UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

IMPLEMENTAZIONE DI UNA CNN PER LA STIMA DELLA PROFONDITA' DI IMMAGINI PER APPLICAZIONI DI GUIDA AUTONOMA SU PIATTAFORMA EMBEDDED

Implementation of a CNN to estimate the depth of images for autonomous driving application on embedded platform

Relatore:
Chiar.mo Prof.
Claudio Turchetti

Presentata da:
Riccardo Rossi

Correlatore:
Dott.ssa.
Laura Falaschetti

A.A. 2020/2021

INDICE

Introduzione ... *4*

CAPITOLO 1 ... *6*

RETI NEURALI ... *6*

1.1 Cosa sono le reti neurali? ... **6**

1.2 Come funzionano le reti neurali? ... **7**

1.3 Background Propagation ... **10**

CAPITOLO 2 ... *11*

RETI NEURALI CONVOLUZIONALI (CNN) ... *11*

2.1 Layer Convoluzionale ... **12**

2.2 Livello ReLu ... **14**

ReLu (Rectified Linear Unit) ... 15

2.3 Pooling Layer ... **16**

Dropout ... 17

2.4 Fully Connected Layer ... **18**

2.5 Funzioni del modello sviluppato ... **19**

Loss Function ... 19

CAPITOLO 3 ... *22*

SVILUPPO DEL MODELLO ... *22*

3.1 Import dei moduli ... **23**

Installazione di TensorfFlow e Keras ... 23

Importo i moduli ... 23

Importo seme e shuffle ... 24

3.2 Preparazione del dataset ... **25**

Caricamento del datset ... 25

Reshape dei dati ... 26

Load dei dati ... 27

Dati di configurazione ... 27

3.3 Loss function ... **27**

3.4 Modello ... **29**

Compile del modello ... 30

Salvataggio su Outdir ... 30

Carico e preprocesso i dati ... 31

Train del modello ... 31

3.5 Valutazione del modello e riduzione size ... **32**

Stima della size ... 33

Salvataggio del modello ... 33

Quantizzazione del modello ... 34

CONCLUSIONI ... *35*

BIBLIOGRAFIA E SITOGRAFIA ... *36*

Introduzione

Negli ultimi anni si è sentito spesso parlare di Intelligenza Artificiale, di come i computer siano in grado di prendere decisioni autonomamente e di come questa nuova frontiera della scienza possa rappresentare un problema in futuro.
Facciamo quindi un po' di chiarezza: cosa si intende per Intelligenza Artificiale?
In termini tecnici, l'Intelligenza Artificiale è un ramo dell'informatica che permette la programmazione e progettazione di sistemi sia hardware che software che permettono di dotare le macchine di determinate caratteristiche che vengono considerate tipicamente umane quali, ad esempio, le percezioni visive, spazio-temporali e decisionali.
Alla base delle problematiche legate allo sviluppo di sistemi e programmi di **Intelligenza Artificiale** vi sono tre parametri che rappresentano i cardini del comportamento umano:

1) Una **conoscenza non sterile**
2) Una coscienza che permetta di prendere **decisioni non solo secondo la logica**
3) L'**abilità di risolvere problemi in maniera differente** anche a seconda dei contesti nei quali ci si trova.

L'uso delle **reti neurali** e di algoritmi in grado di riprodurre ragionamenti tipici degli esseri umani nelle differenti situazioni, hanno permesso ai sistemi intelligenti di migliorare sempre di più le diverse capacità di comportamento. Per poter realizzare ciò, la ricerca si è concentrata su **algoritmi** che potessero imitare i diversi comportamenti a seconda degli stimoli ambientali. Tali algoritmi complessi, inseriti all'interno di sistemi intelligenti, sono quindi in grado di **'prendere decisioni'** ossia di effettuare scelte a seconda dei contesti in cui sono inseriti. Nel caso degli algoritmi connessi ai sistemi intelligenti dei veicoli, ad esempio, un'automobile senza conducente può decidere, in caso di pericolo, se sterzare o frenare a seconda della situazione, ossia a seconda che le informazioni inviate dai vari sensori permettano di calcolare una maggiore percentuale di sicurezza per il conducente e i passeggeri con una frenata o con una sterzata.
Le decisioni di ogni tipo, sia quelle prese da un'auto senza pilota che da altri sistemi di Intelligenza Artificiale, sono prese, come già specificato, grazie alla realizzazione di determinati algoritmi, che permettono di definire una **conoscenza di base e una conoscenza allargata**, ossia creata tramite l'esperienza. Per realizzare algoritmi sempre più precisi e complessi, è sorta un vero e proprio settore specifico, definito **rappresentazione della conoscenza**, che studia tutte le possibilità di ragionamento dell'uomo e, soprattutto, tutte le possibilità di rendere tale conoscenza comprensibile alle macchine tramite un linguaggio e dei comandi sempre più precisi e dettagliati.

Nel caso specifico, all'interno di questa tesi è stata implementata una rete neurale convoluzionale in grado di prendere in ingresso delle immagini e restituire in uscita la profondità degli oggetti contenuti nella stessa.

Per lo sviluppo di questo progetto sono stati necessari ii seguenti passaggi nell'ambiente di sviluppo Google Colab:

1) Preparazione del Dataset NYU Depth
2) Implementazione della rete neurale convoluzionale
3) Test sul modello sviluppato

Per poter analizzare il progetto sviluppato è necessaria una breve trattazione teorica sulle reti neurali semplici e convoluzionali, in modo da facilitare la comprensione del codice scritto in Python utilizzando le librerie TensorFlow e Keras.

CAPITOLO 1

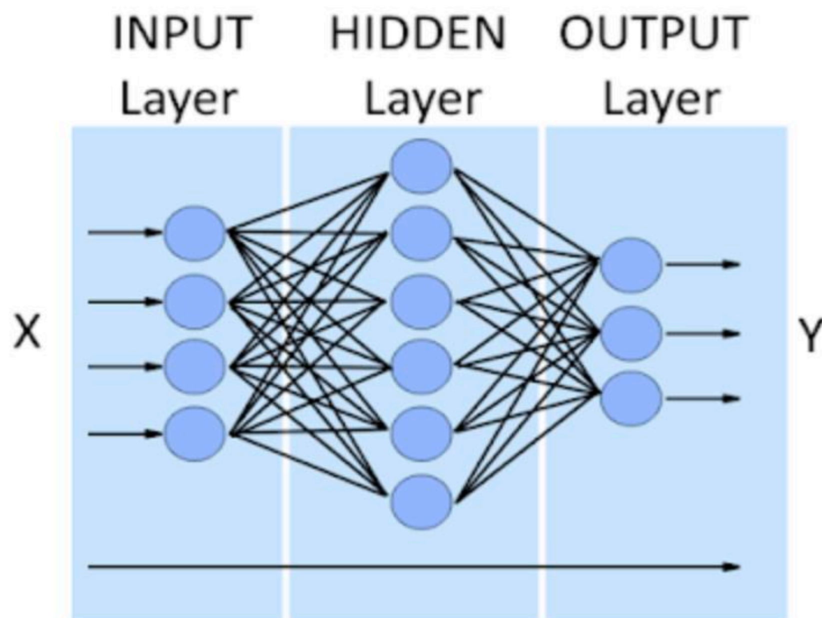
RETI NEURALI

1.1 Cosa sono le reti neurali?

Le reti neurali, note anche come reti neurali artificiali (o ANN, artificial neural network) sono un modello computazionale composto di "neuroni" artificiali, aventi come obiettivo quello di imitare il modo in cui i neuroni biologici si inviano segnali. Le ANN sono composte da diversi livelli di nodi che contengono:

1) Un livello di Input (input layer)
2) Uno o più livelli nascosti (hidden layers)
3) Un livello di output (output layer)

Il primo è quello che ha il compito di ricevere ed elaborare segnali e dati provenienti dall'esterno, il secondo strato ha in carico il processo di elaborazione vero e proprio, mentre lo strato di uscita raccoglie i risultati dell'elaborazione dello strato nascosto, i quali vengono poi adattati alle richieste del successivo livello della rete neurale.
Il processo prevede, in sostanza, che quanto viene prodotto "in uscita" dal primo strato di neuroni artificiali faccia poi da input allo strato successivo e così via, in un ciclo continuo.
Ogni nodo (o neurone artificiale) è in grado di connettersi ad un altro nodo e possiede un certo peso e soglia. Questo significa che solo se l'output di un qualsiasi nodo è superiore al valore di soglia, quel neurone verrà attivato e questo permetterebbe all'informazione di continuare a viaggiare lungo la rete.
Se invece tale valore di soglia dovesse essere insufficiente per l'attivazione del nodo, l'informazione non riuscirebbe ad essere trasmessa al nodo successivo.



