ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Computer Vision and Image Processing M

Stima della profondità da singola immagine per mezzo di una CNN addestrata mediante tecniche di computer graphics

CANDIDATO
Michele Francesco Di Lella

RELATORE:
Chiar.mo Prof. Luigi di Stefano

CORRELATORE
Ing. Alessio Tonioni

Anno Accademico 2015/16

Sessione III

Indice

Introduzione

Nel campo della computer vision, la stima della profondità di una scena ha da sempre rappresentato un problema di massimo interesse. L'informazione sulla profondità costituisce infatti un elemento molto importante, ed in alcuni casi fondamentale, per molte applicazioni pratiche della visione artificiale, come la guida automatica, la ricostruzione 3D e la realtà aumentata. Nel corso degli anni, molte tecniche sono state sviluppate con il fine di migliorare sempre più la qualità della stima della profondità. Dispositivi come le telecamere stereo, i sensori ad infrarossi, i sensori time-of-flight e gli scanner laser 3D hanno permesso, più o meno bene, di risolvere il problema.

Questi dispositivi, tuttavia, sono spesso difficili o scomodi da utilizzare, e spesso possono risultare anche molto costosi. Quelli più accessibili, come ad esempio il kinect (1) di Microsoft, molto spesso, inoltre, non danno risultati molto precisi. Per questi motivi, dispositivi di questo tipo non hanno mai trovato una vera diffusione in attività commerciali. Nell'ultimo decennio, al contrario, il dispositivo di visione artificiale che ha trovato una diffusione grandissima è la semplice telecamera a colori, soprattutto grazie all'enorme sviluppo e diffusione dei dispositivi mobili come gli smartphone.

Tutto questo ha portato l'attenzione sullo sviluppo di sistemi in grado di stimare la profondità di una scena a partire dalla sua rappresentazione come immagine monoculare a colori. Nello stesso periodo, il mondo della computer vision ha trovato nel machine learning uno strumento che ha incrementato moltissimo le prestazioni rispetto alle classiche tecniche di programmazione. E proprio grazie al machine learning è stato possibile sviluppare i primi sistemi accettabili per la stima della profondità da singola immagine. Tuttavia, questi sistemi presentano il limite di dipendere molto dallo scenario nel quale sono stati addestrati.

L'obiettivo del lavoro di questa tesi è stato dunque lo sviluppo di un sistema di machine learning per la stima della profondità da singola immagine, adattabile

a diversi contesti, e testato nel caso specifico dello scenario degli scaffali di un supermercato.

Al fine dello sviluppo del sistema di machine learning, in collaborazione con un altro lavoro di tesi è stato inoltre prodotto un nuovo database di immagini per l'addestramento, o dataset, relativo allo scenario in esame. Al fine di velocizzare il processo, e di produrre mappe di profondità più precise rispetto a quelle ottenibili con una telecamera stereo, il dataset è stato prodotto con l'ausilio di tecniche di computer graphics. Il sistema sviluppato è stato alla fine testato su un dataset di immagini reali per verificarne la qualità.

Al fine di illustrare al meglio il lavoro svolto, questa tesi è stata strutturata nella seguente maniera.

Nel primo capitolo viene data una panoramica sul machine learning e sui principali sistemi utilizzati nel campo specifico della computer vision.

Nel secondo capitolo è illustrato nel dettaglio tutto il lavoro che ha permesso di produrre il dataset sintetico, insieme ad una breve descrizione dei sistemi di computer graphics utilizzati.

Nel terzo capitolo viene data una panoramica sui principali sistemi esistenti di machine learning per la stima della profondità da singola immagine, per poi illustrare nel dettaglio il sistema di machine learning che è stato utilizzato per raggiungere l'obiettivo descritto precedentemente.

Infine, nel quarto capitolo verrà illustrato nel dettaglio il lavoro svolto per l'addestramento del sistema di machine learning prodotto, con l'attenzione su una nuova metrica, detta smoothness, introdotta ed utilizzata per la prima volta nel campo del machine learning come contributo originale di questa tesi. Conclusivamente sono illustrati i risultati che il sistema prodotto ha permesso di raggiungere.

Capitolo 1
Il Machine Learning

In questo capitolo verrà illustrato un quadro generale sul machine learning e sulle varie tecniche relative, con un approfondimento particolare per gli argomenti direttamente correlati con il lavoro svolto in questa tesi.

1.1 Panoramica

L'apprendimento automatico, o **machine learning**, è il campo dell'informatica che fornisce ai computer la capacità di imparare ad eseguire un task senza essere stati esplicitamente programmati per la sua esecuzione. Evolutosi dagli studi sul riconoscimento di pattern e sull'apprendimento computazionale teorico nel campo dell'intelligenza artificiale, il machine learning esplora lo studio e la costruzione di algoritmi che permettono l'apprendimento di informazioni a partire da dati disponibili e forniscono la capacità di predire nuove informazioni alla luce di quelle apprese. Attraverso la costruzione di un modello che impara automaticamente a predire nuovi dati a partire da osservazioni, questi algoritmi superano il classico paradigma delle istruzioni strettamente statiche. Il machine learning trova il suo impiego principale in quell'insieme di problemi di computazione in cui la progettazione e l'implementazione di algoritmi ad-hoc non è praticabile o è poco conveniente. Esempi di applicazioni si possono trovare nei motori di ricerca, nel riconoscimento ottico di caratteri (OCR) o più in generale tra le applicazioni della computer vision, campo dell'informatica del quale ci si è occupati in questo lavoro di tesi.

Il machine learning presenta profondi legami col campo dell'ottimizzazione matematica, il quale fornisce metodi, teorie e domini di applicazione. Molti problemi di apprendimento automatico, infatti, sono formulati come problemi di minimizzazione di una certa funzione di perdita (**loss function**) nei confronti di un determinato set di esempi (**training set**). Questa funzione esprime la discrepanza tra i valori predetti dal modello in fase di allenamento e i valori

attesi per ciascuna istanza di esempio. L'obiettivo finale è dunque quello di insegnare al modello la capacità di predire correttamente i valori attesi su un set di istanze non presenti nel training set (test set) mediante la minimizzazione della loss function in questo insieme di istanze. Questo porta ad una maggiore generalizzazione delle capacità di predizione.

Una più formale ed ampiamente citata definizione per il machine learning è stata formulata da Tom M. Mitchell: "*Si dice che un programma per computer impara dall'esperienza E nei confronti di una certa classe di compiti C e insieme di misure di performance P se le sue performance sui compiti di C, secondo le misure definite in P, migliorano grazie all'esperienza E.*" (2)
Questa definizione è rilevante poiché formula il machine learning in termini fondamentalmente operazionali piuttosto che cognitivi, e dunque segue la proposta formulata da Alan Turing nel suo paper "Computing Machinery and Intelligence" (3) secondo la quale la frase "Le macchine possono pensare?" risulta poco appropriata e dovrebbe essere sostituita da "Possono le macchine fare quello che noi (come esseri pensanti) possiamo fare?".

1.2 Tipi di problemi e task

I diversi task del machine learning sono tipicamente classificati in tre ampie categorie, caratterizzate dal tipo di feedback su cui si basa il sistema di apprendimento:

- **Apprendimento supervisionato:** vengono presentati al computer degli input di esempio ed i relativi output desiderati, con lo scopo di apprendere una regola generale in grado di mappare gli input negli output. Questo scenario è quello di interesse per il lavoro di questa tesi;
- **Apprendimento non supervisionato:** al computer vengono forniti solo dei dati in input, senza alcun output atteso, con lo scopo di apprendere una qualche struttura nei dati d'ingresso. L'apprendimento non supervisionato può rappresentare un obiettivo a se stante (ad esempio per la scoperta di pattern nascosti nei dati) o essere rivolto

all’estrapolazione di caratteristiche salienti dei dati (**feature**) utili per l’esecuzione di un altro task di machine learning;

- **Apprendimento con rinforzo:** il computer interagisce con un ambiente dinamico nel quale deve raggiungere un certo obiettivo (ad esempio, guidare un’automobile o affrontare un avversario in un gioco). Man mano che il computer esplora il dominio del problema, gli vengono forniti dei feedback in termini di ricompense o punizioni, in modo da indirizzarlo verso la soluzione migliore.

Un altro metro di giudizio secondo il quale è possibile distinguere diverse categorie di task è il tipo di output atteso da un certo sistema di machine learning. Tra le principali categorie troviamo:

- La **classificazione**, nella quale gli input sono divisi in due o più classi e il sistema di apprendimento deve produrre un modello in grado di assegnare ad un input una o più classi tra quelle disponibili. Questi tipi di task sono tipicamente affrontati mediante tecniche di apprendimento supervisionato. Un esempio di classificazione è l’assegnamento di una o più etichette ad una immagine in base agli oggetti o soggetti contenuti in essa;
- La **regressione**, concettualmente simile alla classificazione con la differenza che l’output ha un dominio continuo e non discreto. Anch’essa è tipicamente affrontata con l’apprendimento supervisionato. Un esempio di regressione è rappresentato dal task che viene affrontato in questa tesi, ovvero la stima della profondità di una scena a partire dalla sua rappresentazione sotto forma di immagine a colori. Infatti, il dominio dell’output in questione è virtualmente infinito, e non limitato ad un certo insieme discreto di possibilità;
- Il **clustering**, nel quale, come nella classificazione, un insieme di dati viene diviso in gruppi che però, a differenza di questa, non sono noti a priori. La natura stessa dei problemi appartenenti a questa categoria li rende tipicamente dei task di apprendimento non supervisionato.

1.3 Approcci al machine learning

Allo stato attuale, esistono diversi tipi di approcci e tecniche per la progettazione e l'implementazione di sistemi computerizzati per l'apprendimento automatico. Tra i più importanti troviamo:

- Gli **alberi di decisione**: l'apprendimento fa uso, appunto, di un albero di decisione come modello di predizione. Il modello risultante permette di mappare le osservazioni riguardanti un oggetto a determinate conclusioni riguardanti il valore obiettivo relativo a quell'oggetto;
- Il **clustering**: come illustrato precedentemente, è una tecnica che mediante l'analisi di un insieme di dati permette la suddivisione di questo in sottoinsiemi (detti cluster) accomunati da uno o più criteri di similitudine;
- La **programmazione logica induttiva**: dall'inglese inductive logic programming (ILP), costituisce un approccio all'apprendimento di regole che utilizza la programmazione logica per la rappresentazione degli esempi di input, della conoscenza di base e delle ipotesi. A partire da una certa rappresentazione della conoscenza di base e del set di esempi sotto forma di fatti logici, un sistema di ILP può produrre un programma logico in grado di implicare tutti gli esempi positivi e non quelli negativi;
- Gli **algoritmi genetici**: sono algoritmi di ricerca euristica che cercano di imitare il processo della selezione naturale mediante l'uso di tecniche quali la mutazione e l'incrocio, con l'obiettivo di generare un nuovo genotipo (anche detto cromosoma, ovvero un insieme di parametri che definiscono una determinata soluzione) che rappresenti la soluzione migliore ad un determinato problema;
- Le **reti neurali artificiali**: un algoritmo di apprendimento mediante rete neurale artificiale, comunemente chiamato rete neurale, è un algoritmo che si ispira, sia dal punto di vista strutturale che del funzionamento, alle reti neurali biologiche. La computazione è strutturata in termini di gruppi interconnessi di neuroni artificiali e

segue un approccio di tipo connettivista in cui il risultato si manifesta come comportamento emergente di un insieme interconnesso di unità semplici. Le reti neurali vengono spesso impiegate per la modellazione di relazioni complesse tra dati di input e output corrispondenti e per la ricerca di strutture nascoste nei dati;

- Il **deep learning**: rappresenta un'evoluzione delle reti neurali incoraggiata anche dal calo dei prezzi e dal grande sviluppo tecnico nel campo delle GPU general purpose avvenuto negli ultimi anni. Il deep learning si basa sull'utilizzo di particolari reti neurali costituite da una moltitudine di livelli nascosti di neuroni. Questo approccio trae ispirazione dal funzionamento della parte del cervello umano che si occupa della visione e dell'udito, e cerca di modellarne la struttura. Proprio per questo motivo, applicazioni in cui il deep learning ha trovato grande successo sono la visione artificiale (computer vision) ed il riconoscimento vocale (speech recognition).

Nelle sezioni seguenti verrà posta l'attenzione sugli approcci delle reti neurali e del deep learning poiché, come è possibile intuire dalle applicazioni di successo di quest'ultimo indicate precedentemente, costituiscono parte integrante delle conoscenze utilizzate per il raggiungimento degli obiettivi di questo lavoro di tesi.

1.4 Reti neurali artificiali

1.4.1 Modello e struttura

Come già accennato nel paragrafo precedente, le reti neurali sono modelli di apprendimento automatico che cercano di imitare la struttura ed il funzionamento del cervello biologico, costituito da grossi ammassi di neuroni collegati tra loro dagli assoni, mediante l'utilizzo di insiemi di unità neurali, dette neuroni artificiali, interconnesse tra loro a formare una rete.

Ogni unità neurale è connessa con molte altre, ed il collegamento può essere di tipo rafforzativo o inibitorio nei confronti dell'attivazione delle unità a cui è connesso. Ogni neurone contiene una funzione adibita a combinare tra di loro i

valori di tutti i suoi input ed una funzione, detta funzione di attivazione, che restituisce l'output del neurone. La forma generale della funzione complessiva contenuta in un neurone è rappresentata dalla seguente formula:

$$y = f\left(\sum_i w_i x_i + b\right)$$

In questa, w_i sono i pesi assegnati a ciascun input in fase di combinazione e b un termine, detto **bias**, che viene aggiunto in seguito. L'insieme dei pesi e del bias rappresenta l'informazione che il neurone apprende in fase di addestramento e che conserva successivamente. La funzione f rappresenta la funzione di attivazione, che di norma consiste in una funzione di soglia o di limitazione che fa in modo che solo segnali con valori compatibili con la soglia o il limite imposto possano propagarsi al neurone o ai neuroni successivi. Tipicamente la funzione di attivazione è una funzione non lineare, e solitamente si tratta di una funzione gradino, una sigmoide o una funzione logistica.

Le reti neurali sono strutturate tipicamente in tre parti, contenenti quantità distinte di neuroni:

- Un livello di input;
- Un insieme più o meno numeroso di livelli interni nascosti;
- Un livello di output.

I segnali in ingresso attraversano l'intera rete dal livello di input a quello di output, passando dai neuroni degli strati interni, come illustrato in figura 1.1.

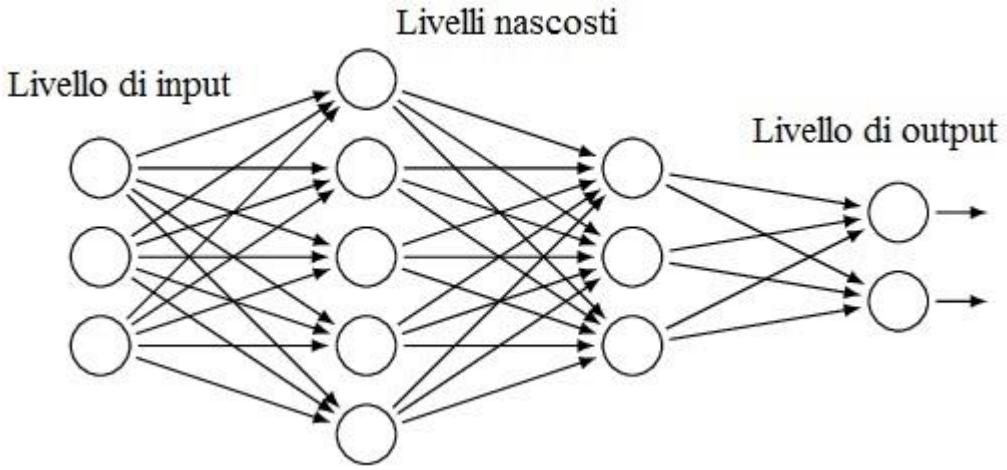

Figura 1.1: Struttura di esempio di una rete neurale con due livelli nascosti. Ogni nodo rappresenta un neurone artificiale e le frecce rappresentano i collegamenti tra i neuroni.

1.4.2 L'addestramento

Nel caso dell'apprendimento supervisionato, la fase di addestramento di una rete consiste nello stimare i pesi w_i, contenuti in ciascun neurone che minimizzano l'errore tra i valori di output attesi dai dati di training e i valori predetti dalla rete. La funzione che calcola questo errore può essere di diverso tipo e, come accennato nei paragrafi precedenti, viene comunemente chiamata loss function.

La stima dei pesi si può ottenere con tecniche note di ottimizzazione matematica. La tecnica che tipicamente viene utilizzata è quella della discesa del gradiente (o **gradient descent**) mediante **back propagation**. Si tratta di una tecnica di tipo ciclico in due fasi: la propagazione e l'aggiornamento dei pesi.

Nella prima fase di propagazione in avanti (**forward step**), gli input attraversano l'intera rete dal livello di input fino a quello di output. Dopo la propagazione, si recuperano gli output prodotti ed attraverso la loss function viene calcolato l'**errore di predizione** rispetto agli output attesi. Questo errore viene quindi utilizzato per il calcolo del gradiente della loss function, che viene poi propagato all'indietro nella rete (da cui back propagation) fino a che ogni neurone ottiene il suo valore di gradiente.

A questo punto inizia la fase di aggiornamento dei pesi. In questa fase il valori calcolati del gradiente vengono dati in pasto all'algoritmo di discesa del gradiente, che li utilizza per aggiornare i pesi di ciascun neurone, con l'intento di minimizzare il valore della loss function.
Nello specifico, per trovare il minimo della loss function (o avvicinarsi il più possibile ad esso in tempi accettabili), la discesa del gradiente aggiorna i pesi della rete utilizzando step proporzionali al valore (negato) che il gradiente assume in un certo momento della computazione. Facendo in questo modo, l'output della rete viene indirizzato nella direzione del gradiente, con l'effetto di ridurre il valore della loss function ciclo dopo ciclo.
La costante di proporzionalità utilizzata per l'aggiornamento dei pesi viene detta **learning rate**. Nell'utilizzo reale questo termine è solitamente variabile e generalmente ha un andamento decrescente col protrarsi dell'addestramento. In questo modo, la rete è in grado di effettuare passi più grandi nella direzione del gradiente all'inizio dell'addestramento, quando si trova generalmente distante dal valore minimo, e ridurre l'entità degli aggiornamenti quando comincia ad avvicinarsi ad esso, in modo da facilitare la convergenza ed evitare divergenze indesiderate.
Un possibile scenario negativo derivante dall'utilizzo di questa tecnica si presenta quando la ricerca del minimo rimane bloccata in corrispondenza di un minimo locale, che impedisce alla rete il raggiungimento del risultato atteso.

Per quanto concerne la gestione dei cicli di addestramento della rete, esistono due modalità di esecuzione: quella **stocastica** e quella in gruppo. Nell'apprendimento di tipo stocastico, ogni passo di propagazione in avanti è seguito immediatamente da un passo di aggiornamento dei pesi. Nell'apprendimento in gruppo, specularmente, viene effettuato il passo di propagazione per ogni esempio nel training set e solo successivamente i pesi vengono aggiornati utilizzando il gradiente accumulato tra tutte le propagazioni. L'apprendimento stocastico introduce generalmente una certa quantità di "rumore" nel processo di addestramento, poiché ad ogni step utilizza il gradiente calcolato in un singolo punto dello spazio dei dati. Una conseguenza di questo è la riduzione del rischio di restare bloccati in un minimo locale ma anche l'aumento del rischio di divergenza o non

convergenza. L'addestramento in gruppo, invece, ha il vantaggio di portare direttamente al risultato finale, dal momento che l'aggiornamento dei pesi viene effettuato tenendo conto dell'intero training set. Purtroppo, la sua implementazione risulta infattibile a causa di grossi limiti per quanto riguarda le performance e le risorse di memoria necessarie. Un compromesso tipicamente adottato nelle applicazioni moderne rappresenta una via di mezzo tra i due approcci e consiste nell'utilizzo dei cosiddetti "**mini-batch**", ovvero insiemi di piccole dimensioni contenenti esempi selezionati in modo casuale tra i dati disponibili.

Uno degli aspetti più importanti ed interessanti dell'addestramento è la capacità dei neuroni degli strati intermedi di organizzarsi autonomamente in modo che ciascuno di essi impari a riconoscere differenti caratteristiche nello spazio degli input. Successivamente alla fase di addestramento, nel momento in cui un nuovo input verrà dato in pasto alla rete, i neuroni dei livelli nascosti saranno in grado di rispondere con uno stato attivo se il determinato input conterrà un certo pattern che in qualche modo somiglia ad una caratteristica che ciascun neurone ha imparato a riconoscere.

1.4.3 Progettazione

Come è possibile evincere dallo schema di funzionamento delle reti neurali, il più grande vantaggio che queste apportano nel campo della computazione artificiale è probabilmente la capacità di approssimare una funzione arbitraria imparando da un insieme di dati osservati. Tuttavia, il funzionamento desiderato da una certa rete non è semplice da ottenere, e la progettazione costituisce un punto delicato dello sviluppo di un sistema di machine learning basato su reti neurali. La progettazione può essere divisa in due punti chiave:

- La **scelta del modello**: è una decisione che dipende dalla rappresentazione dei dati disponibili e dal tipo di applicazione per cui la rete viene progettata. Modelli troppo semplici potrebbero portare ad una fase di addestramento difficile da portare a termine (situazione detta **underfitting**, in cui la rete fatica ad apprendere informazioni), mentre al contrario modelli troppo complessi potrebbero portare la rete a

dipendere troppo dai dati di esempio, limitando le capacità di generalizzazione (situazione nota come **overfitting**, in cui la rete impara a gestire solo le informazioni contenute nei dati di esempio);

- La **scelta dell'algoritmo di apprendimento**: esistono diversi compromessi di cui tener conto tra l'utilizzo di un determinato algoritmo piuttosto che di un altro. Generalmente, qualunque algoritmo lavora bene su un particolare set di dati una volta scelti in modo corretto i cosiddetti **iperparametri**, ovvero le variabili che ne caratterizzano il funzionamento e che sono scelte a priori prima del processo di ottimizzazione. Tuttavia, l'operazione di scelta di un algoritmo e di tuning dei suoi parametri per l'addestramento su nuovi dati richiede un abbondante quantità di sperimentazioni.

Se le operazioni di scelta del modello, della loss function e dell'algoritmo di apprendimento vengono effettuate in maniera corretta, la rete neurale che ne risulta può presentare caratteristiche di robustezza molto elevate.

Inoltre, con una corretta implementazione, le reti neurali possono essere utilizzate con successo anche per l'addestramento di tipo "**online**", in cui i dati non sono presenti nella loro totalità fin dall'inizio ma vengono piuttosto forniti alla rete man mano che diventano disponibili. Inoltre, l'implementazione relativamente semplice e la presenza di dipendenze strutturali per la maggior parte di tipo locale rende possibile implementare le reti direttamente su hardware in maniera parallela e veloce.

