Alma Mater Studiorum · Università di Bologna

Scuola di Scienze

Dipartimento di Fisica e Astronomia

Corso di Laurea Magistrale in Fisica

Metodiche di super-risoluzione con deep neural network

Relatore:
Prof. Gastone Castellani

Presentata da:
Alex Baroncini

Correlatore:
Dott. Nico Curti

Anno Accademico 2017/2018

Abstract

In questo lavoro di tesi ho scritto una libreria in C++ per reti neurali chiamata Byron (Build YouR Own Neural network) ed ho implementato alcune delle reti stato dell'arte nel campo della super-risoluzione di immagini.

Sebbene il mondo delle reti neurali prediliga le GPU grazie alla loro infrastruttura che permette un altissimo grado di parallelizzazione (e quindi una velocità elevata per alcuni tipi di calcoli, come per esempio le convoluzioni presenti in moltissime reti), le principali risorse utilizzate nel mondo della bio-informatica sono computer e server a elevato numero di core. Per questo motivo Byron è ottimizzata per le CPU e per il calcolo parallelo, permettendo di ottenere risultati migliori di molte altre librerie stato dell'arte per l'implementazione di reti neurali in termini di performance.

Indice

Sigle e abbreviazioni

ANN	**A**rtificial **N**eural **N**etwork
BYRON	**B**uild **Y**ou**R** **O**wn **N**eural network
DNN	**D**eep **N**eural **N**etwork
EDSR	**E**nhanced **D**eep **S**uper **R**esolution
GEMM	**GE**neral **M**atrix **M**ultiply
ReLU	**Re**ctified **L**inear **U**nit
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SR	**S**uper **R**esolution
WDSR	**W**ide **D**eep **S**uper **R**esolution
YOLO	**Y**ou **O**nly **L**ook **O**nce

Capitolo 1

Introduzione

1.1 Riassunto dei capitoli

In questo capitolo farò una breve introduzione al mondo delle reti neurali profonde e ai vari framework per implementarle. Nel secondo capitolo parlerò dei costituenti fondamentali delle reti che poi ho implementato, ovvero i layer di convoluzione e di pixel-shuffle, e di alcune altre funzioni necessarie per i miei scopi. Nel terzo capitolo introdurrò la super-risoluzione spiegando da dove è nata e a che punto è ora, con i vari modelli che ho scelto di replicare con la mia libreria. Nel quarto capitolo illustrerò i vari risultati ottenuti confrontando le performance e la bontà dei vari modelli e dei vari framework al variare di alcuni parametri come per esempio il numero di core utilizzati. Infine nell'ultimo capitolo mosterò alcune applicazioni delle tecniche esposte e alcuni possibili lavori futuri.

1.2 Deep neural network

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 attra-

verso 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. Pitts). Da allora le ANN si sono sviluppate diventando un fenomeno emergente della realtà odierna ed evolvendosi nelle cosiddette reti neurali profonde (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 outpu 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 povere, 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 cossidetta 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 risulatato 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 formiti solamente i dati in input, sperando 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 corrispetivi 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 abbastanza usati sono le norme L1 e L2.
- **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 finchè la rete non svolge le funzioni desiderate o smette di apprendere.

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.

1.3 I vari framework

Durante gli anni sono state sviluppate moltissime 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 le principali utilizzate durante questo lavoro di tesi.

1.3.1 Darknet

Darknet è un framework per reti neurali scritto in C da Joseph Redmon, con supporto nativo solo per sistemi operativi Linux (anche se è possibile utilizzarlo anche in ambiente Windows, con delle modifiche e una gran dose di pazienza) e ottimizzato per GPU (solo CUDA, quindi solo schede grafiche NVidia) e CPU (tuttavia la libreria di default per il calcolo parallelo a cui si appoggia è esclusiva Linux). Risulta una delle migliori librerie per reti neurali attualmente disponibili e open source, in termini di performance. Tuttavia ha alcuni aspetti che possono essere migliorati, come la compatibilità tra piattaforme diverse e una migliore ottimizzazione per il 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. 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).

1.3.2 Byron

Come detto sopra, Byron è un framework in C++ basato per la maggior parte sul codice sorgente di Darknet. Tuttavia essendo stata riscritta da zero, questa libreria ha innumerevoli miglioramenti (molti permessi dallo standard del C++ che contiene funzioni molto più avanzate rispetto allo standard del C) e inoltre per alcuni aspetti critici (tra cui la gestione dei core per il calcolo parallelo) adotta strategie nuove permettendo delle performance nettamente superiori a Darknet. Inoltre Byron ha anche alcune funzioni completamente assenti in Darknet, tra cui il layer di pixel-shuffle di cui parlerò più avanti 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.

