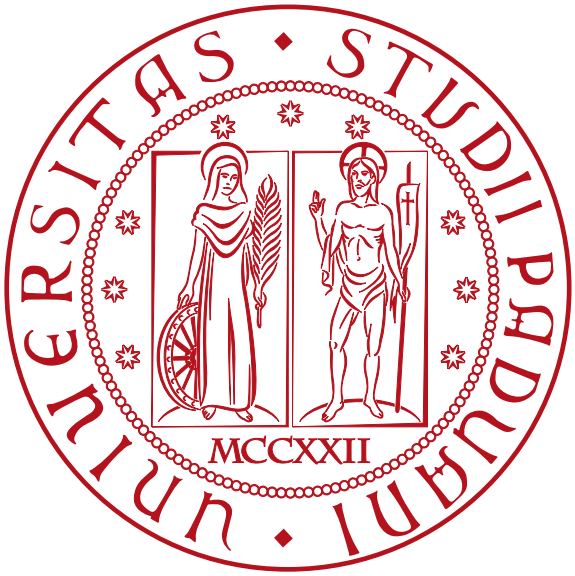
Università degli studi di Padova

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

Co-relatore
Dott. Elena Izzo

Laureando
Riccardo Toniolo

Matricola 2042332

ANNO ACCADEMICO 2023-2024

A mia madre e mio padre, i migliori che potessi avere

Sommario

Questa tesi esplora la predizione della profondità utilizzando tecniche di *deep learning* con immagini provenienti da una telecamera monoculare. Durante uno stage di 320 ore presso il gruppo di ricerca VIMP Group dell'Università degli Studi di Padova, sono stati sviluppati, implementati e validati diversi modelli di rete neurale, tra cui PyDNet e XiNet. È stata successivamente creata PyXiNet, una famiglia di modelli, con l'obiettivo di migliorare la precisione e l'efficienza della stima della profondità.

In particolare, la tesi si è focalizzata su:

- PyDNet: Migrazione del modello da TensorFlow a PyTorch con successiva validazione dei risultati;
- XiNet: Studio, validazione e impiego di un modello con un'architettura più efficiente;
- Moduli di Attenzione: Studio, implementazione e impiego di moduli di attenzione per migliorare le prestazioni dei modelli;
- PyXiNet: esplorazione di combinazioni dei moduli e modelli precedentemente citati.

I risultati dimostrano che l'uso di tecniche di *deep learning* per la stima della profondità da immagini monoculari è promettente, con miglioramenti significativi apportati dall'integrazione di moduli di attenzione.

Indice

Elenco delle Figure

Elenco delle Tabelle

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 dalle 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.

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, relativa al prezzo, 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 o mantenere.

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 in *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, in quanto fortemente leggero (meno di 2 milioni di parametri) e discretamente performante, considerata 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 e V2:
    - 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 PyDNet V1 per verificare i risultati enunciati nel *paper*;
    - Migrare tutto il codice di PyDNet V1 e il codice del modello di PyDNet V2 da TensorFlow a PyTorch;
    - Riprodurre l'allenamento di PyDNet V1 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 PyDNet V1 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 *link* ipertestuali interni al documento utilizzano la seguente formattazione: parola;
- Ogni dichiarazione, risultato o prodotto proveniente da letteratura scientifica, viene accompagnato da una citazione a tale fonte mediante un indice, utilizzabile per rintracciare

la fonte di tale dichiarazione mediante la bibliografia, situata alla fine del presente documento;
- I termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

