldc)
{
    int i,j,k;
    #pragma omp parallel for
    for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            register float A_PART = ALPHA*A[i*lda+k];
            for(j = 0; j < N; ++j){
                C[i*N+j] += A_PART*B[j*ldb+k];
            }
        }
    }
}
```

```

    C[i*ldc+j] += A_PART*B[k*ldb+j];
}
}
}
}

```

dove la direttiva `#pragma omp parallel for` della libreria OpenMP apre una sessione parallela e implica che il ciclo `for` subito dopo di essa verrà svolto in parallelo ed ogni core si occuperà di una iterazione del ciclo. In problema è che la dichiarazione delle variabili di ciclo all'esterno della sessione parallela significa che tutti i thread effettueranno l'accesso alle stesse variabili, causando eventualmente problemi di concurrency.

Per risolvere questo problema, in Byron impiego diverse direttive di OpenMP:

- `#pragma omp parallel` viene usata solamente all'inizio del programma principale per aprire la sessione parallela che resterà aperta fino alla fine dell'esecuzione, in quanto tutti i vari loop della libreria saranno poi eseguiti in parallelo e non solo quelli della GEMM come avviene in Darknet;
- `#pragma omp taskgroup` e `#pragma omp taskloop` permettono di gestire i cicli come il `for` specificando eventuali variabili che i thread devono vedere come private in modo da non sovrascrivere quelle di altri thread e permettono anche di scegliere quanti thread devono occuparsi di una data funzione, permettendo una divisione dei compiti della rete sui vari core in maniera ottimale a seconda della potenza di calcolo disponibile sulla macchina al momento dell'esecuzione del codice. Sebbene durante questo lavoro di tesi non abbia addestrato le reti neurali utilizzate, questa specifica implementazione che è presente in tutte le funzioni della libreria permette di gestire più liberamente la fase di addestramento dividendo per esempio i core del computer in vari compiti quali caricamento delle immagini, propagazione forward e backward nella rete e aggiornamento dei pesi.

Oltre a queste correzioni per quanto riguarda la parallelizzazione del codice, Byron ha anche alcune funzioni completamente assenti in Darknet, tra cui il layer di pixel-shuffle

che vede uso sempre maggiore nei modelli di reti neurali che elaborano le immagini e che permette l'implementazione delle migliori reti per super-risoluzione utilizzate al momento. Rispetto a Keras e Tensforflow il miglioramento principale consiste nell'implementazione del layer YOLO per la object detection. Infatti implementare questo layer direttamente in C++ all'interno di Tensorflow sarebbe parecchio arduo, a causa della struttura enorme e complessa del framework. Ed implementarlo in Python, per quanto leggermente più semplice, ridurrebbe drasticamente le performance. Ciò rende Byron un framework ottimizzato per CPU multi-core e per le reti neurali che si occupano di object detection e super-risoluzione.

Capitolo 4

Risultati

4.1 Tempi e performance

In questo capitolo illustrerò i vari risultati ottenuti confrontando i tempi di calcolo della rete per processare una immagine tra le varie reti e valutando i vari miglioramenti di qualità delle immagini.

4.1.1 EDSR vs WDSR

Come prima analisi ho confrontato i tempi di calcolo delle due reti per super-risoluzione implementate. In figura 4.1 sono rappresentati i risultati ottenuti dopo aver utilizzato per 100 volte le reti su una singola immagine di dimensioni 510x339. Come si può notare la rete WDSR è molto più veloce, oltre un fattore 10 di velocità. Ciò è dovuto principalmente al fatto che questa versione della rete ha molti meno parametri della contendente e i layer di convoluzione hanno quindi molti meno filtri e meno operazioni da svolgere. Tuttavia è stato dimostrato²³ che la struttura della rete WDSR, grazie all'omissione dei layer finali di convoluzione dopo l'upsampling dell'immagine, a parità di parametri è notevolmente più efficiente della struttura della rete EDSR.

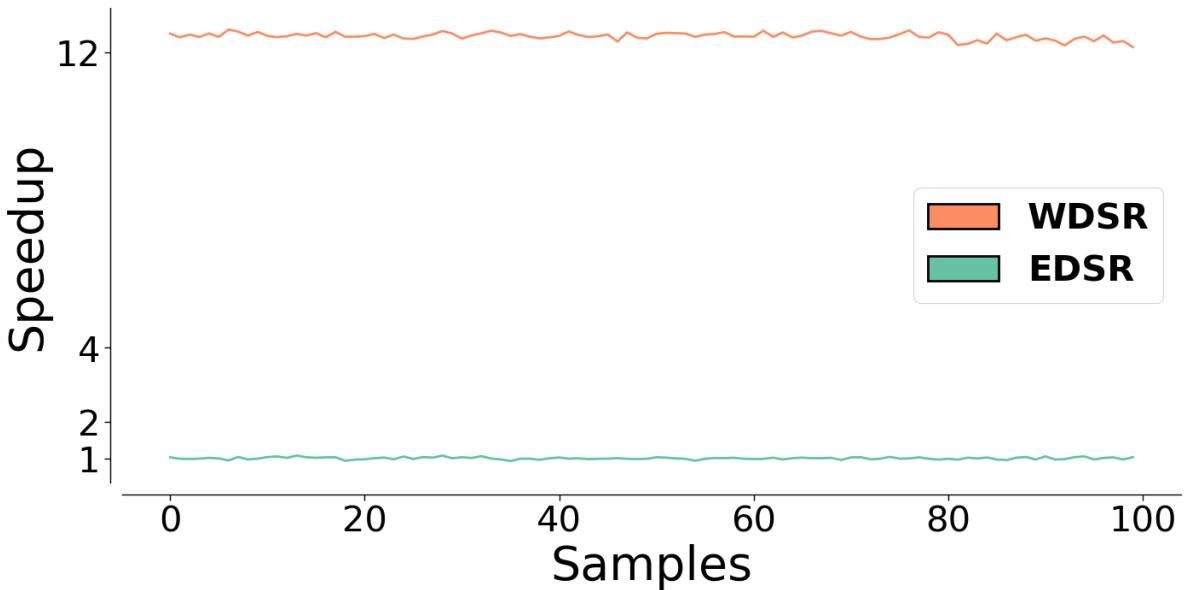


Figura 4.1: Confronto delle velocità della rete WDSR e della rete EDSR, normalizzate sulla velocità media della rete EDSR, per 100 run su immagini 510x339.