1.3.3 Pytorch e Keras

Altri framework per reti neurali molto popolari al momento sono Pytorch e Keras. 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. Come svantaggio hanno tuttavia la lentezza dovuta al linguaggio stesso, che essendo di alto livello gestisce molti parametri automaticamente e non sempre nel modo ottimale. Pytorch è un porting in Byron della libreria Torch, scritta in Lua. Keras invece è un wrapping di un'altra libreria sempre scritta in python e C++ chiamata Tensorflow, pensato per essere più user-friendly senza perdere in termini di performance. 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.

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 momento. 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 funziona su CPU. Questo problema è noto ai programmatori 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.

Per la rete YOLO, che è nativa su Darknet, la conversione dei pesi è invece stata molto più semplice, poichè il formato dei dati è simile.

Capitolo 2

Layer fondamentali

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.

2.1 Immagini

Nel mondo del processing di immagini digitali, queste vengono solitamente rappresentate come tensori tridimensionali dove le tre dimensioni rappresentano l'altezza, la larghezza e il 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 OpenCV 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.

2.2 Convoluzione

La convoluzione consiste nell'applicazione di un filtro di dimensioni ridotte su tutta l'immagine. Ciò avviene scorrendo il filtro lungo tutta l'immagine, ed effettuando l'operazione di convoluzione ad ogni passo. Nel processing di segnali questa operazione è in realtà chiamata correlazione incrociata. L'unica differenza tra le due operazioni è che nella seconda il filtro viene rovesciato prima di essere applicato all'immagine, ma visto che i pesi del filtro vengono imparati dalla rete durante l'addestramento, questo rovesciamento è superfluo. Di conseguenza nel campo delle ANN si usa il termine convoluzione intercambiandolo con correlazione incrociata. Matematicamente l'operazione di convoluzione è rappresentabile come:

$$G = h * F$$

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v] \cdot F[i-u, j-v]$$

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:

- Padding: Definisce se aggiungere dei pixel ai bordi dell'immagine prima di applicare la convoluzione. Solitamente vengono aggiunti pixel neri o riflessi rispetto al bordo. Ciò permette di gestire le dimensioni dell'immagine in output dalla convoluzione.
- Striding: 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. In questo modo la convoluzione verrà effettuata su ogni canale, e il numero di canali dell'output dipenderà dal numero di filtri che scegliamo 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 pesi dei filtri: ciò permette di insegnare alla rete ad applicare trasformazioni anche molto complesse, a seconda del numero di filtri, alla nostra immagine.

2.2.1 Ottimizzazione

Applicare direttamente la formula della convoluzione passando il filtro su tutta l'immagine è un processo molto lento. Per migliorare le performance ci sono vari algoritmi di convoluzione che sfruttano caratteristiche tipiche del calcolo parallelo e della vettorizzazione. Uno dei più utilizzati, che ho implementato in Byron, consiste in due fasi:

2.2.1.1 Im2col

Questa trasformazione "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 perchè 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) ed è molto veloce. 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 (per esempio quello utilizzato per i miei test dispone di 126 GB di ram). In figura 2.1 è rappresentato uno schema dell'operazione di im2col (image to columns) seguita dalla gemm (general matrix multiply).