La presente tesi è organizzata come segue:
- **Il secondo capitolo**: descrive in primo luogo l'architettura, il funzionamento e l'addestramento di PyDNet V1, in secondo luogo descrive il processo di migrazione del modello da **TensorFlow** a **PyTorch** e la successiva verifica di validità del prodotto del processo di migrazione.
- **Il terzo capitolo**: descrive l'architettura, il funzionamento e la validazione di XiNet, una rete neurale convoluzionale parametrica creata per essere particolarmente efficiente in termini di consumo energetico.
- **Il quarto capitolo**: descrive l'architettura, il funzionamento e i casi d'uso di meccanismi di attenzione quali la *self attention* e i *convolutional block attention module* nell'ambito della *computer vision*.
- **Il quinto capitolo**: descrive l'approccio esplorativo nella costruzione di PyXiNet, investigando vari tipi di architetture, progettate mediante iterazioni successive basate sui risultati delle precedenti, al fine di trovare una miglior alternativa a PyDNet.
- **Il sesto capitolo**: analizza i risultati ottenuti dai vari modelli di PyXiNet, sia dal punto di vista quantitativo che qualitativo;
- **Il settimo capitolo**: descrive i traguardi raggiunti e le conclusioni deducibili.

2

PyDNet

PyDNet (**Py**ramidal **D**epth **Net**work) è una famiglia di modelli composta da due versioni [4], [5] (che per comodità verranno riferite come PDV1 e PDV2), che cercano di risolvere il problema della MDE mediante un approccio non supervisionato, con circa 2 milioni di parametri in PDV1 e circa 700.000 in PDV2.

Una loro caratteristica interessante infatti, per l'applicazione in sistemi di tipo embedded, è proprio il numero di parametri estremamente basso. 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, come ad esempio nei cellulari [6].

Questa loro 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.

Di conseguenza i segueti sottocapitoli hanno la seguente funzione:

- Sezione 2.1: descrive l'architettura del modello e dei suoi sotto-componenti;
- Sezione 2.2: descrive come il modello cerca di risolvere il problema dell'MDE, analizzando funzioni di perdita e procedura di allenamento;
- Sezione 2.3: descrive tutta la procedura di configurazione dell'ambiente necessario al fine di poter eseguire il codice originale di PDV1;
- Sezione 2.4: descrive il processo di validazione del codice pubblicato, per verificare che effettivamente conduca a risultati simili a quelli del *paper*;
- Sezione 2.5: descrive il processo di migrazione di PDV1, da TensorFlow a PyTorch, con la sua successiva validazione, su tutti gli scenari proposti dal *paper*;
- Sezione 2.6: descrive in primo luogo degli esperimenti condotti sugli iperparametri di PDV1 per analizzare poi le conseguenze che avranno sulle metriche di valutazione, e in secondo luogo PDV2 per verificare come questo modello si comporta rispetto alla sua versione precedente, a parità di modalità di addestramento.

2.1 Architettura del modello

PDV1 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*) e 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 2×2 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* la mappa di disparità del livello).

L'architettura è quindi la seguente:

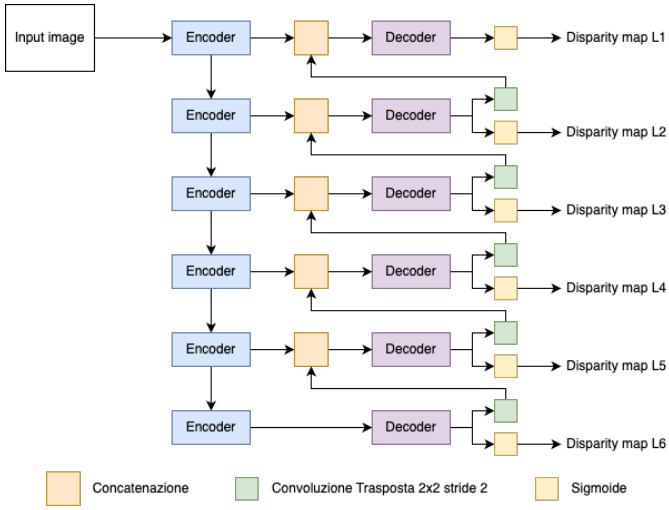


Figura 1: Architettura del modello PDV1

L'encoder è composto da una convoluzione con kernel di dimensione 3×3 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 3×3. 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:

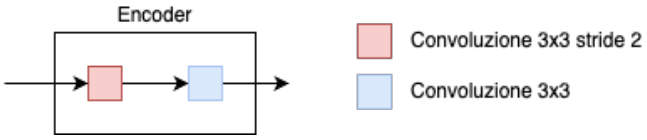


Figura 2: Encoder del modello PDV1.