Le reti neurali fanno affidamento sui dati di addestramento per imparare e migliorare ricorsivamente la loro accuratezza nel tempo, diventando così più precise con l'aumentare del numero ed eterogeneità dei campioni di training.

1.2 Come funzionano le reti neurali?

Possiamo pensare ad ogni nodo come un proprio modello di regressione lineare, con dati di input, soglia, pesi e un output.

Per descrivere il funzionamento della rete, possiamo fare riferimento alla formula della somma pesata degli ingressi, corretta in seguito da una soglia.

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias$$

In base a quanto detto prima, un neurone veniva attivato solo se la somma pesata degli input fosse stata superiore ad un dato valore di soglia.
Volendo tradurre in formule questa condizione, si otterrebbe:

$$\text{output} = f(x) = \begin{cases} 1 \text{ if } \sum w_1 x_1 + b \geq 0 \\ 0 \text{ if } \sum w_1 x_1 + b < 0 \end{cases}$$

I pesi aiutano a determinare l'importanza di una qualsiasi data variabile, con quelli più grandi che contribuiscono in modo più significativo all'output rispetto agli altri input.
Il bias serve per definire quanto facilmente il neurone riesca ad attivarsi. L'output di un nodo diventa quindi l'input del nodo successivo.
Questo processo di passaggio dei dati da un livello a quello successivo definisce questa rete neurale come una rete feedforward.

Possiamo applicare questo concetto ad un esempio più pratico, come ad esempio andare al mare o no (Sì: 1, No: 0). La decisione di andare o non andare è il nostro risultato previsto, o ŷ. Supponiamo che ci siano tre fattori che influenzano il processo decisionale:

1. Ci sono meno di 35 gradi? (Sì: 1, No: 0)
2. Il mio gruppo di amici è presente? (Sì: 1, No: 0)
3. Piove? (Sì: 0, No: 1)

Proseguiamo assegnando i seguenti input:

- X1 = 1, dal momento che ci sono meno di 30 gradi
- X2 = 0, visto che è presente il mio gruppo di amici
- X3 = 1, visto che non sta piovendo al momento

Ora, dobbiamo assegnare dei pesi per determinare l'importanza. Dei pesi maggiori indicano che delle specifiche variabili hanno un'importanza maggiore per la decisione o il risultato.

- W1 = 5, dal momento che il caldo eccessivo è pericoloso
- W2 = 2, visto che posso starmene all'ombra a leggere un libro
- W3 = 4, visto che con la pioggia non si potrebbe fare il bagno

Infine, presupporremo anche un valore di soglia di 3, che si tradurrebbe in un valore di distorsione di -3. Con tutti i vari input, possiamo iniziare a inserire i valori nella formula per ottenere l'output desiderato.

Ŷ = (1*5) + (0*2) + (1*4) – 3 = 6

Se utilizziamo la funzione di attivazione dall'inizio di questa sezione, possiamo determinare che l'output di questo nodo sarebbe 1, poiché 6 è maggiore di 0. In questo caso, andresti al mare.
Se però modificassimo i valori di soglia o i pesi, i risultati potrebbero essere diversi. Quando osserviamo una decisione, come nell'esempio precedente, possiamo vedere in che modo una rete neurale potrebbe prendere delle decisioni sempre più complesse a seconda dell'output delle decisioni o dei livelli precedenti.
In casi più generici e più complessi del precedente, dove la decisione della rete non è binaria e i dati in ingresso sono decine di migliaia o più, non si parlerà in termini di attivo o spento per il neurone, ma di "quanto" è attivo il neurone, quindi le funzioni di attivazione daranno in uscita numeri reali che più sono grandi più il neurone in questione sarà attivo.

Prima tra tutte è la sigmoid function:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

Essendo la z = ∑wixi + bias, ottengo:

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

1.3 Background Propagation

La retro-propagazione è uno dei fondamenti dell’addestramento delle reti neurali. Questa tecnica consiste nel ricalibrare i pesi dati ad ogni input con l’obiettivo di minimizzare i tassi di perdita ottenuti dalla funzione di perdita (LOSS FUNCTION)
.
Nella teoria della decisione, una funzione di costo (LOSS FUNCTION o COST FUNCTION) è una funzione che mappa un evento, o valori di una o più variabili, su un numero reale, intuitivamente rappresenta un "costo" associato all'evento.
Un problema di ottimizzazione cerca di minimizzare una funzione di costo.

Nelle reti neurali, la background propagation rappresenta la capacità di correggere i pesi della rete in relazione al tasso di perdita riportato dalla Loss Function nelle iterazioni precedenti.
Minimizzare l’errore rende la rete più precisa e affidabile.

Nel corso di questa tesi parleremo in maniera più approfondita di come utilizzare una funzione costo (o Loss Function) e di come questi siano strumenti fondamentali nello sviluppo delle reti neurali.

Non tutte le funzioni costo sono uguali e hanno la stessa efficacia infatti: come vedremo meglio nel capitolo relativo alle CNN, la miglior funzione di costo è la Scale Invariant Error, che opera in maniera egregia nei problemi di stima della profondità.

Ci sono invece delle funzioni più adatte per quanto riguarda la classificazioni di numerosi elementi in output ma con un numero intero di label, come ad esempio la Sparse Categorical Crossentropy.

Nel prossimo capitolo andremo ad approfondire tutti gli aspetti più importanti legati alle reti neurali convoluzionali e successivamente metteremo mano al modello sviluppato

CAPITOLO 2

RETI NEURALI CONVOLUZIONALI (CNN)

Le reti neurali convoluzionali o (CNN dall'inglese convolutional neural network) sono uno degli algoritmi di Deep Learning più utilizzati e sono utili per la classificazione delle immagini. Questi algoritmi permettono al computer di riconoscere gli oggetti presenti all'interno di un'immagine e classificarli. C'è da specificare però, che queste reti neurali sono in grado di classificare degli oggetti che rientrano soltanto in alcune categorie, o classi.
Un esempio potrebbe essere cercare di analizzare un volto umano con una CNN che prende in ingresso un dataset di immagini di automobili: essa non capirebbe le forme e i pattern necessari per il riconoscimento facciale.
Una CNN è in grado di prendere un'immagine in input e assegnare dei pesi a diverse parti dell'immagine, in modo da separarne i pattern e velocizzarne l'apprendimento. Questo equivale ad etichettare (classificare) un'immagine in base a delle caratteristiche ottenute sezionandone le parti più importanti.
L'architettura di una CNN è simile a quella delle reti neurali artificiali:

1) Input layer
2) Hidden layer
3) Output layer

L'unica differenza tra le due reti è appunto l'operazione di convoluzione che avviene negli hidden layers.
All'interno di questo layer, sono presenti diversi livelli utili per la CNN:

- Livello Convoluzionale: è il livello principale della rete. Il suo obiettivo è quello di individuare schemi, come ad esempio curve, angoli e quadrati. Sono più di uno e ognuno di essi si concentra nella ricerca di queste caratteristiche nell'immagine iniziale. Maggiore è il numero e maggiore è la complessità della caratteristica che riescono ad individuare.