1.5 Deep learning

Nella sua concezione più generale, il **deep learning** è una branca del machine learning che si basa sull'utilizzo di algoritmi il cui scopo è la modellazione di astrazioni di alto livello sui dati. Fa parte di una famiglia di tecniche mirate all'apprendimento di metodi per rappresentare i dati.

Un'osservazione, ad esempio un'immagine, può essere rappresentata in diversi modi, come un vettore di valori di intensità per ogni pixel, o in una maniera più astratta come un insieme di bordi, di regioni che presentano una particolare forma o una particolare caratteristica saliente. Alcune di queste possibili rappresentazioni possono risultare migliori rispetto ad altre nel task di facilitare l'operazione di addestramento di un altro sistema di apprendimento automatico. A tal riguardo, uno degli impieghi principali del deep learning consiste nella creazione di algoritmi di apprendimento specializzati nell'estrapolazione automatica di caratteristiche salienti (**feature**) in un set di dati, da utilizzare in seguito per l'addestramento di sistemi di machine learning. Il contributo apportato è altamente rilevante se si pensa che senza queste tecniche le suddette feature dovrebbero essere prodotte e valutate manualmente e preliminarmente all'addestramento.

Il concetto cardine su cui si basa il deep learning è quello di sottoporre i dati di ingresso a numerosi livelli di elaborazione in cascata il cui risultato finale è l'emergere delle feature di cui si è parlato. Come verrà illustrato più nel dettaglio nei paragrafi seguenti, nel campo delle reti neurali questo concetto è stato messo in pratica con l'aggiunta di numerosi livelli nascosti di neuroni.

1.5.1 Reti neurali profonde

Nel campo delle reti neurali, il deep learning è stato introdotto attraverso la definizione delle cosiddette reti neurali profonde (**deep neural network**). Il principio di funzionamento è lo stesso delle reti neurali classiche, con la differenza che risiede nel numero molto elevato di livelli nascosti di neuroni intermedi.

Come le reti neurali classiche, le deep neural network sono in grado di modellare relazioni di tipo complesso tra dati di input e output. Tra le applicazioni di maggior successo troviamo la computer vision, con task che includono la classificazione, la regressione di immagini e l'object detection. Nel caso di esempio di quest'ultimo task, una rete neurale profonda è in grado di generare una rappresentazione stratificata degli oggetti in cui ciascun oggetto è individuato da un insieme di caratteristiche aventi la forma di primitive visive, ovvero particolari bordi, linee orientate, texture e pattern ricorrenti. Un esempio di feature prodotte da una rete neurale profonda applicata al campo della computer vision è mostrato in figura 1.2. Questa capacità di modellazione pone le sue fondamenta proprio nell'elevato numero di livelli nascosti di neuroni.

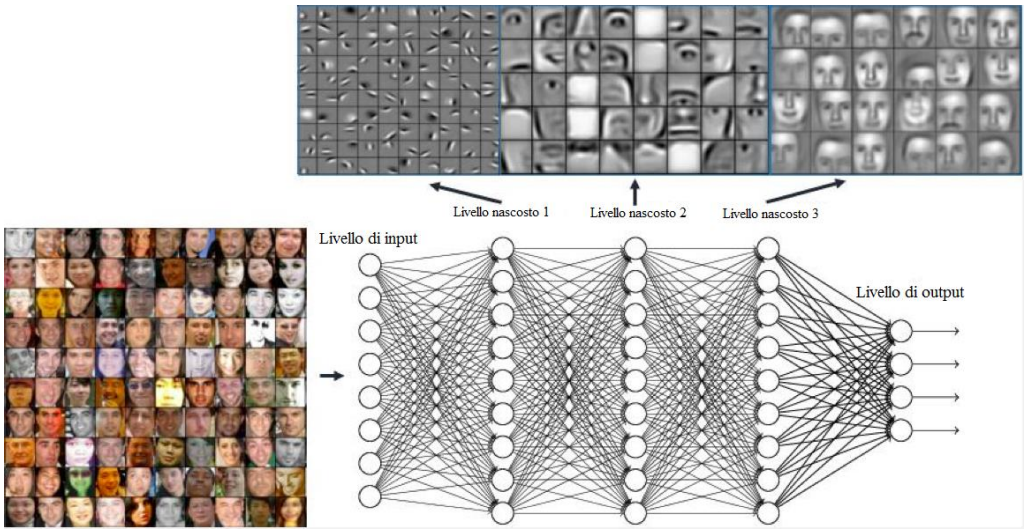

Figura 1.2[1]: Esempio di deep neural network applicata al campo del face recognition. Ogni livello della rete impara a riconoscere particolari tipi di feature provenienti dalle immagini di input. Nella parte finale della rete, tutte queste feature vengono combinate insieme, in modo da generar[1]e la predizione vera e propria.

Per quanto riguarda l'addestramento, il classico algoritmo di back propagation può ancora essere utilizzato. Come nelle reti neurali classiche, anche nelle deep network si può presentare il già citato problema dell'overfitting. Al fine di contrastarlo vengono solitamente utilizzate delle tecniche, dette di regolarizzazione, che vanno ad influenzare il processo di ottimizzazione durante l'addestramento.

[1] RSIP VISION. http://www.rsipvision.com/exploring-deep-learning/

Tra le più utilizzate troviamo:

- la **regolarizzazione l_2** (weight decay), in cui il lavoro dell'ottimizzatore viene influenzato sommando alla loss function la norma l_2 dei pesi della rete moltiplicata per una determinata costante;
- la **regolarizzazione l_1** (sparsity), che si comporta come la precedente ma utilizza la norma l_1;
- il **dropout**, in cui ad ogni passo dell'addestramento un certo numero di neuroni scelti casualmente nei livelli nascosti viene "spento", impedendo che il suo output si propaghi ai neuroni successivi.

Tutte queste tecniche hanno l'effetto di ridurre le dipendenze tra la rete e i dati di esempio, e contribuiscono a ridurre l'entità dell'overfitting.

Benché l'addestramento mediante back propagation rappresenti una buona soluzione, grazie alla semplicità di implementazione ed alla sua tendenza nel convergere verso minimi locali migliori rispetto ad altre tecniche, nel caso delle reti profonde può presentare seri problemi riguardanti la computazione vera e propria. Esiste, infatti, una moltitudine di iperparametri da considerare nelle deep neural network, come le dimensioni (in termini di numeri di livelli e numero di unità per ogni livello), il learning rate e i pesi iniziali, e l'operazione di ottimizzazione di questi parametri può diventare poco gestibile in termini di tempi e di risorse di calcolo.
Diverse soluzioni sono state proposte al riguardo, tra cui il già citato utilizzo di mini-batch di dati che velocizza l'addestramento, ma il vero passo in avanti è stato fatto con lo sviluppo negli ultimi anni di GPU con potenza di calcolo sempre crescente. Nelle reti, infatti, il principale tipo di computazioni effettuate riguarda operazioni tra matrici e vettori, e queste sono particolarmente adatte all'implementazione parallela sull'hardware delle GPU.

Un altro problema derivante dall'utilizzo di tecniche basate sulla discesa del gradiente ed accentuato dalla complessa struttura delle reti profonde è quello della degradazione del gradiente (**vanishing gradient problem**). Il problema deriva dal sistema di calcolo del gradiente in catena e dall'elevato numero di livelli della rete. Le funzioni di attivazione tipicamente utilizzate tendono infatti a generare gradienti con valori molto piccoli (solitamente nel range

[-1,1]) e questo, a causa appunto della computazione in catena, porta a moltiplicare tra loro n valori molto piccoli nel momento in cui viene effettuato il calcolo del gradiente in uno dei livelli iniziali di una rete ad n livelli. Ne consegue che il gradiente decresce esponenzialmente con n e che i livelli inziali apprenderanno in maniera molto lenta.

Questo problema è stato riscontrato fin dalle prime implementazioni di reti neurali profonde, e diverse soluzioni sono state proposte. Tra esse è da menzionare la proposta di Jürgen Schmidhuber (4), secondo il quale è possibile addestrare preliminarmente in modo non supervisionato un livello della rete per volta, per poi effettuare un addestramento completo finale mediante back propagation.

Un altro modo per aggirare il problema, incentivato dal già citato sviluppo delle GPU avvenuto nei tempi recenti, è l'utilizzo di hardware più veloce per andare a contrastare quello che è il sintomo principale del problema, ovvero la lentezza del processo di addestramento.

La soluzione, proposta recentemente, che però sembra più promettente ed efficace è rappresentata dalle cosiddette reti neurali residuali (**residual neural network** (5)). Un'illustrazione approfondita di questo tema sia in ambito generale che specifico è rimandata al capitolo 3, riguardante la rete neurale utilizzata per il progetto, in cui verrà illustrata una particolare architettura che è stata poi sfruttata per il raggiungimento degli obiettivi posti per il progetto.

Un altro punto a sfavore delle deep neural network, nel caso dell'apprendimento supervisionato, è rappresentato dalla quantità molto elevata di dati di esempio (comprensivi di output desiderato) necessari affinché la rete raggiunga i risultati preposti al termine dell'addestramento. Ciò rappresenta un ostacolo non di poco conto poiché, per determinati task come ad esempio quello della stima della profondità affrontato in questa tesi, la produzione degli output attesi per ciascun esempio (nel caso specifico la mappa di profondità di una scena, la cui produzione necessita di un'annotazione pixel per pixel) è un'operazione che può necessitare di tempi molto lunghi. Questo limite rappresenta il motivo principale per cui, come verrà illustrato approfonditamente nel capitolo 2, si è deciso di optare per la produzione di un dataset sintetico mediante l'utilizzo della grafica computerizzata.

Negli anni di sviluppo sono state proposte diverse architetture di reti neurali profonde. Tra queste, le reti neurali convoluzionali (**convolutional neural network**, in breve CNN) hanno riscontrato un grande successo soprattutto nell'ambito della visione artificiale, nonché nel campo del riconoscimento vocale.
Nel paragrafo che segue verrà illustrata in maniera più approfondita l'architettura delle CNN che, forti di risultati che rappresentano lo stato dell'arte attuale nell'ambito del machine learning applicato alla computer vision, sono diventate di fatto la scelta principale in questo campo.

1.5.2 Reti neurali convoluzionali

Nel campo del machine learning, le reti neurali convoluzionali (CNN o ConvNet) rappresentano un tipo di reti neurali in cui il pattern di connessione trai neuroni si ispira alla struttura della corteccia visiva nel mondo animale. I singoli neuroni presenti in questa parte del cervello rispondono a determinati stimoli in una regione ristretta dell'osservazione, definita campo recettivo (**receptive field**). I campi recettivi di neuroni differenti sono parzialmente sovrapposti in modo che complessivamente ricoprano l'intero campo visivo. La risposta di un singolo neurone a stimoli che hanno luogo nel suo campo recettivo può essere approssimata matematicamente da un'operazione di convoluzione. Grazie a tutto questo, come già accennato, le reti convoluzionali rappresentano lo stato dell'arte nella visione artificiale.

Le CNN sono progettate per riconoscere pattern visivi direttamente in immagini rappresentate da pixel e richiedono una quantità di preprocessing nulla o comunque molto limitata. Sono in grado di riconoscere pattern estremamente variabili, come ad esempio la scrittura a mano libera e le immagini rappresentanti il mondo reale. Tipicamente, una CNN consiste in diversi livelli alternati di convoluzione e di sottocampionamento (subsampling o **pooling**) seguiti da uno o più livelli finali completamente connessi (**fully connected**) nel caso della classificazione, o da un certo numero di livelli di sovracampionamento (**upsampling**) nel caso della regressione. In quest'ultimo caso si parla di reti neurali completamente convoluzionali (**fully convolutional**

neural network, o FCN). La rete utilizzata nel progetto finale si tratta proprio di una FCN, e verrà illustrata nel capitolo 3. Una struttura di esempio di rete convoluzionale è mostrata in figura 1.3.

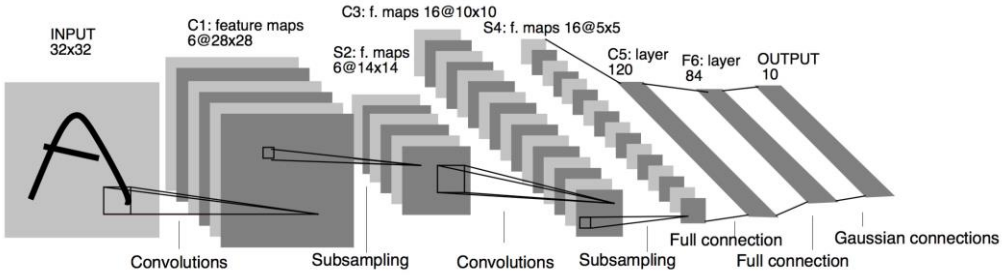

Figura 1.3: Struttura della rete LeNet5 (6), una delle prime reti convoluzionali per l'image recognition che hanno cominciato a far accrescere l'interesse verso questo nuovo tipo di architettura.

Prima dell'avvento delle CNN, architetture classiche di reti neurali sono state utilizzate nel campo della computer vision. I risultati ottenuti hanno messo in mostra uno dei problemi principali che affligge queste architetture nel task della visione artificiale, ovvero la ridotta scalabilità. Quando si lavora con immagini a risoluzioni elevate, infatti, lo schema di collegamento classico tra neuroni, che prevede che ogni neurone di un livello sia connesso con tutti i neuroni del livello successivo, porta all'esplosione del numero di variabili contenute nella rete. Ad esempio, nel caso di un dataset come CIFAR-10 (7), in cui le immagini hanno dimensioni relativamente piccole (32x32x3, larghezza, altezza e numero di canali), un singolo neurone del livello di input della rete avrebbe $32 * 32 * 3 = 3072$ pesi. Se le immagini avessero dimensioni di 200x200, questo numero crescerebbe fino a 120000.

Questa situazione rappresenta un limite architetturale delle reti neurali classiche, che nel caso della predizione su immagini non tengono conto della struttura spaziale dei dati. Infatti, ogni pixel, che sia vicino o lontano rispetto ad un altro, viene trattato allo stesso modo, e questo porta ad uno spreco di risorse nel task della visione artificiale, che inoltre accentua il problema dell'overfitting, causato dell'elevato numero di parametri generati.

Ispirandosi, come già detto, alla struttura e al funzionamento della corteccia visiva, l'intento delle CNN è quello di sfruttare la forte correlazione spaziale presente tipicamente nelle immagini. Le caratteristiche principali di questa tipologia di reti possono essere individuate da tre punti chiave:

- **Volumi tridimensionali di neuroni**: nei livelli di una CNN i neuroni sono disposti in strutture tridimensionali con una certa larghezza, altezza e profondità. Ognuna di queste strutture è connessa solamente ad un sottoinsieme di strutture del livello precedente, che prende il nome di campo recettivo, come già accennato;
- **Connessioni locali**: grazie all'utilizzo di connessioni di tipo parziale, o locale, le CNN sfruttano, come già detto, la correlazione spaziale locale presente nei dati di input. Ad ogni livello della rete vengono appresi specifici filtri di convoluzione (che rappresentano i pesi dei neuroni) che massimizzano la risposta ad un determinato pattern locale di input. Impilando molti livelli di questo tipo, grazie al paradigma di connessione locale i neuroni degli ultimi livelli della rete avranno un receptive field più ampio rispetto a quelli dei livelli iniziali. Questo porta all'apprendimento di filtri non lineari man mano più globali, ovvero che rispondono a stimoli su porzioni più ampie del campo visivo. Il risultato è la creazione preliminare di un insieme di rappresentazioni di piccole parti dell'input (dette feature map) che vengono successivamente assemblate a creare una rappresentazione complessiva di aree più grandi. Un esempio è illustrato nella figura 1.2 proposta nei paragrafi precedenti;
- **Pesi condivisi**: i filtri appresi in un certo livello sono replicati sull'intero campo visivo. Queste unità replicate presentano gli stessi parametri (pesi e bias) e vanno a formare una feature map. Ciò significa che tutti i neuroni in un certo livello della rete imparano a riconoscere la stessa feature nei dati e grazie alla replicazione queste feature vengono individuate indipendentemente dalla loro posizione nel campo visivo, rendendo così le CNN invarianti alla traslazione.

Come già accennato, nelle CNN vengono tipicamente utilizzati diversi tipi di livelli. Tra i principali troviamo:

- **Livelli di convoluzione**: rappresentano il livello fondamentale delle CNN. I parametri consistono in un insieme di filtri apprendibili (**kernel**) che presentano un campo recettivo limitato, ma si estendono attraverso tutta la profondità del volume di input. Durante un passo di predizione, ogni filtro viene convoluto con l'input lungo la larghezza e l'altezza del volume, calcolando così il prodotto scalare tra il filtro e l'input e producendo una mappa di attivazione a due dimensioni relativa a quel filtro specifico. Impilando le mappe di attivazione lungo la profondità si produce l'output completo di un livello di convoluzione. Le dimensioni del volume di output di un certo livello dipendono da tre iperparametri, la **profondità**, il **passo** (stride) e la presenza o assenza di **padding** degli input. La profondità controlla il numero di neuroni nel livello che sono connessi alla stessa regione del volume di input. Ogni strato di neuroni imparerà a riconoscere un certo pattern in questa regione. Il passo controlla il numero e la disposizione delle colonne di profondità nel volume di output. Con un passo piccolo, ad esempio di valore unitario, per ogni posizione spaziale nel volume di input viene allocata una colonna di profondità nel volume di output, mentre con valori di passo più alti verranno allocate nuove colonne solo in corrispondenza di alcune posizioni spaziali nel volume di input. Nel primo caso, come risultato si ottengono campi recettivi con sovrapposizione elevata e conseguentemente volumi di output di dimensioni elevate. In maniera speculare, nel secondo caso le sovrapposizioni trai campi recettivi saranno limitate e con esse le dimensioni del volume di output. Infine, il padding è un parametro che influenza le dimensioni spaziali del volume di output e agisce aggiungendo all'input un bordo di una certa dimensione contenente valori nulli. È particolarmente utile quando si vuole fare in modo che le dimensioni spaziali dell'output e dell'input coincidano;
- **Livelli di sottocampionamento** (pooling): rappresenta uno dei tipi di livello che caratterizzano le CNN e costituisce una forma di

sottocampionamento non lineare. Esistono diverse funzioni non lineari in grado di implementare il pooling, ma la più comune è quella del max pooling. Questa tecnica partiziona l'immagine di input in un insieme di quadrati, e per ciascuna delle regioni risultanti restituisce come output il valore massimo. Il suo scopo è quello di ridurre progressivamente le dimensioni delle rappresentazioni, in modo da ridurre il numero di parametri e la complessità computazionale della rete, contrastando allo stesso tempo il verificarsi dell'overfitting. Si basa sul concetto che, una volta individuata una certa feature, la sua precisa posizione nell'input non è così importante quanto la sua posizione approssimativa nei confronti delle altre feature. Nell'architettura tipica di una CNN vengono alternati ripetutamente livelli di convoluzione e livelli di pooling;

- **Livelli di rettificazione lineare** (ReLU (8), Rectified Linear Unit): si tratta di livelli che svolgono il ruolo della funzione di attivazione dei neuroni nelle reti neurali. Un livello ReLU è composto da neuroni che applicano la funzione $f(x) = \max(0, x)$. Questi livelli incrementano la non linearità della rete ed allo stesso tempo non modificano i campi recettivi dei livelli di convoluzione. La funzione dei ReLU viene preferita ad altre, come la tangente iperbolica o la sigmoide, poiché rispetto a queste porta ad un processo di addestramento molto più rapido, senza incidere in modo significativo sull'accuratezza di generalizzazione;

- **Livelli completamente connessi** (fully connected): rappresentano la parte finale di una CNN, tipicamente nel caso dei task di classificazione. I neuroni di un livello fully connected sono collegati a tutti i neuroni del livello precedente, come accade nelle reti neurali classiche, e hanno lo scopo di compiere i "ragionamenti" di alto livello che portano infine all'output vero e proprio della rete, ovvero la predizione;

- **Livelli di sovracampionalento** (upsampling): come i livelli fully connected, si trovano nella parte finale della rete, ma sono utilizzati

tipicamente nei task di regressione, come la ricostruzione di immagini, la stima della profondità e la segmentazione. Contengono tipicamente dei livelli cosiddetti di **deconvoluzione**, o più propriamente convoluzione trasposta. Questi livelli agiscono in maniera inversa rispetto alla convoluzione ed al pooling ed hanno come risultato la crescita delle dimensioni spaziali degli input. Ciò permette di ottenere in uscita dalla rete delle immagini di dimensioni comparabili con quelle delle immagini di input. Nel caso più semplice questi livelli possono essere implementati come sovracampionamenti statici con interpolazione bilineare, oppure possono presentare dei filtri apprendibili come quelli presenti nel resto della rete;

- **Livelli di loss**: rappresentano tipicamente l'ultimo livello di una CNN. Si occupano del calcolo dell'errore di predizione attraverso la valutazione di una determinata loss function. In base al tipo di task su cui lavora la rete, la funzione utilizzata può essere di vario tipo. Per la classificazione, ad esempio, solitamente viene utilizzata la cross-entropy per la predizione di un valore di probabilità per ogni classe. Nella regressione, invece, è utilizzata una loss di tipo euclideo per la predizione di valori reali.

Tipicamente, le CNN utilizzano un numero più elevato di iperparametri rispetto alle reti neurali classiche. Tra quelli che le differenziano rispetto a queste ultime troviamo:

- Il **numero di filtri**: dal momento che le dimensioni spaziali delle feature map decrescono andando in profondità nella rete, i livelli vicini al livello di input tenderanno ad avere un numero ridotto di filtri mentre livelli vicini all'output avranno più filtri. Per cercare di equalizzare il numero di filtri lungo tutta la rete solitamente si cerca di tenere costante tra tutti i livelli il prodotto tra il numero di feature map e il numero di posizioni spaziali che vengono prese in considerazione nell'input. Facendo in questo modo si va a preservare lungo tutta la rete l'informazione derivante dall'input;

- La **forma dei filtri**: questo parametro solitamente varia da rete a rete e viene scelto in base alle caratteristiche del dataset utilizzato. L'obiettivo è trovare il giusto compromesso tra granularità e dettaglio in modo da creare astrazioni della giusta scala per un particolare dataset;
- La **forma dei filtri utilizzati nel max pooling**: come nel caso precedente, si tratta di un parametro che dipende dallo specifico dataset utilizzato. Immagini con risoluzione elevata potrebbero necessitare di filtri di grandi dimensioni per ridurre in maniera appropriata le dimensioni degli input, mentre per immagini a bassa risoluzione rettangoli troppo grandi potrebbero portare a rappresentazioni troppo piccole negli stadi più avanzati della rete, con conseguente perdita di informazione. In genere, vengono utilizzati rettangoli con dimensione 2x2.

Come per le reti neurali classiche, anche per le CNN è possibile utilizzare le classiche tecniche di regolarizzazione per contrastare l'overfitting. Inoltre è possibile fare uso della cosiddetta tecnica del "**data augmentation**". Questa tecnica consiste nell'apportare piccole modifiche casuali negli input, come rotazioni, traslazioni, ritagli e altre operazioni di image processing, con l'intento di aumentare la quantità effettiva di esempi e conseguentemente contrastare l'overfitting.