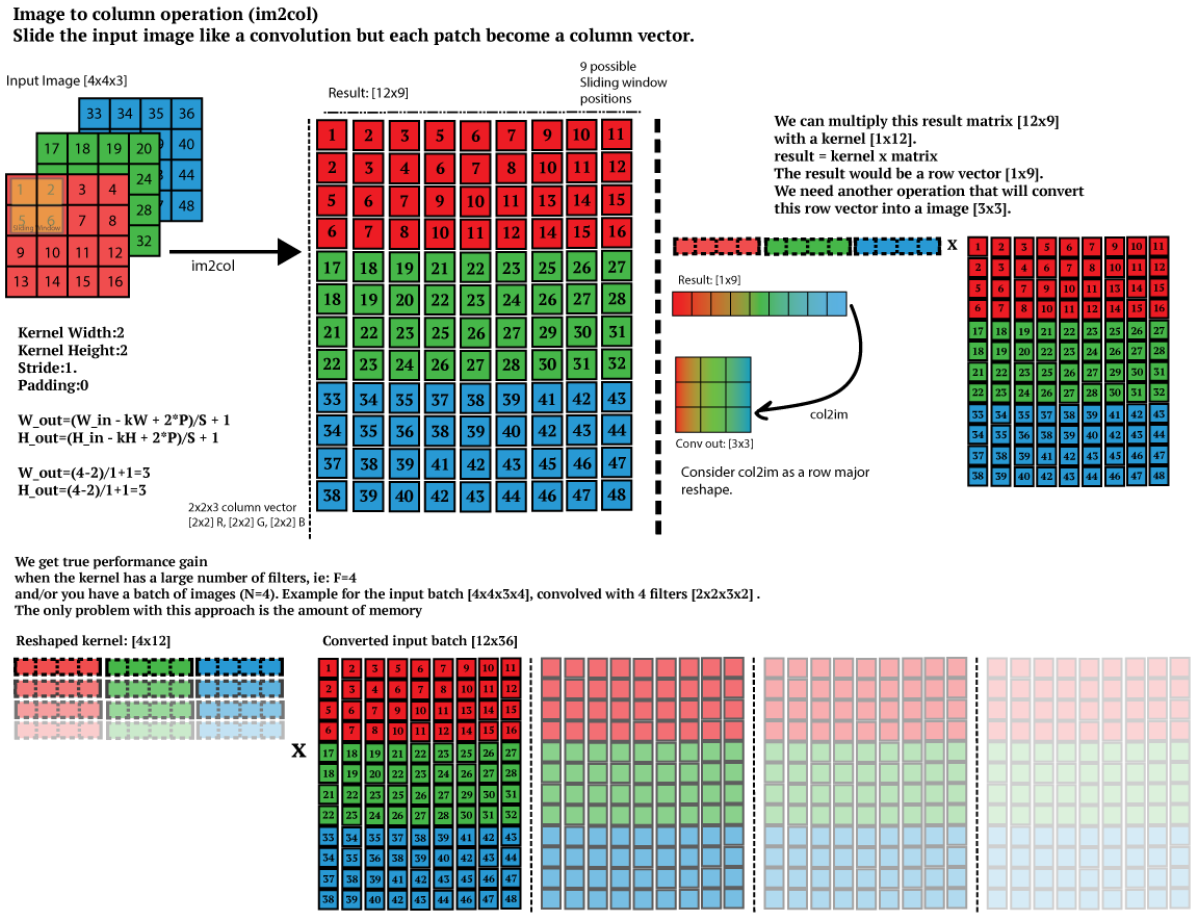

Figura 2.1: Schema esplicativo dell'im2col con un filtro 2x2 su un'immagine a 3 canali.

2.2.1.2 GEMM

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 appiatiti in modo da essere delle matrici in cui ogni riga rappresenta un filtro, e avremo quindi tante righe quanti i filtri da applicare. Questo passaggio rappresenta il vero vantaggio di questo modo di eseguire la convoluzione, in quanto per molti anni anche prima dell'avvento delle reti neurali l'operazione di moltiplicazione tra matrici è stata ottimizzata per essere il più veloce possibile. Per fare ciò è necessario seguire una serie di tecnicismi, tra cui per esempio preservare il Single Input 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 parallelizzazione (che dipende dal processore) a livello di singolo core.

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 molto più piccole. 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.

2.2.2 Problemi di concurrency

Per addestrare (in modo supervisionato) una rete neurale solitamente viene usata una procedura nota come backward propagation. Ciò permette di partire da una funzione di errore scelta a priori che valuti quanto sono differenti l'output atteso e quello ottenuto, e propagare all'indietro nella rete questo errore, correggendo i pesi di un layer alla vol-

ta. Per fare ciò ovviamente i layer hanno bisogno di funzioni che dicono 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 o problematiche. Un esempio di ciò è la funzione inversa dell'im2col, chiamata banalmente col2im. Questa funzione ovviamente deve trasformare la matrice dell'immagine appiatita e ricostruire l'immagine originale. Tuttavia come ho fatto notare precedentemente, ogni pixel dell'immagine originale viene mappato in più di una posizione nella matrice! Ne consegue che quando dobbiamo tornare indietro, vari numeri dalla matrice vanno a contribuire a un singolo pixel dell'immagine. Fin qui nessun problema: infatti basta semplicemente sommare i vari contributi per determinare il valore finale del pixel. I problemi sorgono, tuttavia, quando usiamo più di un core (o in generale più di un thread) per eseguire questa operazione: se due core cercano di scrivere nella stessa locazione di memoria insieme, non possiamo essere sicuri che il risultato sia quello che ci aspettiamo. Questo problema è chiamato concurrency. In Darknet la funzione di col2im per le CPU non risolve questo problema: ne consegue che durante l'addestramento su CPU possono nascere degli errori numerici che, sebbene piccoli, spesso si propagano in tutta l'immagine e in tutta la rete al susseguirsi dei cicli di addestramento. Questo problema ovviamente non si presenta solamente per la funzione di col2im ma è ricorrente in varie funzioni di backward di Darknet, che tuttavia non starò a specificare in quanto non saranno usate nelle reti di interesse.

