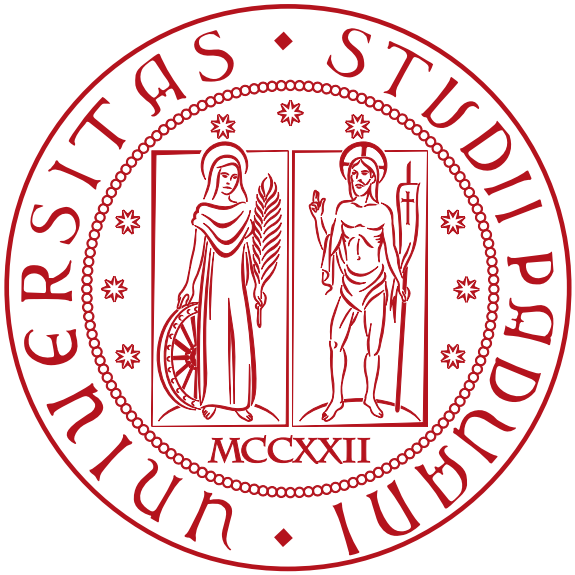
**Università degli studi di Padova**

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica


# Predizione della Profondità con Deep Learning da Immagini di Telecamera Monoculare

*Tesi di laurea*


*Relatore*

Prof. Lamberto Ballan

*Laureando*

Riccardo Toniolo

*Matricola* 2042332


---


ANNO ACCADEMICO 2023-2024

# Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dal laureando Riccardo Toniolo, affiancante la dottoranda Elena Izzo, presso il gruppo di ricerca VIMP Group, Università degli studi di Padova.
Gli obbiettivi da raggiungere erano molteplici. In primo luogo era richiesto lo sviluppo di … In secondo luogo era richiesta l'implementazione di un … Tale framework permette di registrare gli eventi di un controllore programmabile, quali segnali applicati Terzo ed ultimo obbiettivo era l'integrazione …

# Indice



# Elenco delle Figure

# Elenco delle Tabelle

# Capitolo 1.

# Introduzione

## 1.1. Stimare le profondità

L'avere la possibilità di misurare la profondità di un'immagine e, quindi, potenzialmente la profondità di ciascun frame all'interno di uno streaming video, apre le porte alla risoluzione di una vasta gamma di problemi che richiedono una stima precisa delle distanze tra gli oggetti all'interno di un determinato campo visivo.
Alcuni problemi appartenenti a questa classe includono:

- Prevenzione delle collisioni: lo sviluppo di algoritmi che, controllando un oggetto fisico in movimento, cercano di evitare impatti con altre entità lungo il suo percorso;
- Percezione tridimensionale: una serie di algoritmi che analizzano dati di profondità per ricostruire una scena tridimensionale dell'ambiente circostante;
- Realtà aumentata: un settore che prevede, attraverso l'uso di visori e altri dispositivi, il posizionamento di elementi grafici virtuali nel campo visivo dell'utente in modo che si integrino naturalmente con la realtà presente.

Esistono soluzioni hardware per avere delle misurazioni di profondità con alta precisione. I sistemi hardware più famosi ed utilizzati sono:

- Sensori di profondità, come ad esempio il **LiDAR**;
- Sistemi di fotografia stereoscopica: come ad esempio la **stereocamera**.

Il problema con questi sistemi hardware risiede, tuttavia, in due aspetti principali:

- Nel caso dei sensori LiDAR, il costo elevato può rendere il prodotto finale meno competitivo sul mercato o ridurre i margini di profitto per l'azienda che lo fornisce. Ad esempio, se si volesse integrare un LiDAR in un robot da giardino, il costo aggiuntivo potrebbe influire negativamente sulla competitività del prodotto.
- D'altro canto, l'integrazione di un sistema basato su fotocamere stereoscopiche presenta altre sfide pratiche. Non è sempre semplice trovare spazio per le due fotocamere necessarie e gestire la loro calibrazione può essere complicato. Questi problemi possono limitare l'applicabilità della tecnologia in ambienti dove lo spazio è ristretto o dove la calibrazione precisa è difficile da ottenere.

## 1.2. *Monocular Depth Estimation*

Un'altra soluzione promettente è quella di sviluppare una rete neurale in grado di prevedere correttamente le profondità di un'immagine a partire da una singola immagine di input, un approccio noto come *Monocular Depth Estimation* (**MDE**). Se riuscisse ad essere realizzata con successo, tale soluzione permetterebbe il vantaggio di utilizzare una sola fotocamera, con il potenziale di un sensore LiDAR.
Tuttavia, questo tipo di approccio, oltre alle tradizionali sfide del *machine learning*, come la ricerca di dataset adeguati e la costruzione di un modello adatto, presenta ulteriori difficoltà, in particolare nei sistemi **embedded**, che hanno vincoli significativi in termini di memoria, energia e di potenza computazionale.
Modello particolarmente interessante di *machine learning* è PyDNet [1], [2], in quanto fortemente leggero (meno di 2 milioni di parametri) e decentemente performante per la sua dimensione. Per questo motivo è stato scelto per essere il modello di riferimento da usare come punto di partenza del tirocinio.