Capitolo 2
La produzione del dataset

Come illustrato nel capitolo 1, l'insieme dei dati di esempio, ovvero il dataset, rappresenta un requisito obbligatorio per l'addestramento delle reti neurali. Nel campo specifico delle reti neurali profonde, e quindi delle CNN, il dataset richiesto presenta inoltre requisiti particolarmente stringenti per quanto riguarda il numero di esempi necessari.

Nel caso del task di cui si occupa questa tesi, inoltre, trattandosi di un problema di apprendimento supervisionato ogni esempio contenuto nel dataset dovrà essere corredato dal relativo output atteso (detto **label**). Per la stima della profondità queste label consistono nella profondità reale nella scena contenuta in ciascun esempio Nella pratica, le label sono rappresentate mediante un'immagine (detta **depth map**), o più generalmente una matrice, delle stesse dimensioni dell'immagine di esempio ed in cui per ogni pixel è indicata la distanza del relativo punto della scena rispetto al sensore del dispositivo di acquisizione.

La soluzione più diretta per produrre queste mappe di profondità sarebbe l'utilizzo di un dispositivo di acquisizione 3d. Tra i principali troviamo le telecamere stereo, i sistemi ad infrarossi (come ad esempio la prima versione del kinect (1) di Microsoft), i laser scanner 3d ed i time-of-flight sensor (come la seconda versione del kinect). Questi dispositivi presentano però alcune problematiche, specialmente per quanto riguarda quelli con costi abbordabili. Le telecamere stereo, ad esempio, producono depth map molto rumorose, mentre i kinect, soprattutto nella prima versione, forniscono una risoluzione bassa ed inoltre, sia in versione ad infrarossi che time of light, funzionano solo fino a circa cinque metri di distanza. Una rete neurale addestrata con l'utilizzo di dati contenenti imprecisioni finirebbe per apprendere anche queste, e ciò rappresenta uno scenario non ideale. Risultati più precisi possono essere ottenuti con i laser scanner, i quali però hanno tempi di funzionamento più lunghi e presentano costi molto levati.

Per questi motivi è stato deciso di intraprendere la strada dell'addestramento mediante **dati sintetici** generati attraverso tecniche di computer graphics, ispirandosi anche al lavoro svolto nel 2016 presso l'università di Friburgo, in cui per la prima volta è stato proposto l'addestramento di CNN mediante l'utilizzo di dataset generati sinteticamente (9) (10).

Per lo sviluppo del dataset virtuale è stata necessaria la scelta di un software per la grafica computerizzata. Dopo una ricerca sono state individuate due alternative valide: il software gratuito ed open source **Blender** (11) ed il motore grafico gratuito per videogame **Unity 3D** (12). Dopo un'attenta analisi, la scelta è ricaduta sul software Blender, poiché permette di raggiungere livelli di fotorealisticità superiori rispetto ad Unity 3D, che dalla sua conta prestazioni in realtime, aspetto trascurabile per i nostri scopi.

Nel prossimo paragrafo verrà fornita una panoramica sul software Blender, e successivamente verrà illustrato nel dettaglio il lavoro svolto per la creazione del dataset.

2.1 Blender

Blender è un software professionale gratuito ed open source per la grafica 3D. È disponibile su diversi sistemi operativi, tra cui Microsoft Windows, MacOS e Linux. Viene utilizzato per svariate applicazioni, come film di animazione, effetti speciali, arte, creazione di modelli 3D per la stampa, applicazioni 3D interattive e videogames.
Tra le funzionalità principali che sono state sfruttate per la creazione del dataset troviamo la **modellazione 3D**, l'applicazione di **texture**, la creazione di **materiali** virtuali, la gestione del **movimento della telecamera**, la gestione delle **luci**, il **rendering**, la **composizione** della scena e lo **scripting** in linguaggio python.

Blender gestisce un file system interno che permette di impacchettare diverse scene in un file singolo con estensione **.blend**. Questi file hanno la caratteristica di essere compatibili sia con versioni diverse di blender che con sistemi operativi diversi. Nel file blend è possibile salvare scene, oggetti, texture, immagini e molto altro senza la necessità di trasportare file aggiuntivi.

2.1.1 L’interfaccia grafica

L’interfaccia grafica di Blender è progettata sui seguenti concetti:

- Le **modalitá di editing**: sono disponibili diverse modalità di modifica degli oggetti tridimensionali, tra cui le principali sono la “**Object Mode**” e la “**Edit Mode**”. La Object Mode viene utilizzata per manipolare un particolare oggetto come se fosse un’unità atomica, mentre la Edit Mode permette di manipolare direttamente il modello tridimensionale (**mesh**) dell’oggetto. Ad esempio, la Object Mode può essere utilizzata per muovere, ruotare o scalare l’intero oggetto, mentre la Edit Mode permette di andare ad agire sui singoli vertici che compongono la sua mesh;
- L’utilizzo di **scorciatoie da tastiera**: la maggior parte delle funzionalità del software sono accessibili tramite scorciatoie da tastiera (**hotkey**), caratteristica che rende il lavoro molto veloce;
- L’**input numerico**: i campi di immissione di valori numerici sono caratterizzati dalla possibilità di essere modificati direttamente cliccando e trascinando con il mouse. Mediante l’utilizzo contemporaneo di tasti come Ctrl e Shift è possibile inoltre andare ad agire sugli step di modifica. Un’altra funzionalità molto potente è quella di poter immettere espressioni in linguaggio python, e quindi di poter indicare valori numerici risultanti da espressioni matematiche;
- La **gestione dell’area di lavoro**: l’interfaccia di Blender permette di modellare a piacimento la configurazione delle varie finestre. L’interfaccia principale è suddivisa in sezioni, la disposizione e le dimensioni delle quali sono personalizzabili dall’utente. Le disposizioni create possono inoltre essere salvate sotto forma di “schermate” (dette **screen**) ed è possibile passare da una schermata all’altra agendo su un menu di scelta. In questo modo è possibile definire aree di lavoro specializzate per task differenti, come ad esempio la modellazione, lo scripting o l’applicazione di texture.

Nella figura 2.1 è mostrato un esempio di schermata base.

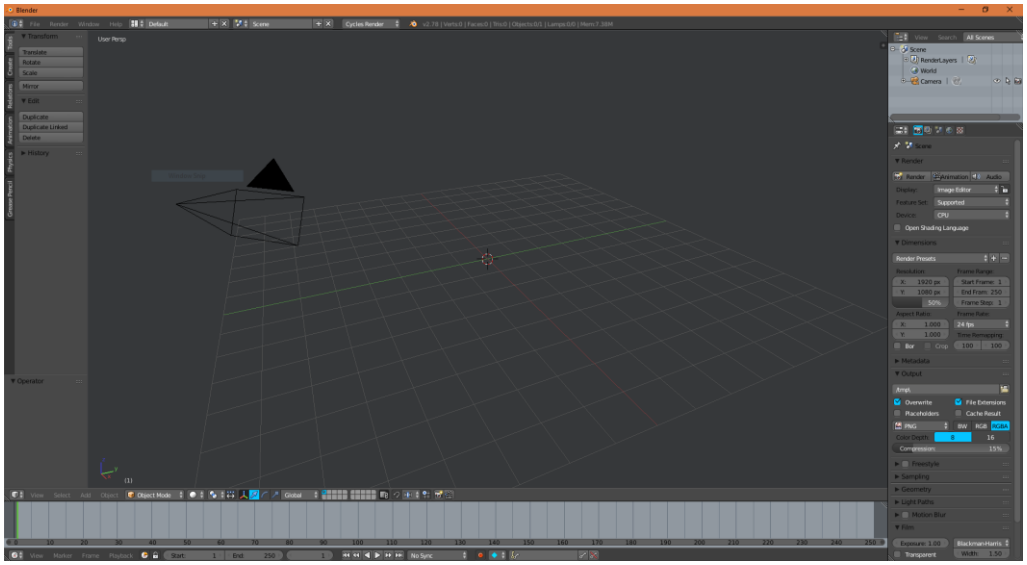

Figura 2.1: Interfaccia base di Blender per la modellazione 3D. Al centro si trova il viewport, a destra una toolbar in cui è possibile modificare varie impostazioni.

2.1.2 Il motore di rendering

In Blender sono presenti due motori di rendering: il **Blender Render** ed il **Cycles Render**.

Il Blender Render rappresenta il motore impostato di default. Utilizza un algoritmo di rendering di tipo **scanline**. In questa tecnica i poligoni presenti nella scena vengono prima ordinati in base alla loro distanza dalla telecamera, e successivamente viene scansionato riga per riga il campo visivo in modo da disegnare in ogni pixel il relativo punto del poligono che si trova in primo piano in quella determinata posizione. Questo motore di rendering è caratterizzato soprattutto dalla sua velocità di esecuzione, che però va a discapito della fotorealisticità delle immagini ottenibili.

Per quanto riguarda il Cycles Render, che è stato introdotto solo recentemente nel software, si tratta di un algoritmo di rendering di tipo **path-tracing** progettato per essere interattivo e facile da usare. Il path-tracing si basa sull'integrazione sull'insieme dei raggi di luce che, direttamente o indirettamente, raggiungono un certo punto della scena. L'illuminazione risultante viene quindi ridotta da una certa funzione di riflettività in modo da determinare l'effettiva luminosità del punto. Questo procedimento viene

ripetuto per tutti i pixel dell'immagine finale, e restituisce immagini in cui l'illuminazione globale è rispondente alla realtà. Quando accoppiata con modelli fisicamente accurati dei materiali delle superfici, modelli accurati delle fonti di luce e modelli di telecamere con proprietà ottiche realistiche, questa tecnica può restituire immagini molto simili ad una fotografia.
Nell'algoritmo di path-tracing implementato in blender i raggi di luce vengono tracciati dalla telecamera ai punti della scena. Questi raggi rimbalzano tra i vari punti finché non impattano in una fonte di luce. Viene analizzata sia la luce diretta che quella indiretta, ovvero riflessa da un oggetto e che illumina parzialmente la scena.
Attraverso un parametro, il numero di **samples**, è possibile indicare quanti cicli di path-tracing devono essere effettuati per ogni pixel dell'immagine di output. Valori bassi di samples portano ad un'immagine che presenta notevole rumore, ma comporta tempi di rendering bassi. Specularmente, valori di samples alti portano a immagini di qualità superiore, a discapito della velocità. Parte del processo di rendering consiste anche nel determinare questo valore, in modo da trovare un giusto compromesso tra tempistiche e qualità.

Per le sue potenzialità in termini di fotorealisticità delle immagini prodotte, per la produzione del dataset è stato deciso di utilizzare proprio il Cycles Render. Altro punto a favore è la possibilità di rendering su GPU, caratteristica che riduce non poco i tempi di rendering e quindi permette la creazione di un numero più elevato di esempi di training.

Un esempio della qualità che è possibile raggiungere con questo motore di rendering utilizzando valori di samples relativamente alti è mostrato in figura 2.2. Si tratta di un rendering di test ad alta risoluzione prodotto in fase di creazione del dataset.

Figura 2.2: render di prova in alta risoluzione e ad alto numero di samples realizzato utilizzando la scena creata per la produzione del dataset.

Nei prossimi paragrafi verrà illustrato nel dettaglio il lavoro che ha portato alla creazione del dataset utilizzato per il processo di training, che sarà illustrato a sua volta nel capitolo 4.

2.2 Il dataset

Come già accennato nell'introduzione, il dataset è stato creato nell'ambito di un progetto più ampio ed in collaborazione col lavoro di un'altra tesi. In questo paragrafo verrà illustrato il processo nella sua totalità, approfondendo maggiormente le implementazioni prodotte esclusivamente nel lavoro di questa tesi.

La produzione del dataset sintetico può essere suddivisa in cinque fasi:

- La **creazione dei prodotti**;
- La **creazione dello scaffale**;
- Il **posizionamento dei prodotti**;

- La **creazione della scena**;
- Il **rendering**.

Nelle sezioni seguenti verrà illustrata nello specifico ciascuna delle fasi.

2.2.1 La creazione dei prodotti

Questa fase ha portato alla realizzazione di numerosi modelli tridimensionali di altrettanti prodotti venduti comunemente nei supermercati. Per la selezione è stato utilizzato un database commerciale contenente diverse centinaia di prodotti. Per ogni prodotto, nel database è presente un file xml che contiene tutte le informazioni che lo caratterizzano.

Nello specifico, le informazioni che sono state utilizzate per la creazione dei modelli riguardano le dimensioni del prodotto e le immagini che lo rappresentano dai 6 punti di vista principali (fronte, retro, sopra, sotto, sinistra, destra).

Per l'estrapolazione delle informazioni dai file xml è stato implementato un piccolo software scritto in C# che esegue il parsing del file e recupera le dimensioni e i link da cui scaricare le immagini. Lo stesso software in seguito salva le dimensioni in un file di testo e scarica e salva su disco le immagini. Inoltre, prima di salvarle, rimpicciolisce le immagini per ridurre l'occupazione di memoria su disco (le dimensioni di base sono pari a 2000x2000, livello di dettaglio non necessario per i nostri fini) e le ritaglia in modo che nell'immagine rimanga solo l'effettiva faccia del prodotto. Oltre a dimensioni e immagini, dal file xml legge e salva anche le informazioni riguardanti il nome del prodotto ed il marchio, informazioni necessarie per il lavoro di tesi con cui c'è stata la collaborazione.

L'obiettivo posto fin dall'inizio per la fase di creazione dei modelli dei prodotti è stato quello dell'automazione del processo. La maggior parte dei prodotti contenuti nel database rappresenta scatolati di forma rettangolare, sacchetti o buste, tutti formati che possono essere approssimati in maniera buona con dei parallelepipedi.

Data la semplicità del modello risultante, per la creazione di queste tipologie di prodotti è stata dunque sfruttata la funzionalità già accennata di blender che permette di implementare script in linguaggio python in cui è possibile effettuare, in modo automatico, la maggior parte delle operazioni che normalmente si effettuerebbero in maniera manuale dall'interfaccia.
Lo script implementato per prima cosa crea le 6 facce del parallelepipedo virtuale, quindi applica ad ognuna di esse un materiale che simula il cartoncino tipico utilizzato per gli scatolati, ed in seguito applica una texture utilizzando le immagini ottenute dal database. A questo punto assembla assieme le 6 facce e salva il modello risultante in un file blend che contiene tutto il necessario. Questo processo è eseguito ciclicamente per un insieme di prodotti selezionati.

In figura 2.3 sono mostrati due esempi di prodotti generati.

Figura 2.3: due esempi di modelli tridimensionali di prodotti.

Per quanto riguarda il resto dei prodotti, la maggior parte si tratta di bottiglie. In questo caso l'approssimazione come parallelepipedo avrebbe portato a risultati non buoni. L'automazione stessa della modellazione delle bottiglie è risultata infattibile a causa della varietà della forma di queste, non approssimabile neanche da un semplice cilindro, ad esempio. L'unica soluzione affrontabile era quella di modellare manualmente questi prodotti. Chiaramente, date le dimensioni del database, non era pensabile operare su tutti i prodotti, ma un piccolo insieme sarebbe stato accettabile. Come test, sono stati dunque modellati manualmente due prodotti, una bottiglia in plastica di forma abbastanza standard ed una in vetro con forma più particolare. I prodotti in questione sono visionabili in figura 2.4.

Figura 2.4: I due modelli di bottiglia creati, posizionati in una scena di prova. È possibile notare, soprattutto nella bottiglia in vetro, il leggero disturbo (granulosità) di cui si è parlato nel paragrafo sul motore di rendering Cycles.

I test di rendering effettuati hanno evidenziato delle problematiche anche per quanto riguarda i tempi. I modelli delle bottiglie, infatti, contengono molti più vertici rispetto a dei meri parallelepipedi, e questo, soprattutto dopo aver posizionato nella scena numerose copie dello stesso prodotto, ha portato ad un rallentamento marcato del processo di rendering sull'hardware disponibile, per non parlare del tempo necessario alla modellazione vera e propria di questi prodotti. È stato deciso dunque, per gli scopi della tesi, di lavorare solamente con prodotti di forma semplice modellati in automatico tramite lo script illustrato precedentemente.

2.2.2 La creazione dello scaffale

Il passo successivo alla creazione dei prodotti è consistito nella creazione di uno scaffale sul quale posizionarli. Per ridurre i tempi, è stata fatta una ricerca in rete per trovare un modello 3d gratuito. È stato individuato e scaricato un modello rispondente alle specifiche, ma che purtroppo era fornito nel formato .max, non supportato da Blender. Ci si è forniti dunque della versione di prova gratuita del software 3D Studio Max (13) per la conversione in formato .obj,

che è possibile importare in Blender senza problemi. Una volta importato in Blender, lo scaffale è stato lievemente modificato in modo da renderlo più realistico. Nello specifico, sono stati modificati alcuni materiali, aggiunte le texture e aggiunta ad ogni ripiano la barra dove solitamente vengono inseriti i prezzi dei prodotti. Il risultato finale è mostrato in figura 2.5.

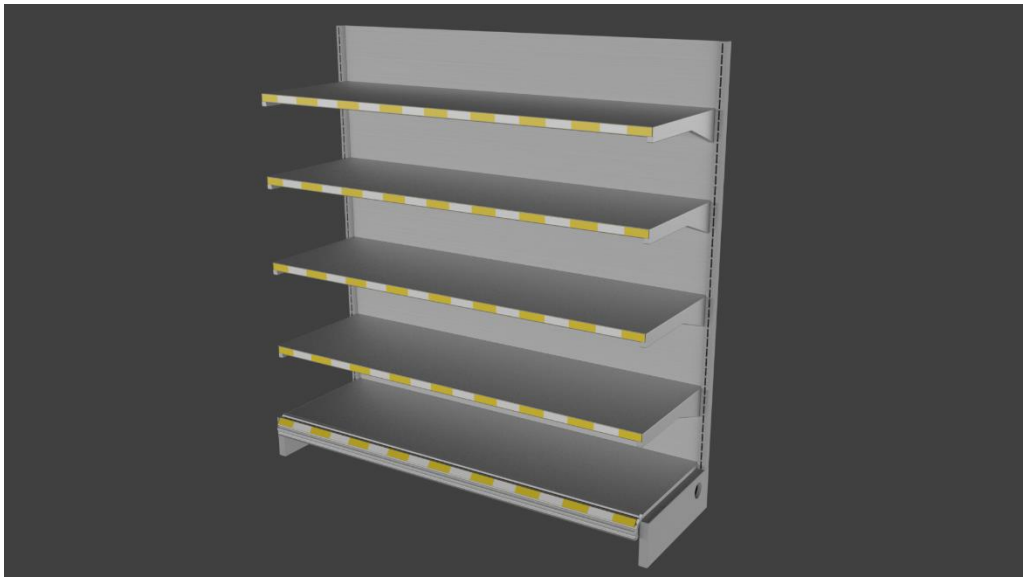

Figura 2.5: modello definitivo dello scaffale.

2.2.3 Il posizionamento dei prodotti

Con lo scaffale pronto ad essere popolato con i prodotti, il passo successivo è stato appunto la definizione di una procedura di posizionamento di questi. Sempre sfruttando le potenzialità di scripting di Blender, è stato implementato uno script python che seleziona casualmente i prodotti tra quelli contenuti in una delle categorie disponibili e li posiziona sui ripiani dello scaffale con un certo criterio. Nello specifico, ogni ripiano è stato suddiviso in 3 slot, ognuno dei quali contiene un solo tipo di prodotto. La quantità di istanze di ogni prodotto contenute in uno slot viene calcolata in base alle dimensioni del prodotto ed in base ad una certa distanza minima tra un prodotto ed i suoi vicini, impostata a priori. Il posizionamento di ogni singola istanza è inoltre influenzato in modo casuale al fine di simulare prodotti lievemente spostati e ruotati rispetto alla posizione di partenza. È stata modellata anche la possibilità di avere, in maniera casuale, file vuote o solo parzialmente riempite, così da

simulare la situazione tipica in un vero supermercato. Un esempio del risultato finale raggiunto è mostrato in figura 2.6.

Figura 2.6: esempio di posizionamento dei prodotti sullo scaffale.

2.2.4 La creazione della scena

A questo punto è stato necessario definire una scena in cui inserire gli scaffali popolati. La prima soluzione adottata è stata quella di rappresentare un intero supermercato. A tal fine, come per lo scaffale, per ridurre i tempi è stato scaricato un modello 3d gratuito molto accurato, al quale sono state apportate alcune modifiche quali la definizione dei materiali, l'applicazione delle texture, la modellazione di porte e finestre, comprese di vetrate, l'aggiunta di una struttura di luci che simula dei neon e la creazione di un ambiente esterno in cui posizionare l'edificio.

È stata dunque inserita qualche decina di scaffali di prova per testare il rendering. Il risultato ottenuto è visibile in figura 2.7.

Figura 2.7: rendering di test ad alta definizione della scena del supermercato.

Dai test di rendering effettuati è subito saltato fuori un problema: la complessità della scena. Come accennato anche nel paragrafo sulla creazione dei prodotti, il rendering di oggetti complessi comporta tempi considerevoli, e questo avrebbe rappresentato un problema rilevante dal momento che l'obiettivo finale è quello di produrre diverse migliaia di immagini. Inoltre, le immagini necessarie per il training consistono in inquadrature frontali degli scaffali, a distanze diverse, e quindi la presenza dell'intero supermercato risulta relativamente significativa.

Per questi motivi alla fine è stato deciso di adottare un compromesso optando per un altro approccio, ovvero una scena contenente un solo scaffale, una stanza aperta strutturata con tre muri ed il pavimento, ed un sistema di illuminazione costituito da quattro luci a cono posizionate in alto in corrispondenza degli angoli della stanza. La scena risultante è mostrata in figura 2.8.

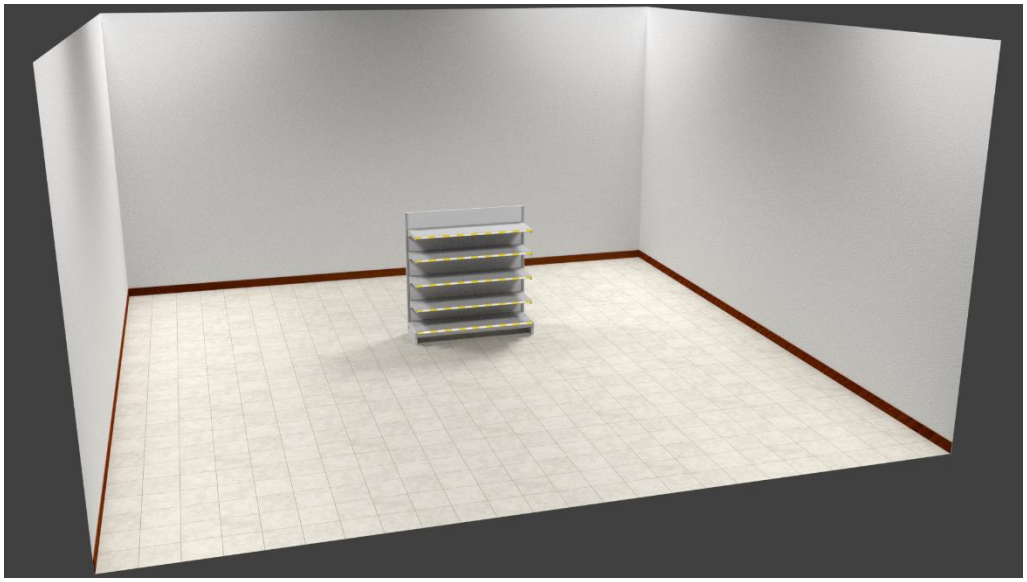

Figura 2.8: la scena finale utilizzata per il rendering.