- Livello ReLu (Rectified Linear Unit) : in questo livello vengono posti a zero tutti i valori negativi dei livelli precedenti

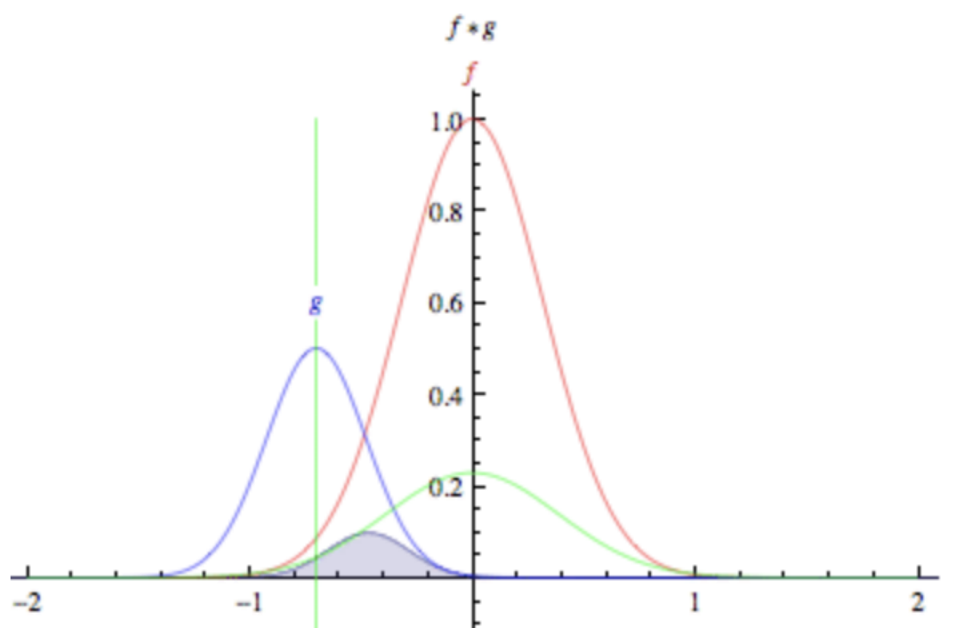
- Livello Pool: ci permette di identificare se nell'immagine in questione è presente la caratteristica che cercavamo

- Livello Fully Connected: connette tutti i neuroni precedenti per poter stabilire le classi secondo una probabilità.

Ora procederemo con la descrizione di ognuno di questi livelli, approfondendone i concetti chiave.

2.1 Layer Convoluzionale

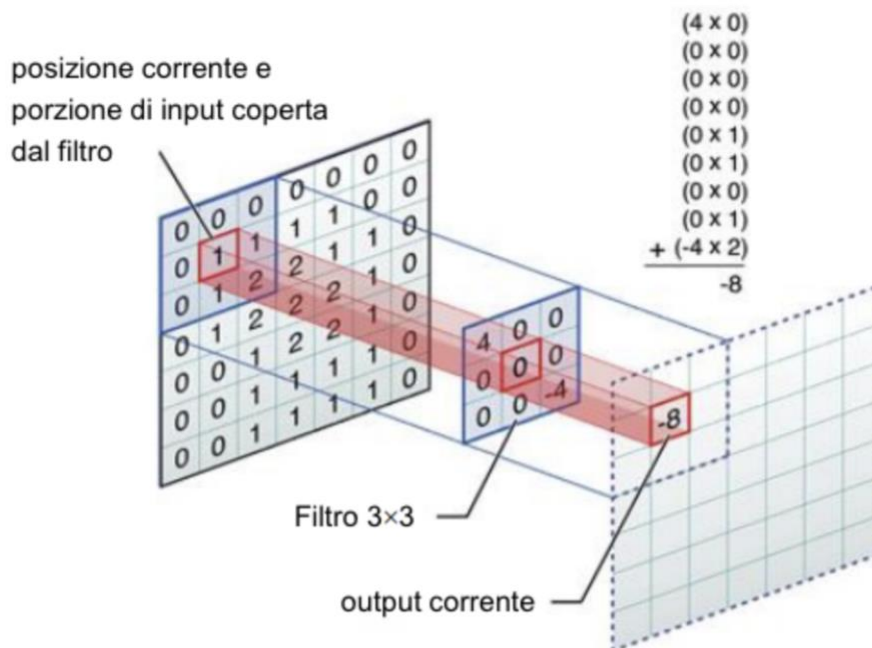
Per poter descrivere cosa accade nel layer convoluzionale, è bene partire con il concetto di convoluzione.
L'operazione di convoluzione consiste nel prodotto tra due funzioni, in cui una delle due viene fatta scorrere sopra l'altra.
La funzione in blu scorre sopra la funzione in rosso, ottenendo la funzione in verde come risultato



La funzione in rosso rappresenta la base di partenza e possiamo vederla come l'immagine di input che viene fornita alla rete, mentre la funzione in blu può essere visto come un "filtro" che viene applicato.

Nell'elaborazione delle immagini, utilizziamo delle matrici di convoluzione, chiamate Kernel o maschera.
Ma come funziona il livello di convoluzione? in pratica un filtro digitale (maschera) è fatta scorrere sulle diverse posizioni dell'immagine in input; per ogni posizione

viene generato un valore di output, eseguendo il prodotto scalare tra la maschera e la porzione dell'input coperta (entrambi trattati come vettori).
All'interno del layer, questa operazione avviene per poter identificare dei pattern di complessità sempre maggiore all'interno dell'immagine.
All'inizio, infatti, i pattern che la rete sarà in grado di riconoscere saranno molto semplici, come ad esempio solo le linee orizzontali oppure le circonferenze. Queste caratteristiche sono chiamate di basso livello, perché i pattern riconosciuti sono abbastanza elementari.
Man mano che avanziamo nell'addestramento, le caratteristiche che la rete riuscirà a riconoscere saranno sempre più complesse, fino ad arrivare a riconoscere una mano od un viso.



È molto importante che le matrici di convoluzioni (i filtri) siano di ordine dispari, dal momento che un aspetto fondamentale di questo processo è riuscire ad individuare il centro di tale matrice, per poi applicare il filtro correttamente.

Facciamo un esempio per comprendere meglio come funziona questo filtro.

Prendiamo una matrice A, la base, e una matrice B, che sarà il nostro filtro. Dobbiamo centrare la matrice B in modo da avere il suo centro in corrispondenza del pixel della matrice A su cui vogliamo lavorare.

Il valore di ciascun pixel della matrice A oggetto di elaborazione viene ricalcolato come la somma pesata dei prodotti di ciascun elemento della matrice kernel con il corrispondente pixel della matrice A sottostante.

Questo layer convoluzionale si basa su un sistema di pesi e bias, in cui i pesi sono gli elementi che compongono la matrice filtro. Lo strato convoluzionale è in grado di scorrere e filtrare l'immagine, facilitando il riconoscimento di pattern all'interno della stessa.

Grazie a queste caratteristiche, un campione può essere individuato all'interno di una scena anche se ruotato o spostato rispetto allo sfondo.

2.2 Livello ReLu

Dopo uno strato di convoluzione, solitamente l'uscita è soggetta ad una funzione di attivazione. Questa funzione giace all'interno dei neuroni e determina la loro attivazione o meno, per poi mandare il risultato in output. Possiamo vedere quindi la funzione di attivazione come un modo per mappare l'ingresso con l'uscita.

Ora la domanda è : cosa succede se non usiamo alcuna funzione di attivazione e permettiamo a un neurone di fornire la somma ponderata degli input così come gli output? Bene, in quel caso il calcolo sarà molto difficile poiché la somma ponderata dell'input non ha alcun intervallo e, a seconda dell'input, può assumere qualsiasi valore. Quindi un uso importante della funzione di attivazione è di mantenere l'uscita limitata a un intervallo particolare. Un altro uso della funzione di attivazione è l'aggiunta di non linearità nei dati. Scegliamo sempre funzioni non lineari come funzioni di attivazione.

La non linearità è fondamentale nelle funzioni di attivazione.
Se usiamo le funzioni di attivazione lineare in una rete neurale profonda, indipendentemente dalla profondità della nostra rete, sarà equivalente ad una semplice rete neurale lineare senza strati nascosti perché quelle funzioni di attivazione lineare possono essere combinate per formare un'altra singola funzione lineare. Quindi fondamentalmente tutta la nostra rete sarà ridotta ad un singolo neurone con quella funzione lineare combinata come sua funzione di attivazione e quel singolo neurone non sarà in grado di apprendere relazioni complesse nei dati. Poiché la maggior parte dei problemi del mondo reale sono molto complessi, abbiamo bisogno di funzioni di attivazione non lineari in una rete neurale. La rete neurale senza funzioni di attivazione non lineare sarà solo un semplice modello di regressione lineare. Tuttavia, nello strato finale della rete neurale, possiamo scegliere funzioni di attivazione lineare.