## 1.3. Obbiettivi

Il tirocinio si è quindi strutturato su due macro obbiettivi:

1. Studiare come il problema di **MDE** è stato affrontato da PyDNet V1 [1] e V2 [2]:
   - Studiarne i paper e i relativi codici;
   - Inserire nella procedura di allenamento un sistema di *logging* per analizzare i costi provenienti dalle varie *loss function*;
   - Riprodurre l'allenamento di [1] per verificare i risultati enunciati nel *paper*;
   - Migrare tutto il codice di [1] e il codice del modello di [2] da **Tensorflow** a **PyTorch**;
   - Riprodurre l'allenamento di [1] nella sua versione migrata per verificare che si ottengano gli stessi risultati e che quindi la versione migrata sia equivalente all'originale.
2. Esplorare soluzioni per ottenere modelli migliori in termini di efficacia e efficienza:
   - Verificare come variano le prestazioni di [1] al variare degli iperparametri;
   - Esplorare eventuali nuove tecniche e strategie al fine di creare un modello migliore nel compito di **MDE**.

## 1.4. Organizzazione del documento

Relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- I termini riportati nel glossario utilizzano la seguente formattazione: **parola**;
- Dopo la prima citazione del soggetto di un articolo o di un testo di riferimento ritrovabile in bibliografia, questo verrà poi menzionato solo dal suo numero di riferimento (i.e. [1]) e non più dal suo nome;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Relativamente ai capitoli:

# Capitolo 2.

# PyDNet

PyDNet(**Py**ramidal **D**epth **Net**work) è una famiglia di modelli composta da due versioni [1], [2], che cercano di risolvere il problema della **MDE** mediante un approccio non supervisionato. con circa 2 milioni di parametri in [1] e circa 700.000 in [2].
Una peculiarità di questa famiglia di modelli è il numero di parametri estremamente basso, con circa 2 milioni di parametri in [1] e circa 700.000 in [2]. L'obbiettivo di questi modelli infatti, è quello di essere abbastanza leggeri da poter essere direttamente eseguiti su un processore, senza il supporto di una scheda grafica, per poter essere integrati in sistemi **embedded**, come ad esempio nei cellulari [3].
Questa sua caratteristica li rende molto interessanti come punto di partenza per sperimentare con tecniche innovative o apportare modifiche ai loro blocchi, di modo da migliorarne le prestazioni.
Tuttavia, essendo modelli relativamente datati, sono stati scritti in una versione di **Tensorflow** ormai deprecata, il che li rende non più facilmente utilizzabili. I sottocapitoli seguenti descriveranno quindi l'intero processo di migrazione da **Tensorflow** all'ultima versione di **PyTorch**.

## 2.1. Architettura del modello

[1] è una rete convoluzionale profonda strutturata su sei livelli, dove ogni livello riceve l' input dal livello superiore (fatta eccezione per il primo livello che riceve l'immagine di input), elabora l'input ricevuto mediante un **encoder**, che restituisce un output il quale farà da input al livello inferiore (fatta eccezione per l'ultimo livello). L'output dell'**encoder** viene poi concatenato all'output del livello inferiore sulla dimensione dei canali, e passato ad un **decoder**, il cui output verrà:

- Passato attraverso la funzione di attivazione sigmoide, il cui output corrisponderà alla mappa di **disparità** del livello preso in analisi;
- Passato attraverso una convoluzione trasposta con **kernel** di dimensione $2 \times 2$ e **stride** 2, per raddoppiare le dimensioni di altezza e larghezza del tensore in ingresso, il cui output verrà passato al livello superiore (eccezione fatta per il primo livello, che ha come unico output il risultato la mappa di **disparità** del livello).

L'architettura è quindi la seguente:

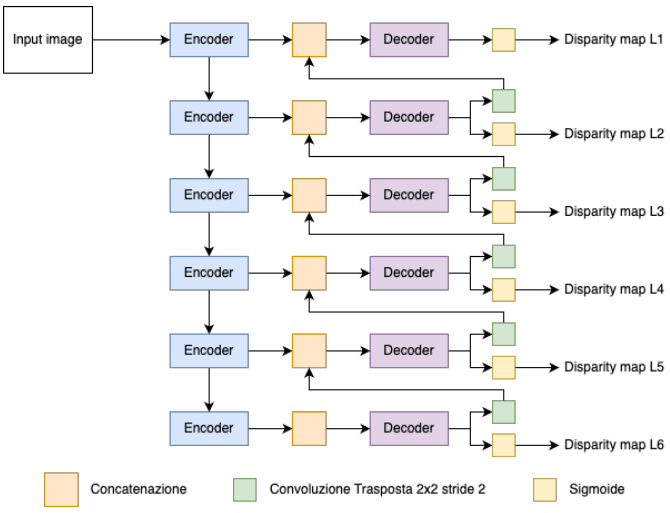



Figura 1: Architettura del modello [1]