In Byron questo problema è stato risolto garantendosi che l'accesso ad ogni locazione di memoria in scrittura fosse sempre sequenziale. Di conseguenza non capita mai che due o più core vadano a scrivere nella stessa locazione contemporaneamente.

2.3 Problema della scomparsa del gradiente

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 sto per parlare servano effettivamente alla rete, è importante parlare dell 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 backward propagation 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 0. 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.

Due soluzioni sempre più comuni a questo problema sono le attivazioni ReLU e l'utilizzo di blocchi residui.

2.3.1 Blocci residui

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 senza essere modificati, come se prendessero una scorciatoia, e vengono poi sommati a layer successivi della rete. Questi layer avranno quindi un residuo da un livello meno profondo della rete. Durante l'addestramento ci saranno quindi alcuni layer speciali 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.

2.3.2 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 svolgono due compiti:

- introducono un grado di non linearità nel sistema, che permette solitamente alla rete di apprendere funzioni complesse in minor tempo;
- riducono il problema di scomparsa del gradiente in quanto non sono limitate a un intervallo che, se moltiplicato molte volte, tende a zero.

Nei modelli analizzati per questo motivo molti layer avranno attivazioni ReLU mentre altri avranno semplici attivazioni lineari (ovvero il risultato dell'attivazione è identico all'output del layer).

2.4 Pixel-shuffle

Molte delle prime reti neurali per super-risoluzione preprocessavano l'immagine a bassa risoluzione di input con un upsample bicubico, di cui parlerò più avanti. 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 esponenzialmente i calcoli richiesti durante i forward dei vari layer. Per ovviare a questo problema è stato introdotto qualche anno fa un layer chiamato pixel-shuffle, 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:

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

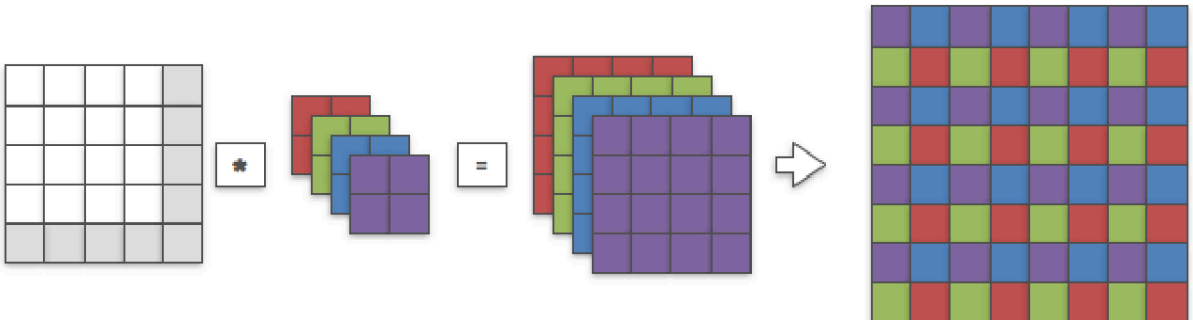

Figura 2.2: Schema esplicativo del layer di pixel shuffle applicato dopo un layer di convoluzione.

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.

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

Capitolo 3

Super-risoluzione

3.1 Le origini

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 desiderata è nascosto nel contenuto a bassa frequenza dell'immagine 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 a bassa risoluzione con la corrispetiva immagine ad alta risoluzione. Queste patch venivano solitamente prese dopo aver filtrato l'immagine con un filtro di edge detection o dalla trasformata di Fourier dell'immagine per avere direttamente il contenuto in frequenza. Una volta imparato un "dizionario" di queste associazioni, da un insieme di coppie di immagini alta e bassa risoluzione note, era possibile applicarlo su un'immagine a bassa risoluzione e ottenerne una versione ad alta risoluzione. Si noti che in questo caso non necessariamente le immagini di input e output avevano 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 automatizzata un analogo del dizionario delle patch, che in questo caso diventava un insieme di moltissime feature. Nacque così la prima ANN per super-risoluzione, chiamata SRCNN (Super Resolution Convolutional Neural Network), che consisteva semplicemente in tre layer di convoluzione. Il primo estraeva le patch a bassa risoluzione dall'immagine, il secondo collegava queste patch organizzate in un vettore a molte dimensioni a un altro vettore che idealmente rappresentava le patch ad alta risoluzione. Infine l'ultimo layer riorganizzava le patch ad alta risoluzione e le combinava per ottenere le immagini ad alta risoluzione di output.

Da allora sono stati fatti molti progressi nella ricerca in questo campo, ma le idee fondamentali non sono cambiate. Semplicemente i modelli sono molto più profondi e utilizzano quindi contromisure per riuscire a gestire l'addestramento con un numero enorme di parametri imparabili.