4.1.2 Numero di core

Come seconda analisi ho studiato l'andamento della velocità di calcolo in funzione del numero di core fisici utilizzati dalla macchina durante i test. In questo caso il confronto è tra 100 run della rete WDSR su una singola immagine di dimensioni 510x339. I test sono stati effettuati con 2, 4, 8, 16 e 32 core. Come si può notare dal grafico in figura 4.2, l'andamento della velocità in funzione del numero di core è parabolico, non lineare. Questo andamento è normale in quanto all'aumentare del numero di core aumenta il tempo necessario in cui il master thread (che gestisce tutti gli altri) deve distribuire le informazioni necessarie per i calcoli ad ogni core o recuperare i risultati ottenuti per procedere al successivo ciclo di istruzioni. In generale aumentare il numero di core non è solo dispendioso in termini economici ma anche energetici, soprattutto nel caso in cui sia necessario addestrare una rete, procedura che può richiedere svariati giorni. Da questo grafico si evince quindi un tradeoff tra utilizzo dei core della macchina e corrispettivo consumo energetico ed effettivo guadagno in velocità.

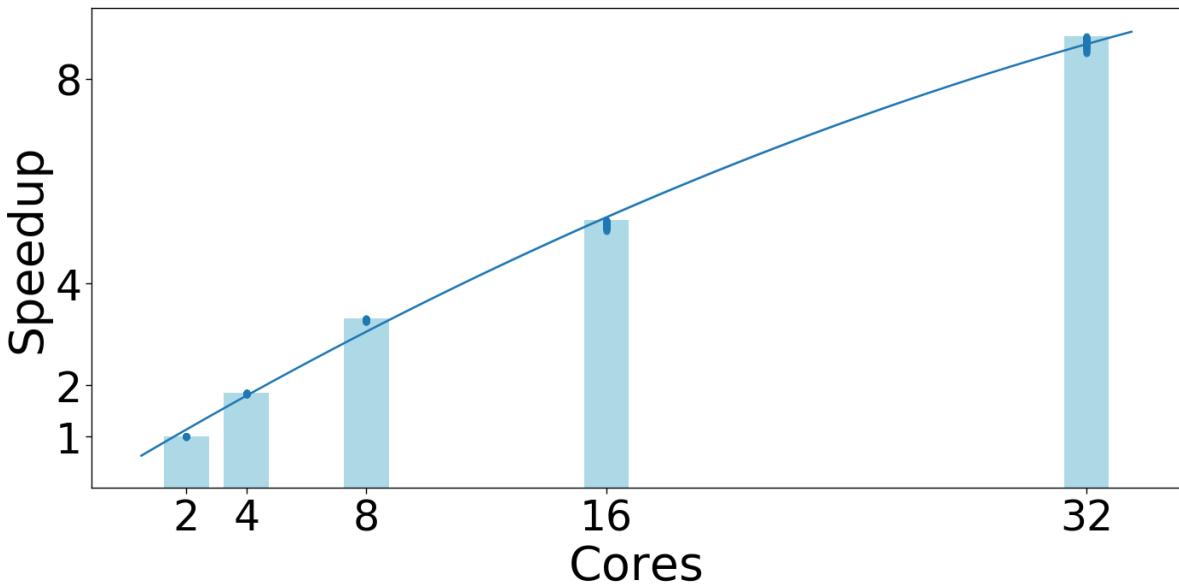


Figura 4.2: Confronto della velocità della rete WDSR in funzione del numero di core utilizzati durante il forward della rete. Per ogni numero di core sono state effettuate 100 run su immagini 510x339.