L'**encoder** è composto da una convoluzione con **kernel** di dimensione $3 \times 3$ e **stride** 2, che va quindi a ridurre l'altezza e la larghezza del tensore di ingresso della metà, seguita da una convoluzione con **kernel** di dimensione $3 \times 3$. L'**encoder** del livello $i, \forall i \in \{1, 2, 3, 4, 5, 6\}$ avrà quindi come tensore in uscita un tensore con dimensioni di altezza e larghezza pari a $\frac{1}{2^i}$ delle dimensioni dell'input iniziale. In più, andando dal primo al sesto livello, i canali di uscita prodotti dalla seconda convoluzione dell'encoder sono 16, 32, 64, 96, 128, 196.
L'architettura dell'encoder è quindi la seguente:

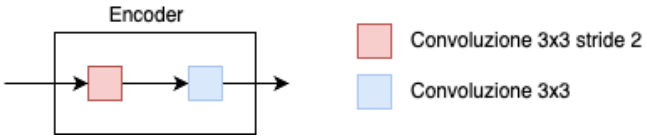



Figura 2: **Encoder** del modello [1].

Il **decoder** invece è composto da una successione di quattro convoluzioni con **kernel** di dimensione $3 \times 3$ e **stride** 2, i quali rispettivamente producono delle **feature map** con un numero di canali pari a 96, 64, 32 e 8, mantenendo invece le dimensioni di altezza e larghezza dell'*input*. L'architettura del **decoder** è quindi la seguente:

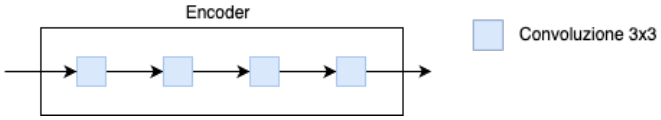



Figura 3: **Decoder** del modello [1].

Successivamente ad ogni convoluzione, tranne per l'ultima del decoder, viene applicata la funzione di attivazione *leaky ReLU* con coefficiente di crescita per la parte negativa di 0,2.

## 2.2. Funzionamento

Lo scopo di [1], [2] è quello di riuscire nel compito di **MDE**. Per farlo [1], [2] imparano, date due immagini stereo dello stesso scenario, ad applicare uno sfasamento a ogni pixel dell'immagine di sinistra, in modo da renderla quanto più simile possibile alla corrispondente immagine di destra, e uno sfasamento a ogni pixel dell'immagine di destra per renderla quanto più simile possibile alla corrispondente immagine di sinistra.

Quanto enunciato però viene fatto solamente durante l'addestramento del modello, poichè durante l'inferenza ogni immagine di *input* viene introdotta nel modello come immagine di sinistra. Si prende infatti come *output* del modello, solo il primo canale delle **feature map** uscenti dalla sigmoide del livello, che infatti corrisponde alla mappa di **disparità** per le immagini di sinistra. Questa strategia funziona perchè progressivamente, durante l'allenamento, viene forzato un allineamento tra le predizioni per le immagini di sinistra e quelle di destra, rendendo ambivalente l'*output* del modello sul canale di sinistra rispetto a quello di destra.

### 2.2.1. Funzioni di perdita

***Image error loss*** ($\mathcal{L}_{\text{ap}}$):

$$\mathcal{L}_{\text{ap}}^{l} = \frac{1}{N} \sum_{i,j} \alpha \frac{1 - \text{SSIM}\left(I_{i,j}^{l}, \tilde{I}_{i,j}^{l}\right)}{2} + (1 - \alpha) \left\| \left(I_{i,j}^{l}, \tilde{I}_{i,j}^{l}\right) \right\|_1$$

Figura 4: *Image error loss* calcolata per l'immagine di sinistra.

Questa è la funzione di perdita che penalizza quanto più l'immagine originale di sinistra $I^l$ è diversa dall'immagine di destra con lo sfasamento applicato $\tilde{I}^l$ (appunto per diventare l'immagine di sinistra). La prima parte della sommatoria, utilizzando la funzione SSIM, serve per misurare la similarità strutturale tra le due immagini, mentre con la seconda parte, mediante la norma 1,

serve per misurare la distanza tra i corrispondenti pixel delle due immagini. Il parametro $\alpha$ viene usato per regolare il peso tra la prima e la seconda parte, il quale viene impostato a 0,85, andando quindi a dare molta più importanza alla prima.