3.2 Preprocessing

Ogni rete a super-risoluzione deve ovviamente essere addestrata ed ha quindi bisogno di un dataset di immagini di input e di output attesi. Nelle reti di cui mi sono occupato in questo lavoro di tesi, l'immagine di input consiste di una versione ridimensionata dell'immagine di output. 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 milioni di parametri: 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. Anche se i modelli di queste reti sono pensati per uno scopo diverso, ciò non evita che le feature estratte dai numerosi layer di convoluzione presenti risentano dei filtri applicati all'input e di conseguenza durante l'addestramento i pesi

verranno aggiustati diversamente.

Per le analisi svolte in questa tesi ho considerato per semplicità (e anche per la facilità nel reperire i pesi dei modelli già precedentemente addestrati) 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.

3.2.1 Ricampionamento bicubico

Il ricampionamento bicubico, che solitamente viene diviso in upsampling e downsampling o analogamente chiamati upscaling e downscaling, consiste in un metodo di interpolazione per determinare i 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:

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

Tra i più popolari cito le opzioni B=0, C=0.75 usata da OpenCV e Photoshop o B=0, C=0.5 solitamente chiamato filtro di Catmull-Rom usato da GIMP e Matlab. 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 miglio-

ri (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 diminiuzione della intensità complessiva dell'immagine.

3.3 EDSR

Il primo modello che ho analizzato e riprodotto in Byron si chiama Enhanced Deep Super Resolution (EDSR) ed è appunto un DNN per la super-risoluzione. Questa rete si è classificata al primo posto nella NTIRE (New Trends in Image Restoration and Enhancement) challenge del 2017, in cui vari team di ricerca proponevano modelli di network con l'obiettivo di migliorare la risoluzione di un'immagine ricampionata bicubicamente. La squadra che riusciva ad avere il PSNR (Peak Signal to Noise Ratio) medio sulle immagini di validazione del dataset DIV2K più alto si aggiudicava il primo posto. Delle misure della qualità delle immagini e delle performance parlerò nel prossimo capitolo.

La struttura di base dell'EDSR è la SRResNet, una modifica della ResNet (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 normalization 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 dimostrato (ref) che per task di cosiddetta low-level vision come la super-risoluzione, dove non è necessario svolgere compiti difficili come l'object detection, mantenere una dinamicità del range di output è benefico per i risultati e non ha ripercussioni

sull'addestramento.

La struttura della rete EDSR è illustrata nell'immagine 3.1. Essa consiste in:

- Un layer di convoluzione che prende l'immagine downsampled 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 compoente ad alta frequenza estratta dai 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. A causa del grande numero di filtri e delle dimensioni delle immagini in input nei layer di

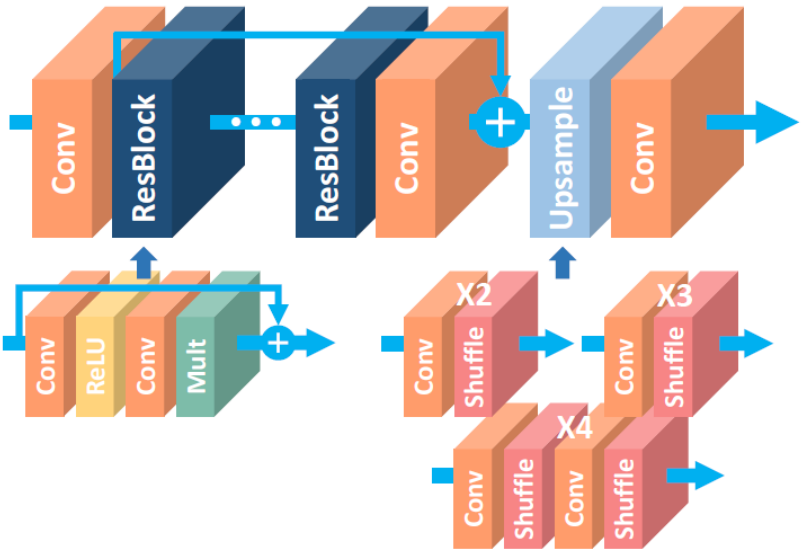

Figura 3.1: Struttura dei layer che compongono la rete EDSR.

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 3.1 riporto il numero di parametri utilizzati nei layer del modello. In totale sono oltre 43 milioni di pesi imparabili dalla rete.

Tabella 3.1: Parametri per ogni layer della rete EDSR.