4.1.3 Byron vs Darknet

Come ultima analisi temporale ho confrontato la velocità di calcolo tra Byron e la libreria su cui è basata, Darknet. Visto che quest'ultima non ha implementato il layer di pixel-shuffle, non è possibile testare le reti per super-risoluzione come mezzo per valutare a parità di rete la velocità delle due librerie. Tuttavia visto che il punto di forza di Darknet sulle altre librerie è la velocità nell'object detection con YOLO, ho confrontato le due librerie e calcolato lo speedup relativo di Byron per 100 run della rete YOLOv3 su una singola immagine di dimensioni 608x608. Come si può notare dal grafico in figura 4.3, c'è un aumento di velocità di circa un fattore 2. Inoltre dal grafico si evince anche che la distribuzione delle velocità di Byron è più piccata mentre quella di Darknet è più ampia: ciò significa che Byron è più consistente, fatto probabilmente dovuto alla diversa gestione dei core delle librerie. Ritengo che questo speedup abbia margini di miglioramento in quanto la libreria può ancora essere migliorata dal punto di vista dell'implementazione dei layer più costosi in termini di tempi di calcolo come per esempio il layer di convoluzione²⁹.

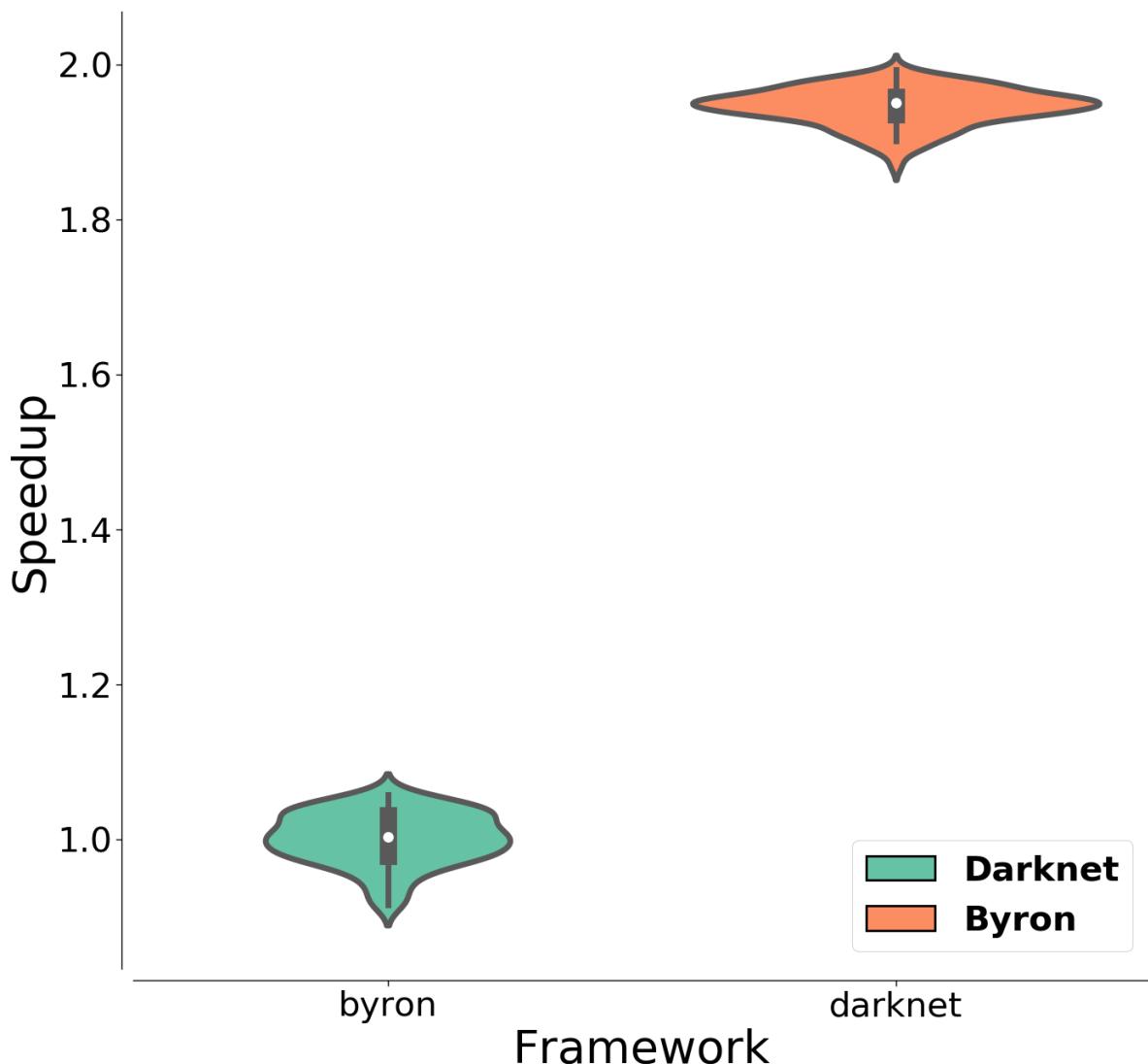


Figura 4.3: Confronto delle velocità della libreria Darknet e della libreria Byron, normalizzate sulla velocità media della libreria Darknet, per 100 run della rete YOLOv3 su immagini 608x608.