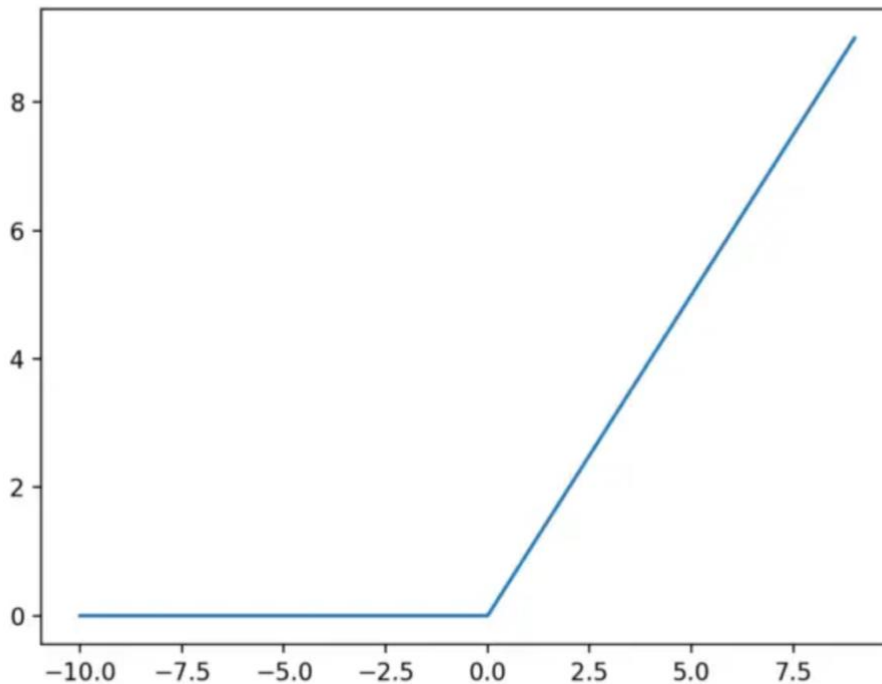
Andiamo a vedere le funzioni di attivazione utilizzate nel modello sviluppato.

ReLu (Rectified Linear Unit)

È forse la funzione di attivazione più utilizzata nelle reti neurali, chiamata anche raddrizzatore.

È definita come la parte positiva del suo argomento.

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



La ReLu è una funzione di attivazione molto facile da usare e dato che non attiva tutti i neuroni, velocizza il funzionamento della rete. D’altra parte, non è centrata sullo zero e disattiva alcuni neuroni per sempre se il loro valore è inferiore a zero.

Linear

La seconda funzione di attivazione utilizzata in questo modello è proprio quella lineare.
Questa funzione può essere a soglia, binaria: 0, 1.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Dove:

x = input
w = il peso moltiplicato ad ogni input
b = soglia di attivazione

nel caso ci siano più valori in input, si effettua il prodotto scalare tra x e w.

2.3 Pooling Layer

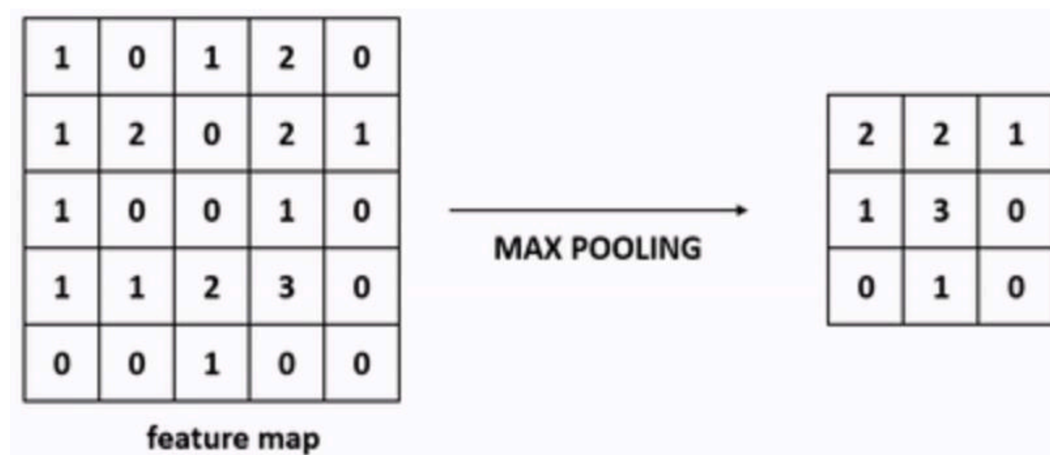
Successivamente ad un layer convoluzionale, vien posto un pooling layer.
Questo accade perché lo scopo del pooling è quello di ridurre le dimensioni dell'immagine dopo essere passata attraverso la convoluzione, che restituisce un oggetto più denso di quello che si potrebbe maneggiare.

Giusto per darci un'idea, un'immagine a colori di piccolissime dimensioni (32x32 pixel), porterebbe ad avere nel primo layer già 3072 nodi (32x32 pixel e 3 canali di colore), con 3072 connessioni ciascuno, più di 9 milioni in totale.
Un'immagine di dimensioni più ragionevoli, 1000x1000, porterebbe ad avere un totale di 10^12 connessioni! È quindi evidente che una soluzione di questo tipo non è accettabile.

Un livello di pooling esegue un'aggregazione delle informazioni nel volume di input, generando feature map di dimensione inferiore. Obiettivo è conferire invarianza rispetto a semplici trasformazioni dell'input mantenendo al tempo stesso le informazioni significative ai fini della discriminazione dei pattern.

Esistono due tipi di Pooling: Max Pooling ed Average Pooling.

Per spiegare il max pooling prendiamo ad esempio una matrice 5x5, cui viene applicata una matrice di pooling 2x2, che scorre sopra la matrice di partenza.
Di questa sotto-matrice noi prendiamo solo il valore massimo e lo utilizziamo per costruire la matrice di uscita, che sarà in questo caso una matrice 3x3.



discorso analogo per il Pooling Average, che al posto di prendere il valore massimo della sotto-matrice, calcola il valore medio di tutti i suoi elementi.

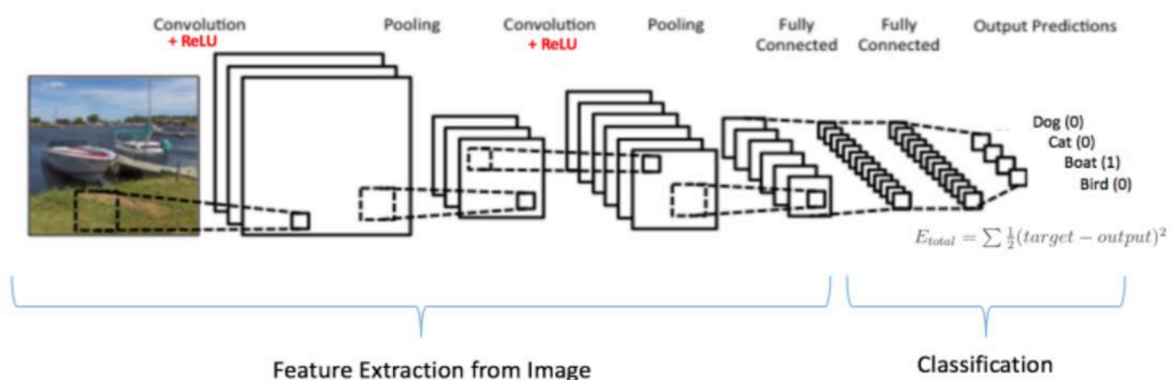
Se invece si volesse mantenere la dimensione dell'immagine di partenza, si potrebbe applicare lo zero-padding, ovvero l'aggiunta di righe e colonne di 0 nei bordi dell'immagine.
Logicamente il pooling svolgerà il suo lavoro di riduzione dell'informazione abbassando la risoluzione dell'immagine, ma manterrà le dimensioni originali.