***Disparity smoothness loss*** ($\mathcal{L}_{\text{ds}}$):

$$\mathcal{L}_{\text{ds}}^{l} = \frac{1}{N}\sum_{i,j}\left|\delta_x d_{i,j}^{l}\right|e^{-\left\|\delta_x I_{i,j}^{l}\right\|} + \left|\delta_y d_{i,j}^{l}\right|e^{-\left\|\delta_y I_{i,j}^{l}\right\|}$$

Figura 5: *Disparity smoothness loss* calcolata per l'immagine di sinistra.

In questo caso invece, questa funzione di perdita disincentiva discontinuità di profondità calcolate mediante norma 1, a meno che non ci sia una discontinuità sul gradiente dell'immagine. La prima parte della sommatoria analizza le discontinuità sull'asse orizzontale, mentre la seconda parte sull'asse verticale.

***Left-right consistency loss*** ($\mathcal{L}_{\text{lr}}$):

$$\mathcal{L}_{\text{lr}}^{l} = \frac{1}{N}\sum_{i,j}\left|d_{i,j}^{l} - d_{i,j+d_{i,j}^{l}}^{r}\right|$$

Figura 6: *Left-right consistency loss* calcolata per l'immagine di sinistra.

Infine, l'ultima funzione di perdita, formula nota nel campo degli algoritmi stereo, serve per forzare coerenza tra le predizioni di **disparità** di destra $d^r$ e di sinistra $d^l$.

**Funzione di perdita completa** ($\mathcal{L}_{\text{lr}}$):
Le precedenti funzioni di perdita vengono calcolate anche per l'immagine di sinistra, per poi essere combinate nel seguente modo, creando la funzione di perdita completa $\mathcal{L}_s$:

$$\mathcal{L}_s = \alpha_{\text{ap}}\left(\mathcal{L}_{\text{ap}}^{l} + \mathcal{L}_{\text{ap}}^{r}\right) + \alpha_{\text{ds}}\left(\mathcal{L}_{\text{ds}}^{l} + \mathcal{L}_{\text{ds}}^{r}\right) + \alpha_{\text{lr}}\left(\mathcal{L}_{\text{lr}}^{l} + \mathcal{L}_{\text{lr}}^{r}\right)$$

I pesi per i vari termini della funzione completa sono impostati nel seguente modo:
- $\alpha_{\text{ap}} = 1$;
- $\alpha_{\text{lr}} = 1$;
- $\alpha_{\text{ds}} = \frac{1}{r}$ dove $r$ è il fattore di scala a ciascun livello di risoluzione.

### 2.2.2. Allenamento

Per l'allenamento viene utilizzato l'ottimizzatore **Adam** con i seguenti parametri: $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\varepsilon = 10^{-8}$.
Il *learning rate* parte da $10^{-}4$ per il primo 60% delle epoche, e viene dimezzato ogni 20% successivo.
Infine vengono applicate, con una probabilità del 50%, le seguenti *data augmentation*:
- Capovolgimento orizzontale delle immagini;
- Trasformazione delle immagini:
  - Correzione gamma;
  - Correzione luminosità;
  - Sfasamento dei colori.

Il dataset viene suddiviso in batch da 8 immagini, e verranno eseguite un totale di 50 epoche di allenamento.

## 2.3. Configurazione dell'ambiente

La versione di Tensorflow usata per l'ambiente di allenamento e per i modelli [1], [2] è la 1.8, ormai deprecata da anni e non più scaricabile dai package manager come **pip** o **Anaconda**. Una versione retrocompatibile con la 1.8 e ancora scaricabile tramite **pip** è la 1.13.2, che però dipende da una versione del pacchetto protobuf non più disponibile. Fortunatamente, la versione 3.20 di protobuf è ancora scaricabile da **pip** e compatibile con Tensorflow 1.13.2. Il codice si basava anche

su una versione deprecata del pacchetto scipy, facilmente sostituibile con la versione 1.2, ancora disponibile tramite **pip**. L'ultima configurazione necessaria per eseguire il codice è la corretta versione di **Python**, ancora scaricabile e che riesca ad essere compatibile con tutti i pacchetti sopra menzionati e con le relative dipendenze. Grazie ad **Anaconda** è possibile scaricare la versione 3.7 che è utilizzabile per questo scopo.
I comandi da terminale per ottenere la seguente configurazione, previa corretta installazione di **pip** e **Anaconda** sono:

```Bash
# Creare l'ambiente Anaconda (usare il nome che si preferisce)
conda create -n <nomeAmbiente> python=3.7
# Attivare l'ambiente Antaconda creato
conda activate <nomeAmbiente>
# Installare i paccheti richiesti
pip install protobuf==3.20 tensorflow_gpu=1.13.2 scipy=1.2 matplotlib wandb
```

Tra i pacchetti installati mediante **pip** è presente anche **Wandb**, un sistema che permette di registrare, gestire e catalogare il *plot* delle *loss function* per i vari esperimenti che verranno condotti.