4.1.4 PSNR e SSIM

In figura 4.4 riporto il confronto tra i PSNR misurati su 60 immagini del validation set del dataset DIV2K per tre diversi metodi di upsample: bicubico, super-risoluzione con WDSR e super-risoluzione con EDSR. Come si può notare c'è un notevole miglioramento nelle immagini super-risolute rispetto al semplice upsample bicubico. Infatti un aumento di PSNR di 0.25 è già visibile a occhio nudo, come si può notare nelle figure ???. Tra le due reti invece, sebbene la differenza sia meno evidente, prevale la EDSR come qualità. Tuttavia è importante ricordare che questa implementazione della rete WDSR ha meno di 1/10 dei parametri della contendente, e quindi i risultati sono ragionevolmente peggiori. Se avessimo avuto lo stesso numero di parametri per le due reti, la struttura della WDSR avrebbe riportato risultati notevolmente migliori²³. Ciò avrebbe comportato tuttavia un notevole aumento dei tempi di calcolo: in questo caso c'è un tradeoff tra qualità del risultato e tempi richiesti.

In figura 4.5 riporto il confronto tra i SSIM misurati sulle stesse 60 immagini del validation set del dataset DIV2K utilizzate anche per calcolare il PSNR, ed anche in questo caso distinguendo i tre metodi impiegati. I risultati ottenuti sono concordi con le misure di PSNR precedentemente illustrate, e confermano che le reti per super-risoluzione migliorano notevolmente la qualità di un'immagine ricampionata ripristinandola fedelmente.

4.1.5 Confronto visuale

Nelle figure in questo paragrafo illustro in maniera qualitativa e riporto PSNR e SSIM per varie immagini che sono state super-risolute dal validation set del DIV2K, confrontando l'immagine originale con l'immagine LR dopo un ricampionamento bicubico e dopo aver applicato le due reti per super-risoluzione implementate.

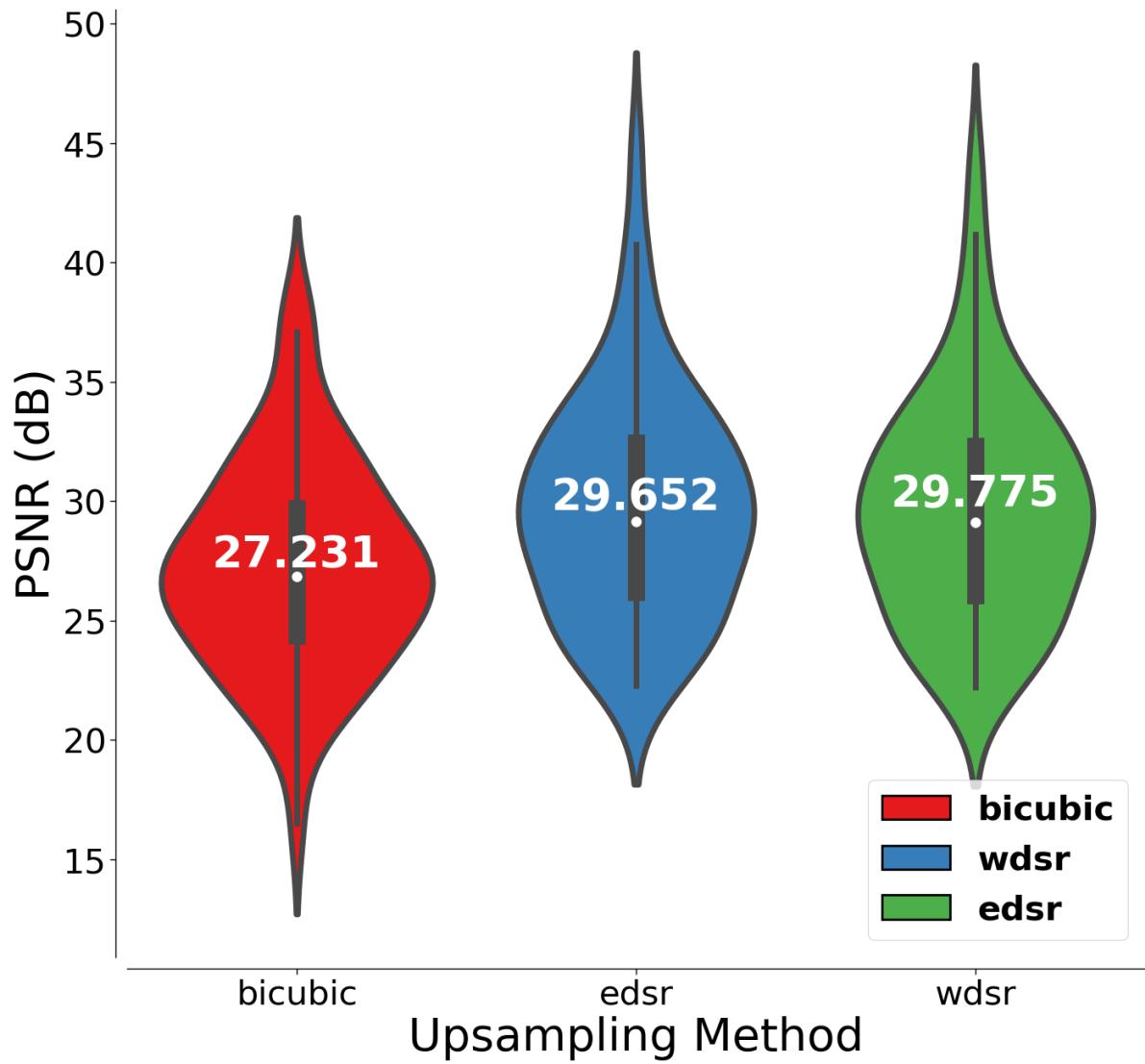


Figura 4.4: Confronto dei risultati ottenuti su 60 immagini del validation set del DIV2K in termini di PSNR in funzione del metodo di upsample utilizzato. In bianco viene riportato il valore medio del PSNR sul set in esame.