Dropout

Prima di arrivare all'ultimo layer, quello fully-connected, si passa per un livello chiamato Dropout.
In questa fase, viene spenta una certa quantità di neuroni per ogni layer nascosto.
Questo ragionamento può sembrare inopportuno, ma serve per prevenire dei comportamenti difettosi da parte della rete, come l'overfitting.
L'overfitting si verifica quando la rete non "ragiona" sui dati che ha a disposizione (perché troppi), ma li impara a memoria.
Facendo così, le prestazioni potrebbero essere notevolmente inferiori e si preferisce spegnere una certa percentuale di neuroni al fine di evitare tale fenomeno.

2.4 Fully Connected Layer

Il livello FC (Fully connected) solitamente è l'ultimo di una rete neurale.
Questo livello prende un volume di input e genera un vettore di dimensione N, dove N è il numero di classi tra cui il programma deve scegliere. Ogni classe sarà associata alla probabilità che nell'immagine sia rappresentata quella classe. La somma delle probabilità è 1, quindi ogni classe avrà un numero compreso fra 0 e 1. Ad esempio, in un programma di classificazione delle cifre N sarà 10, poiché le cifre sono 10 (0,1,2,3,4,5,6,7,8,9). Ogni numero rappresenta la probabilità di una certa classe. Se il vettore risultante per un programma di classificazione di cifre è [0 0 15 10 0 0 0 65 0 10] allora questo rappresenta una probabilità del 15% che l'immagine sia 2, una probabilità del 10% che l'immagine rappresenti un 3, una probabilità del 65% che l'immagine sia un 7 e una probabilità del 10% che l'immagine sia un 9 (tutti gli altri numeri hanno probabilità nulla di essere scelti). Questo livello completamente connesso funziona poiché guarda l'output del livello precedente e determina quali caratteristiche sono maggiormente correlate a una particolare classe. Se per esempio, il programma prevede che un'immagine sia un cane, avrà identificato nei livelli precedenti caratteristiche di alto livello come una zampa o il muso. Per il risultato finale, un livello FC guarda quali caratteristiche di alto livello sono maggiormente correlate ad una particolare classe e calcola i prodotti tra i pesi e il livello precedente per ottenere le probabilità corrette per le diverse classi.



2.5 Funzioni del modello sviluppato

Loss Function

Scale Invariant Error

La Loss Function utilizzata in questo caso è la Scale Invariant Error, particolarmente utile per quanto riguarda il problema di depth prediction.
Il modello sviluppato si incentra sull'analisi globale della scena, che da sempre rappresenta un grosso ostacolo nella stima della profondità.
Utilizziamo quindi questa funzione per misurare le relazioni tra i punti della scena, indipendentemente dalla scala globale assoluta.

Per una mappa della profondità prevista y e la sua reale distanza y*, ciascuno con n pixel indicizzati da i, definiamo la scale invariant error nello spazio logaritmico come:

$$D(y, y^*) = \frac{1}{2n} \sum_{i=1}^{n} (\log y_i - \log y_i^* + \alpha(y, y^*))^2,$$

L'ultimo termine dentro la sommatoria rappresenta il valore di alpha che minimizza l'errore per un dato set di (y, y*).

Optimizer SGD

In ottimizzazione ed analisi numerica il metodo di discesa del gradiente, detto anche metodo di discesa più ripida, `e una tecnica che consente di determinare i punti di massimo e minimo di una funzione di più variabili. Questo metodo `e il più utilizzato per minimizzare la funzione costo, e quindi ottimizzare le reti neurali, e ciò avviene aggiornando i parametri relativi alla funzione in direzione opposta rispetto al gradiente della funzione stessa.

Come detto precedentemente, addestrare una rete significa scegliere i parametri, ossia pesi e biases, che minimizzano la funzione costo. Essi assumono la forma di

matrici e vettori, ma `e più conveniente immaginarli memorizzati come un singolo vettore che viene indicato con p. Generalmente, si suppone p ∈ R s e si scrive la funzione costo come Cost(p) per enfatizzare la dipendenza dai parametri.

È possibile individuare tre varianti del metodo di discesa del gradiente che differiscono ` per il numero di dati utilizzati nel calcolo del gradiente della funzione. La quantità di dati di cui l'algoritmo si avvale crea un trade-off tra l'accuratezza del parametro aggiornato ed il tempo impiegato per ottenere tale aggiornamento.

Discesa del gradiente

Il metodo di discesa del gradiente convenzionale, noto anche come metodo di discesa del gradiente batch, funziona in modo iterativo e calcola una sequenza di vettori con l'obiettivo di farli convergere ad un vettore che minimizzi la funzione costo.

Il metodo si basa sull'aggiornamento iterativo di un parametro θ lungo la direzione opposta a quella del gradiente della funzione obiettivo C(θ).

L'aggiornamento viene fatto in modo da convergere verso il minimo della funzione.

Uno dei parametri del metodo è il learning rate η che determina l'ampiezza della variazione di θ in ciascuna iterazione e quindi influenza il numero di iterazioni necessarie a raggiungere il valore ottimo della funzione obiettivo.

Va specificato che in base alla convessità della funzione di partenza si otterranno conclusioni diverse: se la funzione fosse convessa, allora il punto trovato sarebbe un punto di minimo globale.

Al contrario, se la funzione non dovesse essere convesse, si tratterebbe di un punto stazionario o un minimo locale.

Questa funzione infatti calcola per ogni iterazione il gradiente $\nabla C(\theta)$ utilizzando tutto il training set.

Discesa del gradiente stocastico (SGD)

L'utilizzo dell'intero training set è uno dei punti deboli del metodo di discesa del gradiente tradizionale.

Per ovviare a questo impiego di risorse, si è pensato di selezionare casualmente, per ogni iterazione, un campione del training set ed utilizzarlo per calcolare il gradiente. Uno dei vantaggi è senz'altro la maggiore velocità della rete e la minore complessità, a discapito di una minima perdita di precisione.

Questo metodo molto più efficace è chiamato discesa del gradiente stocastico.

Mini Batch

Nonostante l'utilizzo di un singolo campione possa sembrare estremamente efficace, talvolta può indurre in effetti collaterali indesiderati, come l'elevata varianza del gradiente.

Questo fenomeno si manifesta dal momento che selezionando un singolo campione, è come se procedessimo in modo casuale per trovare la soluzione migliore.

Una soluzione è stata quella di introdurre un numero n di campioni del training set per ogni iterazione su cui calcolare il gradiente. Questa iterazione di chiama epoca.

Grazie a questo nuovo metodo, chiamato mini-batch gradient descent, riusciamo sia a facilitare la convergenza della funzione verso un suo punto di minimo globale, sia a diminuire la varianza del gradiente.

CAPITOLO 3

SVILUPPO DEL MODELLO

In questo capitolo verrà presentato e spiegato il codice scritto in Python per la realizzazione della rete neurale.
Il lavoro può essere suddiviso in 5 parti:

- Import dei moduli
- Preparazione del dataset
- Loss Function
- Modello
- Valutazione del modello Riduzione size

Nella fase di Import dei moduli verranno scaricati sul server di Google i diversi moduli necessari per il completamento delle operazioni successive; poi verrà preparato il dataset andando a lavorare sui dati che entreranno nella rete; si definirà una Loss Function specifica per questa rete; verrà definito il modello, compilato, addestrato e salvato come file.h5; infine verrà stimata la dimensione del modello e successivamente ridotta mediante un algoritmo integer-only.

Per implementare tale rete è stato utilizzato l'ambiente di sviluppo COLAB fornito da Google, in linguaggio Python e utilizzando le librerie Keras e TensorfFlow.
Questo ambiente di sviluppo è stato particolarmente vantaggioso perché permette allo sviluppatore di dividere in sezioni il codice e mandarle in esecuzione una per volta, facilitando la correzione degli errori.
Ogni sezione può essere utilizzata per scrivere codice o del testo, molto utile per spiegare cosa si andrà a fare successivamente nel programma e aiutando la "pulizia" visiva dell'elaborato.

Keras è una libreria open source scritta in Python ed è stata progettata per lo sviluppo di reti neurali profonde. È un'interfaccia di astrazione superiore rispetto a librerie simili e supporta librerie back-end come TensorFlow, Microsoft Cognitive Toolkit (CNTK) e Theano.