L'utilizzo di questa scena, inoltre, semplifica molto anche la disposizione dei prodotti. Utilizzando il supermercato completo, infatti, sarebbe stato necessario progettare un modo per riempire il supermercato con scaffali popolati. Una soluzione sarebbe stata adattare lo script che popola il singolo scaffale in modo da funzionare su tutti gli scaffali del supermercato, un'altra generare un certo numero di scaffali già popolati casualmente con lo script esistente, per poi inserirli nel supermercato in un secondo momento. La prima soluzione presenta il problema di dover implementare una versione più complessa dello script che dispone i prodotti, la seconda invece lascia lo script intatto ma, per ottenere un livello di casualità accettabile nelle immagini finali, necessita della creazione di molte scene contenenti disposizioni diverse, e per rendere questo processo automatico sarebbe necessario comunque implementare uno script ad-hoc. In ogni caso, la gestione della generazione di configurazioni casuali sarebbe stata problematica.

L'utilizzo della scena con il singolo scaffale non necessita di script aggiuntivi e, come verrà illustrato nel prossimo paragrafo, si integra perfettamente con il processo di rendering finale, ed alla fine permette di ottenere immagini sostanzialmente equivalenti a quelle ottenibili nella scena del supermercato completo.

2.2.5 Il rendering

In questa fase tutto il lavoro precedente è stato messo insieme per implementare una procedura automatica per il rendering dell'intero dataset.

Oltre alle immagini ed alle mappe di profondità, nel dataset completo sono state incluse anche le informazioni riguardanti le bounding box dei loghi di ogni prodotto relative ad ogni singola immagine, dati necessari al lavoro dell'altra tesi con cui si è collaborato. A tal fine è stato utilizzato uno script, implementato dall'altro tesista, che a partire dalle coordinate degli oggetti rispetto al sistema di riferimento della telecamera genera le coordinate di ciascuna bounding box nell'immagine relativa e le salva in un file di testo.

Il primo step effettuato consiste nella definizione di un percorso che la telecamera deve seguire nella scena, in modo da produrre inquadrature frontali a due distanze. L'implementazione è stata effettuata manualmente con l'utilizzo degli strumenti dell'interfaccia di Blender. Il percorso risultante consiste in due passaggi completi sull'intero scaffale, in cui nel primo vengono inquadrati due ripiani e nel secondo uno solo. Il percorso completo è stato suddiviso in 100 frame, di cui 40 per le inquadrature a due ripiani e 60 per quelle ad uno. Come requisito dell'altra tesi, la telecamera è stata inoltre impostata per l'acquisizione di immagini stereoscopiche. Il numero totale di immagini singole prodotte per ogni percorso completo è pari dunque a 200.

Messo a punto il percorso della telecamera, lo script che dispone i prodotti sullo scaffale, descritto nei paragrafi precedenti, e quello che genera le bounding box sono stati combinati nell'implementazione di un terzo script che, ciclicamente, genera una disposizione di prodotti ed effettua un certo numero di render completi su di essa.

Per velocizzare la generazione delle bounding box, allo script che popola lo scaffale è stata inoltre aggiunta la creazione di alcuni oggetti "segnaposto" che rappresentano la faccia frontale dei prodotti posti in primo piano. Senza di questi segnaposto, infatti, per generare le bounding box sarebbe stato necessario analizzare tutti i prodotti della scena successivamente al posizionamento, in modo da individuare quelli in primo piano (i quali sono gli unici ad avere il logo visibile), mentre la creazione dei segnaposto non

necessita di passaggi aggiuntivi, dal momento che può essere effettuata nel momento stesso in cui un oggetto viene posizionato in primo piano.

Come già detto, questo nuovo script genera ciclicamente una disposizione casuale di prodotti ed effettua alcuni passaggi di rendering su questa stessa disposizione. Nello specifico, il primo passaggio viene effettuato con la telecamera perfettamente allineata frontalmente, mentre negli altri vengono introdotte da codice delle rotazioni casuali della telecamera lungo l'asse verticale e quello orizzontale parallelo allo scaffale, in modo da ottenere una maggiore varietà tra le immagini.

Il database di prodotti che è stato utilizzato presenta un'organizzazione in categorie. È stato deciso dunque, nel momento di generare una singola disposizione, di selezionare un certo numero (nello specifico 15, ovvero 3 slot su 5 ripiani) di prodotti casuali da una singola categoria. Questa decisione non limita la varietà poiché alla fine, durante il training, le immagini verranno utilizzate singolarmente ed in modo casuale, senza un riferimento alla sequenza dalla quale provengono.

Per ogni categoria di prodotti vengono generate diverse disposizioni. Il numero di categorie disponibili, insieme al numero di disposizioni per ogni categoria ed al numero di passaggi di rendering effettuati per ogni disposizione definisce il numero totale di immagini prodotte. Tutti questi valori sono stati dunque definiti come parametri da impostare nello script secondo la particolare necessità.

Ultimato lo script per il rendering automatico, si è passati alla calibrazione dei parametri di rendering interni a Blender. Come illustrato nel paragrafo 2.1, esistono infatti diversi parametri modificabili che influenzano la qualità delle immagini renderizzate e le tempistiche, tra cui i principali sono il numero di samples per il Cycles Render e la risoluzione delle immagini.

Per quest'ultima è stato scelto il valore 640x480, il più piccolo possibile compatibilmente alle necessità del training. L'utilizzo di una risoluzione maggiore avrebbe infatti allungato inutilmente i tempi di rendering, mentre un valore più basso avrebbe reso più difficoltoso il riconoscimento dei dettagli più piccoli.

Per la scelta del numero di samples sono stati effettuati vari test su singola immagine, in modo da trovare il giusto compromesso tra qualità e tempistiche.

Oltre alla risoluzione e ai samples, sono stati valutati altri parametri che influenzano, in maniera più leggera, sia la qualità che i tempi. Tra i principali troviamo il numero massimo di rimbalzi dei raggi di luce che l'algoritmo di path-tracing analizza, parametro che influenza la velocità del rendering e la qualità dei riflessi ed in generale dell'illuminazione nella scena, e le dimensioni delle sezioni dell'immagine che vengono analizzate in parallelo dall'algoritmo di rendering, parametro che dipende dalle caratteristiche dell'hardware utilizzato e che può influire in modo rilevante sui tempi.

Grazie al tuning di tutti questi parametri è stato possibile arrivare ad una durata di rendering su singolo frame stereoscopico pari a circa 30 secondi. Questo valore si riferisce al rendering su GPU utilizzando la scheda nVidia Titan X, disponibile in laboratorio.

L'ultimo passo effettuato prima di poter avviare il processo di rendering è stata l'impostazione della composizione dell'output in Blender. Si tratta di un sistema che permette di definire, mediante uno schema a blocchi, le componenti che dovranno essere date in output per il rendering di ciascun fotogramma, e permette anche di definire delle operazioni di post processing su di esse.
È in questa fase che è stata definita la creazione delle mappe di profondità. Queste mappe sono disponibili direttamente dopo ogni singolo rendering nel cosiddetto "**z-buffer**', che consiste in una matrice delle stesse dimensioni dell'immagine e che contiene per ogni pixel la distanza dalla telecamera nell'unità di misura impostata in Blender (nel caso specifico, in millimetri). Attraverso l'interfaccia a blocchi è stato dunque reperito il contenuto dello z-buffer ed è stato impostato il salvataggio in due formati. Il primo consiste in un'immagine in formato OpenEXR (14), ovvero un particolare tipo di immagine che permette di registrare valori float a 32 bit al posto dei classici interi a 8 bit senza segno. Il file exr risultante è quello che verrà utilizzato per il training vero e proprio, come verrà illustrato nel capitolo 4. Il secondo formato

è una versione normalizzata nel range 100-5000mm salvata in formato png ed utilizzata come semplice verifica visiva del rendering.

Nello stesso schema è stato infine creato un blocco per il salvataggio in formato png delle immagini a colori. In figura 2.9 è mostrato lo schema definito.

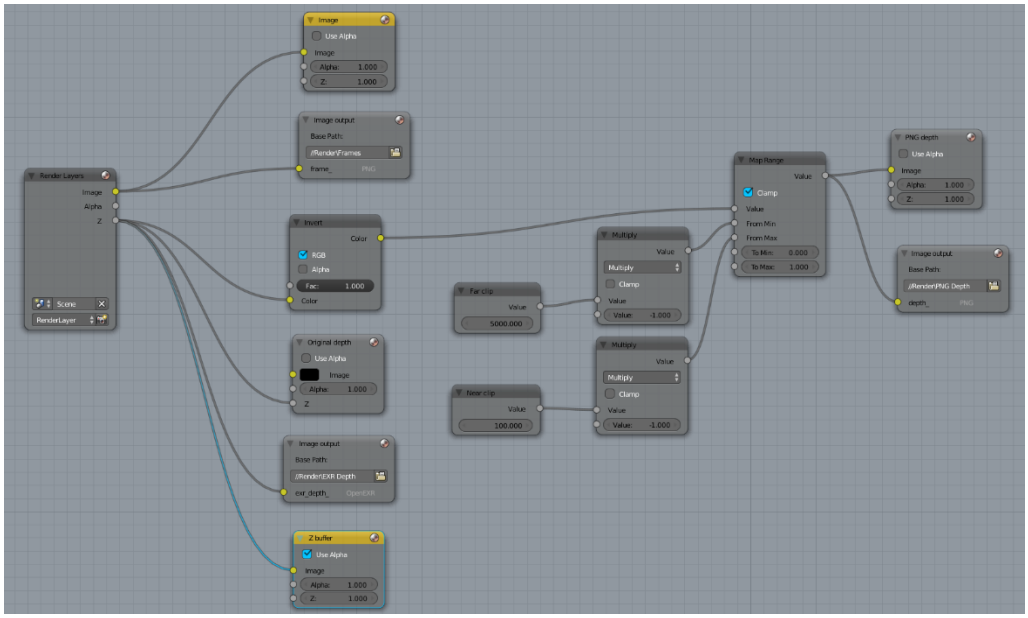

Figura 2.9: lo schema a blocchi creato per la composizione dell'output del rendering. In alto è presente la sezione per il salvataggio dell'immagine a colori, in basso quella per l'immagine OpenEXR della profondità e al centro lo schema per la normalizzazione di questa e il salvataggio come immagine png.

Recapitolando, l'output finale del rendering consiste nelle seguenti informazioni:

- Le immagini a colori delle camere sinistra e destra di un sistema stereo simulato;
- La mappa di profondità in formato OpenEXR e png normalizzato, per entrambi gli occhi;
- Un file di testo contenente le coordinate delle bounding box per l'immagine sinistra.

Nella figura 2.10 sono mostrate due immagini prese a campione dal risultato del rendering.

Figura 2.10: immagini renderizzate di esempio, con relative mappe di profondità in basso (valori chiari per gli oggetti vicini, scuri per quelli lontani). I puntini neri presenti nelle depth map sono causati da errori nella valutazione della profondità in corrispondenza di punti trasparenti. Come verrà mostrato nel capitolo 4, sono stati sistemati come procedura di preprocessing.

Per testare la correttezza del calcolo delle coordinate delle bounding box è stato implementato, inoltre, un piccolo script python che legge il file di testo con le coordinate e disegna le bounding box di una specifica immagine. In figura 2.11 sono mostrati i risultati relativi ai frame di figura 2.10.

Figura 2.11: rappresentazione delle bounding box dei frame mostrati in figura 2.10.

Prima di passare alla fase di progetto della rete, nel tempo a disposizione è stato possibile renderizzare circa 13000 frame stereoscopici. Grazie al fatto che con questa tecnica è possibile avere la mappa di profondità sia per l'immagine destra che per quella sinistra, nella pratica, per il task affrontato in questa tesi, le immagini utilizzabili in realtà per il training ammontano al doppio. Come verrà mostrato nel capitolo 4, questa quantità di immagini, accoppiata a qualche operazione di data augmentation, ha permesso un training soddisfacente della rete.

Capitolo 3
Predizione di depth da singola immagine

In questo capitolo verrà illustrato nel dettaglio il processo di valutazione e di selezione della rete neurale utilizzata per il progetto. A tal fine verrà fatto il punto sulle reti attualmente più performanti nell'ambito della predizione della profondità da singola immagine, per poi illustrare, in termini generali, l'architettura su cui si basa la rete selezionata, la quale verrà in seguito analizzata approfonditamente.

3.1 Reti neurali per depth prediction da singola immagine

La stima della profondità (**depth estimation**) rappresenta da sempre uno dei task più ricercati nel campo della computer vision, poiché la conoscenza della profondità di una scena permette di intuirne la geometria tridimensionale, aprendo la strada ad altri task come la ricostruzione 3D e l'orientamento automatico.

Come è stato già illustrato nel capitolo precedente, esistono diverse tecniche che permettono di valutare, in maniera più o meno precisa, la profondità di una scena. Tali tecniche tuttavia necessitano spesso di strumenti costosi o difficili da utilizzare.

Per questi motivi, il task della predizione della depth da singola immagine a colori ha sempre rappresentato un tema di grande importanza, poiché necessita di una semplice telecamera a colori, nonché un'ambiziosa sfida. Infatti, se nel caso, ad esempio, della visione stereo la corrispondenza locale tra le immagini destra e sinistra è sufficiente per stimare la profondità, nel caso dell'utilizzo di una singola immagine ci si trova di fronte a diversi elementi di ambiguità, che necessitano di un'analisi più ampia che include anche informazioni globali oltre che locali.

Alla luce di questo, fin dalle loro prime implementazioni, le reti neurali profonde sono state ampiamente sfruttate per la predizione della depth da

singola immagine, forti proprio delle loro capacità di analisi sia sul piano globale che locale.

È comunque da tener presente che l'utilizzo delle reti neurali nel campo della depth prediction da singola immagine comporta anche diversi limiti, rispetto alle tecniche di acquisizione diretta. Il più stringente tra questi è rappresentato dalla necessità di avere un dataset molto grande, al fine di carpire al meglio tutte le feature dell'immagine monoculare che rappresentano un collegamento tra la rappresentazione a colori e la mappa di profondità. Produrre un dataset di grandi dimensioni non è un'operazione banale, e non solo per le dimensioni. A causa dell'ambiguità insita nel task della depth prediction da singola immagine, infatti, le reti, una volta addestrate, dipenderanno molto dalla struttura delle scene (colori, scala di dettaglio, contenuto) presenti nel dataset, e questo suggerisce che queste debbano essere selezionate con particolare cura dipendentemente dal contesto in cui la rete verrà in seguito utilizzata. Ed è anche per questi motivi che, come è stato illustrato nel capitolo 2, per il progetto è stato deciso di produrre un dataset sintetico. Lo scenario di applicazione del supermercato, infatti, presenta caratteristiche abbastanza particolari, e dunque non sarebbe stato possibile utilizzare altri dataset simili poiché semplicemente non esistenti. Come già detto nel capitolo 2, inoltre, la scelta di usare la computer grafica viene dalla difficoltà, con l'utilizzo di immagini reali, di produrre delle label di qualità accettabile con i mezzi a disposizione. Strumenti di qualità maggiore presentano infatti costi troppo elevati, e comunque in molti casi presentano tempi di funzionamento lunghi che andrebbero ad allungare molto il processo di produzione del dataset. Un altro limite è rappresentato anche dai tempi necessari per l'addestramento che, nel caso di dataset molto grandi, possono diventare molto importanti.

Esistono comunque alcuni dataset largamente utilizzati per il testing di reti neurali per depth prediction. Tra i più famosi troviamo il dataset **NYU Depth V2** (15), creato dalla New York University e contenente sequenze video catturate in una varietà di scene indoor registrando le immagini RGB e le depth map mediante Microsoft Kinect (1), ed il dataset **Make3D** (16) (17), contenente scene in ambito outdoor le cui depth map sono state prodotte mediante l'utilizzo di laser scanner.

Come Illustrato nel capitolo 1, lo stato dell'arte nei principali task della computer vision è stato raggiunto grazie all'utilizzo delle reti neurali convoluzionali, o CNN, ed il task della depth estimation fa parte di questi. La rete che ha dato inizio all'utilizzo delle CNN nell'ambito della computer vision è stata LeNet5 (6), già citata nel capitolo 1, sviluppata da Yann LeCun nel 1998. Ispirate da questo lavoro, negli ultimissimi anni sono state presentate numerose architetture di CNN che hanno permesso di raggiungere risultati nettamente superiori rispetto alle precedenti tecniche di computer vision. Tra le principali troviamo:

- **AlexNet** (18): rappresenta l'architettura che nel 2012, vincendo la **ILSVRC** (19) (ImageNet Large-Scale Visual Recognition Challenge, competizione internazionale di riferimento nel campo della computer vision basata sul dataset ImageNet, contenente più di un milione di immagini corredate da label e progettato specificatamente per il benchmarking), ha dato inizio al trend di vasto utilizzo delle CNN in ambito computer vision. Il 2012 fu infatti il primo anno in cui la vittoria andò ad un'architettura basata su CNN. Oltre alla vittoria in se, quello che fece più scalpore nella community della computer vision fu il risultato raggiunto sull'errore top 5 (ovvero la probabilità che, data un'immagine, la label corretta non si trovi tra le 5 predizioni di valore più alto). Ottenne infatti un errore top 5 del 15,4%, con ben 10,6 punti percentuali di distacco sul secondo classificato, fermo al 26,2%. Rispetto alle reti odierne, AlexNet presenta una struttura relativamente semplice. Essa consiste infatti in 5 livelli di convoluzione, pooling e dropout, e 3 livelli fully connected;

- **VGG Net** (20) (Visual Geometry Group): si tratta di un'architettura CNN realizzata nel 2014 e caratterizzata da una struttura molto semplice ma al contempo molto profonda. Contiene ben 19 livelli, in cui vengono utilizzate solamente convoluzioni 3x3 con stride e padding unitari, insieme a pooling 2x2 con stride 2 e a 3 livelli finali fully connected. Presenta diverse configurazioni, di cui quella con risultati migliori ha registrato un errore minimo pari al 7,3% sul dataset ImageNet. Il risultato più importante che questa rete ha portato nel

campo delle CNN è l'aver dimostrato che il punto chiave per il miglioramento delle performance non è rappresentato tanto dall'aumento della complessità della rete, quanto dalla maggiore profondità di questa;

- **ResNet** (5) (residual network): si tratta di un'architettura che, mediante l'utilizzo di un nuovo ed innovativo tipo di blocchi (detti **residual block**) ed al concetto del **residual learning**, ha permesso di raggiungere profondità impensabili con il modello feed forward classico a causa del problema della degradazione del gradiente. Esistono implementazioni con profondità diverse, di cui la più profonda, infatti, conta ben 152 livelli. Esiste anche un prototipo con 1202 livelli, che però ha raggiunto risultati peggiori a causa dell'overfitting. Questa architettura ha vinto la ILSVRC 2015 facendo segnare un errore pari al 3,6%. Per intuire il valore di questo risultato, basti pensare che l'errore generalmente raggiunto da un essere umano si aggira intorno al 5-10%, in base alle sue capacità e conoscenze. Grazie a questi risultati, il modello ResNet rappresenta attualmente lo stato dell'arte nel campo della computer vision. Una più approfondita analisi di questa architettura verrà fornita nel prossimo paragrafo.

Per il task specifico della depth prediction da singola immagine RGB, due reti hanno attirato l'attenzione nella fase di ricerca di un'architettura da utilizzare per il progetto. Si tratta di Eigen *et al.* (21) e Laina *et al.* (22).

Il modello sviluppato da Eigen *et al.* consiste in un'architettura classica di deep CNN **multiscala**, ovvero in grado di analizzare le immagini di input a diversi livelli di dettaglio in modo da restituire risultati di maggiore precisione. È stata progettata per essere utilizzata su tre differenti task, ovvero stima della profondità, stima delle normali alle superfici e predizione delle label semantiche, tutto partendo da una singola immagine a colori. È possibile addestrare la rete separatamente per ognuno dei task applicando solo lievi modifiche. In base all'addestramento eseguito, la rete permette di predire le pixel map relative allo specifico task attraverso l'utilizzo di un livello finale

fully connected, normalmente utilizzato nei task di classificazione, i cui output vengono rimodellati in modo da creare la mappa di output.

Come già detto, si tratta di un'architettura multiscala che per prima cosa produce una predizione grossolana dell'output a partire dall'intera immagine, ed in seguito rifinisce questa predizione mediante l'utilizzo di reti locali a scala più piccola che analizzano i dettagli dell'immagine. Lo schema della rete è mostrato in figura 3.1.