Il codice è tecnicamente eseguibile, solo se si dispone di una sceda video all'interno della macchina. Tuttavia il cluster del dipartimento di matematica, ha versioni troppo aggiornate dei driver **CUDA** e della libreria **cuDNN**, per essere utilizzabili da **Tensorflow** 1.13.2. Ho quindi ritrovato le versioni adatte: per **CUDA**, la versione 10.0, scaricabile seguendo le istruzioni presenti nell'**archivio CUDA**, e per **cuDNN**, la versione 7.4.2, scaricabile seguendo le istruzioni presenti nell'**archivio cuDNN**.
Infine bisogna scaricare il dataset KITTI, dataset utilizzato per l'addestramento e valutazione del modello, utilizzando questo comando:

```Bash
wget -i utils/kitti_archives_to_download.txt -P ~/my/output/folder/
```

Successivamente bisogna effettuare l'*unzip* di tutte le cartelle compresse e convertire tutte le immagini da .png a .jpg, mediante i seguenti comandi:

```Bash
cd <pathCartellaDataset>
find <pathCartellaDataset> -name '*.zip' | parallel 'unzip -d {.} {}'
find <pathCartellaDataset> -name '*.png' | parallel 'convert {.}.png {.}.jpg && rm {}'
```

Dove <pathCartellaDataset> è il path che conduce alle cartelle .zip precedentemente scaricate.

## 2.4. Validazione

Seguendo le istruzioni ritrovabili nella *repository* di [1] e [4], si possono recuperare le seguenti istruzioni per effettuare l'esecuzione dell'allenamento, il *testing* e la successiva valutazione di [1]. Quindi, una volta impostata una *codebase* come scritto nella documentazione di [1] possono essere utilizzati i seguenti comandi.
**Per l'allenamento**:

```Bash
conda activate <nomeAmbiente>
python3 <pathFileEseguibile>/monodepth_main.py \
  --mode train \
  --model_name pydnet_v1 \
  --data_path <datasetPath> \
```

```
  --filenames_file <fileNamesDatasetPath>/eigen_train_files.txt \
  --log_directory <outputFilesPath>
```

Dove:
- <nomeAmbiente>: è il nome dell'ambiente **Anaconda** da dover attivare;
- <pathFileEseguibile>: è il path che conduce al file monodepth_main.py, file che dovrà essere eseguito per eseguire l'allenamento;
- <datasetPath>: è il path che conduce alla cartella contenente il dataset;
- <fileNamesDatasetPath>: è il path che conduce al file eigen_train_files.txt;
- <outputFilesPath>: è il path che conduce alla cartella dove verranno salvati tutti i file di output prodotti dalla procedura di allenamento.

Questa procedura produrrà dei file di checkpoint, ritrovabili nella cartella <outputFilesPath>.
**Per il *testing*:**

```Bash
conda activate <nomeAmbiente>
python3 <pathFileEseguibile>/experiments.py \
  --datapath <datasetPath> \
  --filenames <fileNamesDatasetPath>/eigen_test_files.txt \
  --output_directory <outputFilesPath> \
  --checkpoint_dir <checkpointPath>
```

Dove:
- <nomeAmbiente>: è il nome dell'ambiente **Anaconda** da dover attivare;
- <pathFileEseguibile>: è il path che conduce al file experiments.py, file che dovrà essere eseguito per calcolare e generare il file disparities.npy;
- <datasetPath>: è il path che conduce alla cartella contenente il dataset;
- <fileNamesDatasetPath>: è il path che conduce al file eigen_test_files.txt;
- <outputFilesPath>: è il path che conduce alla cartella dove verrà salvato il file disparities.npy;
- <checkpointPath>: è il path che conduce alla cartella dove è posizionato il checkpoint da usare per impostare i pesi del modello, precedentemente creato dalla fase di *training*.

Questa procedura produrrà un file disparities.npy, contenente tutte le **disparità** prodotte dal modello, avente avuto come input le immagini appartenenti al test set.
**Per la valutazione**:

```Bash
conda activate <nomeAmbiente>
python3 <pathFileEseguibile>/evaluate_kitti.py \
  --split eigen \
  --gt_path <datasetPath> \
  --filenames_path <fileNamesDatasetPath> \
  --predicted_disp_path <disparitiesPath>/disparities.npy \
```

Dove:
- <nomeAmbiente>: è il nome dell'ambiente **Anaconda** da dover attivare;
- <pathFileEseguibile>: è il path che conduce al file evaluate_kitti.py, file che dovrà essere eseguito per valutare il file disparities.npy, precedentemente creato dalla fase di *testing*;
- <datasetPath>: è il path che conduce alla cartella contenente il dataset;
- <fileNamesDatasetPath>: è il path che conduce alla cartella al cui interno è posizionato eigen_test_files.txt;
- <disparitiesPath>: è il path che conduce alla cartella dove è posizionato il file disparities.npy.