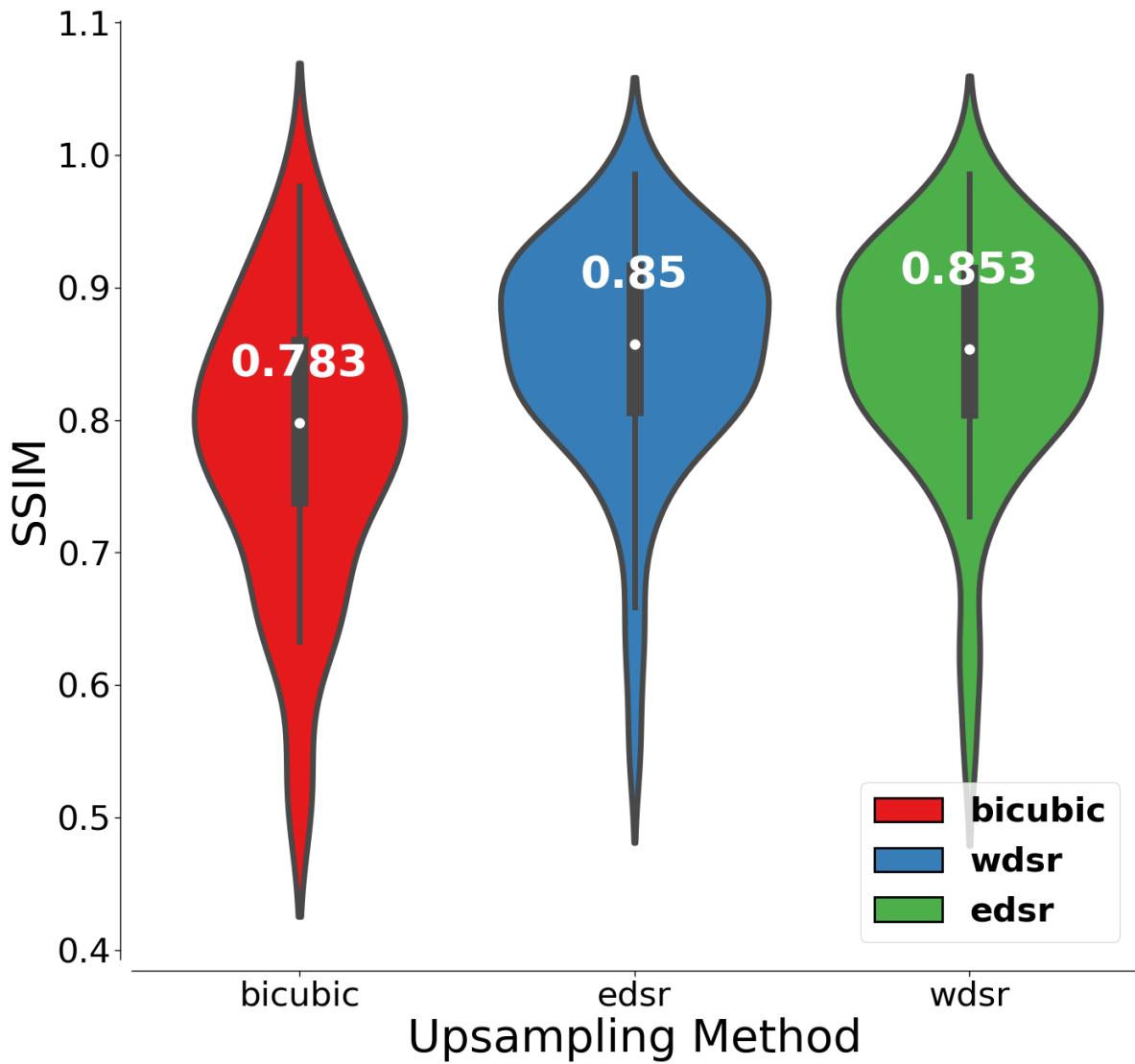
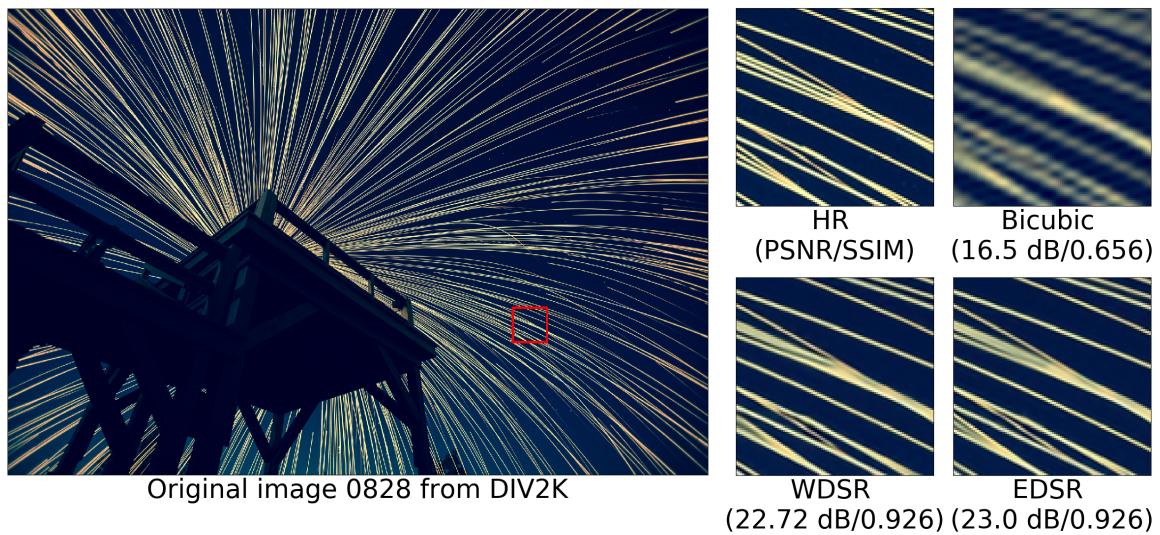
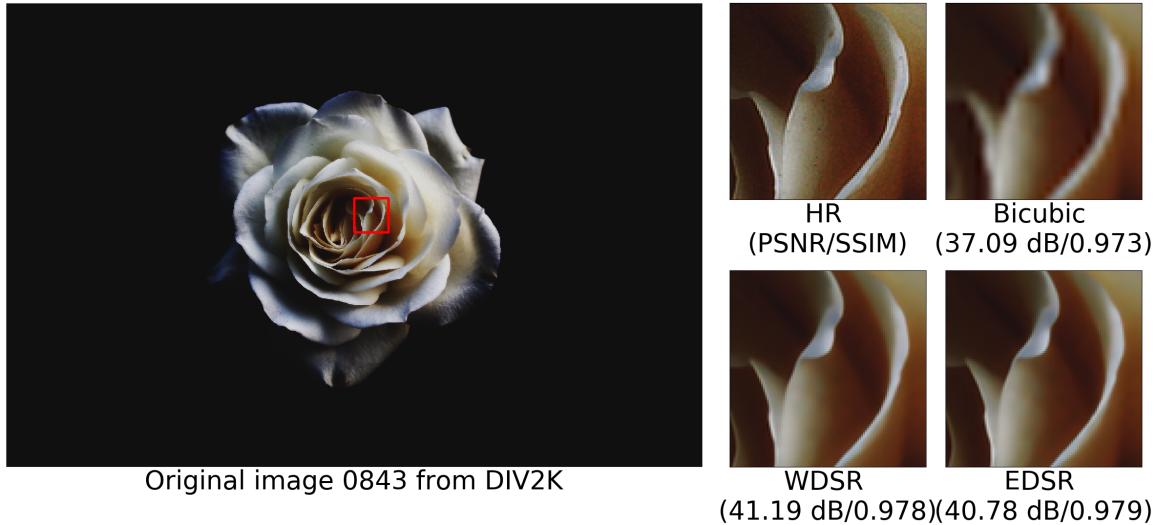


Figura 4.5: Confronto dei risultati ottenuti su 60 immagini del validation set del DIV2K in termini di SSIM in funzione del metodo di upsample utilizzato. In bianco viene riportato il valore medio del SSIM sul set in esame.





4.2 Super-risoluzione e object detection

Uno dei problemi principali di YOLO, oltre alla precisione nella localizzazione, è la detection di oggetti piccoli e vicini²⁵. Per questo motivo è plausibile aspettarsi che l'utilizzo di una rete per super-risoluzione