TensorFlow è una piattaforma open source end-to-end per l'apprendimento automatico. Dispone di un ecosistema completo e flessibile di strumenti, librerie e risorse della comunità che consente ai ricercatori di creare e addestrare facilmente modelli di Machine Learning utilizzando API come Keras.
È una API di seconda generazione ed è ampiamente usata in ambiti di ricerca e produzione, essendo alla base di prodotti molto conosciuti di Google come riconoscimento vocale, Gmail, Google Foto, ecc

3.1 Import dei moduli

In questa sezione verranno spiegati i moduli utilizzati nel codice, utili per lo svolgimento delle operazioni successive.
Va specificato che i moduli non vengono scaricati sul computer (lo stesso vale per le librerie Keras e TensorFlow), bensì sul server di Google. Questo è possibile perché il COLAB fa si che Google riservi uno spazio nei loro server per le nostre operazioni, senza andare ad utilizzare le risorse del nostro computer come GPU e CPU, fondamentali nello sviluppo di reti neurali.

Installazione di TensorfFlow e Keras

```
1 !pip install tensorflow
2 !pip install keras
```

Prima di iniziare, bisogna scaricare le due librerie su cui andremo a lavorare, ovvero TensorFlow e Keras. L'istruzione !pip install serve per usare quella cella di codice come fosse il nostro terminale, con l'unica differenza che non stiamo scaricando queste librerie sul nostro computer, ma sul server di Google.

Importo i moduli

```
1 print('### IMPORTING MODULES ###')
2 import tensorflow as tf
3 import cv2
4 import datetime
5 import numpy as np
6 import os
7 import time
8 import yaml
9 from tensorflow import keras
10 from tensorflow.keras import models, layers
11 from tensorflow.keras.models import Sequential, model_from_json
12 from keras.layers import Dense, Dropout, Activation, Flatten, Input, Reshape, merge
13 from keras.layers import Conv2D, MaxPooling2D
14 from tensorflow.keras.optimizers import SGD
15 from keras.utils import np_utils
16 from keras.callbacks import History, ModelCheckpoint
17 print(tf.__version__)
18 print(tf.keras.__version__)
```

In questa sezione ho importato le librerie utili per lo svolgimento del codice :

- numpy : ci permette di lavorare con array e matrici, velocizzando le operazione che di solito si potrebbero fare con delle liste Python. Gli oggetti numpy sono chiamati ndarray

- os: questa libreria ci permette di muoverci all'interno del nostro sistema operativo e in questo esempio ci permetterà di muoverci all'interno della directory del nostro Drive

- cv2: è una libreria che integra numpy per la lettura di immagini. Permette quindi di caricare un'immagine dal file specificato, l'immagine viene letta sotto forma di un array multidimensionale, se l'immagine non può essere letta (a causa di file mancanti, autorizzazioni improprie, formato non supportato o non valido), questo metodo restituisce una matrice vuota.

- time: questa libreria ci permette di lavorare con il tempo all'interno del codice, potendo rappresentarlo come oggetto, numero o stringa.

- yaml: è un formato per la serializzazione di dati utilizzabile da esseri umani. Il linguaggio sfrutta concetti di altri linguaggi come il C, il Perl e il Python e idee dal formato XML e dal formato per la posta elettronica.
  Successivamente vado ad importare dalle librerie di keras anche tutti gli strumenti utili nella composizione della rete neurale, che verranno spiegati in seguito.
  Come ultima istruzione vengono stampate a video le versioni installate di TF e Keras.

Importo seme e shuffle

```
1 seed_value = 1
2 tf.random.set_seed(seed_value)
3 dateTimeStr = datetime.datetime.now()
4 print(dateTimeStr)
```

Questa sezione definisce in che modo i dati verranno rimescolati e stampo a video l'orario e la data attuali.
Serve rimescolare i dati (farne uno shuffle) perché è importante prendere dei campioni casuali in fase di training, seguendo uno split che solitamente è 80/20 (ovvero 80% dei campioni rimescolati vengono usati per l'addestramento e il 20% per il test).

3.2 Preparazione del dataset

Ora è importante caricare il dataset e prepararlo in modo da fornire al modello vero e proprio dei dati utili per un funzionamento corretto ed efficiente.
Come prima cosa carico il dataset

Caricamento del datset

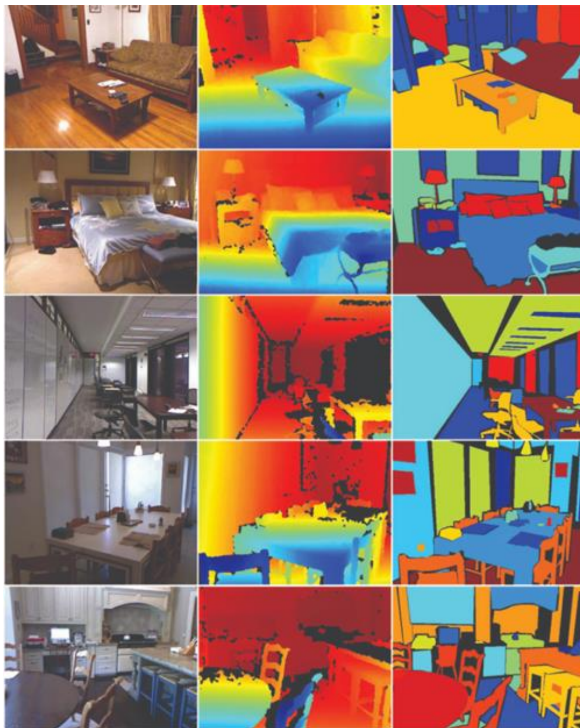
```
1 from google.colab import drive
2 drive.mount('/content/drive/')
3 DATADIR = '/content/drive/My Drive/TIROCINIO/DATASET/nyudataset.tar.grz.zip'
4 OUTDIR = '/content/drive/My Drive/TIROCINIO/OUTDIR'
```

Prima di tutto ho scaricato sul mio Drive di Google il dataset che vado ad utilizzare e
per poterlo utilizzare ho dovuto scrivere il comando
`from google.colab import drive` , che mi permetterà di accedere al Drive.
Da qui ho "montato" il mio drive nel comando successivo e ho permesso l'accesso alla cartella DATADIR dove il dataset era salvato.
OUTDIR rappresenta un'altra cartella all'interno del drive in cui andrò a salvare i risultati della rete.

Parliamo ora del Dataset che sto utilizzando: NYU Depth V2.
Questo dataset è stato realizzato utilizzando sia una camera RGB che una camera Depth fornite da un dispositivo Kinekt della Microsoft.
Nel dataset sono presenti :

- 1449 immagini con label RGB e Depth accoppiate
- 464 scene di tre città
- 407024 frames presi senza label.

Un esempio di queste immagini del dataset potrebbe essere



Reshape dei dati

```
1 def reshapeAndScale(data):
2     if len(data.shape) == 3:
3         data = data.reshape(data.shape[0], data.shape[2], data.shape[1])
4     else:
5         data = data.reshape(data.shape[0], data.shape[2], data.shape[1], data.shape[3])
6     data = data.astype('float32')
7     data /= 255
8     return data
```

In questa porzione di codice ho deciso di eseguire un reshape dei dati, che potrebbero essere non in formato RGB.
Per adempiere a tale compito ho scritto una funzione dal titolo esaustivo: reshapeAndScale, che andrà a modificare il formato delle immagini del dataset e normalizzarle.
Inizio verificando che la lunghezza di data.shape sia 3, procedo con il reshape e assegno il corretto ordine ai canali: Rosso, Verde e Blu.
Può succedere infatti che le immagini non siano in formato RGB, ma BGR, determinando un andamento non corretto della nostra rete.
In seguito procedo con la normalizzazione dei punti (pixel) dell'immagine, dividendoli per 255.
La funzione poi ritornerà i dati una volta effettuate queste operazioni.