Il decoder invece è composto da una successione di quattro convoluzioni con kernel di dimensione 3×3 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:

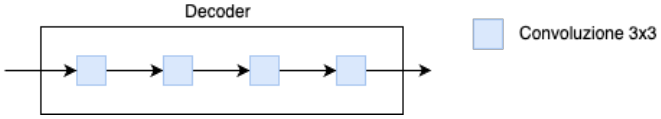


Figura 3: Decoder del modello PDV1.

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 PDV1 e PDV2 è quello di riuscire a portare a termine il compito di MDE. Per farlo 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 α 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 ($\mathcal{L}_{\text{ds}}$):

$$\mathcal{L}^l_{\text{ds}} = \frac{1}{N}\sum_{i,j} |\delta_x d^l_{i,j}| e^{-\|\delta_x I^l_{i,j}\|} + |\delta_y d^l_{i,j}| e^{-\|\delta_y I^l_{i,j}\|}$$

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 ($\mathcal{L}_{\text{lr}}$):

$$\mathcal{L}^l_{\text{lr}} = \frac{1}{N}\sum_{i,j} \left| d^l_{i,j} - d^r_{i,j+d^l_{i,j}} \right|$$

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. Questo viene reso possibile andando a penalizzare differenze tra le $d^l_{i,j}$, e le $d^r_{i,j}$ ma con uno sfasamento applicato sull'asse orizzontale di una quantità corrispondente a $d^l_{i,j}$ (ovvero lo stesso sfasamento che viene applicato alle immagini di destra per renderle quanto più simili alle immagini di sinistra).

Funzione di perdita completa ($\mathcal{L}_{\text{lr}}$):

Le precedenti funzioni di perdita vengono calcolate anche per l'immagine di destra, per poi essere combinate nel seguente modo, creando la funzione di perdita completa $\mathcal{L}_s$:

$$\mathcal{L}_s = \alpha_{\text{ap}}(\mathcal{L}^l_{\text{ap}} + \mathcal{L}^r_{\text{ap}}) + \alpha_{\text{ds}}(\mathcal{L}^l_{\text{ds}} + \mathcal{L}^r_{\text{ds}}) + \alpha_{\text{lr}}(\mathcal{L}^l_{\text{lr}} + \mathcal{L}^r_{\text{lr}})$$

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 i codici dei modelli PDV1 e PDV2 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 configurazione citata, 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 pacchetti 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 scheda 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) e ho proceduto con il configurare una macchina con tali librerie e *driver*.

Infine bisogna scaricare il *dataset* KITTI, il quale verrà 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 PDV1 e Monodepth[7], si possono recuperare le istruzioni per effettuare l'esecuzione dell'allenamento, il *testing* e la successiva valutazione.

Quindi, una volta impostata una *codebase* come scritto nella documentazione di PDV1 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**
PDV1 *paper*	**0.163**	**1.399**	**6.253**	**0.262**	**0.759**	**0.911**	**0.961**
PDV1 ri-allenato	0.164	1.427	6.369	0.266	0.757	0.908	0.96

Tabella 1: PDV1 vs. PDV1 ri-allenato

Come si può notare i risultati sono estremamente vicini e di conseguenza il *paper* [4] 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 in PDV2 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 [8]), 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 analizzare 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 nell'ecosistema 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_{sx}.

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

2.5.2 I modelli

I modelli di PDV1 e PDV2 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, di modo da 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 configurazione

Il codice originale fa un forte uso degli argomenti da terminale per definire le varie impostazioni di esecuzione del programma, mentre il codice migrato 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.

Nel mio caso ho scritto un *file* di configurazione `ConfigHomeLab.py` per la configurazione del programma di modo da poterlo eseguire sul mio computer di casa, e un *file* di configurazione `ConfigCluster.py` per poterlo invece eseguire sul cluster di dipartimento.