Questa procedura mostrerà a terminale i valori calcolati per ciascuna metrica di valutazione del modello.
Una volta seguita questa procedura ho ottenuto i seguenti risultati:

| | Minore è meglio | | | | Maggiore è meglio | | |
|---|---|---|---|---|---|---|---|
| **Fonte** | **Abs Rel** | **Sq Rel** | **RMSE** | **RMSE log** | **d1** | **d2** | **d3** |
| [1] | 0.163 | 1.399 | 6.253 | 0.262 | 0.759 | 0.911 | 0.961 |
| [1] ri-allenato | 0.164 | 1.427 | 6.369 | 0.266 | 0.757 | 0.908 | 0.960 |

Tabella 1: Confronto tra i valori della valutazione riportata in [1] e i risultati della valutazione sul modello di [1] ri-allenato.

Come si può notare i risultati sono estremamente vicini e di conseguenza [1] è stato dimostrato valido.

## 2.5. Migrazione da Tensorflow a PyTorch

Verificati i risultati ottenuti nel paper, si può quindi partire con la migrazione dell'intera *codebase* da **Tensorflow** a **PyTorch**, standard del mondo della ricerca nel campo del *machine learning*, che ci permetterà successivamente di integrare ad esso tecniche innovative, altrimenti impossibili da sperimentare.

### 2.5.1. Il dataset

La migrazione è cominciata con l'entità che governa l'approvvigionamento di immagini alla procedura di addestramento, per allenare il modello. In **PyTorch** questa entità è chiamata `Dataset` e può essere implementata mediante l'omonima interfaccia.
L'interfaccia espone i seguenti due metodi astratti:

- `__len__(self)`: il quale deve restituire la lunghezza del dataset;
- `__getitem__(self, i: int)`: il quale dato un indice, deve restituire l'elemento o gli elementi del dataset corrispondenti ad esso.

Siccome i nomi dei vari file da recuperare per il dataset sono presenti all'interno di determinati file di testo (nello specifico `eigen_train_files.txt` per il training e `eigen_test_files.txt` per il testing, secondo lo split presentato in [5]), organizzati in un formato simile al `.csv`, nell'implementazione di questa entità ho scelto di appoggiarmi alla libreria *Pandas*, la quale solitamente viene utilizzata apposta per leggere grandi file `.csv` in modo efficiente. Inoltre, grazie alle *API* di *Pandas* è molto facile reperire la dimensione del dataset (ogni riga del file di testo corrisponde ai path della coppia di immagini stereo della stessa scena), ed è molto facile dato un indice reperire i path delle corrispondenti immagini stereo.

Appoggiandomi poi alla libreria *Pillow* (standard di lettura efficiente delle immagini nel mondo **Python**) e **PyTorch**, mi sono occupato della lettura delle immagini selezionate mediante *Pandas*, della succesiva loro conversione in tensori e dell'applicazione di un eventuale *data augmentation* da applicare a questi, prima che vengano restituiti dal metodo `__getitem__`. Il `Dataset` è stato creato in modo da far restituire una tupla di tensori $(T_{\text{sx}}, T_{\text{dx}})$ se questo è in modalità *training* altirmenti, se in modalità *testing*, resitituirà solo il tensore di sinistra $T_{\text{sx}}$.

Ho implementato infine un metodo di utilità che a partire dal `Dataset` genera un `DataLoader`, il quale sarà il diretto usufruitore del `Dataset` per fornire alla procedura di addestramento i corretti batch di immagini.

### 2.5.2. I modelli

I modelli sono stati ricreati con una corrispondenza 1:1 rispetto a quanto ritrovabile nella *codebase* originale (cambia solo la sintassi con la quale sono stati implementati, dovuta solo alla differenza

di API tra **Tensorflow** e **PyTorch**), in quanto entrambe le parti devono rappresentare gli stessi modelli matematici.

Tuttavia, ho approfittato dei vari metodi, interfacce e classi che **PyTorch** offre per:
- Creare moduli, mediante l'implementazione dell'interfaccia torch.nn.Module per poter costruire l'**encoder** e il **decoder** come due moduli a se stanti, poi integrati come sotto-moduli dei modelli, per rendere il codice più leggibile e compartimentalizzato;
- Creare sequenze di blocchi o *layer*, mediante l'impiego di oggetti torch.nn.Sequential, per poter rendere il codice più semplice e sequenziale, migliorandone la leggibilità.