Load dei dati

```
1 def loadData(dataDir):
2     X_files = [os.path.join(dataDir, f) for f in os.listdir(dataDir) if '_image' in f]
3     X = np.array([cv2.pyrDown(cv2.imread(f)) for f in X_files])
4     X = reshapeAndScale(X)
5 
6     Y_files = [f.replace('_image', '_depth') for f in X_files]
7     Y = np.array([cv2.pyrDown(cv2.pyrDown(cv2.pyrDown(cv2.imread(f, 0)))) for f in Y_files])
8     Y = reshapeAndScale(Y)
9     return X, Y
```

Avendo effettuato il reshape e normalizzazione dei dati provenienti dal dataset, ora procedo con il caricamento di tali dati in sostituzione dei precedenti.

Dati di configurazione

```
1 BATCH_SIZE = 32
2 EPOCHS =  1000
3 IMG_ROWS = 640
4 IMG_COLS = 480
5 LEARNING_RATE = 0.1
6 MOMENTUM = 0.9
7 LAMBDA = 0.5
8 input_shape = (int(IMG_ROWS/2), int(IMG_COLS/2), 3)
```

In questa sezione ho deciso di salvare tutte le variabili con i loro valori che userò all'interno del modello, per evitare ripetizioni inutili nel proseguimento del codice. I loro valori e cosa rappresentano verranno spiegati meglio nella costruzione del modello.

3.3 Loss function

Come avevamo detto nel capitolo precedente, ci sono diversi tipi di Loss Function e possono essere più o meno utili a seconda del problema in esame.
Per questo modello, si è deciso di implementare una funzione chiamata scale invariant error, particolarmente adeguata per problemi legati alla stima di profondità di immagini.
Il problema maggiore quando si vuole stimare la profondità di immagini è la valutazione globale (global scale) della scena. Proprio per questo si è deciso di implementare la scale invariant error: fornisce prestazioni migliori nel misurare la distanza tra i punti all'interno di un'immagine.

All'interno del codice la funzione si presenta come:

```python
def scale_invariant_error(self, y_true, y_pred):
        first_log = K.log(K.clip(y_pred, K.epsilon(), np.inf) + 1.)
        second_log = K.log(K.clip(y_true, K.epsilon(), np.inf) + 1.)
        return K.mean(K.square(first_log - second_log), axis=-1) - LAMBDA * K.square(K.mean(first_log - second_log, axis=-1))

```

La funzione prende in ingresso y_true e y_pred, ovvero i valori reali e quelli desiderati.

Definiamo quindi i due logaritmi utilizzati nella formula finale

$$D(y, y^*) = \frac{1}{2n} \sum_{i=1}^{n} (\log y_i - \log y_i^* + \alpha(y, y^*))^2,$$

Tutti i dettagli di questa formula sono stati descritti in precedenza.

3.4 Modello

Ora procederemo con la scrittura del modello, la sua compilazione, l'addestramento e lo testeremo per verificarne l'efficienza.
Partiamo con la struttura del modello:

```
1 model = keras.Sequential()
2 # Coarse 1:
3 # 11x11 conv, 4 stride, ReLU activation, 2x2 pool
4 model.add(Conv2D(96, (11, 11), activation = 'relu', input_shape = input_shape)
5 model.add(MaxPooling2D((2, 2)))
6 # Coarse 2:
7 # 5x5 conv, 1 stride, ReLU activation, 2x2 pool
8 model.add(Conv2D(256, (5, 5), activation = 'relu'))
9 model.add(MaxPooling2D((2, 2)))
10 # Coarse 3:
11 # 3x3 conv, 1 stride, ReLU activation, no pool
12 model.add(Conv2D(384, (3, 3), activation = 'relu'))
13
14 # Coarse 4:
15 # 3x3 conv, 1 stride, ReLU activation, no pool
16 model.add(Conv2D(384, (3, 3), activation = 'relu'))
17
18 # Coarse 5:
19 # 3x3 conv, 1 stride, ReLU activation, 2x2 pool
20 model.add(layers.Conv2D(256, (3, 3), activation = 'relu'))
21 model.add(layers.MaxPooling2D((2, 2)))
22 # Coarse 6:
23 # Fully-connected, ReLU activation, followed by dropout
24 model.add(Flatten())
25 model.add(Dense(4096, activation = 'relu'))
26 model.add(Dropout(0.5))
27
28
29 # Coarse 7:
30 # Fully-connected, linear activation
31 model.add(Dense(((IMG_ROWS)/8)*((IMG_COLS)/8)), activation = 'linear')
32 model.add(Reshape((IMG_ROWS/8, IMG_COLS/8)))
```

Il modello che viene implementato è un modello di tipo sequenziale, in quanto si ha una semplice pila di layer dove ogni layer ha esattamente un tensore* di input e un tensore di output.
Prima di dare in ingresso alla rete le immagini, bisogna ridimensionare il tensore di ingresso secondo la forma richiesta da input_shape. A questo sono servite le funzioni definite in precedenza nella sezione reshape.

model = Sequential() ci permette di inizializzare model come un modello sequenziale

Notiamo che il nostro modello è stato sviluppato secondo una struttura composta da 7 layers chiamati Coarse, ciascuno con caratteristiche e compiti diversi.
Partiamo quindi con la descrizione di ciò che avviene in ciascun layer:

Coarse 1 : in questo layer si effettua una convoluzione in 2D mediante 96 filtri di dimensione 3x3 sul tensore di ingresso. In seguito viene utilizzata la funzione di attivazione ReLu e viene fatto un Max Pooling di dimensioni 2x2.

Coarse 2: le operazioni fatte in questo layer sono simili al precedente, con l'unica differenza rappresentata dalla dimensione del filtro di convoluzione (96 al posto di 256).

Coarse 3 e Coarse 4: in entrambi è presente solamente uno strato di convoluzione in 2D, che utilizza 384 filtri mediante una matrice avente dimensioni 3x3.

Coarse 5: questo layer vede l'applicazione di uno strato convolutivo che si avvale di 256 filtri grandi 3x3, una funzione di attivazione ReLu e uno strato di Max Pooling con dimensioni 2x2.

Coarse 6 e Coarse 7: questi layer rappresentano i livelli fully-connected e final layer della struttura. È presente un Dropout di 0.5, ovvero la rete spegne il 50% dei neuroni ad ogni iterazione, per evitare il fenomeno dell'overfitting.

Compile del modello

La rete è così costruita e si può procedere con la compilazione del modello.

```
1 # compile model
2 print("### COMPILE THE MODEL###")
3 model.compile(optimizer = optimizer.SGD(lr=LEARNING_RATE, momentum=MOMENTUM, nesterov=False, name='SGD', **kwargs),
4               loss = scale_invariant_error(),
5               metrics = ['accuracy'])
6 model.summary()
```

Nelle impostazioni di compilazione sono state aggiunte informazioni sul tipo di Loss Function (scale invariant error descritta in precedenza), l'ottimizzatore (SGD, anch'esso descritto in precedenza) e le metriche, ovvero cosa si vuole misurare con questa rete.
Come ultimo comando troviamo model.summary(), che stampa a video l'intera struttura del modello e mostra la struttura della rete.

Salvataggio su Outdir

```
1 # save the model architecture to file
2
3 print('### SAVING MODEL ARCHITECTURE ###')
4 modelDir = dateTimeStr;
5 os.mkdir(os.path.join('OUTDIR', modelDir))
6 modelFile = os.path.join('OUTDIR', modelDir, 'depth_coarse_model_{}.json'.format(dateTimeStr))
7 print(model.to_json(), file=open(modelFile, 'w'))
```

Con questi comandi vado a salvare i risultati di questa rete sulla cartella OUTDIR, creata in precedenza.

Carico e preprocesso i dati

```
1 # load and preprocess the data
2 print('### LOADING DATA ###')
3 X_train, Y_train = loadData(os.path.join('DATADIR', 'train/'))
4 X_test , Y_test = loadData(os.path.join('DATADIR', 'test/'))
5 print('X_train shape:', X_train.shape)
6 print('Y_train shape:', Y_train.shape)
7 print(X_train.shape[0], 'train samples')
8 print(X_test.shape[0], 'test samples')
```

Inizio con lo stampare a video la volontà di caricare i dati, per poi prelevarli da DATADIR e suddividerli in train e test, ovvero una parte di questi mi serviranno per l'addestramento della rete e una parte per testarla su dati ancora mai visti.
Mando a video la forma di questi dati come ultime istruzioni della sezione.