Di conseguenza le fasi di *training*, utilizzo e valutazione hanno tutte bisogno di due argomenti da terminale:

- `--mode`: che serve a specificare la modalità di esecuzione del codice (se per l'allenamento, utilizzo o valutazione);
- `--env`: che serve a specificare la configurazione da utilizzare (nel mio caso tra `ConfigHomeLab`, ovvero la scelta di default se non viene inserito niente e `ConfigCluster`).

2.5.4 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 è implementata nella propria funzione Python. Successivamente la funzione di perdita completa richiama tutte le altre, come la corrispondente formula matematica, così da rendere più comprensibile e compartimentalizzato il codice;
- 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.

2.5.5 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 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 corrispondente.

2.5.6 La procedura di valutazione

Tutto il codice per la valutazione e per il *testing* è stato realizzato dentro i corrispondenti *file* `evaluating.py` e `testing.py`, i quale vengono eventualmente richiamati 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 [8].

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.

2.5.7 Risultati della migrazione

Per eseguire il codice bisogna innanzitutto avere un ambiente Anaconda con tutte le dipendenze necessarie, e per farlo bisogna eseguire i seguenti comandi da terminale:

```Bash
conda create -n <nomeAmbiente>
conda activate <nomeAmbiente>
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c nvidia
pip install wandb pandas matplotlib Pillow
```

Dove `<nomeAmbiente>` è il nome che diamo all'ambiente Anaconda che poi utilizzeremo. Se il proprio *computer* usa CUDA `11.8`, bisogna sostituire nei comandi sovrastanti `pytorch-cuda=12.1` con `pytorch-cuda=11.8`

Successivamente bisogna impostare il *file* di configurazione che si decide di usare, fornendo tutte le impostazioni richieste (entrambi i *file* sono commentati al di sotto di ogni impostazione per spiegare che valori riporvi).

Fatto ciò possiamo allenare il modello tramite il seguente comando:

```Bash
python3 main.py --mode=train --env=<configurazioneUsata>
```

Dove `<configurazioneUsata>` è il nome della configurazione che decidiamo di utilizzare.

Dopo averlo allenato avremo due *file* di *checkpoint* generati nella directory specificata all'interno del *file* di configurazione. Sempre nel *file* di configurazione dobbiamo ora specificare quale *checkpoint* dobbiamo utilizzare.

Una volta specificato il *checkpoint* da utilizzare testiamo il modello con il seguente comando:

```Bash
python3 main.py --mode=test --env=<configurazioneUsata>
```

Questa procedura avrà generato un *file* `disparities.npy` nella directory specificata all'interno del *file* di configurazione.

Ora possiamo andare a valutare l'*output* del modello (ovvero l'*output* della fase precedente), utilizzando il seguente comando:

```
python3 main.py --mode=eval --env=<configurazioneUsata>
```
Bash

Seguendo questi passaggi ho ottenuto i seguenti risultati:

	Minore è meglio				Maggiore è meglio		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	0.163	**1.399**	6.253	0.262	0.759	0.911	0.961
PDV1 in PyTorch	**0.16**	1.52	**6.229**	**0.253**	**0.782**	**0.916**	**0.964**

Tabella 2: PDV1 vs. PDV1 riscritto in PyTorch (*training* su *KITTI*, 50 epoche)

Ho poi allenato il modello sul *dataset CityScapes* per poi fare ***fine tuning*** sul *dataset KITTI*, ottenendo i seguenti risultati:

	Minore è meglio				Maggiore è meglio		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	0.148	**1.318**	5.932	0.244	0.8	0.925	**0.967**
PDV1 in PyTorch	**0.147**	1.378	**5.91**	**0.242**	**0.804**	**0.927**	**0.967**

Tabella 3: PDV1 vs. PDV1 riscritto in PyTorch (*training* su *CityScapes+KITTI*, 50 epoche)

Infine ho allenato il modello per 200 epoche, ottenendo i seguenti risultati:

	Minore è meglio				Maggiore è meglio		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	**0.153**	**1.363**	**6.03**	0.252	**0.789**	**0.918**	0.963
PDV1 in PyTorch	**0.153**	1.473	6.23	**0.251**	**0.789**	**0.918**	**0.964**

Tabella 4: PDV1 vs. PDV1 riscritto in PyTorch (*training* su *KITTI*, 200 epoche)

Si può quindi constatare che la migrazione a **PyTorch** è stata un successo, permettendoci di arrivare nell'intorno dei risultati enunciati in [4], in tutti gli esperimenti proposti.

2.6 Iperparametri e PyDNetV2

2.6.1 Esplorazione degli iperparametri

È stata condotta una procedura investigativa relativa a come il cambiamento degli iperparametri influenzi le *performance* del modello, per poter capire poi come eventualmente migliorare la procedura di training al fine di avere un modello che a parità di architettura abbia una valutazione migliore.

Sono quindi state analizzate le seguenti situazioni date le seguenti ipotesi:

- Cosa succede se il *dataset* è in bianco e nero? Questa ipotesi va a verificare come la semplificazione del *dataset* impatta la prestazioni del modello. Uso il termine semplificare, in quanto la rappresentazione delle immagini passa dall'essere (secondo la norma $\text{numCanali} \times \text{altezzaImmagine} \times \text{larghezzaImmagine}$) $3 \times H \times W$ a $1 \times H \times W$, andando a rappresentare con quell'unico canale, solamente la luminosità del *pixel*. I risultati sono i seguenti:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	**0.16**	**1.52**	**6.229**	**0.253**	**0.782**	**0.916**	**0.964**
PDV1 *Luminance dataset*	0.164	1.553	6.423	0.263	0.763	0.906	0.959

Tabella 5: In PyTorch: PDV1 vs. PDV1 (*Luminance dataset*)

Come si può vedere, i risultati peggiorano, questo a significare che avere effettivamente i canali dei colori introduceva informazione significativa.

- Cosa succede se il *dataset* è in formato *HSV*? Il formato *HSV* è un formato di rappresentazione dei colori che si appoggia sempre su tre canali, ma rispetto alla rappresentazione *RGB*, i tre in questo caso sono usati per rappresentare tonalità, saturazione e luminosità. Si cerca quindi di capire se un'altra rappresentazione dei colori possa portare beneficio alle *performance* del modello. I risultati sono i seguenti:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	**0.16**	**1.52**	**6.229**	**0.253**	**0.782**	**0.916**	**0.964**
PDV1 *HSV dataset*	0.237	2.451	8.259	0.377	0.605	0.798	0.899

Tabella 6: In PyTorch: PDV1 vs. PDV1 (*HSV dataset*)

Si può quindi evincere dai risultati che il miglior formato per la rappresentazione dei colori per un buon addestramento del modello è l'*RGB*.

- Cosa succede se si applica un ribaltamento verticale alle immagini? In questo caso si vuole verificare come vengono impattate le prestazioni del modello, aggiungendo il ribaltamento verticale alle immagini. Questo perchè alla base è presente l'ipotesi di una possibile migliore generalizzazione del modello, se addestrato anche su scenari poco probabili, poichè ad esempio, abituarsi al far si che il cielo sia sempre nelle parti superiori delle immagini, creerà nel modello un'influenza forte. I risultati sono i seguenti:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	0.16	1.52	**6.229**	**0.253**	**0.782**	**0.916**	**0.964**
PDV1 *VFlip*	**0.159**	**1.403**	6.277	0.26	0.765	0.909	0.961

Tabella 7: In PyTorch: PDV1 vs. PDV1 (con *vertical flip*)

Si nota che, sebbene le prime due metriche di valutazione sono leggermente migliorate, nel resto le prestazioni degradano significativamente.