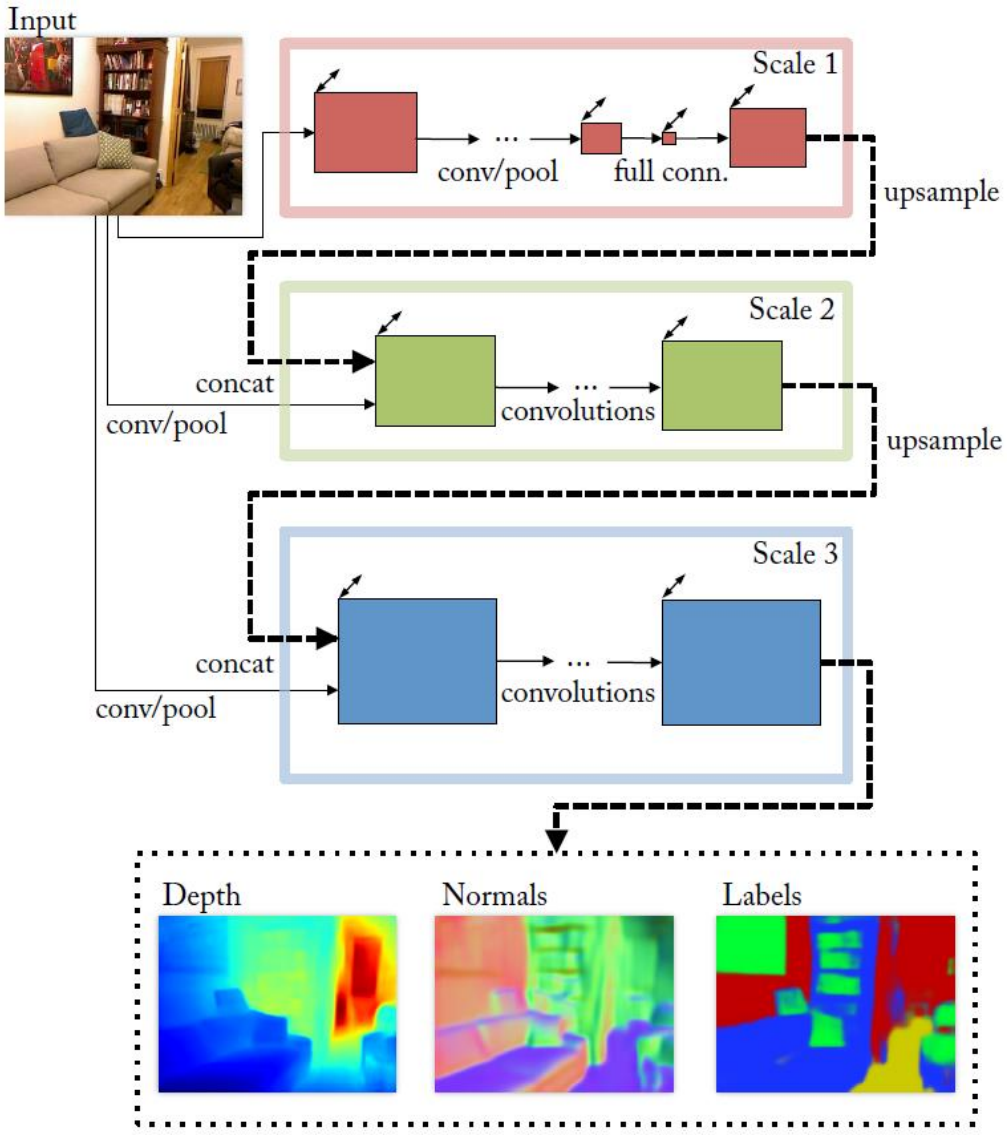

Figura 3.1 (21)**:** modello dell'architettura proposta da Eigen *et al.*

Nello specifico:

- nella prima fase (scala 1) la rete produce un set di feature grossolane ma spazialmente variabili a partire dall'intera immagine di input;
- nella seconda fase (scala 2), le feature predette nella prima fase, insieme ad un set di altre feature più dettagliate ma con un receptive field più ristretto sull'immagine di input, vengono utilizzate per produrre una predizione dell'output ad un livello di risoluzione intermedio;
- nella terza ed ultima fase (scala 3), le predizioni della fase precedente vengono rifinite aumentandone il livello di dettaglio mediante l'utilizzo di feature map generate a partire dall'immagine di partenza. Al termine del processo la risoluzione dell'output della rete è pari alla metà di quella degli input.

Al momento della sua pubblicazione nel 2015, questa rete ha raggiunto risultati pari allo stato dell'arte in tutti e tre i task di cui si occupa.

La seconda architettura, quella di Laina *et al.*, sviluppata per il solo task della depth prediction, si basa invece sul modello di ResNet, ed in particolare sulla versione a 50 livelli, detta ResNet-50. Come verrà illustrato nel dettaglio più avanti, si tratta di una versione fully convolutional di ResNet-50 in cui è stato introdotto un nuovo sistema per l'upsampling. Anche questa rete, al momento della pubblicazione nel 2016, ha raggiunto risultati pari allo stato dell'arte attuale nel campo della depth prediction.

In figura 3.2 è possibile visionare un confronto qualitativo tra i risultati raggiunti da AlexNet, VGG-16 (VGG Net in versione a 16 livelli), ResNet-50, Laina *et al.* ed Eigen *et al.* sul dataset di riferimento **NYU Depth v2**.

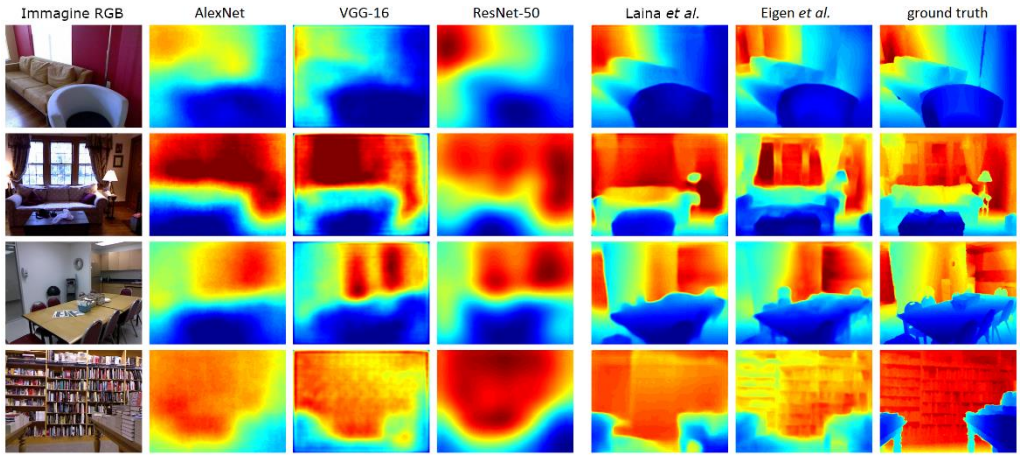

Figura 3.2 (22): confronto qualitativo dei risultati sul dataset NYU Depth v2 tra le diverse architetture menzionate. Le color map sono scalate allo stesso modo per facilitare il confronto.

Grazie agli ottimi risultati raggiunti nel task della depth prediction da singola immagine, corrispondenti come detto allo stato dell'arte nel settore, e spinti anche dall'interesse per l'utilizzo di un'architettura come ResNet, per il progetto è stato scelto di utilizzare il modello prodotto da Laina *et al.*

Nel prossimo paragrafo verrà illustrato più nel dettaglio il modello ed il funzionamento di ResNet, in modo da introdurre al meglio l'analisi dettagliata della rete di Laina *et al.* che verrà fornita nel paragrafo successivo.

3.2 Reti neurali residuali

Come si è già accennato nel paragrafo precedente, le reti neurali residuali, o ResNet, sono un particolare tipo di CNN, sviluppato nel 2015 in Microsoft, che utilizza un innovativo tipo di blocco, il **residual block**, e che sfrutta il concetto del **residual learning**.

Il concetto su cui si basa il residual block è quello di sottoporre un input x alla sequenza di operazioni convoluzione-ReLU-convoluzione, ottenendo una certa $F(x)$, e di sommare al risultato lo stesso x. In uscita a questo blocco si ha dunque $H(x) = F(x) + x$. In una CNN tradizionale di tipo feed forward si avrebbe invece, in pratica, che $H(x) = F(x)$. In figura 3.3 è mostrato il modello del residual block.

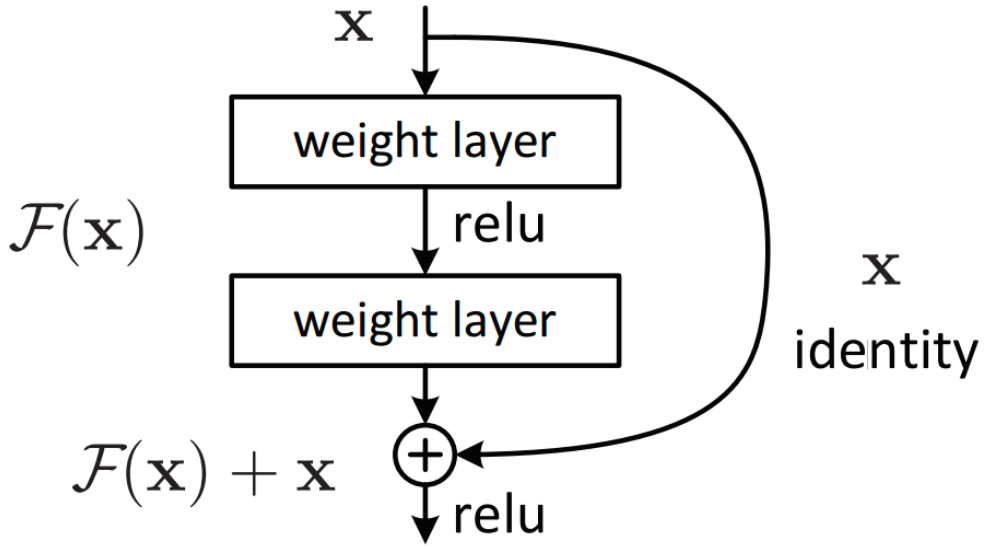

Figura 3.3 (5): modello del residual block

Attraverso la concatenazione di diversi blocchi di questo tipo, ResNet impara a predire un certo output non attraverso l'apprendimento di una trasformazione diretta dai dati di input all'output, ma attraverso l'apprendimento di un certo termine $F(x)$ da sommare al dato di input per arrivare all'output minimizzando l'errore, che viene chiamato errore residuale. Questo approccio rappresenta quello che è stato già accennato col nome di residual learning.

Un altro componente largamente utilizzato in ResNet è rappresentato dai livelli di **batch normalization** (23), utilizzati dopo ogni convoluzione e attivazione. La batch normalization è un'operazione che permette di normalizzare i dati presenti nei mini-batch, e grazie a questo riduce le limitazioni sul valore del learning rate che tipicamente sussistono nel training delle reti neurali profonde. Rende inoltre meno complessa la fase di inizializzazione dei pesi. Tutto questo porta ad una riduzione notevole dei tempi necessari per il processo di addestramento della rete.

L'idea centrale nel paper di ResNet è che, nella costruzione di una rete neurale con un elevato numero di livelli, la rappresentazione dei dati di ingresso dovrebbe rimanere quanto più possibile inalterata andando in profondità nella rete, in modo da preservare l'informazione.

Si supponga, ad esempio, di avere una rete poco profonda in grado di produrre delle predizioni perfette, e di aggiungere alcuni livelli ad essa. Dato che la rete predice già alla perfezione gli output desiderati, l'operazione migliore che i nuovi livelli potrebbero imparare ad applicare sarebbe la funzione identità. Se invece la rete originaria avesse imparato a fare delle predizioni con dei piccoli errori, il comportamento desiderato per i nuovi livelli sarebbe quello di sfruttare i risultati già raggiunti dalla rete ed applicare piccoli cambiamenti all'output, senza stravolgerlo, in modo da correggere questi errori.

In altri termini, l'apprendimento di un'operazione in grado di correggere un errore residuale derivante da una predizione già esistente rappresenta un problema più semplice rispetto all'apprendimento di una nuova trasformazione in grado di produrre una predizione completa.

Il residual block è stato introdotto proprio per implementare questo approccio all'apprendimento. Un altro importante motivo per cui questo blocco è particolarmente efficace è rappresentato dal fatto che, grazie all'operazione di addizione, nel passo di propagazione del gradiente della back propagation questo viene distribuito tra i livelli con i pesi ed il livello in testa al blocco. Questo permette di contrastare in maniera importante il problema, già descritto nel capitolo 1, della degradazione del gradiente, problema che si presenta in maniera marcata nelle reti con molti livelli e che ostacola il flusso del gradiente verso i livelli iniziali della rete, rallentando così l'addestramento. La capacità di contrastare questo problema rappresenta una delle caratteristiche principali che permette di costruire reti estremamente profonde.

3.2 La rete neurale selezionata

Come già illustrato precedentemente, la rete che è stata utilizzata per il progetto finale consiste nell'architettura proposta da Laina *et al.* (22) nel 2016. Come anticipato, si tratta di una CNN basata sull'architettura di ResNet-50 in versione fully convolutional, e modificata con l'introduzione di un nuovo sistema di upsampling che introduce il concetto di residual learning anche in

questa parte della rete, e che permette di ottenere depth map di dimensioni pari a circa la metà dell'immagine di input.

3.2.1 Architettura della CNN

Come già detto, si tratta di una CNN fully convolutional. Dal momento che manca la parte finale fully connected, che fornirebbe alla rete un receptive field completo su tutta l'immagine di input, il dimensionamento di questo rappresenta un aspetto molto importante per la progettazione dell'architettura. Nello specifico, la rete prende in ingresso immagini di dimensioni 304x228, e come detto predice depth map di dimensioni pari a circa la metà. Grazie all'utilizzo di ResNet-50, progettata per input di dimensioni 483x483, anche superiori rispetto a quelle utilizzate effettivamente, i requisiti per il receptive field sono pienamente soddisfatti.

L'utilizzo di input di dimensioni 304x228 in ResNet-50 porta ad avere 2048 feature map di dimensioni 10x8 nell'ultimo livello di convoluzione, senza considerare l'ultimo livello di pooling. Come verrà mostrato più avanti, il modello finale, che utilizza up-convoluzioni residuali per l'upsampling, produce un output di dimensioni 160x128. Se fosse stato utilizzato un livello fully connected, sarebbero stati introdotti ben 3,3 miliardi di parametri, corrispondenti a circa 12,6GB in memoria, rendendo l'implementazione infattibile coi mezzi a disposizione.

Il modello proposto può essere visionato in figura 3.4. Le dimensioni delle feature map corrispondono a quelle della rete addestrata con immagini di dimensione 304x228 appartenenti al dataset NYU Depth v2 (15). La prima parte della rete è quella che si basa su ResNet-50 ed è inizializzata con pesi pre-addestrati per classificazione su ImageNet. La seconda parte permette alla rete di apprendere l'upscaling attraverso una sequenza di livelli di unpooling e di convoluzione. Alla fine di questa sequenza è applicato il dropout, ed infine c'è un ultimo livello di convoluzione che fornisce le predizioni.

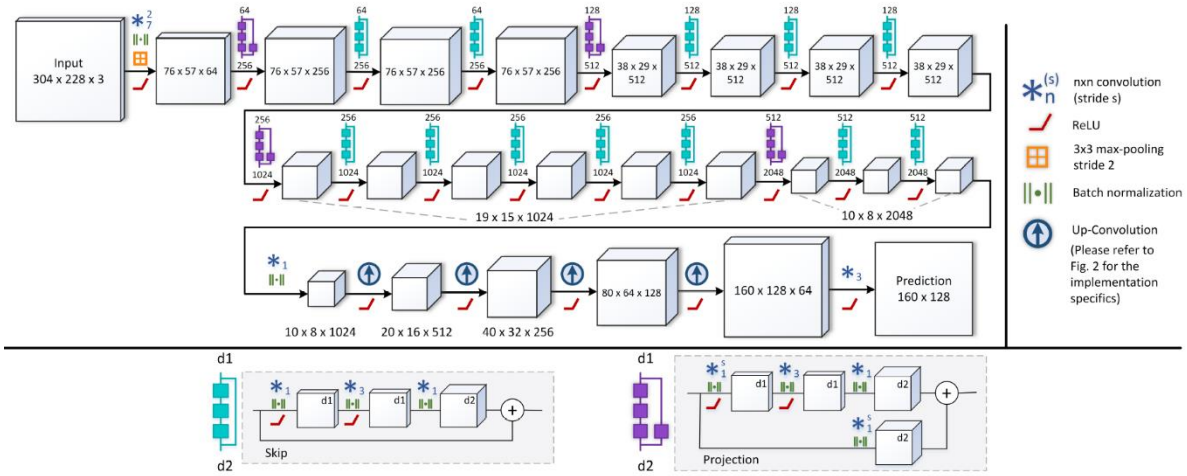

Figura 3.4 (22)**:** l'architettura della rete basata su ResNet-50. Il livello finale originale fully connected è sostituito dal nuovo blocco di upsampling, fornendo un output di risoluzione pari a circa la metà rispetto all'immagine di input.

3.2.1.1.Blocchi di Up-Projection

I livelli di unpooling effettuano l'operazione inversa del pooling, ed incrementano la risoluzione spaziale delle feature map. In una prima versione del modello proposto, l'upscaling è realizzato mediante un livello di unpooling che mappa i singoli valori di input nell'angolo sinistro superiore di un filtro 2x2 composto da valori nulli. Questo tipo di unpooling permette di raddoppiare le dimensioni delle feature in input. Ognuno di questi livelli è seguito quindi da una convoluzione 5x5, in modo che sia applicata a più di un elemento non nullo in ogni posizione. Infine, è aggiunto un livello di attivazione ReLU. Il blocco risultante viene definito up-convoluzione. Nel modello sono presenti 4 blocchi consecutivi di questo tipo, che risultano in un ingrandimento pari a 16 volte le dimensioni di partenza.

In una seconda versione del modello, che corrisponde a quella definitiva, questi blocchi di up-convoluzione sono stati estesi con l'introduzione del concetto del residual learning. L'idea alla base è l'inserimento di una semplice convoluzione 3x3 dopo il blocco di up-convoluzione, insieme all'inserimento di una connessione di proiezione dalle feature map di risoluzione inferiore verso il risultato. Il blocco risultante è mostrato in figura 3.5 (c).

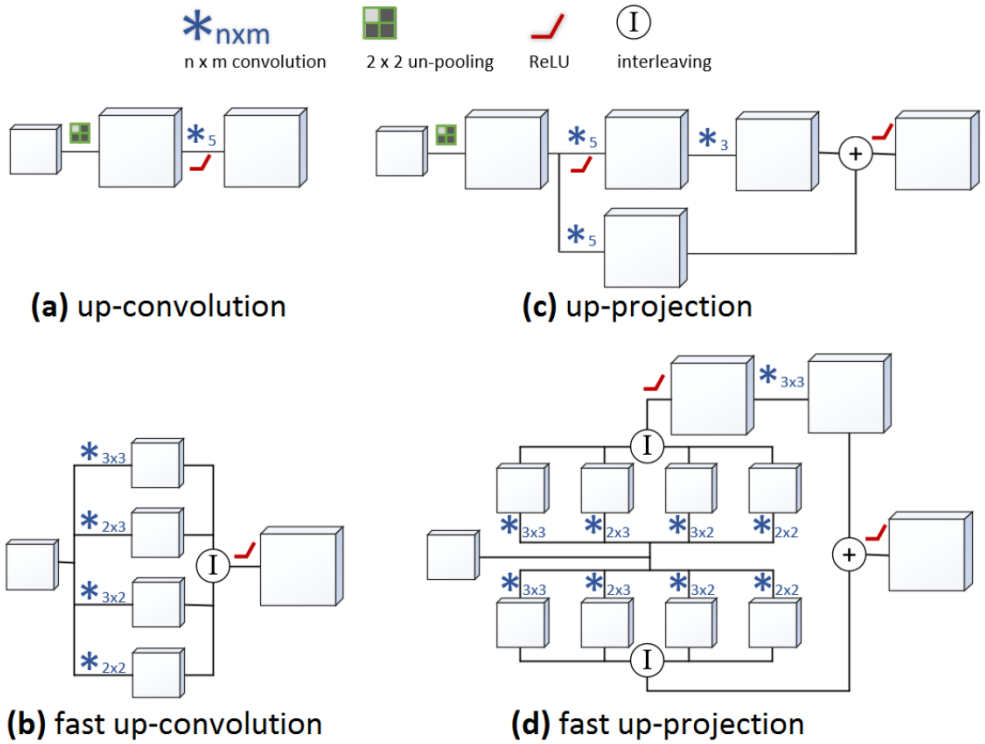

Figura 3.5 (22): confronto tra i blocchi di up-convoluzione e di up-projection. (a) Up-convoluzione standard. (b) Versione più veloce dell'up-convoluzione. (c) Nuovo blocco di up-projection. (d) Versione più veloce del blocco di up-projection.

A causa delle dimensioni diverse, le feature con risoluzione più bassa dovrebbero essere sottoposte ad upsampling utilizzando un altro blocco di up-convoluzione nel branch relativo, ma dal momento che l'unpooling può essere effettuato un'unica volta per entrambi i branch, l'unica operazione che viene effettuata a parte sul secondo branch è la convoluzione 5x5, come si può vedere in figura 3.5 (c). Il nuovo blocco di upsampling risultante è stato chiamato up-projection. La concatenazione di questi blocchi permette all'informazione di alto livello di fluire più efficacemente lungo la rete, e al contempo permette di incrementare gradualmente le dimensioni delle feature map. La figura 3.5 mostra le differenze tra un blocco di up-convoluzione e questo nuovo blocco di up-projection. Mostra inoltre le corrispondenti versioni più rapide, che verranno descritte nella prossima sezione.

3.2.1.2 Up-Convoluzioni veloci

L'ulteriore contributo che gli autori della rete hanno portato con questo lavoro è stato la riformulazione dell'operazione di up-convolution per renderla più efficiente, portando così ad un decremento del tempo di training per l'intera rete pari a circa il 15%. Quest'operazione di ottimizzazione è stata applicata anche al nuovo blocco di up-projection.

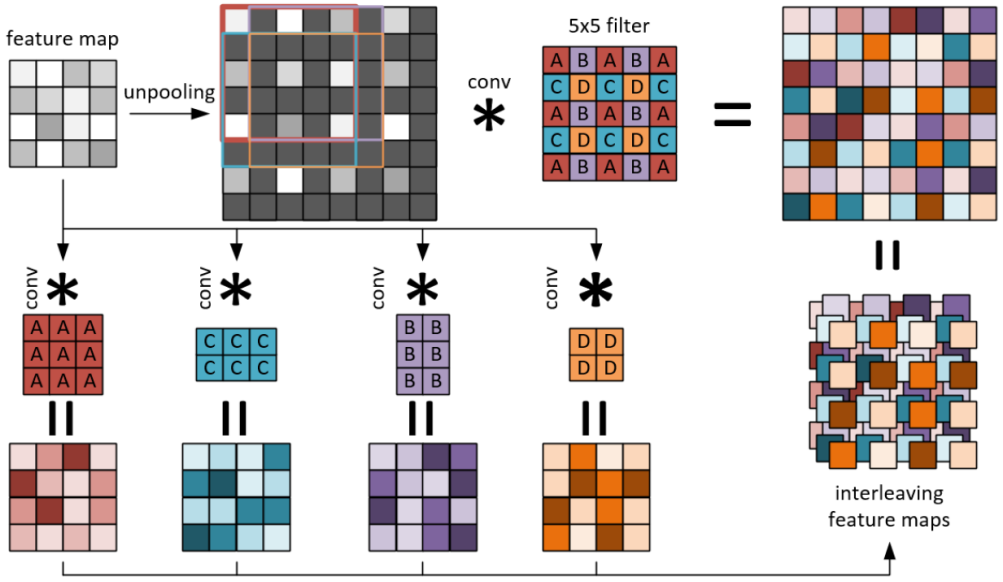

Figura 3.6 (22): illustrazione del processo di riformulazione della convoluzione 5x5.

L'idea alla base è la seguente: dopo l'operazione di unpooling, il 75% dei valori contenuti nelle feature map risultanti sarà nullo, e dunque la convoluzione 5x5 che segue lavorerà principalmente su valori nulli, per l'appunto. Una riformulazione di questa convoluzione permette però di evitare questa situazione. Il concetto è illustrato graficamente in figura 3.6. In alto a sinistra, la feature map originale viene sottoposta ad unpooling, come mostrato alla sua destra, e quindi convoluta con il filtro 5x5. È stato osservato che nel risultato dell'unpooling della feature map, in base alla posizione del filtro 5x5 (bounding box rossa, blu, viola ed arancione), solo alcuni valori del filtro, o pesi, vengono moltiplicati effettivamente con valori potenzialmente diversi da zero. Questi pesi ricadono in quattro gruppi non sovrapposti, indicati nella figura dai quattro colori e dalle label A, B, C, D. Basandosi su questi gruppi, il filtro originale è stato riorganizzato in quattro filtri di dimensioni 3x3 (A), 3x2 (B), 2x3 (C) e 2x2 (D). A questo punto, lo stesso esatto risultato prodotto

dall’operazione originale (unpooling e convoluzione) può essere ottenuto attraverso l’interleaving degli elementi delle quattro feature map risultanti, come mostrato nella figura 3.6. In figura 3.5 (b) e (d) sono mostrati i blocchi di up-convoluzione e up-projection dopo la riformulazione della convoluzione 5x5.