per migliorare la qualità di un'immagine prima di applicarvi la rete YOLO per object detection ne migliori i risultati e la precisione. Per verificare questa ipotesi ho effettuato due test, entrambi su immagini contenenti persone. In entrambi i casi ho analizzato con YOLO tre immagini: l'immagine di partenza LR (che viene ridimensionata dalla rete a 608x608 linearmente prima dei calcoli), l'immagine

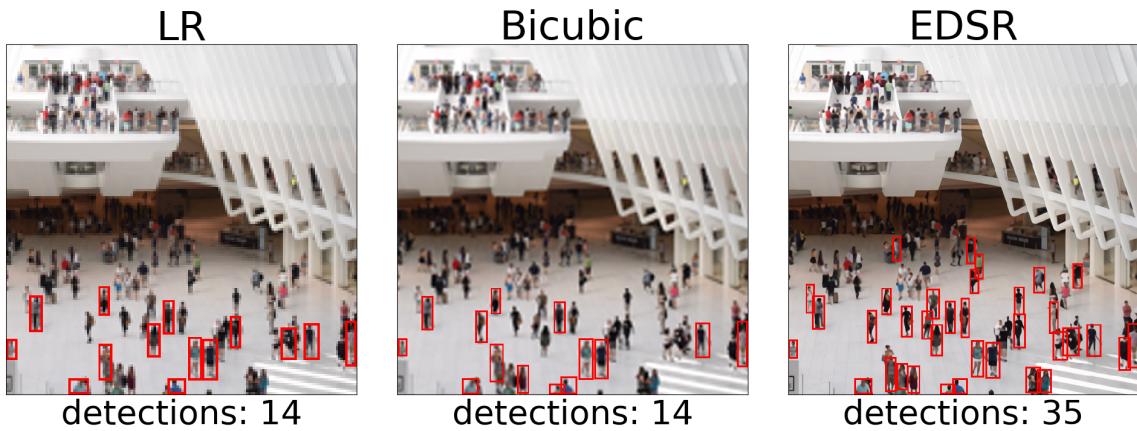


Figura 4.6: Confronto dell’efficacia della rete YOLO per object detection su una patch LR di dimensioni 152x152 al variare del metodo di upsample utilizzato.