- Cosa succede se si applica un ribaltamento verticale alle immagini rimuovendo il ribaltamento orizzontale? Volendo provare a vedere che conseguenza avrebbero portato i soli ribaltamenti verticali, rimuovendo quindi quelli orizzontali, ho ottenuto i seguenti risultati:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	**0.16**	**1.52**	**6.229**	**0.253**	**0.782**	**0.916**	**0.964**
PDV1 *VFlip* no *HFlip*	0.173	1.636	6.536	0.269	0.746	0.9	0.957

Tabella 8: In PyTorch: PDV1 vs. PDV1 (con *vertical flip* senza *horizontal flip*)

Si può quindi notare che mettere il modello in condizioni poco probabili, non lo aiuta a generalizzare meglio, ma introduce solo più confusione.

- Cosa succede al cambiare della dimensione delle immagini di *input*? Si vuole in questo caso verificare come gradualmente aumentando la dimensione delle immagini, sulla quale il modello viene addestrato, si modificano le metriche di valutazione per il modello. Sono riportati in seguito i risultati per ciascuna risoluzione di *input* provata:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1 512×256	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV1 640×192	0.163	1.525	6.203	0.251	0.778	0.916	0.963
PDV1 1024×320	0.151	1.373	5.919	0.245	0.794	0.923	0.966
PDV1 1280×384	**0.139**	**1.249**	**5.742**	**0.234**	**0.816**	**0.932**	**0.969**

Tabella 9: In PyTorch: confronto tra varie risoluzioni di *input* per PDV1

Si può notare come aumentando la dimensione delle immagini di *input*, le metriche di valutazione migliorano notevolmente. È tuttavia da fare un'osservazione.

Avere immagini di *input* più grandi determina una *performance* in termini di tempo di inferenza e in termini di consumo di memoria molto peggiore, è quindi preferibile concentrarsi sulla costruzione di un modello migliore su un *input* di dimensioni accettabili, che sull'uso di modelli leggeri ma costosi per la dimensione dell'*input* che elaborano.

2.6.2 PyDNet V2

PDV2 è un modello ancor più leggero rispetto alla versione precedente, infatti si passa dai 2 milioni di parametri per PDV1 a circa 700.000 parametri per PDV2.

Questo è stato reso possibile rimuovendo solamente gli ultimi due livelli della piramide, senza cambiare ne gli encoder ne i decoder, andando quindi ad avere la seguente architettura:

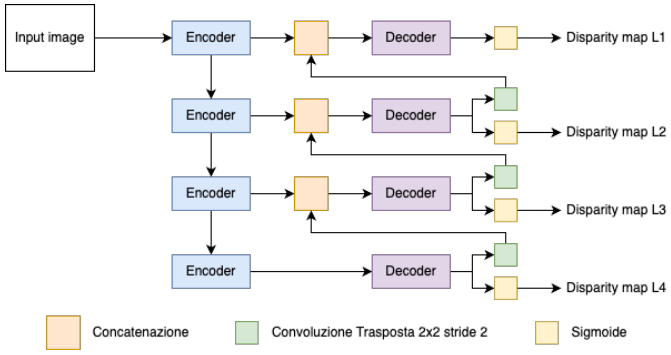


Figura 7: Architettura del modello PDV2

Sebbene la procedura di allenamento di PDV2 differisce dalla procedura di PDV1, si è voluto verificare a parità di ambiente di addestramento, la performance del modello di PDV2 migrato anch'esso in PyTorch (come menzionato precedentemente), ottenendo quindi i seguenti risultati:

	Minore è meglio				**Maggiore è meglio**		
Fonte	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	0.16	1.52	6.229	**0.253**	0.782	0.916	**0.964**
PDV2	**0.157**	**1.487**	**6.167**	0.254	**0.783**	**0.917**	**0.964**

Tabella 10: In PyTorch: PDV1 vs. PDV2

Dai risultati si può dedurre che forse avere sei livelli invece che quattro, avendo di conseguenza circa 1.3 milioni di parametri in più, fa imparare al modello più rumore che informazione utile.