### 2.5.3. La procedura di *training*

Tutto il codice per il *training* è stato realizzato dentro un file apposito training.py, il quale viene eventualmente richiamato dal file main.py.
Anche nel caso della procedura di *training* c'è una corrispondenza 1:1 rispetto a quanto ritrovabile nella *codebase* originale, poichè per ottenere gli stessi risilutati è necessario che il modello segua lo stesso addestramento, tuttavia sono state applicate le seguenti scelte stilistiche e organizzative:
- Ogni funzione di perdita ha la propria funzione **Python**. Successivamente la funzione di perdita completa richiama tutte le altre la corrispondente formula matematica, così da rendere più comprensibile e compartimentalizzato il codice;
- Il codice originale fa un forte uso degli argomenti da terminale per definire le varie impostazioni, il codice migrato invece fa uso di file di configurazione scritti in **Python**, così da poter specificare anche i tipi delle varie impostazioni inseribili e da poter sfruttare il **linter** di **Python** per avere suggerimenti riguardo alle impostazioni durante la scrittura del codice. Questo approccio è stato utilizzato anche dalla **procedura di utilizzo** che dalla **procedura di valutazione**;
  - In questo modo, l'unico argomento da terminale che è possibile andare ad impostare è l'argomento --mode, il quale può essere impostato a train, use, webcam, test e eval, dove il primo specifica che si vuole addestrare il modello, mentre i successivi due si impostano nel caso della **procedura di valutazione**, mentre gli ultimi due si impostano nel caso della **procedura di utilizzo**.
- Rigurado al salvataggio dei *checkpoint*, ho scelto di salvare sia l'ultimo *checkpoint* che quello della versione del modello con la valutazione migliore sul *test set*. Questo perchè dopo ogni epoca viene fatta una valutazione del modello sul *test set*.

Successivamente, come precedentemente fatto per la *codebase* originale, è stato aggiunto **Wandb** per effettuare la registrazione delle funzioni di perdita per ogni esperimento.

### 2.5.4. La procedura di utilizzo

Tutto il codice per l'utilizzo è stato realizzato dentro un file apposito using.py, il quale viene eventualmente richiamato dal file main.py.
Come per la respository originale ho fatto in modo che si possano utilizzare i modelli nei seguenti modi:
- Se, come citato nella **procedura di training** si imposta --mode=use si può fornire un secondo argomento --img_path dove si specifica il path dell'immagine di cui si vuole ottenere la mappa delle **disparità**. In questo modo verrà generata una mappa delle disparità con nome omonimo al file inserito come input, che verrà posizionata nella medesima cartella del file di input;
- Se si desidera utilizzare il modello attraverso la *webcam* integrata del computer, si deve impostare --mode=webcam. Bisogna tuttavia assicurarsi di avere il comando ffmpeg disponibile mediante terminale;

Inoltre, nel caso in cui si voglia integrare il modello in un'altro programma, è stata creata la funzione use() la quale, una volta forniti come parametri: il modello da utilizzare, l'immagine sotto forma di immagine *Pillow* o tensore di **PyTorch**, le dimensioni delle immagini accettate dal modello, le dimensioni originali dell'immagine e il dispositivo sulla quale si vuole eseguire il modello (cuda o cpu), restituisce in output un tensore di **PyTorch**, rappresentante la mappa delle **disparità**.

### 2.5.5. La procedura di valutazione

Tutto il codice per la valutazione è stato realizzato dentro un file apposito evaluating.py, il quale viene eventualmente richiamato dal file main.py. Tutto il codice per il *testing* è stato realizzato dentro un file apposito testing.py, il quale viene eventualmente richiamato dal file main.py.

La procedura di valutazione si divide in due parti:

- *testing*: la fase di *testing* si occupa di fornire le predizioni per tutte le immagini del *test set*, e di salvarle in un file chiamato disparities.npy;
- valutazione: la fase di valutazione si occupa di analizzare il file disparities.npy, al fine di produrre delle valutazioni sulle metriche presentate in [5].

La procedura di testing è stata riscritta completamente sempre con corrispondenza 1:1 con la *codebase* originale per poter sfruttare poi le stesse procedure di valutazione. Infatti le procedure di valutazione, essendo scritte in **Python** utilizzando solamente *Numpy*, non sono dipendenti da un framework di *machine learning* specifico e non sono quindi state migrate, ma tenute come sono.