ridimensionata bicubicamente con scala 4 e l’immagine super-risoluta dalla rete EDSR. Nel primo caso YOLO opera su una patch di dimensioni 608x608 nelle immagini upsampled e di dimensioni 152x152 nell’immagine LR. Nel secondo caso la patch ha dimensioni rispettivamente 300x300 e 75x75. Si possono notare dalle figure 4.6 e 4.7 alcuni dettagli:

- Nel caso 1 le persone sono di dimensioni considerevoli e già nell’immagine a bassa risoluzione YOLO riesce a trovarne qualcuna. Il metodo di upsample bicubico non migliora il numero di detection ma migliora leggermente le probabilità di classificazione. Nell’immagine super-risoluta invece aumenta notevolmente il numero di detection e anche le probabilità di classificazione sono migliori.
- Nel caso 2 le immagini sono a una risoluzione inferiore e infatti YOLO non riesce a trovare nessuna persona nell’immagine LR. Con l’upsample bicubico vengono trovate solo 2 persone, mentre nell’immagine super-risoluta le detection aumentano a 7. Ciò valida l’ipotesi dell’efficacia della super-risoluzione applicata in congiunzione con l’object detection soprattutto nel caso di immagini piccole.

4.3 Conclusioni

In questo lavoro di tesi ho implementato Byron, una libreria per reti neurali stato dell’arte in termini di performance nel campo dell’object detection e della super-risoluzione su CPU,

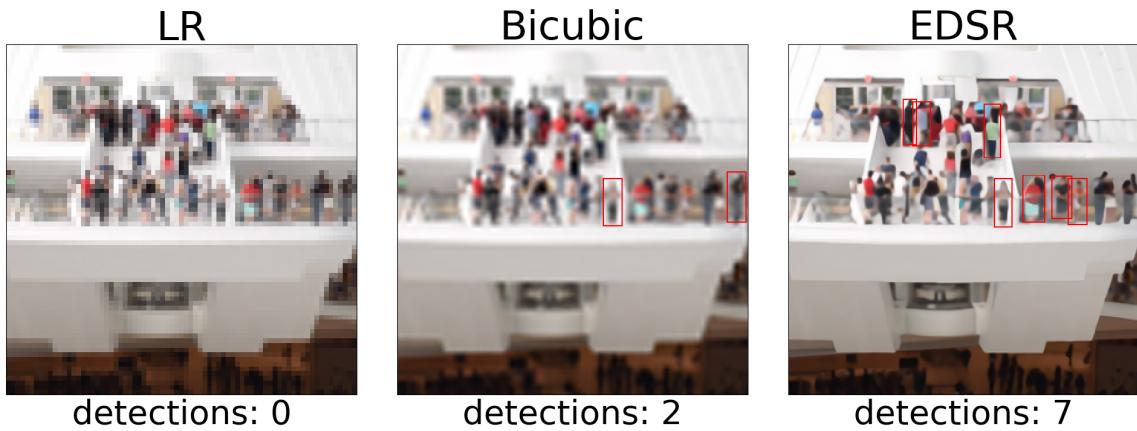


Figura 4.7: Confronto dell’efficacia della rete YOLO per object detection su una patch LR di dimensioni 75x75 al variare del metodo di upsample utilizzato.

ottimizzata per sistemi informatici con elevato numero di core, dimostrando anche una possibile applicazione congiunta delle due metodologie di elaborazione immagini al fine di migliorare i risultati della object detection.

Il miglioramento delle performance delle reti di object detection su immagini super-risolute è noto già da alcuni lavori³⁰, ma non è l’unico possibile campo di applicazione della super-risoluzione. Potendo addestrare i modelli delle reti su dataset particolari come per esempio immagini di microscopia o di risonanze magnetiche, probabilmente sarebbe possibile migliorarne la qualità per scopi pratici in campo medico. Anche in questo caso sono già state svolte delle ricerche (32) ma l’argomento è ancora una novità in moltissimi ambienti e quindi ha molto potenziale per essere sviluppato e molte possibili applicazioni.

Altri lavori futuri più centrati sulla libreria che sui modelli implementabili comprendono:

- Byron su tutti i sistemi operativi: per ora Byron è stato testato esaustivamente solo su ambiente Linux ma è stato progettato per essere multi piattaforma e compatibile anche con sistemi operativi Windows e Mac. Ulteriori test sono necessari per verificare la compatibilità e l’ottimizzazione su altri sistemi e sicuramente avere una libreria che funzioni in qualsiasi ambiente è un ottimo obiettivo da raggiungere.
- Byron su GPU: vero che l’idea alla base di Byron è l’ottimizzazione per CPU multi-core quali i server di bio-informatica, ma qualsiasi libreria per reti neurali che si rispetti deve avere anche un’implementazione su GPU di pari passo a quella CPU in

modo da poter sfruttare qualsiasi hardware disponibile nel miglior modo possibile. A questo scopo sarebbe ideale implementare sia una versione in CUDA che una versione in OpenCL della libreria, in modo da garantirsi il funzionamento sulla maggioranza delle GPU moderne.