3

XiNet

Il seguente capitolo parla di XiNet [1], una rete neurale convoluzionale profonda, parametrizzata, orientata all'efficienza energetica per compiti relativi alla *computer vision*. Il *paper* tratta in primo luogo lo XiConv ovvero un blocco convoluzionale parametrizzato che combina tecniche alternative per riuscire a migliorare l'efficienza energetica, rispetto ad una tradizionale convoluzione, e in secondo luogo XiNet ovvero una rete neurale che combina gli XiConv per riuscire ad ottenere il massimo dei risultati.

Di conseguenza nelle seguenti sezioni verranno trattati:

- Sezione 3.1: L'architettura del blocco XiConv e della rete XiNet;
- Sezione 3.2: La validazione dei risultati espressi nel paper;

3.1 Architettura

Il blocco convoluzionale XiConv è detto parametrico in quanto sono in esso impostabili due parametri:

- α: è il coefficiente di riduzione dei canali. Se per esempio si utilizza uno XiConv con $C_{\text{in}} = 16$ e $C_{\text{out}} = 32$ e l'α è impostato a 0.4, i veri canali di *input* e di *output* accettati saranno rispettivamente $C_{\text{in}} = \lfloor \alpha 16 \rfloor = 6$ e $C_{\text{out}} = \lfloor \alpha 32 \rfloor = 12$;
- γ: è il coefficiente di compressione. Questo perchè la convoluzione principale, effettuata dal blocco XiConv è in realtà divisa in due passi:
  1. Comprimere il numero di canali dell'*input* da αC_{in} a $\frac{\alpha C_{\text{in}}}{\gamma}$ mediante una convoluzione *pointwise* (quindi con kernel di dimensione 1×1);
  2. Successivamente applicare la convoluzione principale, con kernel 3×3, che porta il numero di canali da $\frac{\alpha C_{\text{in}}}{\gamma}$ a αC_{out}.

Tra la convoluzione di compressione e quella principale è presente una somma tensoriale tra l'*output* della convoluzione di compressione e l'*input* originale passato alla rete, passato in *broadcasting* e propriamente ridimensionato nei canali (mediante una convoluzione *pointwise*) e nelle dimensioni (mediante un *pooling* medio adattivo) dal seguente blocco di elaborazione:

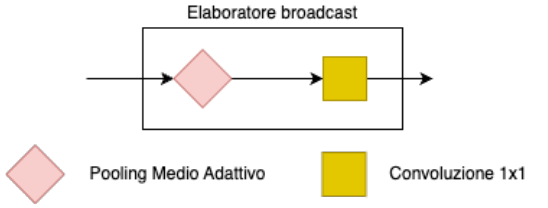


Figura 8: Architettura blocco di elaborazione del *broadcasted input* in [1]

Infine, successivamente alla convoluzione principale, viene applicato un blocco di attenzione mista, dove viene combinata l'attenzione sui canali a quella spaziale, il cui *output* moltiplicherà l'*output* della convoluzione principale mediante **prodotto di Hadamard** (per appunto applicare l'attenzione calcolata), restituendo quindi l'*output* finale dell'intero blocco.

L'architettura è la seguente:

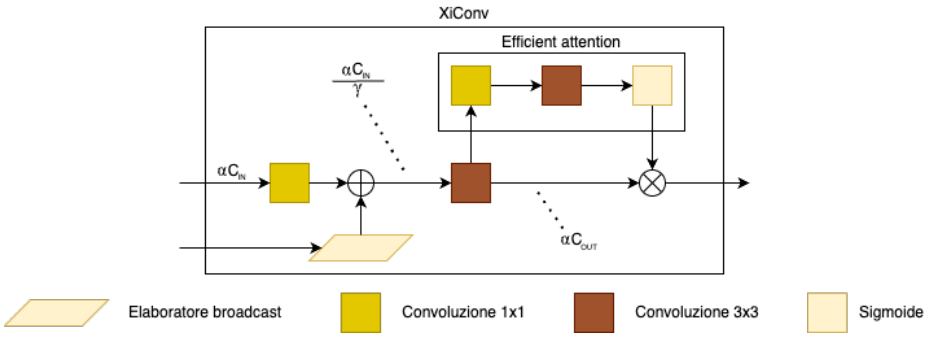


Figura 9: Architettura dello XiConv proposto in [1]