Il modello risultante dall’aggiunta di queste nuove tipologie di blocchi rappresenta l’architettura finale proposta da Laina *et al.*, architettura che come già detto rappresenta lo stato dell’arte attuale nel campo della depth prediction da singola immagine RGB. I risultati qualitativi comparati a quelli delle reti descritte precedentemente possono essere visionati in figura 3.2, mostrata nei paragrafi precedenti.

Nel prossimo capitolo verrà illustrato il lavoro che ha permesso di implementare una procedura di training per questa rete, e che ha quindi permesso di raggiungere i risultati che verranno mostrati conclusivamente.

Capitolo 4
Il training ed i risultati

Come è stato illustrato nel capitolo precedente, per il progetto è stato deciso di utilizzare l'architettura di CNN fully convolutional (o FCN, fully convolutional network) basata su ResNet e progettata da Laina *et al.* (22). Il codice della rete è open source ed è fornito in due implementazioni diverse:

- per il framework **MatConvNet** (24);
- per il framework **TensorFlow** (25).

Si tratta di due framework utilizzati per l'implementazione di reti neurali convoluzionali e, nel caso di TensorFlow, anche di altre applicazioni generiche di calcolo numerico.

Per quanto riguarda MatConvNet, si tratta di un insieme di tool per MATLAB espressamente sviluppati per l'implementazione di CNN per la computer vision. Comprende anche diverse architetture di CNN già implementate e pre-addestrate per task di classificazione, segmentazione, riconoscimento facciale e riconoscimento testuale.

TensorFlow rappresenta, invece, una libreria software open source per il calcolo numerico che utilizza un approccio a grafo del flusso di dati (**data flow graph**). È stato progettato per rendere disponibile la computazione parallela su più CPU o GPU, sia su singola macchina che nel distribuito. TensorFlow venne sviluppato originariamente da ricercatori e ingegneri del Google Brain Team per scopi di ricerca nel campo del machine learning e delle deep neural network, ma venne progettato con un approccio più generale in modo da poter essere utilizzato in molte altre applicazioni diverse di computazione.

Rispetto a MatConvNet, TensorFlow offre un'interfaccia di utilizzo in diversi linguaggi, tra cui python e C++. Proprio per motivi di maggiore affinità con questi linguaggi rispetto a MATLAB, ed anche per motivi di maggiore compatibilità con altri progetti paralleli a quello di questo lavoro di tesi, è stato deciso di utilizzare l'implementazione fornita per TensorFlow.

Nello specifico, si tratta di un'implementazione ottenuta dagli autori della rete attraverso l'utilizzo del tool di conversione **caffe-tensorflow** (26), sviluppato da Saumitro Dasgupta, che permette di convertire una certa architettura di rete implementata in Caffe (27), un altro framework per il machine learning molto utilizzato, nella corrispondente implementazione TensorFlow.

L'implementazione fornita utilizza le API TensorFlow per python, e comprende il modello della rete ed uno script basilare per testare la rete in inferenza. Sono forniti, inoltre, anche i pesi pre-addestrati sul dataset NYU Depth v2 sotto forma di file npy contenente degli array NumPy.

A partire da questa implementazione, come verrà mostrato più avanti, è stato implementato il codice che ha permesso di addestrare la rete sul dataset sintetico in primis e sul dataset di immagini reali in seguito. Prima di entrare nel dettaglio del processo di training, nel prossimo paragrafo verrà illustrato brevemente il framework TensorFlow e le sue principali funzionalità.

4.1 Tensorflow

Come accennato precedentemente, TensorFlow è una libreria software open source per il calcolo numerico basata sulla modellazione a grafi (**data flow graph**). Un grafo è definito come pipeline astratta di operazioni matematiche operanti su **tensori**, ovvero array multidimensionali. Ogni grafo è composto da **nodi**, ovvero operazioni sui dati, ed **archi**, rappresentanti i tensori, che passano attraverso le varie operazioni.

È una libreria utilizzata soprattutto nel campo del machine learning e delle reti neurali. Presenta numerose API, tra cui quella di più basso livello, TensorFlow Core, permette un controllo completo sulla programmazione. Queste API sono quelle tipicamente utilizzate nel campo del machine learning, poiché rendono possibile controllare nel dettaglio tutti gli elementi del modello che si sta implementando. Le API disponibili di livello più alto sono costruite a partire da TensorFlow Core. In alcuni casi possono rendere alcune operazioni, come task ripetitivi e predefiniti, più veloci e semplici, ma generalmente precludono la possibilità di andare nel dettaglio, e nell'implementazione di una rete neurale è

spesso necessario avere un controllo più preciso sulle operazioni. Possono comunque tornare utili per lo sviluppo di modelli standard di machine learning.

Come accennato precedentemente, TensorFlow fornisce interfacce per diversi linguaggi, tra cui python e C/C++ con pieno supporto e go/java in fase beta. Nel caso specifico di questo progetto è stato deciso di utilizzare l'interfaccia python. Supporta inoltre la computazione parallela su GPU o CPU, nonché la computazione distribuita. Permette l'esecuzione anche su dispositivi mobili.

Tra i principali vantaggi dell'utilizzo di TensorFlow troviamo:

- calcolo automatico delle derivate;
- tool di visualizzazione grafica;
- ottima documentazione;
- API molto simili a quelle NumPy.

Esistono comunque alcuni punti a sfavore di TensorFlow, tra cui:

- codice molto verboso;
- utilizzo alto della memoria;
- non esistono molti modelli di reti pre-addestrate (parzialmente vero, poiché, come accennato precedentemente, esistono tool per la conversione dei modelli);
- più lento rispetto ad altri framework, come ad esempio Caffe.

4.1.1 Struttura di un programma

Un programma TensorFlow è tipicamente strutturato in due fasi ben separate:

- **Fase di costruzione**: in questa fase vengono definite le varie operazioni del grafo che verranno eseguite sui tensori di input;
- **Fase di esecuzione**: in questa fase le operazioni definite nella fase precedente vengono valutate in modo da recuperarne l'output numerico. L'esecuzione delle operazioni è gestita mediante l'oggetto "**session**" di TensorFlow.

4.1.2 Tensori

L'unità di dato fondamentale in TensorFlow è rappresentata dal **tensore**. Un tensore consiste in un insieme di valori di tipo primitivo modellato sotto forma di array multidimensionale. In TensorFlow, la quasi totalità delle funzioni contenute nelle API prende in input dei tensori e dà in output sempre tensori.

Ogni tensore contenuto nel grafo ha un nome univoco che può essere indicato dall'utente, o altrimenti assegnato in maniera automatica.

Il comportamento predefinito in TensorFlow è quello di allocare tutte le componenti (tensori ed operazioni) nella memoria della GPU (se è stato installato con il supporto alla computazione su GPU). È tuttavia possibile specificare manualmente dove allocare ogni tensore ed operazione. È comunque consigliato di ridurre al minimo gli switch tra CPU e GPU poiché questi rallentano molto l'esecuzione.

TensorFlow permette inoltre di definire degli scope per le variabili, attraverso i quali viene gestito un meccanismo di namespace che facilita la definizione di modelli complessi. Gli scope sono molto importanti anche per la condivisione di variabili tra più grafi.

4.1.3 Gestione dell'input

TensorFlow fornisce due modalità per la lettura dei dati di ingresso (ovvero i batch di esempi nel caso delle reti neurali):

- **manuale**: gli esempi vengono manualmente letti e organizzati in batch ed in seguito passati al modello. È un meccanismo semplice da utilizzare, ma può diventare molto lento poiché i dati devono essere continuamente copiati dall'environment python a quello TensorFlow;
- **integrata**: tutte le operazioni per leggere i dati e organizzarli in batch di esempi sono implementate all'interno del grafo. È un meccanismo meno intuitivo da utilizzare ma presenta un grande aumento di performance, dal momento che tutti i dati rimangono sempre nell'environment di TensorFlow.

Per quanto riguarda la lettura vera e propria dei dati su disco, esistono diverse soluzioni utilizzabili:

- **file immagine**: gli esempi sono memorizzati in memoria direttamente come file immagine, da leggere nella maniera preferita;
- **file binari**: le informazioni sono codificate in file binari, che nel caso specifico vengono chiamati **TFRecord**. TensorFlow fornisce strumenti sia per la lettura che per la scrittura di questi file binari. L'utilizzo di file TFRecord riduce notevolmente il numero di accessi al disco, migliorando così l'efficienza.

TensorFlow mette inoltre a disposizione numerose funzioni per il data augmentation.

4.1.4 Operazioni

Tutte le operazioni basilari per le reti neurali sono implementate in TensorFlow come nodi del grafo. Il framework gestisce automaticamente tutto il necessario per l'implementazione del forward e backward pass, incluso il calcolo automatico delle derivate.

Tra le principali operazioni disponibili per la costruzione dei modelli di reti neurali troviamo:

- **convoluzioni**;
- **somma dei bias**;
- **livelli fully connected**;
- **funzioni di attivazione**;
- **pooling**'
- **funzioni per la produzione delle predizioni**.

4.1.5 Loss function ed ottimizzazione

All'interno di un programma TensorFlow, ogni operazione che produce un singolo valore come output può essere utilizzata come loss function. Il grande vantaggio è che il framework gestisce in maniera completamente automatica il calcolo del gradiente.

Inoltre, le funzioni più comunemente utilizzate per i task di classificazione e regressione sono già implementate nel framework

TensorFlow offre inoltre, già implementati, diversi algoritmi classici per l'ottimizzazione della loss function, come ad esempio il Gradient descent, il Momentum e l'Adam. Come per il calcolo del gradiente, anche la fase di ottimizzazione è gestita in automatico dal framework.

Recapitolando, il processo per il training di un qualsiasi modello di machine learning in TensorFlow può essere suddiviso nelle seguenti fasi fondamentali:

1. Creazione delle operazioni di input che caricano gli esempi dal disco;
2. Creazione del modello della rete;
3. Creazione della loss function;
4. Definizione dell'ottimizzatore;
5. Valutazione ciclica dell'operazione di training fino al raggiungimento della convergenza;
6. Salvataggio su disco del modello addestrato.

Per quest'ultima fase, TensorFlow mette a disposizione degli strumenti appositi per il salvataggio su disco ed il ripristino dei pesi del modello. Questi strumenti possono essere utilizzati anche per il salvataggio dei pesi a certi intervalli di tempo, in modo da avere un checkpoint da cui ripartire in caso di errori o in caso si voglia suddividere l'addestramento in più periodi.

4.1.6 TensorBoard

TensorBoard è una suite di strumenti per la visualizzazione grafica completamente integrata in TensorFlow. Permette di visualizzare il grafo computazionale del modello e molte altre statistiche utili per l'analisi del processo di training. In figura 4.1 è mostrata l'interfaccia base di TensorBoard.

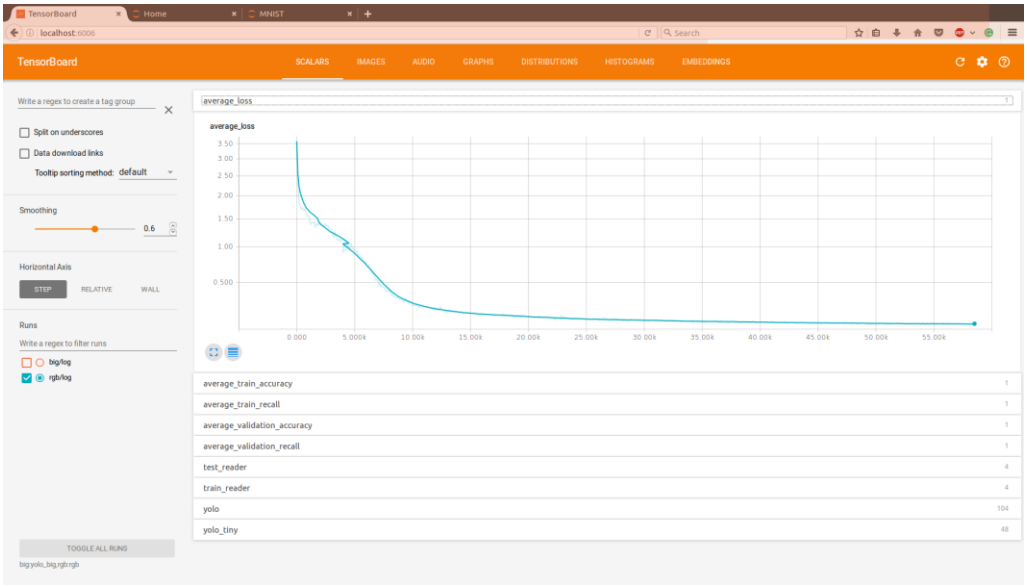

Figura 4.1: interfaccia grafica di TensorBoard.

TensorBoard opera su dei file di eventi di TensorFlow. Questi file contengono un sommario dei dati relativi ai nodi che è stato deciso di monitorare. TensorFlow mette a disposizione strumenti appositi per la creazione del file eventi e per la scrittura dei dati al suo interno.

Una volta messo in esecuzione, l'interfaccia di TensorBoard può essere visualizzata mediante un web browser.

4.2 Il processo di training

Alla luce delle informazioni su TensorFlow date nel paragrafo precedente, in questo paragrafo verrà illustrato il lavoro che ha premesso di implementare il training della rete neurale e di produrre i risultati che verranno mostrati infine.

Per lo sviluppo è stato utilizzato TensorFlow in ambiente Ubuntu 16.04. Come già accennato, sono state utilizzate le API TensorFlow per python. Per rendere l'esecuzione il più efficiente possibile, è stato deciso di seguire l'approccio dei file TFRecord per la gestione dell'input.

Il lavoro è stato suddiviso in tre fasi:

- La conversione dei dataset in file TFRecord;
- L'implementazione del codice per il training;
- L'implementazione del codice per produrre le predizioni;

Nei prossimi paragrafi verrà illustrata nel dettaglio ognuna di queste fasi.

4.2.1 Conversione dei dataset in TFRecord

La prima operazione per preparare il processo di training è stata quella di produrre i file TFRecord a partire dai file immagine contenuti nei due dataset, quello sintetico prodotto e quello con immagini reali che è stato utilizzato in seguito per il finetuning e per i test.

Il processo di conversione è stato diviso in due fasi:

- La creazione di uno script per la creazione di una lista di nomi file;
- La creazione dello script di conversione vero e proprio.

Nel primo script vengono prima letti da disco i nomi file delle immagini rgb e delle corrispondenti depth map, ed in seguito le coppie di nomi file vengono inserite in un array. Su questo array viene applicata dunque una funzione di ordinamento casuale, in modo da distaccare i singoli esempi dalla relativa sequenza di rendering, ed infine l'array risultante viene salvato su disco in formato npy.
Facendo in questo modo è possibile avere sempre disponibile un supporto con il quale reperire facilmente tutti gli esempi del dataset, e soprattutto poterli leggere sempre nello stesso ordine se necessario, pur mantenendo

l’indipendenza tra singoli esempi e sequenza di rendering dai quali provengono.
Sono stati creati due script di questo tipo, uno per il dataset sintetico ed uno per il dataset reale. È stato necessario differenziare poiché l’organizzazione dei file su disco varia tra un dataset e l’altro.

Una volta prodotti i file con le liste dei nomi file, si è passati all’implementazione dello script di conversione vero e proprio.
Per prima cosa, lo script legge i dati dai file creati nel passo precedente. In seguito, in base a degli indici passati come parametro, seleziona una particolare porzione della lista dei nomi file su cui applicare la conversione. Questa operazione è stata inserita per permettere, in modo semplice, l’esecuzione di più istanze dello stesso script per la conversione in parallelo del dataset, funzionalità utile per ridurre i tempi dal momento che questo contiene diverse migliaia di esempi.
Una volta selezionata la porzione del dataset da convertire, vengono letti da file gli esempi in essa contenuti. Mediante le funzioni TensorFlow apposite, i dati letti vengono dunque salvati a gruppi in diversi file TFRecord, che contengono una struttura dati a dizionario.

Nel particolare, ogni esempio in un file TFRecord rappresenta un piccolo dizionario, le cui tuple, dette feature, rappresentano una certa informazione. In fase di lettura, ogni informazione può recuperata attraverso il nome che le è stato assegnato in fase di creazione. Nel caso in questione, le feature per ogni esempio saranno due, una contenente i byte dell’immagine rgb e l’altra contenente i byte della depth map.
Gli esempi sono stati distribuiti in un certo numero di file TFRecord. Facendo in questo modo si facilita la successiva distribuzione degli esempi tra training set, validation set e test set, ed allo stesso tempo si mantengono file abbastanza grandi (leggibili in maniera sequenziale) da non rallentare le operazioni di I/O.

Nel caso del dataset sintetico, è stato necessario implementare, inoltre, una procedura per la correzione di alcuni errori presenti nelle depth map. Si tratta di pixel per cui l’algoritmo di rendering non è riuscito a valutare la profondità, e che quindi nella mappa di profondità sono stati salvati come valori all’infinito.

Questi errori sono stati mostrati in figura 2.10, nel capitolo 2. Sostanzialmente, per correggerli viene applicato un passo di interpolazione bilineare sui pixel con errori, in modo da lasciare il più possibile inalterata la rappresentazione.

Anche per quanto riguarda il dataset reale, nelle depth map sono presenti alcuni errori derivanti dall'algoritmo stereo che le ha prodotte. La loro correzione, come verrà illustrato nel prossimo paragrafo, è stata rimandata però alla fase di training, poiché in questo caso gli errori sono più estesi rispetto al dataset sintetico, e per correggerli senza impattare troppo sulla rappresentazione era necessario avere disponibili gli output della rete, prodotti in fase di training.

4.2.2 Implementazione del codice di training

In questa fase è stato prodotto il codice per il training vero e proprio. Per l'implementazione ci si è basati in parte sul codice per l'inferenza fornito dagli autori della rete ed in parte su alcuni esempi forniti da TensorFlow. Lo script può essere suddiviso nelle seguenti parti:

- Definizione delle operazioni per la gestione dell'input;
- Definizione del grafo;
- Definizione della funzione di loss;
- Definizione delle operazioni di ottimizzazione;
- Definizione delle operazioni per l'inizializzazione delle variabili;
- Definizione del ciclo di esecuzione.

Nei paragrafi seguenti verrà illustrata ogni parte nello specifico.

4.2.2.1 Gestione dell'input

In questa fase sono state definite le operazioni per la gestione dell'input mediante TFRecord. Sono state create due funzioni, una per la lettura da file di un singolo esempio ed un'altra per la creazione dei mini-batch a partire dagli esempi.

La prima funzione utilizza una coda di nomi di file, da cui recupera i file TFRecord dai quali estrarre i singoli esempi. Una volta recuperato l'esempio, vengono lette le feature contenute in esso, ovvero l'immagine rgb e la depth map. A questo punto i dati contenuti nelle feature vengono decodificati e

rimodellati, in modo da ottenere le immagini rgb e le depth map in forma matriciale. I valori della matrice dell'immagine rgb vengono inoltre convertiti in float, poiché la rete lavora su questo tipo di dati.
A questo punto viene effettuato un primo passo di data augmentation che applica all'immagine, in modo casuale, una riflessione rispetto all'asse verticale.

La seconda funzione implementata si occupa della creazione dei mini-batch, che verranno poi utilizzati per il training e per la validazione, e della creazione della coda che li conterrà. La funzione per prima cosa crea una lista contenente i nomi dei file TFRecord disponibili. In seguito invoca la prima funzione, passandogli come input la lista creata, in modo da ottenere i singoli esempi e da inserirli in una coda. La capacità della coda dipende dalla durata desiderata di addestramento, espressa in numero di epoche. Nello specifico, un'epoca rappresenta un ciclo di training nel quale vengono utilizzati tutti gli esempi contenuti nel training set. Un singolo passo di training che analizza un solo mini-batch viene detto invece step. Creata questa coda di esempi, la funzione pesca in modo casuale un certo numero di esempi da essa, ed in base al batch-size specificato li assembla in un tensore delle giuste dimensioni rappresentante il mini-batch
L'ultima operazione effettuata prima di restituire in output i tensori dei mini-batch consiste in un secondo passo di data augmentation, rappresentato dall'estrazione di crop casuali dalle immagini di ogni mini-batch.

4.2.2.2 Definizione del grafo

Una volta definite le operazioni per la gestione dell'input e la creazione dei mini-batch, il passo successivo è stato la definizione del grafo del modello. A questo scopo, è stata utilizzata l'implementazione esistente e fornita dagli autori della rete. Nello specifico, sono stati creati due grafi a partire dallo stesso modello. Uno verrà infatti utilizzato per il training, e l'altro per la validazione. Per permettere al grafo di validazione l'utilizzo dei pesi che man mano vengono aggiornati durante il training nel grafo di training, i due grafi sono stati creati in uno scope apposito che permette la condivisione dei pesi tra di essi.

4.2.2.3 Funzione di loss e termine di smoothness

Una volta definiti i grafi della rete, è stata definita la funzione di loss per il training e la funzione per il calcolo dell'errore di predizione in validazione. Per la prima è stato deciso di implementare la possibilità di scelta tra tre diverse operazioni di loss: la norma $l1$, la norma $l2$ e la funzione di Huber inversa, detta Berhu (28). È stato deciso di includere questa loss function poiché si tratta di quella che gli autori della rete hanno utilizzato con ottimi risultati per il loro training. La loss Berhu consiste nella seguente funzione matematica:

$$B(x) = \begin{cases} |x| & |x| \leq c \\ \dfrac{x^2 + c^2}{c^2} & |x| > c \end{cases}$$

in cui x rappresenta la matrice di differenza, o errore, tra la depth map attesa e quella predetta, mentre c è una costante scalare. Sostanzialmente, in base al valore di c la funzione di Berhu restituisce la norma $l1$ o la norma $l2$ dell'input. Il valore del termine c è stato scelto come quello utilizzato dagli autori della rete, ovvero pari al 20% dell'errore massimo rispetto ad un singolo mini-batch, ovvero:

$$c = \frac{1}{5} max_i(|x_i|)$$

in cui x_i rappresenta la matrice di errore della singola predizione i nel mini-batch.

Esistono prove empiriche secondo le quali l'utilizzo della loss Berhu rappresenta un buon equilibrio tra le due norme l_1 ed l_2 nell'utilizzo con le reti residuali, caratteristica che porta a risultati migliori nella fase di ottimizzazione.