- Ottimizzazioni di codice: altri possibili miglioramenti su cui sto già lavorando riguardano l'implementazione e lo sviluppo di nuovi algoritmi per ottimizzare i layer più intensivi delle reti quali per esempio layer di convoluzione e batch-normalization. Uno di essi per esempio è l'algoritmo Winograd²⁹ per la convoluzione che dovrebbe garantire un notevole speedup per i layer con filtri di dimensione 3x3 che ormai sono alla base della maggior parte delle reti per elaborazione immagini.

Ringraziamenti

Cose varie / ringraziamenti

Bibliografia

1. McCulloch, W. S. & Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5**, 115–133 (1943).
2. Glorot, X., Bordes, A. & Bengio, Y. Deep sparse rectifier neural networks. in *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (eds. Gordon, G., Dunson, D. & Dudík, M.) **15**, 315–323 (PMLR, 2011).
3. Gomez, A. N., Ren, M., Urtasun, R. & Grosse, R. B. The Reversible Residual Network: Backpropagation Without Storing Activations. *arXiv e-prints* arXiv:1707.04585 (2017).
4. Robbins, H. & Monro, S. A stochastic approximation method. *Ann. Math. Statist.* **22**, 400–407 (1951).
5. Redmon, J. Darknet: Open source neural networks in c. (2013).
6. Paszke, A. *et al.* Automatic differentiation in pytorch. in *NIPS-w* (2017).
7. Chollet, F. & others. Keras. (2015).
8. Bradski, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).
9. Dong, C., Change Loy, C., He, K. & Tang, X. Image Super-Resolution Using Deep Convolutional Networks. *arXiv e-prints* arXiv:1501.00092 (2014).
10. MathWorks. *MATLAB version 9.5 (r2018b)*. (The MathWorks Inc., 2018).
11. Hore, A. & Ziou, D. Image quality metrics: PSNR vs. SSIM. in *2010 20th international conference on pattern recognition* 2366–2369 (2010). doi:10.1109/ICPR.2010.579
12. Walt, S. van der *et al.* Scikit-image: Image processing in python. *PeerJ* **2**, e453 (2014).
13. Agustsson, E. & Timofte, R. NTIRE 2017 challenge on single image super-resolution: Dataset and study. in *The ieee conference on computer vision and pattern recognition (cvpr) workshops* (2017).

14. Chetlur, S. *et al.* cuDNN: Efficient Primitives for Deep Learning. *arXiv e-prints* arXiv:1410.0759 (2014).
15. Shi, W. *et al.* Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. *arXiv e-prints* arXiv:1609.05158 (2016).
16. Ioffe, S. & Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints* arXiv:1502.03167 (2015).
17. Lecun, Y., Bottou, L., Orr, G. & Müller, K.-R. Efficient backprop. (2000).
18. Ren, S., He, K., Girshick, R. & Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv e-prints* arXiv:1506.01497 (2015).
19. Redmon, J. & Farhadi, A. YOLO9000: Better, Faster, Stronger. *arXiv e-prints* arXiv:1612.08242 (2016).
20. Lim, B., Son, S., Kim, H., Nah, S. & Lee, K. M. Enhanced Deep Residual Networks for Single Image Super-Resolution. *arXiv e-prints* arXiv:1707.02921 (2017).
21. Timofte, R. *et al.* NTIRE 2017 challenge on single image super-resolution: Methods and results. in *The ieee conference on computer vision and pattern recognition (cvpr) workshops*
22. He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. *arXiv e-prints* arXiv:1512.03385 (2015).
23. Yu, J. *et al.* Wide Activation for Efficient and Accurate Image Super-Resolution. *arXiv e-prints* arXiv:1808.08718 (2018).
24. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L.-C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv e-prints* arXiv:1801.04381 (2018).
25. Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints* arXiv:1506.02640 (2015).
26. Redmon, J. & Farhadi, A. YOLOv3: An incremental improvement. in (2018).
27. Nickolls, J., Buck, I., Garland, M. & Skadron, K. Scalable parallel programming with cuda. *Queue* **6**, 40–53 (2008).
28. Dagum, L. & Menon, R. OpenMP: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.* **5**, 46–55 (1998).
29. Lavin, A. & Gray, S. Fast Algorithms for Convolutional Neural Networks. *arXiv e-prints* arXiv:1509.09308 (2015).

30. Cheng, N., Zhao, T., Chen, Z. & Fu, X. Enhancement of underwater images by super-resolution generative adversarial networks. in *Proceedings of the 10th international conference on internet multimedia computing and service* 22:1–22:4 (ACM, 2018). doi:10.1145/3240876.3240881
31. Keshk, H. M. & Yin, X. Satellite super-resolution images depending on deep learning methods: A comparative study. in *2017 ieee international conference on signal processing, communications and computing (icspcc)* 1–7 (2017). doi:10.1109/ICSPCC.2017.8242625
32. Agard, D. A. & Sedat, J. W. Three-dimensional architecture of a polytene nucleus. *Nature* **302**, 676–681 (1983).