## 2.6. Esplorazione degli iperparametri e *data augmentation*

# Glossario

***Anaconda***: Distribuzione del linguaggio di programmazione Python per la computazione scientifica, che cerca di semplificare la gestione dei pacchetti e la messa in produzione del software. **5.**, **6.**, **7.**

***CUDA***: **C**ompute **U**nified **D**evice **A**rchitecture è un'architettura hardware per l'elaborazione parallela creata da NVIDIA. **6.**

***stereocamera***: Particolari tipi di fotocamere dotate di due obbiettivi paralleli. Questo tipo di fotocamera viene utilizzata per ottenere due immagini della stessa scena a una distanza nota. Queste immagini vengono successivamente introdotte in un algoritmo che, cercando di trovare la corrispondenza dei vari pixel tra le due immagini e conoscendo la distanza tra i due obbiettivi, triangola la profondità di tali pixel. **1.**

***LiDAR***: Strumento di telerilevamento che permette di determinare la distanza di una superficie utilizzando un impulso laser. **1.**

***MDE***: Monocular depth estimation, è il campo che si occupa di trovare soluzioni in grado di stimare le profondità a partire da una sola immagine in input. **1.**, **2.**

***PyTorch***: Libreria *open source* per l'apprendimento automatico sviluppata da Meta AI. **2.**, **3.**, **8.**, **9.**

***Python***: Linguaggio di programmazione interpretato con tipizzazione dinamica e forte, diventato standard per la scrittura di codice orientato al *machine learning* e alla *data science*. **6.**, **8.**, **9.**

***Tensorflow***: Libreria *open source* per l'apprendimento automatico sviluppata da Google Brain. **2.**, **3.**, **6.**, **8.**, **9.**

***Wandb***: Sistema online per il logging e la gestione dei log mediante *report*, per registrare l'andamento di variabili di interesse, specialmente utilizzato nel campo del *machine learning*. **6.**, **9.**

***Adam***: L'Adaptive Moment Estimation è un ottimizzatore che utilizzando stime adattive del momento di primo e secondo ordine (media e varianza dei gradienti) aggiorna i pesi, migliorando la velocità e stabilità della convergenza durante l'addestramento dei modelli di apprendimento profondo. **5.**

***cuDNN***: **cu**da **D**eep **N**eural **N**etwork è una libreria sviluppata da NVIDIA, che espone una serie di primitive per permettere l'esecuzione di codice accellerata su schede video NVIDIA, specialmente utile per reti neurali profonde. **6.**

***decoder***: Rete neurale che ha lo scopo di analizzare un input compresso da un **encoder**, per generare la predizione. **3.**, **4.**, **9.**

***disparità***: Nel contesto delle fotocamere stereoscopiche, la disparità è la differenza nella posizione orizzontale di un pixel tra due immagini catturate da due fotocamere posizionate ad una certa distanza l'una dall'altra. Questa differenza è causata dalla variazione di angolo con cui ogni fotocamera vede gli oggetti nella scena. **3.**, **4.**, **5.**, **7.**, **9.**

***embedded***: Un dispositivo si dice *embedded* quanto, è progettato per eseguire operazioni di elaborazione e analisi dei dati localmente, vicino alla fonte dei dati stessi, piuttosto che inviarli a un server centrale o al cloud. **1.**, **3.**

***encoder***: Rete neurale che comprime un input in una rappresentazione di dimensioni ridotte, estraendo le caratteristiche essenziali. **3.**, **4.**, **9.**

***feature map***: Il risultato delle operazioni del kernel sull'immagine, rappresentando le caratteristiche rilevate come bordi e texture. **4.**

***kernel***: Matrice di pesi utilizzata per filtrare l'immagine, eseguendo operazioni di somma e prodotto su sotto-regioni dell'immagine per estrarre caratteristiche come bordi, texture e dettagli. **3.**, **4.**, **12.**

***linter***: Strumento che analizza il codice sorgente per individuare errori, bug, stile non conforme e altri problemi di qualità. **9.**

***pip***: *Package-management system* scritto in *Python* e usato per installare e gestire pacchetti software. **5.**, **6.**

***stride***: Il passo con cui il **kernel** si sposta sull'immagine, determinando la distanza tra le posizioni successive del **kernel**. **3.**, **4.**

# Bibliografia


[1] M. Poggi, F. Aleotti, F. Tosi, e S. Mattoccia, «Towards real-time unsupervised monocular depth estimation on CPU». IEEE/JRS Conference on Intelligent Robots and Systems (IROS), 2018.

[2] M. Poggi, F. Tosi, F. Aleotti, e S. Mattoccia, «Real-time Self-Supervised Monocular Depth Estimation Without GPU». IEEE Transactions on Intelligent Transportation Systems, 2022.

[3] F. Aleotti, G. Zaccaroni, L. Bartolomei, M. Poggi, F. Tosi, e S. Mattoccia, «Real-time single image depth perception in the wild with handheld devices», vol. 21. 2021.

[4] Clément Godard, Oisin Mac Aodha, Michael Firman, e Gabriel J. Brostow, «Real-time Self-Supervised Monocular Depth Estimation Without GPU». CVPR, 2017.

[5] E. David, P. Christian, e F. Rob, «Depth Map Prediction from a Single Image using a Multi-Scale Deep Network». Advances in neural information processing systems, 2014.