Prima del calcolo della loss function è inoltre stata applicata la correzione degli errori per le depth map del dataset reale di cui si è parlato nel paragrafo sulla conversione dei dataset. Nello specifico, questi errori corrispondono a pixel con valori di depth prossimi allo zero o maggiori di 7 metri. Per annullare l'effetto di questi errori sul processo di ottimizzazione, tutti i valori dei pixel di errore delle label sono stati sostituiti con i corrispondenti valori dei pixel dell'output della rete. Facendo in questo modo, infatti, la differenza tra i pixel

della label e quelli dell'output in corrispondenza dei pixel di errore sarà pari a zero, e quindi non influirà sul valore della loss function. Sostanzialmente, facendo in questo modo i pixel di errore vengono scartati dal processo di training.

In questa fase è stata inoltre definita un'altra metrica da aggiungere come termine della loss function. Si tratta di un termine che rappresenta la similarità tra il valore di depth in ogni pixel ed i valori di depth nei pixel vicini. Questa metrica si trova spesso in letteratura quando si parla di valutazione della profondità di una scena, ma il suo utilizzo nel campo delle reti neurali rappresenta un contributo originale di questa tesi. È stata definita da noi col nome di "**smoothness**". In termini matematici è rappresentata dalla seguente formula:

$$S(p) = \frac{1}{w * h} \sum_{i=1}^{h} \sum_{j=1}^{w} \frac{|p_{i,j} - p_{i-1,j}| + |p_{i,j} - p_{i+1,j}| + |p_{i,j} - p_{i,j-1}| + |p_{i,j} - p_{i,j+1}|}{4}$$

in cui w ed h sono la larghezza e l'altezza dell'immagine in pixel e p è la matrice della predizione a cui è stato aggiunto un padding di un pixel a tutti e quattro i bordi. Sostanzialmente, per ogni pixel si calcola la media delle differenze in valore assoluto tra il pixel ed i suoi pixel vicini 4-connected, ed in seguito viene calcolata una media su tutti i valori ottenuti. I valori del padding ai quattro bordi sono stati scelti uguali a quelli della riga/colonna vicina, in modo tale che la differenza sui pixel di bordo non influisca sul calcolo.

Intuitivamente, questo termine dovrebbe aiutare a definire in maniera migliore le zone a profondità uniforme, come ad esempio le facce piane ed altamente texturate dei prodotti da supermercato. Nel prossimo paragrafo, riguardante i risultati finali, verrà mostrato come questo termine ha portato effettivamente dei benefici.

Recapitolando, la funzione complessiva da minimizzare in fase di training è dunque rappresentata dalla seguente formula:

$$F(x) = L(x) + c * S(p)$$

in cui x rappresenta la differenza tra la label e la predizione, p la predizione, $L(x)$ la loss function (che come detto può essere scelta tra l_1 l_2 e $Berhu$), $S(p)$ la smoothness e c una costante scalare minore di 1 con la quale viene pesato il termine di smoothness.

Come metrica per il calcolo dell'errore di predizione sul dataset di validazione è stata utilizzata la loss function priva del termine di smoothness.

4.2.2.4 Operazioni di ottimizzazione

Il passo successivo è stato la definizione delle operazioni per l'ottimizzazione della loss function. È stato deciso di utilizzare l'ottimizzatore momentum, poiché si tratta di quello utilizzato dagli autori della rete per il loro training. È stata inoltre impostata la decrescita esponenziale nel tempo del learning rate. L'operazione di ottimizzazione vera e propria è stata definita mediante le apposite funzioni di TensorFlow che, come detto nel paragrafo precedente, gestiscono automaticamente il calcolo del gradiente e l'aggiornamento dei pesi del modello in fase di training.

4.2.2.5 Inizializzazione delle variabili

La fase di inizializzazione delle variabili è stata suddivisa in due parti: l'inizializzazione casuale e il caricamento dei pesi pre-addestrati.

La prima parte di inizializzazione casuale è necessaria per inizializzare tutte quelle variabili che non sono comprese tra i presi pre-addestrati, ed è implementata mediante le apposite funzioni di TensorFlow.

La seconda parte sovrascrive i valori della prima inizializzazione per quanto riguarda i pesi del modello. Per questa operazione sono state implementate due possibilità: una utilizza il file npy fornito dagli autori della rete e contenente i pesi pre-addestrati sul dataset NYU Depth v2, mentre l'altra fa uso dei checkpoint TensorFlow.

Nello specifico, per il caricamento dei pesi dal file npy è stata utilizzata l'apposita funzione contenuta nell'implementazione del modello. La seconda possibilità di caricamento consiste invece nell'utilizzo dei file di checkpoint di TensorFlow. Il motivo per cui è stata introdotta questa possibilità è rappresentato dal fatto che il caricamento con file npy risulta molto lento, mentre il ripristino di un checkpoint è gestito in maniera molto efficiente da TensorFlow. È stata implementata dunque la possibilità di eseguire lo script di training in una modalità che permette la conversione dei pesi da file npy a checkpoint TensorFlow. Sostanzialmente, il programma carica prima i pesi dal file npy, ed in seguito esegue subito un salvataggio di un checkpoint del modello, per poi terminare l'esecuzione. In questo modo tutte le esecuzioni successive potranno caricare i pesi iniziali direttamente dal checkpoint, senza passare dal file npy, velocizzando così il training.

I sistema dei checkpoint viene inoltre utilizzato non solo per il caricamento dei pesi iniziali, ma anche per il ripristino dei pesi salvati in un certo momento dell'addestramento. In questo modo è possibile mettere in pausa il processo di training e riattivarlo in un secondo momento, in base alla disponibilità delle risorse di calcolo.

4.2.2.6 Ciclo di esecuzione

L'ultimo passo per l'implementazione del training è stata la definizione del ciclo di esecuzione.

La prima operazione effettuata è stata la creazione dell'oggetto della sessione TensorFlow, necessario per la valutazione di tutte le operazioni definite nei passi precedenti.

A questo punto è stato definito il ciclo vero e proprio, che continua l'esecuzione finché la coda dei mini-batch non è vuota.

Ad ogni step di esecuzione, vengono valutate allo stesso momento le operazioni dell'ottimizzatore e del calcolo della loss function, in modo da effettuare un passo di ottimizzazione su un singolo mini-batch. Con una certa frequenza viene inoltre stampato a video un riepilogo sui valori attuali della loss e dell'errore di predizione e sul numero di step completati. Vengono

inoltre salvate in un file eventi TensorFlow diverse statistiche sul processo di training in esecuzione, nonché un file di checkpoint con i pesi del modello.

4.2.3 Implementazione del codice per la predizione

Implementato il codice per il training, il passo successivo è stato l'implementazione di un altro script che utilizza la rete in inferenza per la predizione delle depth map.

Sostanzialmente, lo script è strutturato come quello per il training, con la differenza che è presente solo un grafo e non sono presenti le operazioni per l'ottimizzazione, per il calcolo della loss function e per il salvataggio dei checkpoint e del file eventi.

In realtà, sono stati creati più script, in modo da rendere possibile la predizione di immagini rappresentate diversamente. È stata creata una versione per la predizione di una singola immagine letta da file, una per la predizione di immagini multiple lette sempre da file, una per la predizione di immagini contenute in file TFRecord ed una per la predizione dei frame di un filmato.

Nello specifico, per le prime due versioni i file immagine vengono letti manualmente su disco e le immagini vengono direttamente date in pasto alla rete.

Nella versione che predice i frame di un video, il file del filmato viene letto manualmente su disco, ed in seguito ogni suo frame viene estrapolato e trattato come un'immagine singola data in pasto alla rete.

Nella versione che effettua predizioni su immagini contenute in file TFRecord, invece, la gestione dell'input è molto simile a quella che è stata descritta per il training. Le funzioni utilizzate per leggere gli esempi sono le stesse illustrate precedentemente. L'unica differenza risiede nell'eliminazione delle procedure di data augmentation e nell'eliminazione della componente casuale nella scelta degli esempi. In questo modo, infatti, è possibile effettuare diverse predizioni sullo stesso file TFRecord, per confrontare i risultati di diversi training ad esempio, rimanendo certi di ottenere le predizioni sulle stesse immagini e non su immagini prese a caso tra quelle nel TFRecord.

A parte la fase di input, le varie versioni dello script funzionano in maniera analoga. Per prima cosa vengono ripristinati i pesi della rete utilizzando un checkpoint specificato, quindi viene creata la sessione TensorFlow nella quale viene valutata la rete in inferenza, in modo da produrre le predizioni. Nel caso della predizione della singola immagine quest'operazione viene eseguita una sola volta, mentre per il caso delle immagini multiple, del video e dei TFRecord viene creato un ciclo di esecuzione in cui ad ogni step viene predetto un mini-batch di immagini di dimensioni pari al batch-size utilizzato per l'addestramento. Infine, ogni singola predizione viene salvata su disco come immagine in formato png. Nel caso della predizione da TFRecord, vengono salvate su disco anche l'immagine rgb di input e quella della label della depth map.

4.3 Valutazione dei risultati

Una volta pronti tutti gli script necessari, è stato possibile procedere con l'addestramento della rete e con la successiva valutazione dei risultati.

4.3.1 Addestramento sul dataset sintetico

Tutti i training svolti in questa fase sono stati effettuati a partire dai pesi pre-addestrati sul dataset NYU Depth v2, forniti dagli autori della rete.

Il primo addestramento che è stato fatto consiste sostanzialmente in un test per verificare l'efficacia del termine di smoothness descritto nei paragrafi precedenti. Gli iperparametri del training sono stati impostati nel seguente modo:

- **durata del training**: 2 epoche, corrispondenti a circa 7000 step di esecuzione;
- **batch-size**: 6, il più alto possibile con l'hardware a disposizione, ovvero una GPU nVidia Titan X;
- **learning-rate**: 0,01, lo stesso utilizzato in fase di training dagli autori della rete;
- **loss function**: Berhu.

Con questa configurazione sono stati quindi effettuati due training, di cui il primo senza il temine di smoothness ed il secondo con questo termine. Per la costante moltiplicativa della smoothness è stato utilizzato il valore 0,001. Ognuno dei due training ha impiegato circa 60 minuti. In figura 4.2 sono presenti due grafici comparativi che mostrano l'andamento del valore della loss e dell'errore di predizione nei due training.

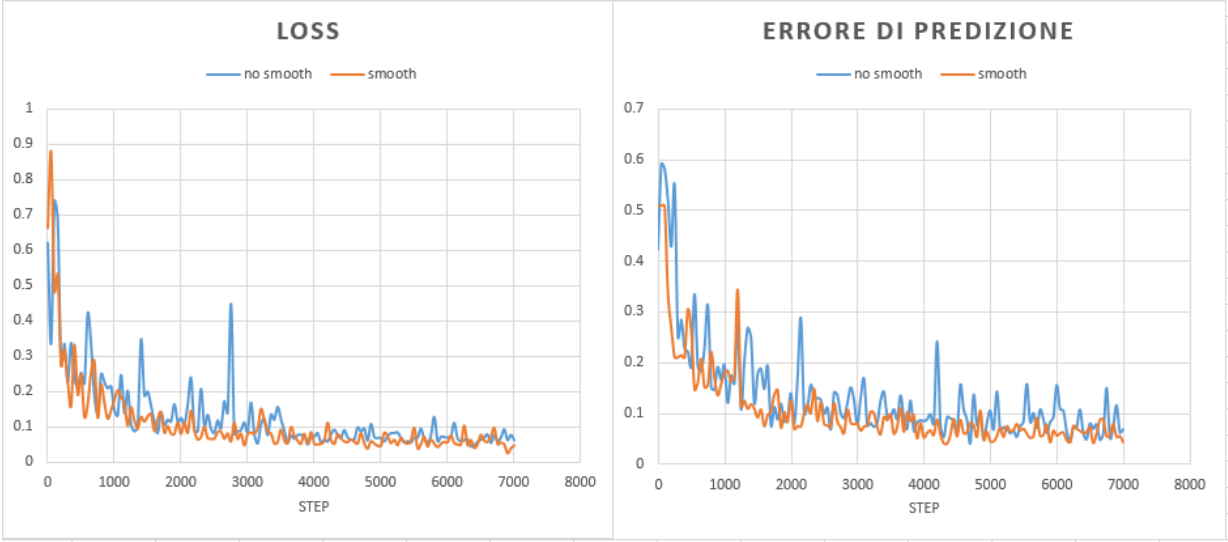

Figura 4.2: confronto tra l'andamento di loss ed errore di predizione nei due training effettuati.

È possibile apprezzare un andamento leggermente migliore nel caso dell'utilizzo del termine di smoothness. Considerando una media sugli ultimi 1000 step, nel caso del training senza smoothness i valori si attestano a circa 6,7cm per la loss e 8,1cm per l'errore di predizione, mentre nel caso del training con smoothness i valori sono pari a 5,9cm per la loss e 6,1cm per l'errore di predizione. In figura 4.3 il miglioramento dei risultati è apprezzabile anche visivamente in due esempi di predizione.

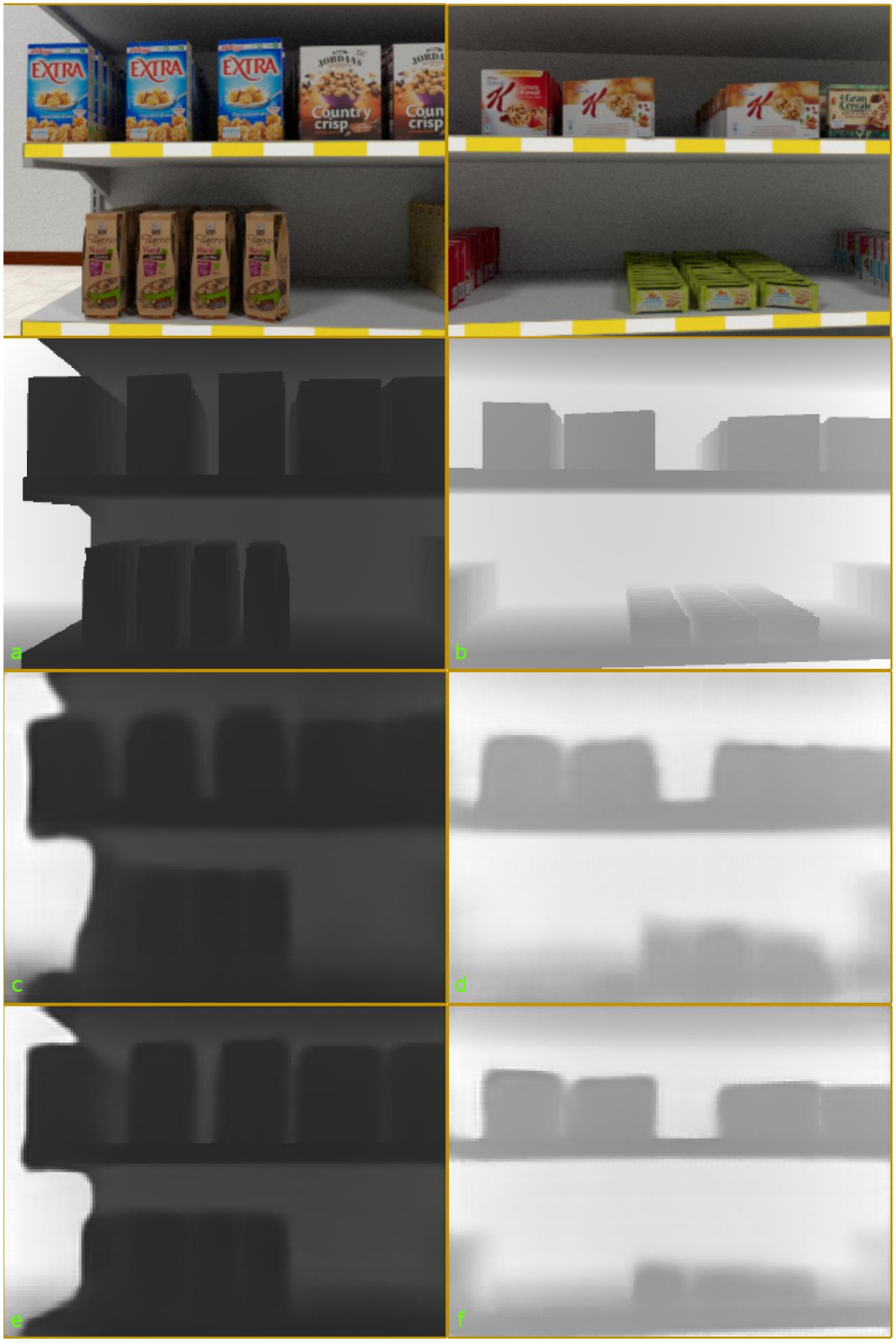

Figura 4.3: confronto qualitativo tra le label (a,b), i risultati senza smoothness (c,d) ed i risultati con smoothness (e,f). In alto sono mostrate anche le rispettive immagini rgb di partenza.

È possibile vedere come l'utilizzo del termine di smoothness, già dopo sole due epoche, risulta in un netto miglioramento nella precisione dei contorni degli oggetti nelle immagini predette. Si può notare anche un leggero miglioramento della predizione del range di distanze nella scena. Nei risultati con smoothness, infatti, gli oggetti più vicini nella scena presentano una gradazione di grigio più simile a quella della label rispetto ai corrispettivi senza smoothness.

Alla luce di questi risultati, è stato dunque deciso di procedere al training definitivo della rete sul dataset sintetico utilizzando anche il termine di smoothness. La configurazione del training è stata la stessa dei test effettuati precedentemente, dunque learning rate iniziale pari a 0,01, batch-size pari a 6 e Berhu loss. È stato impostato il training per la durata di 30 epoche, corrispondenti a circa 107000 step. È stato inoltre attivato il decadimento del gradiente ogni 6000 step di esecuzione. Il training è stato eseguito in più momenti mediante l'utilizzo di checkpoint, ed ha impiegato complessivamente circa 17 ore su GPU nVidia Titan X. In figura 4.4 sono mostrati i grafici di andamento della loss e dell'errore di predizione nell'arco dell'intero training.

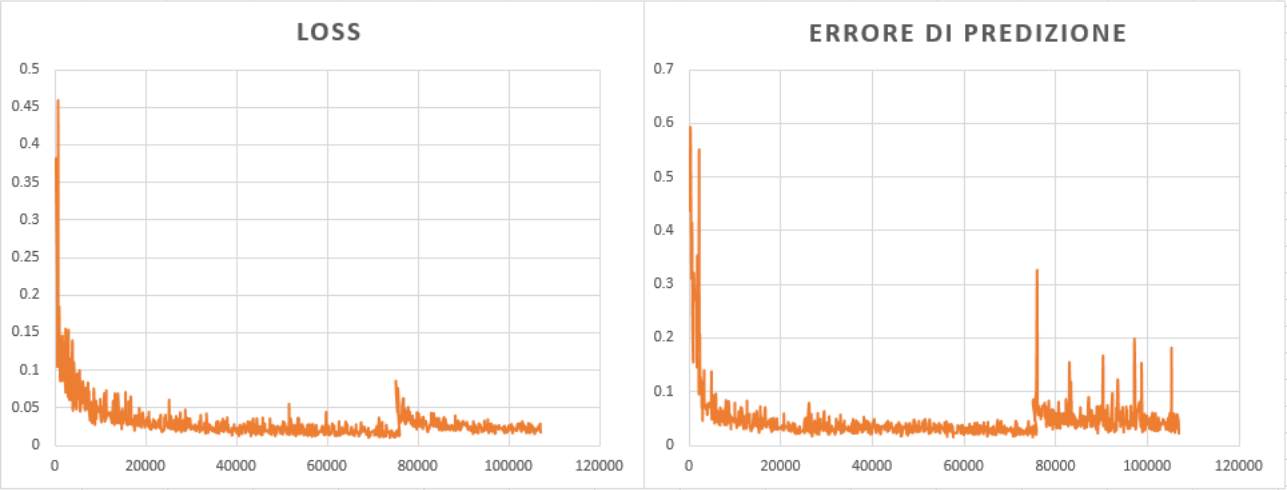

Figura 4.4: andamento della loss e dell'errore di predizione durante il training definitivo della rete.

Il leggero incremento dei valori che si verifica poco prima dello step 80000 è dovuta al fatto che, al momento dell'esecuzione del training, il rendering del dataset sintetico non era ancora stato finito completamente. Intorno allo step 75000 il rendering è stato completato, e dunque sono stati inseriti nel dataset altri esempi, che hanno dapprima causato una leggera destabilizzazione della

rete, ma poi sono stati assorbiti senza problemi, e la loss e l'errore di predizione sono tornati al loro andamento discendente. Prendendo una media sugli ultimi 2000 step, la rete è stata in grado di arrivare ad un valore minimo pari a circa 2cm per la loss e circa 4.9cm per l'errore di predizione. In figura 4.5 e possibile constatare anche visivamente la qualità ottenuta per le predizioni, confrontate con le predizioni della rete prima di effettuare il training.

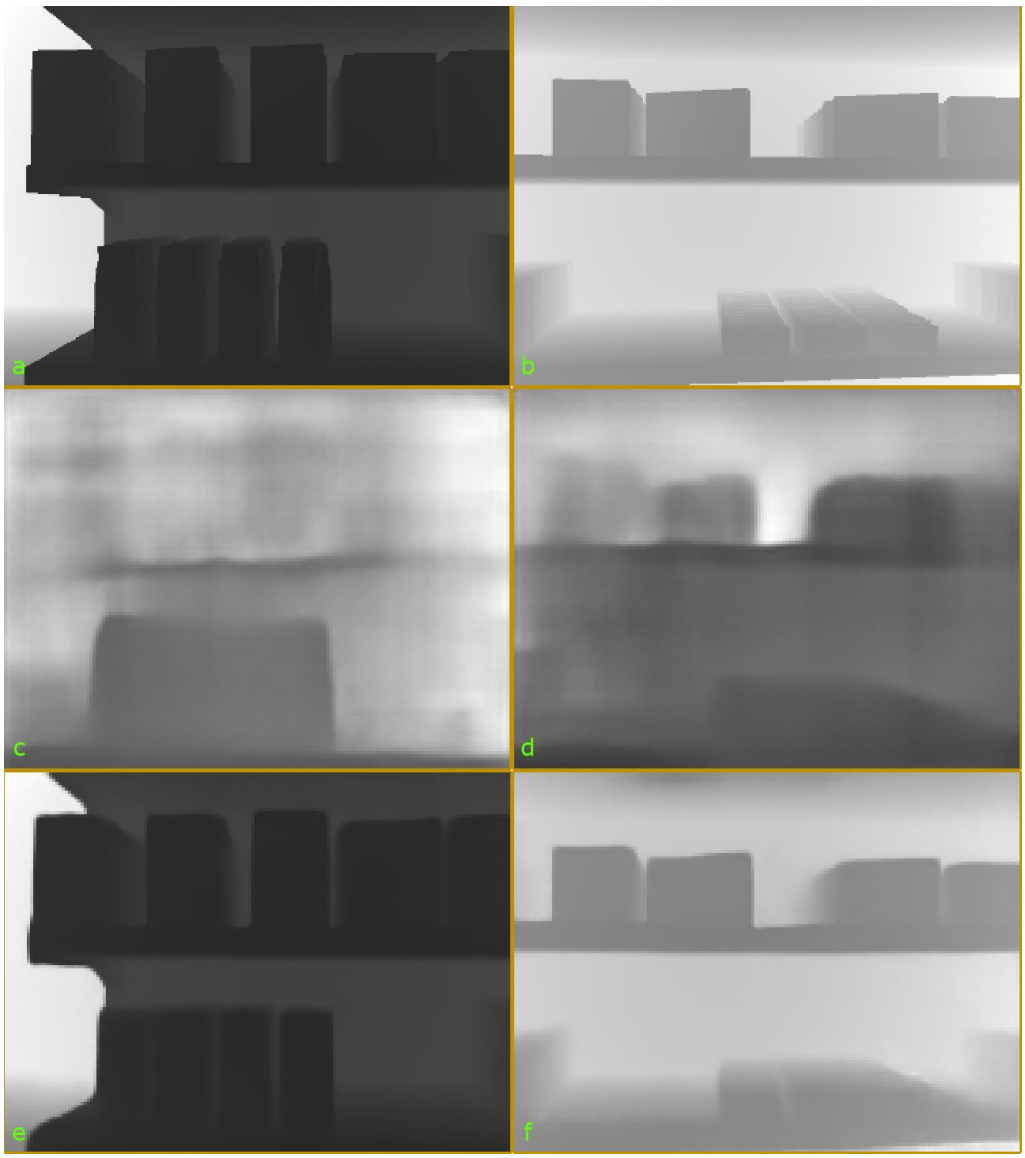

Figura 4.5: confronto tra label (a,b), predizioni fatte prima del training (c,d) e predizioni fatte al termine del training definitivo (e,f). Le immagini di partenza sono le stesse di figura 4.3.