Layer	Canali input/ouput	Dimensione filtri	Parametri
Convoluzione input	3 / 256	3x3	6912
Convoluzione 1 e 2 (blocco residuo)	256 / 256	3x3	589824
Convoluzione (pre shuffle)	256 / 256	3x3	589824
Convoluzione 1 e 2 (blocco upsample)	256 / 1024	3x3	2359296
Convoluzione output	256 / 3	3x3	6912

3.4 WDSR

Il secondo modello che ho analizzato e riprodotto in Byron si chiama Wide Deep Super Resolution (WDSR) 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 3.3, 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 una scorciatoia dopo il primo layer di convoluzione viene aggiunta prima del 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 3.2.
- **I blocchi residui**: Aumentare la profondità e i parametri delle reti neurali generalmente migliora le performance a discapito dei tempi di calcolo. Per migliorare effettivamente le performance senza cambiare complessità computazionale (e quindi senza aggiungere parametri), la WDSR propone dei blocchi residui leggermente diversi basati sulla congettura (ref) che i layer di attivazione ReLU, sebbene garantiscano la non-linearità 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

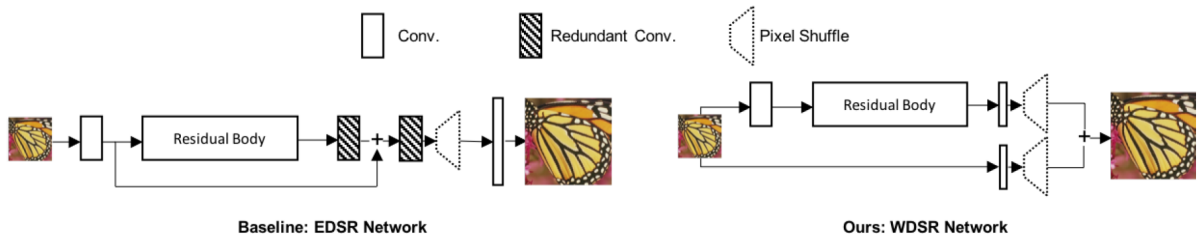

Figura 3.2: Output dei due rami della rete WDSR. A sinistra il contenuto a bassa frequenza, ottenuto semplicemente con un layer di convoluzione e un pixel-shuffle. A destra il contenuto ad alta frequenza, ottenuto dopo layer di convoluzione, blocchi residui e pixel-shuffle. In basso l'immagine finale ottenuta dalla somma dei due contributi.

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-linearità 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 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 parametri, meno di 1/10 della precedente rete utilizzata. Nella tabella 3.2 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 dimostrato (ref) 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.

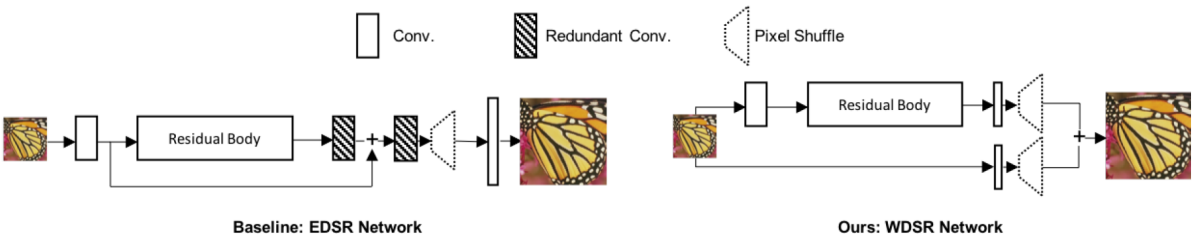

Figura 3.3: Confronto delle strutture delle reti EDSR e WDSR.

Tabella 3.2: Parametri per ogni layer della rete WDSR.

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

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 **??** 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 dimostrato (ref) 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.

boxplot

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. Come si può notare dal grafico in figura **??**, non c'è una corrispondenza uno a uno tra numero di core e speedup. Ciò significa che, per esempio, raddoppiare il numero di core non comporta un raddoppiamento della velocità di calcolo. 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. Sicuramente si può ridurre questo tempo di overhead gestendo meglio le funzioni della libreria OpenMP utilizzata per gestire il multi-threading.

catplot e spezzata collegata

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. Per questo motivo ho implementato anche in Byron una delle reti più utilizzate della libreria Darket, chiamata YOLO, di cui parlerò più approfonditamente nel prossimo capitolo. Di seguito riporto quindi 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 **??**, c'è un aumento di velocità di circa un fattore 2. Ritengo che questo speedup abbia ampi margini di miglioramento in quanto Darknet non solo utilizza in modo non ottimale il multi-threading ma può anche essere migliorata dal punto di vista dell'implementazione dei layer più costosi in termini di tempi di calcolo come per esempio il layer di convoluzione, utilizzando algoritmi recenti quali il Winograd che sono risultati molto più efficienti dell'im2col (ref).

boxplot

4.2 Qualità delle immagini

4.2.1 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. Rappresenta il rapporto tra la massima potenza del segnale e il rumore di fondo. Viene solitamente espresso in decibel (dB) perchè le immagini hanno una gamma dinamica molto ampia e quindi avere una scala logaritmica rende i numeri più gestibili. Nel caso della super-risoluzione viene usato per confrontare l'immagine super-risoluta, in output dalla rete, con quella originale ad alta risoluzione prima che venisse ricampionata bicubicamente. 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. Matematicamente 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 uno degli indici di qualità più popolari tra le immagini, 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.