Train del modello

```
1 print('### TRAINING ###')
2 history_cb = History()
3 checkpointFile = os.path.join('OUTDIR', modelDir, 'coarse-weights-improvement-{epoch:02d}-{val_acc:.2f}.hdf5')
4 checkpoint_cb = ModelCheckpoint(filepath=checkpointFile, monitor='val_loss',
5                                 verbose=1, save_best_only=True,
6                                 save_weights_only=True, mode='auto')
7 model.fit(X_train,
8           Y_train,
9           epochs=EPOCHS,
10          batch_size=BATCH_SIZE,
11          verbose=1,
12          validation_split=0.2,
13          callbacks=[history_cb, checkpoint_cb])
14 histFile = os.path.join('OUTDIR', modelDir, 'depth_coarse_hist_{}.h5'.format(dateTimeStr))
```

Questo è l'addestramento vero e proprio della rete, in cui utilizzo X_train e Y_train come array usati per l'operazione.
Vado anche a specificare il numero di epoche, le dimensioni della batch e il validation split, ovvero in che modo vado a suddividere il training set in un altro dataset che valuta le performance della rete ad ogni epoca.
In questo caso sto prendendo il 20% del training set.

3.5 Valutazione del modello e riduzione size

Abbiamo concluso la stesura della rete neurale, abbiamo definito le sue caratteristiche e come essa debba essere allenata.
Manca ora la fase di valutazione, in cui la rete viene sottoposta a dei campioni random e viene valutato il suo comportamento nel riconoscere le immagini.
È una fase di valutazione delle prestazioni.

Solitamente le prestazioni della rete sono ritenute accettabili se sopra all'80/90%, ma il risultato può dipendere anche dalle caratteristiche del dataset: se le immagini sono state pre-elaborate è più probabile che la rete funzioni meglio.
Come detto prima, di solito si valuta la rete anche utilizzando una parte del training set (set di validazione), per poter valutare le prestazioni della CNN anche su campioni random mai osservati durante l'addestramento.

```
1 print('### EVALUATING ###')
2 score = model.evaluate(X_test, Y_test, verbose=1)
3 print('Test score:', score[0])
4 print('Test accuracy:', score[1])
```

In questo caso stampiamo a video la volontà di effettuare l'operazione di valutazione, per poi specificare l'utilizzo degli array X_test e Y_test, attribuendo uno score (risultato) all'operazione.

Il passo successivo alla valutazione del modello sarà quello di valutarne le dimensioni.
Questa operazione è importante perché solitamente questo genere di modelli vengono utilizzati poi in sistemi embedded, che non dispongono di grandi capacità di memoria.
Avere infatti un modello che occupi poco spazio migliora le prestazioni del sistema in cui opera la rete ed è quindi di vitale importanza ridurre queste dimensioni il più possibile.

Va quindi operata una riduzione della size attraverso una tecnica di quantizzazione chiamata integer-only.
Uno dei lati negativi di questa operazione è il peggioramento dell'accuratezza della rete, ma se non si esagera con la quantizzazione, i risultati rimangono accettabili.
Spiegheremo meglio in seguito come fare questa operazione.

Stima della size

```
# Define function to estimate model size

def get_file_size(file_path):
    size = os.path.getsize(file_path)
    return size

def convert_bytes(size, unit=None):
    if unit == "KB":
        return print('File size: ' + str(round(size / 1024, 3)) + ' KB')
    elif unit == "MB":
        return print('File size: ' + str(round(size / (1024 * 1024), 3)) + ' MB')
    else:
        return print('File size: ' + str(size) + ' bytes')
```

In questa sezione di codice stiamo andando a definire una funzione in grado di definire la grandezza del file contenente il modello.
La funzione convert_bytes prende in ingresso la size del file del modello e la traduce in KB o MB, unità più comode rispetto ai Bytes.

La funzione get_file_size invece restituisce il peso finale una volta effettuata questa conversione.

Salvataggio del modello

```
# Save model to a HDF5 file format
import os
!pip install -q pyyaml h5py  # Required to save models in HDF5 format

saved_model = "my_model.h5"
model.save(saved_model)
!ls
```

```
# Compute model size
convert_bytes(get_file_size(saved_model), "MB")
get_file_size(saved_model)
```

Salvo ora il modello come file in formato h5 per poi chiamare le funzioni che mi restituiscono la grandezza della size del modello.

Manca ora eseguire la quantizzazione del modello.

Quantizzazione del modello

```
1 # Convert using integer-only quantization
2 # Now you have an integer quantized model that uses integer data for
3 # the model's input and output tensors, so it's compatible with integer-only hardware
4
5 def representative_data_gen():
6   for input_value in tf.data.Dataset.from_tensor_slices(x_train).batch(1).take(100):
7     yield [input_value]
8
9 # quantization
10 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
11 converter.inference_input_type = tf.int8
12 converter.inference_output_type = tf.int8
13 converter.optimizations = [tf.lite.Optimize.DEFAULT]
14 converter.representative_dataset = representative_data_gen
15 tflite_model_quantInt = converter.convert()
16
17 interpreter = tf.lite.Interpreter(model_content=tflite_model_quantInt)
18 input_type = interpreter.get_input_details()[0]['dtype']
19 print('input: ', input_type)
20 output_type = interpreter.get_output_details()[0]['dtype']
21 print('output: ', output_type)
22
23 # Save the quantized int model:
24 import pathlib
25 tflite_models_dir = pathlib.Path("./")
26 tflite_models_dir.mkdir(exist_ok=True, parents=True)
27 tflite_model_quantInt_file = tflite_models_dir/"mnist_model_quantInt.tflite"
28 tflite_model_quantInt_file.write_bytes(tflite_model_quantInt)
29
30 # Estimate size
31 convert_bytes(get_file_size(tflite_model_quantInt_file), "MB")
```

La quantizzazione è una strategia di ottimizzazione che converte i numeri in virgola mobile a 32 bit (ad esempio pesi e uscite di attivazione) nei numeri in virgola mobile a 8 bit.
Logicamente avremo un modello più piccolo e con una maggiore velocità (qualità che oggi sono fondamentali), a discapito di una parte di precisione che viene per forza persa durante il processo.

CONCLUSIONI E SVILUPPI FUTURI

Le reti neurali per la stima della profondità delle si sono dimostrati estremamente utili ed efficaci per l'applicazione su sistemi embedded, in particolare per sistemi di guida autonoma.
A tal fine nel lavoro di tesi viene presentata l'architettura di una rete neurale per la stima della profondità delle immagini in applicazioni automotive e gli step implementativi per la realizzazione di un codice Python in libreria Keras adatta all'implementazione su sistemi embedded.
In particolare, la tesi inizia con una presentazione teorica sulle reti neurali, per poi approfondire l'argomento studiandone il funzionamento e la struttura.
È stato implementato un codice Python utilizzando le librerie Keras e TF per caricare il Dataset fornito dalla NYU e pre-elaborarne le immagini, per poi sviluppare il modello di rete neurale necessario per stimare la profondità.
Il modello è stato infine quantizzato e salvato, riducendone le dimensioni per una futura applicazione su dispositivo embedded.
Sviluppi futuri prevedono quindi la validazione della rete implementata su piattaforme basate su microcontrollori a basso consumo per valutarne le prestazioni.

BIBLIOGRAFIA E SITOGRAFIA

Paper: "Depth Map Prediction from a Single Image using a Multi-Scale Deep Network"
Eigen, David, Christian Puhrsch, and Rob Fergus. "Depth map prediction from a single image using a multi-scale deep network.

Libro: “Deep Learning with Python”, Francois Chollet

https://github.com/jrussino/eigen_depth

https://www.tensorflow.org/tutorials

https://www.tensorflow.org/guide

https://keras.io/guides/

https://docs.python.org/3/tutorial/index.html

https://stackoverflow.com/

https://www.freecodecamp.org/learn/scientific-computing-with-python/#python-for-everybody

https://www.freecodecamp.org/learn/machine-learning-with-python/

https://www.w3schools.com/python/default.asp

https://www.w3schools.com/python/exercise.asp?filename=exercise_variables4

https://www.codecademy.com/catalog