È possibile subito notare come le predizioni prima del training siano largamente errate, sia per quanto riguarda i contorni dei singoli oggetti, sia per

quanto riguarda il range delle distanze nella scena. In basso, si può vedere come il training effettuato abbia permesso di ottenere predizioni molto accurate, a conferma dei risultati numerici illustrati precedentemente.

4.3.2 Addestramento sul dataset reale

Alla luce degli ottimi risultati ottenuti con il dataset sintetico, il passo successivo è stato quello di effettuare un finetuning della rete utilizzando il dataset reale, in modo da ottenere la rete definitiva da testare sul campo per il task dell'individuazione dei buchi a scaffale.

A questo fine, è stato effettuato un training sul dataset reale per 90 epoche, corrispondenti a circa 70000 step su questo dataset. I parametri utilizzati sono stati gli stessi del training effettuato sul dataset sintetico. I risultati ottenuti per la loss e l'errore di predizione sono mostrati nei grafici in figura 4.6.

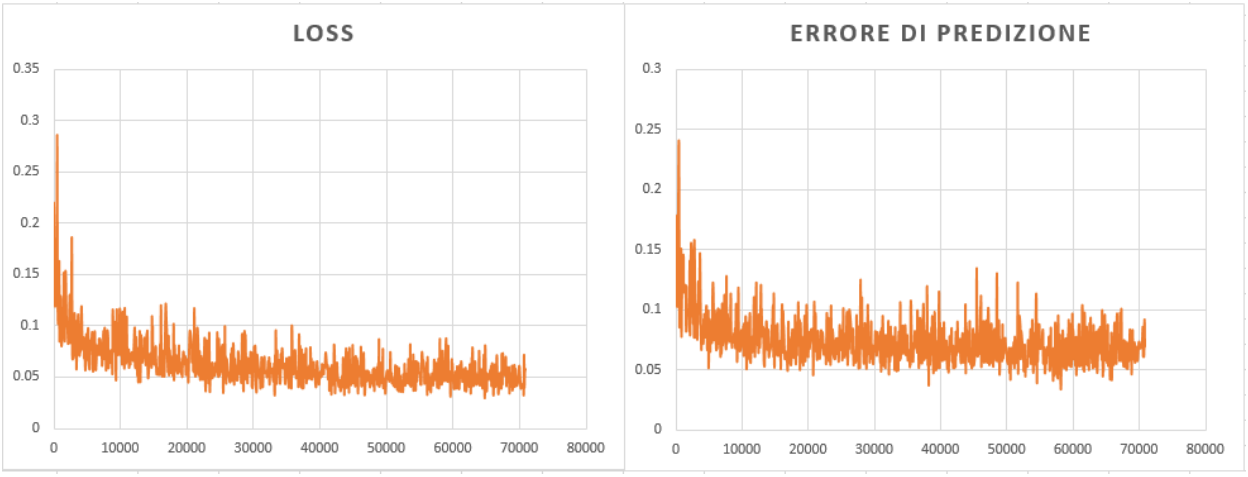

Figura 4.6: andamento della loss e dell'errore di predizione durante il finetuning sul dataset reale.

Si può notare come, intorno allo step 45000, l'andamento medio diventi pressoché costante, attestandosi su un valore medio di circa 5cm per la loss e circa 6,5cm per l'errore di predizione, considerando gli ultimi 2000 step. In figura 4.7 è mostrato un confronto tra le label, le predizioni prima del finetuning e le predizioni dopo il finetuning.

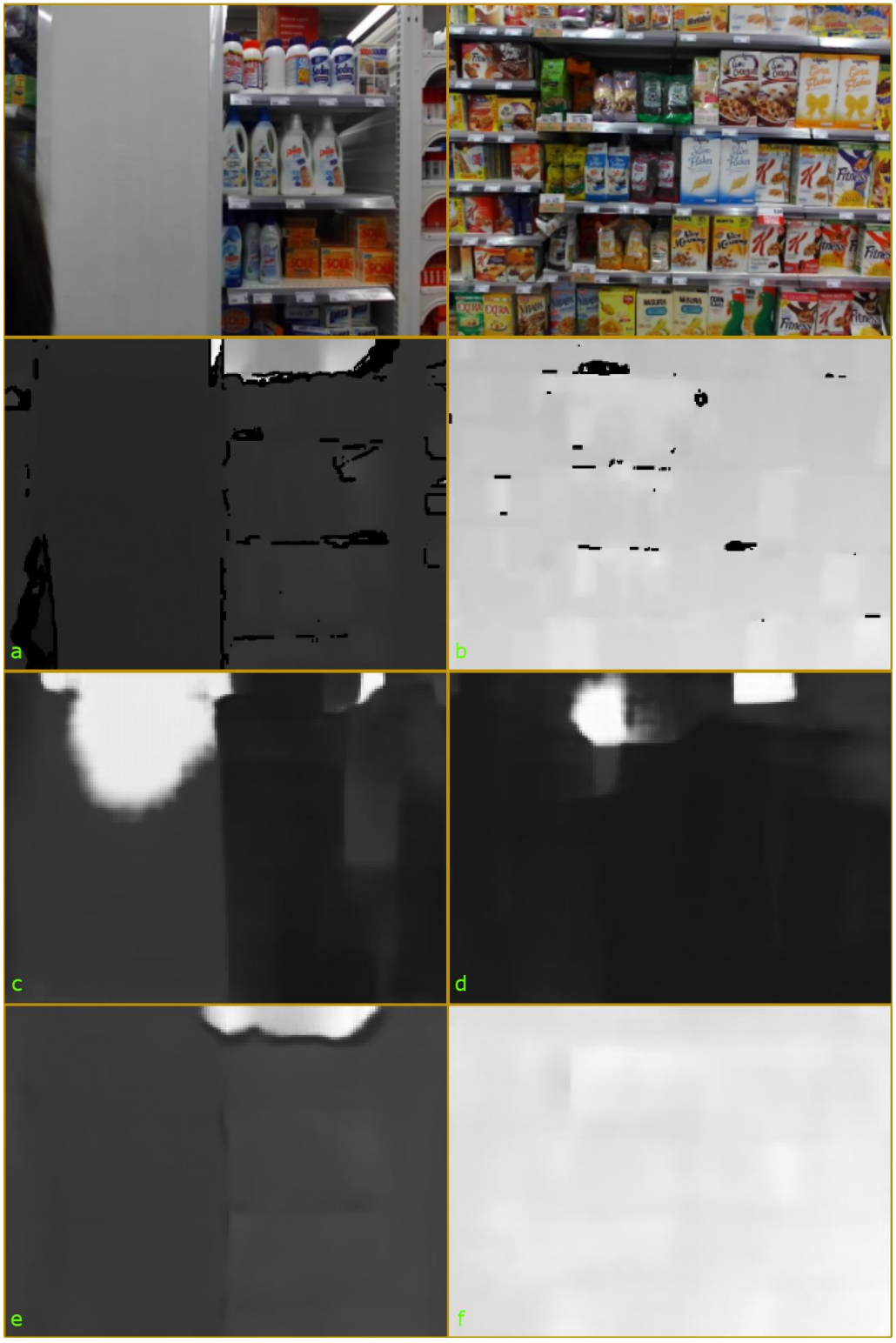

Figura 4.7: confronto tra label (a,b), predizioni prima del finetuning (c,d) e predizioni dopo il finetuning sul dataset reale (e,f)

È facile notare come le predizioni precedenti al finetuning sbaglino completamente il range delle distanze nella scena ed anche l'individuazione dei

contorni degli oggetti. Come si vede in basso, il finetuning ha permesso di risolvere in maniera molto buona questi problemi. Inoltre, grazie al filtraggio dei pixel di errore nelle label (rappresentati dalle macchie nere nelle immagini in prima fila) che è stato illustrato nel paragrafo sull'implementazione del training, la rete riesce a predire delle depth map di precisione pari a quelle prodotte con l'algoritmo stereo, ma col vantaggio di non avere errori grossolani.

4.3.3 Test sui buchi a scaffale

Alla luce degli ottimi risultati raggiunti dopo il finetuning sul dataset reale, l'ultimo passo del progetto è stata la definizione di una metrica di confronto tra le nuove predizioni e le depth map stereo.

A causa dei numerosi errori presenti in queste ultime, utilizzare l'errore di predizione come metrica non avrebbe dato risultati affidabili. Per questo motivo è stato deciso di testare la qualità delle nuove predizioni confrontando le performance derivanti dall'utilizzo delle predizioni e delle depth map stereo per un task di computer vision basato sulla profondità della scena. Nello specifico, il task selezionato consiste dell'individuazione dei buchi a scaffale, ovvero parti dello scaffale completamente prive di prodotti.

Per il confronto è stato utilizzato uno script python che, attraverso un algoritmo di thresholding sui valori di profondità nelle depth map, estrapola dei blob che potenzialmente possono rappresentare dei buchi a scaffale. Questo algoritmo non si presta certamente per una individuazione precisa dei buchi, ma rappresenta un ottimo strumento per confrontare le prestazioni tra l'utilizzo delle depth map predette e l'utilizzo delle depth map prodotte dall'algoritmo stereo.

Nello specifico, una volta individuati i potenziali buchi a scaffale nella depth map in esame, lo script calcola alcune metriche a partire dai risultati ottenuti e dalle coordinate vere delle bounding box dei buchi contenute in un file di testo. Ognuna di queste metriche viene calcolata più di una volta in base a varie soglie di un termine detto **Intersection Over Union** (IOU). Questo termine rappresenta il rapporto tra l'area dell'intersezione tra due bounding box e la

somma delle aree delle bounding box stesse. Rappresenta una misura per valutare la precisione di allineamento tra le due bounding box. Il passo preliminare al calcolo di ognuna delle metriche consiste dunque nel recuperare tutte le bounding box predette dall'algoritmo, e per ognuna di esse trovare la bounding box vera che presenta l'intersezione maggiore, se esistente. A questo punto, sulla coppia di bounding box viene calcolato il valore della IOU ed in base al risultato la relativa bounding box predetta viene considerata buona oppure scartata, in base ad un certo valore di soglia per la IOU.

Si ripete questa operazione per diversi valori di soglia della IOU, ottenendo diversi insiemi di bounding box valide. In seguito, per ognuno di questi insiemi, vengono calcolate tre metriche: la **precision**, il **recall** e **l'fmeasure**. Ai fini del confronto, le metriche significative sono quelle relative agli insiemi con IOU > 0.0 e IOU > 0.5. Nel primo caso, le metriche sono calcolate su un insieme di bounding box predette che individua bene la posizione del buco, ma non le sue dimensioni. Nel secondo caso, invece, le bounding box predette su cui vengono calcolate le metriche rappresentano una buona predizione sia della posizione che delle dimensioni del buco.

Tornando alle metriche, la precision rappresenta la percentuale dei buchi predetti che effettivamente corrisponde ad un vero buco nell'immagine. La recall indica invece quanti dei buchi effettivamente presenti nell'immagine sono stati individuati dalle predizioni. L'ultima metrica, la fmeasure, rappresenta infine una media armonica tra precision e recall.
Al fine di effettuare un confronto significativo relativo ad una misura di media, mediante lo script appositamente implementato per la predizione di filmati, sono state predette le depth map per i frame di 26 diversi filmati catturati in un supermercato. Una volta ottenute le predizioni, per tutti i frame sono state calcolate le metriche descritte precedentemente, ed in seguito ne è stato calcolato un valore medio rispetto a tutti i frame di tutti i filmati. La stessa operazione è stata poi effettuata utilizzando, al posto delle predizioni, le depth map prodotte dall'algoritmo stereo.

In figura 4.8 sono mostrati i risultati relativi a IOU > 0.0 (sinistra) ed IOU > 0.5 (destra).

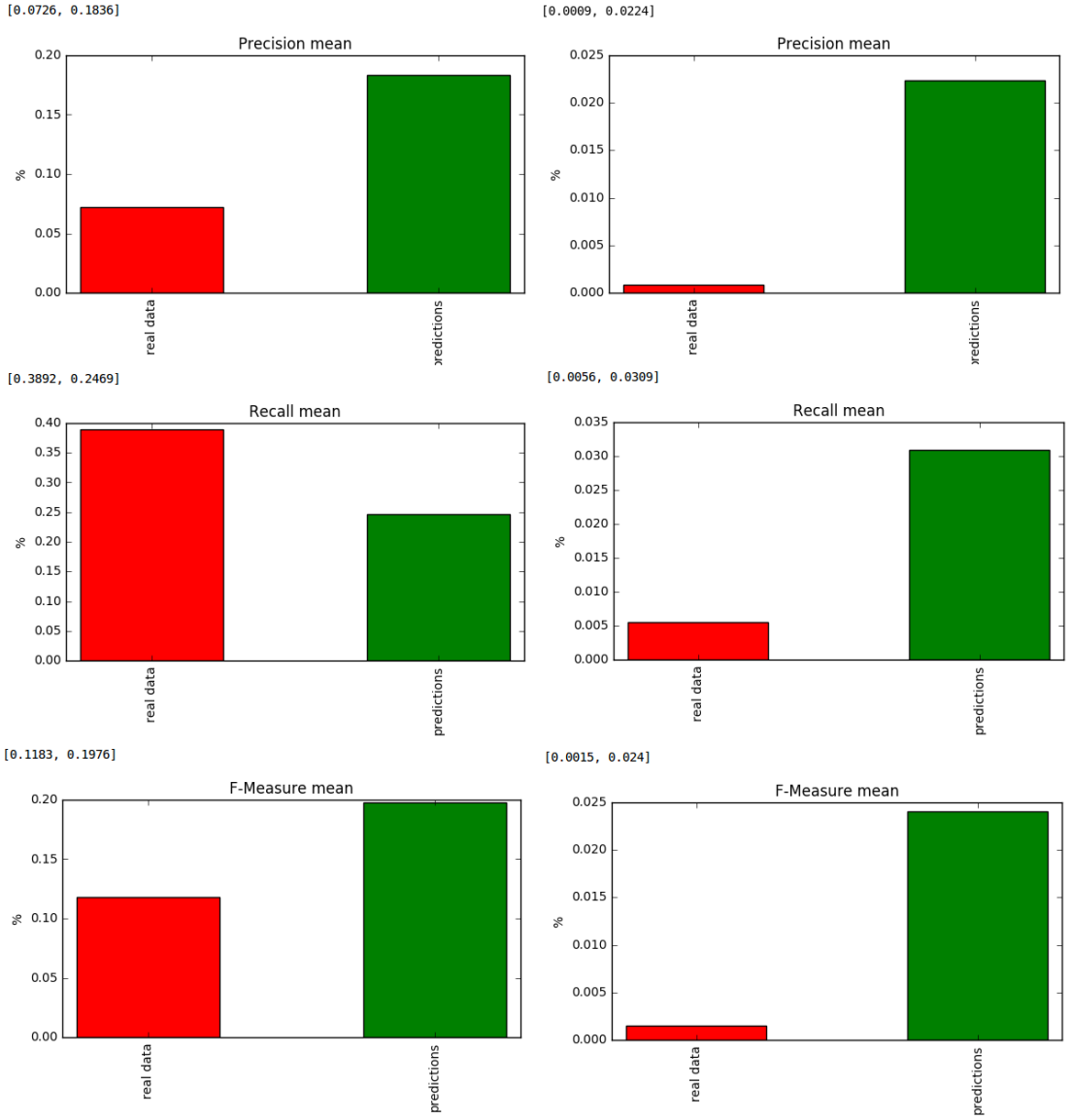

Figura 4.8: confronto tra i risultati relativi a IOU > 0.0 (sinistra) e a IOU > 0.5 (destra).

Ad IOU > 0.0 si può notare un miglioramento nel caso della precision, ed un leggero peggioramento nella recall. La fmeasure mostra comunque una tendenza generale al miglioramento, anche se di piccola entità. Con IOU > 0.0 si tratta comunque di un confronto parziale, poiché si prende in considerazione solo la precisione di localizzazione dei buchi, e non la precisione della ricostruzione delle bounding box relative.

Ad IOU > 0.5 si cominciano ad apprezzare, invece, risultati nettamente migliori nel caso delle predizioni. Nel caso della precision, si arriva ad un miglioramento pari a circa il 2400% rispetto all'utilizzo delle depth stereo,

mentre per la recall ci si aggira intorno ad un 450%. Infine, la fmeasure, che rappresenta un andamento medio rispetto a precision e recall, fa segnare un miglioramento di circa il 1600%. Questi risultati sono molto più importanti rispetto a quelli con IOU > 0.0, poiché tengono in considerazione sia la precisione di individuazione dei buchi sia la precisione di ricostruzione delle bounding box.

I risultati mostrati rappresentano la prova che il lavoro svolto ha effettivamente portato ad un netto miglioramento nel task dell'individuazione dei buchi da scaffale, e questo sancisce dunque il raggiungimento degli obiettivi che ci si era posti per il progetto.

Capitolo 5
Conclusioni

In questa tesi ci si è posto l'obiettivo di produrre un sistema in grado di stimare la profondità di una scena in maniera comparabile rispetto agli attuali algoritmi di stereo vision.

Come si è visto nel capitolo 3, il sistema prodotto fa uso dell'approccio di programmazione del machine learning, e nello specifico delle reti neurali. Al fine di addestrare questo sistema, come è stato illustrato nel capitolo 2, è stato sviluppato un nuovo dataset di immagini sintetiche mediante tecniche di computer graphics.

I risultati riportati nel capitolo 4 hanno mostrato come il dataset creato si è dimostrato molto valido per l'addestramento della rete neurale, e che il nuovo termine di smoothness introdotto ha effettivamente apportato benefici tangibili sulla qualità delle depth map predette. Il test finale sulle immagini reali ha infine sancito il raggiungimento dell'obiettivo che ci si era posti per il progetto.

Seppure i risultati ottenuti sono risultati molto buoni, il margine di miglioramento è ampio, ed una futura ottimizzazione dei parametri di training e di altri particolari della rete potrebbe portare a risultati ancora migliori.

Bibliografia

1. **Microsoft.** Kinect per Windows. https://developer.microsoft.com/it-it/windows/kinect.

2. **Michell, T.** *Machine Learning.* s.l. : McGraw Hill, 1997.

3. **Turing, Alan.** Computing Machinery and Intelligence. s.l. : Mind, 1950.

4. **Schmidhuber, Jürgen.** Learning complex, extended sequences using the principle of history compression. 1992. p. 234-242.

5. **Kaiming, He, et al.** Deep Residual Learning for Image Recognition. 10 Dicembre 2015. arXiv:1512.03385v1.

6. **LeCun, Yann, et al.** Gradient-Based Learning Applied to Document Recognition. s.l. : IEEE, Novembre 1998.

7. **Krizhevsky, Alex.** Learning Multiple Layers of Features from Tiny Images. Toronto : s.n., 8 Aprile 2009.

8. **Nair, Vinod e Hinton, Geoffrey E.** Rectified Linear Units Improve Restricted Boltzmann Machines. Toronto : Department of Computer Science, University of Toronto, 2010, ICML.

9. **Fischer, Philipp, et al.** FlowNet: Learning Optical Flow with Convolutional Networks. Friburgo : s.n., 2015. arXiv:1504.06852v2.

10. **Mayer, Nikolaus, et al.** A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Flow Estimation. Friburgo : Computer Vision Foundation, CVPR 2016. arXiv:1512.02134v1.

11. **Blender.** https://www.blender.org/.

12. **Unity 3D.** https://unity3d.com/.

13. **Autodesk.** http://www.autodesk.it/products/3ds-max/overview.

14. **Industrial Light & Magic.** OpenEXR file format. http://openexr.com/.

15. **Silberman, Nathan, et al.** Indoor Segmentation and Support Inference from RGBD Images. European Conference on Computer Vision : s.n., 2012.

16. **Saxena, Ashutosh, Chung, Sung H. and Ng, Andrew Y.** Learning Depth from Single Monocular Images. NIPS 2005.

17. **Saxena, Ashutosh, Sun, Min and Ng, Andrew Y.** Make3D: Learning 3D Scene Structure from a Single Still Image. IEEE Transactions of Pattern Analysis and Machine Intelligence (PAMI), vol. 30, no. 5, pp 824-840, 2009..

18. **Krizhevsky, Alex, Sutskever, Ilya e Hinton, Geoffrey E.** *ImageNet Classification with Deep Convolutional Neural Networks.* Toronto : s.n., 2012.

19. **Fei-Fei, Li, et al.** ImageNet Large Scale Visual Recognition Challenge. IJCV : s.n., 2015.

20. **Karen, Simonyan e Andrew, Zisserman.** Very Deep Convolutional Networks for Large-Scale Image Recognition. Oxford : s.n., 10 Aprile 2015. arXiv:1409.1556v6.

21. **Eigen, David e Fergus, Rob.** Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture. 17 Dicembre 2015. arXiv:1411.4734v4.

22. **Laina, Iro, et al.** Deeper Depth Prediction with Fully Convolutional Residual Networks. 19 Settembre 2016. arXiv:1606.00373v2.

23. **Ioffe, Sergey e Szegedy, Christian.** Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2 Marzo 2015. arXiv:1502.03167v3.

24. **MatConvNet.** http://www.vlfeat.org/matconvnet/.

25. **Tensorflow.** https://www.tensorflow.org/.

26. **Dasgupta, Saumitro.** Caffe-TensorFlow conversion tool. https://github.com/ethereon/caffe-tensorflow.

27. **Jia, Yangqing, et al.** Caffe: Convolutional Architecture for Fast Feature Embedding. 2014. arXiv:1408.5093.

28. **Zwald, L. e Lambert-Lacroix, S.** The berhu penalty and the grouped effect. arXiv preprint arXiv:1207.6868, 2012.