In figura 4.2.1 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. Tra le due reti invece, sebbene la differenza sia meno evidente, prevale la EDSR come qualità. Tuttavia è importante notare che la 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 migliori (ref). Ciò avrebbe comportato tuttavia un notevole aumento dei tempi di calcolo.

boxplot

4.2.2 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 4.1 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:

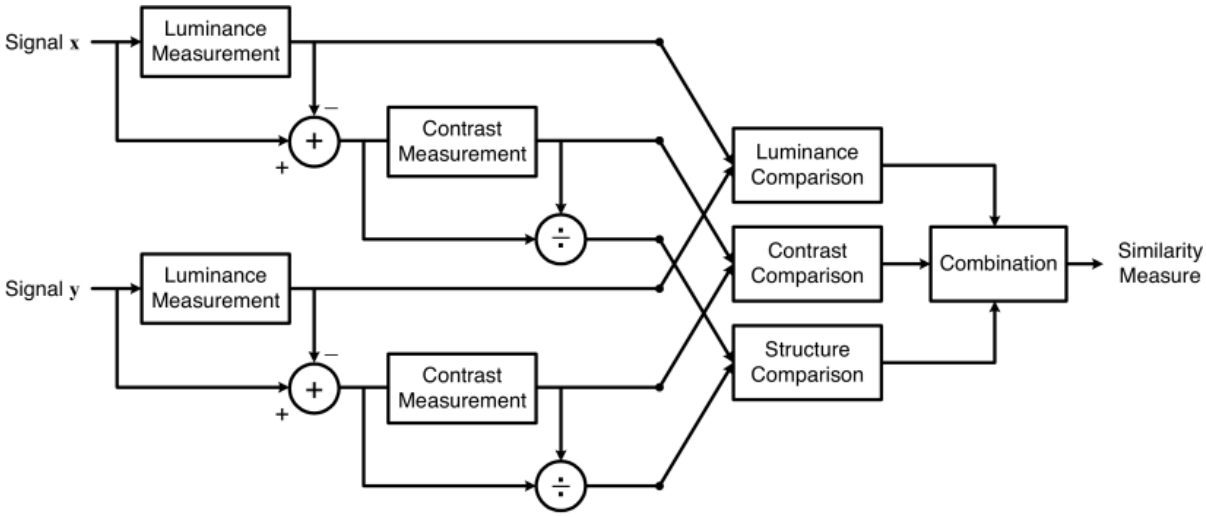

Figura 4.1: Diagramma per il calcolo dell'indice SSIM.

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

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 calcoli statistici *skimage*.

In figura 4.2.2 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.

boxplot

4.2.3 Confronto visuale

grafico tempi vs psnr di bicubica, edsr, wdsr

confronto patch bicubica / HR / EDSR / WDSR

Capitolo 5

Applicazioni

5.1 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 poi usavano classificatori su queste regioni per determinare a quale classe appartenesse l'oggetto. Con la rete YOLO (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 videocamere per esempio, e inoltre commette molti meno 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 precise.

5.1.1 Funzionamento: estrazione delle feature e classificazione

L'immagine di input viene divisa in una griglia S x 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(oggetto)* x *IOU(previsione/realtà)*. *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: x,y,w,h e la confidenza. Le coordinate rappresentano il centro del box rispettivamente ai bordi della cella e le sue dimensioni rispetto all'immagine, quindi sono tutti valori compresi tra 0 e 1. Ogni cella prevede anche C probabilità condizionali di classe *Pr(Classei /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 viene codificato in un tensore S x S x $(B$ x $5 + C)$. Il 5 è dato dalle 4 coordinate e della confidenza.

5.1.2 Modifiche delle versioni successive

La versione attuale di YOLO è la v3. Rispetto alla prima rete utilizzata la struttura del network ha subito grossi cambiamenti e nella v3 si basa sulla rete Darknet53. Alcune delle modifiche rilevanti sono:

- **Batch normalization:** Questa tecnica consiste nel normalizzare l'output di un layer scalandone e aggiustandone le attivazioni. Applicandola su tutti i layer di convoluzione del network il modello diventa molto più regolare e stabile. Ciò è vero nel caso dell'object detection, tuttavia è stato dimostrato (ref) che questa tecnica non funziona per la super-risoluzione, in quanto riduce troppo la flessibilità della rete e per ripristinare il contenuto ad alta frequenza c'è bisogno di una gamma dinamica ampia.
- **Addestramento a scala multipla:** 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.

- **Box fissi:** Per determinare i box YOLOv1 usava dei layer fully connected nel cui output si trovano direttamente le coordinate del box. Tuttavia è stato dimostrato in altre reti come la Faster R-CNN che usare dei box a dimensione fissa come priori e usare un layer di convoluzione per determinare l'offset e la confidenza dei box per la detection a partire da essi migliora le performance e rende l'addestramento più stabile. Visto che il layer è di convoluzione l'output sarà una feature map di questi offset e confidenze. Togliendo un pooling layer dalla rete per avere le dimensioni dell'output del layer di convoluzione maggiori e usando un input 608x608, avremo una feature map 19x19. Con questo cambiamento separiamo anche il meccanismo di previsione delle classi dalla posizione spaziale nell'immagine e invece calcoliamo le probabilità relative delle classi per ogni box. Avere box fissi peggiora lievemente la precisione della rete (definita come il rapporto tra i veri positivi della classificazione e la somma di veri positivi e falsi positivi) ma ne migliora notevolmente il recupero (definito come il rapporto tra i veri positivi e la somma di veri positivi e falsi negativi).
- **Cluster dimensionali:** Nella Faster R-CNN le dimensioni dei box fissi sono decise a priori e a mano. In YOLOv2 e v3 sono decise a priori ma sono state scelte dopo aver effettuato un k-mean clustering sui dataset VOC e COCO per avere un buon rapporto efficienza / precisione. Si è optato per avere 5 box fissi diversi, da cui la rete parte per trovare i box finali.
- **Previsione diretta della posizione:** Per migliorare la stabilità del network, invece di prevedere le coordinate del box fisso rispetto all'intera immagine come nella Faster R-CNN, viene usato un approcio nel quale ogni cella (della feature map) prevede 5 box e le loro coordinate relative alla cella. Ciò limita gli output tra 0 e 1 (limite imposto nel network da attivazioni logistiche), rendendo la parametrizzazione

più efficace per far imparare il network.

- **Previsione a scala multipla:** Per migliorare la precisione nella detection di oggetti piccoli e/o vicini, viene preso l'output di più layer precedenti al layer di convoluzione che crea la feature map per creare una seconda e una terza feature map, di dimensioni maggiori. Queste mappe conterranno informazioni a risoluzione spaziale più ampia, e vengono concatenate alla prima dopo essere state ridimensionate a 19x19 (per fare ciò viene aumentato il numero di canali in modo da mantenere le dimensioni effettive). In YOLOv3 abbiamo quindi il contributo di 3 scale, e ogni scala prevede 3 box (quindi abbiamo in questo caso 9 box fissi a priori).
- **Classificazione a label multipla:** Visto che alcuni dataset come per esempio l'Open Image Dataset contengono classi che si possono sovrapporre (come "donna" e "persona"), per rendere YOLOv3 più versatile su questi database si introducono indipendenti classificatori logistici in modo da prevedere per ogni box più di una label.

5.2 Super-risoluzione e object detection

Uno dei problemi principali di YOLO, oltre alla precisione nella localizzazione, è la detection di oggetti piccoli e vicini. 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 contententi persone. In entrambi i casi ho analizzato con YOLO quattro immagini: l'immagine di partenza di dimensioni 152x152 (che viene ridimensionata dalla rete a 608x608 linearmente prima dei calcoli), l'immagine ridimensionata bicubicamente con scala 4 di dimensioni 608x608 e le due immagini super-risolute dalle reti EDSR e WDSR, anche esse di dimensioni 608x608. Si possono notare dalle figure **??** e **??** alcuni dettagli:

- nel caso 1 le persone sono di dimensioni considerevoli e già nell'immagine a bassa risoluzione YOLO riesce senza problemi a trovarle tutte. Tuttavia in questo caso le probabilità di classificazione della rete sono peggiori delle immagini super-

risolute, dimostrando una precisione inferiore. Inoltre alcuni oggetti piccoli (come per esempio lo zaino nella figura) vengono rilevati solo nelle immagini SR;
- nel caso 2 le persone sono di dimensioni inferiori e infatti abbiano un numero maggiore di detection nelle immagini super-risolute su persone che YOLO non riesce a trovare nell'immagine a bassa risoluzione.

far vedere come migliora sull'immagine la detection e come migliorano le probabilità

5.3 Lavori futuri

Il miglioramento delle performance delle reti di object detection su immagini super-risolute è noto già da alcuni lavori (ref), 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 (ref super res microscopy) 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 multicore 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 l'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 Winograd 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.

Appendice

Cose varie / ringraziamenti

Capitolo 6

Bibliografia