La rete XiNet è invece detta parametrica in quando sono in essa impostabili i seguenti parametri:

- β: è il coefficiente che controlla il compromesso tra numero di parametri e operazioni. Questo perchè, fatta eccezione del primo blocco XiConv della rete, gli altri blocchi seguono la seguente formula per calcolare i propri canali di *output* (e di conseguenza i canali di *input* del blocco successivo):

$$C_{\text{out}}^{i} = 4\left\lfloor \alpha 2^{D_i - 2}\left(1 + \frac{(\beta - 1)i}{N}\right) C_{\text{out}}^{0} \right\rfloor$$

Dove:

- C_{out}^{i}: rappresenta il numero dei canali di *output* che avrà il blocco i-esimo;
- D_i: rappresenta il numero di volte che l'*input* è stato dimezzato nelle dimensioni, prima del blocco i-esimo;

    - Questo perchè per ogni coppia successiva di blocchi XiConv, il primo blocco va ad applicare la propria convoluzione principale con uno **stride** 2 (dimezzando l'altezza e la larghezza del tensore di *input*);
    - Vengono sempre aggiunti due XiConv all'inizio della rete, a prescindere dal numero di XiConv specificati, quindi ci sarà sempre almeno un dimezzamento delle dimensioni.
- N: il numero di XiConv utilizzati nella rete.

L'architettura della rete è quindi la seguente:

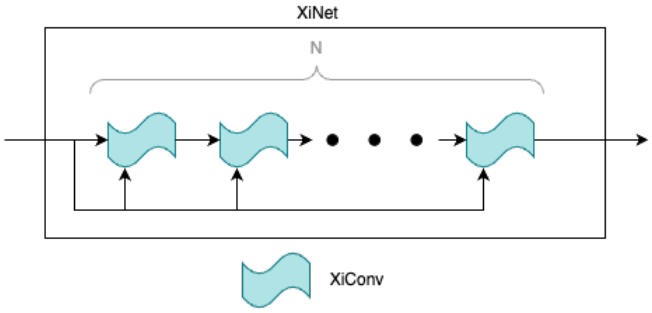


Figura 10: Architettura di XiNet, composta da N XiConv

Come si può osservare dalla figura e come precedentemente menzionato, l'*input* della rete viene poi passato in *broadcast* ad ogni XiConv che la compone.

3.2 Validazione

Purtroppo non viene menzionato alcun *benchmark* rigurardo alle prestazioni sul *dataset CIFAR10*, si può tuttavia fare una comparazione con lo stato dell'arte per vedere quanto si discosta da esso.

Si è voluto quindi verificare il risultato andando a svolgere i seguenti passi:

- Clonare la *repository* da **GitHub**, mediante il comando:

```bash
git clone https://github.com/micromind-toolkit/micromind.git
```

- Installare il pacchetto *micromind* (pacchetto nella quale è presente XiNet) in locale, mediante i seguenti comandi:

```bash
# Per entrare nella directory clonata
cd ./micromind/
# Per installare il pacchetto micromind
pip install -e .
# Per installare requisiti e dipendenze aggiuntive
pip install -r ./recipes/image_classification/extra_requirements.txt
```

- Effettuare l'allenamento con successiva valutazione mediante il seguente comando:

```bash
# Per entrare nella cartella corretta
cd ./recipes/image_classification/
# Per eseguire l'allenamento con successiva valutazione
python train.py cfg/xinet.py
```

Seguendo i precedenti comandi ho quindi ottenuto un' *accuracy* del 81.44% con ∼7.8 milioni di parametri. Il modello presentato in [9] è in cima alle classifiche con un'*accuracy* del 99.61% con 11 milioni di parametri.

Possiamo quindi constatare come, seppur con ∼3 milioni di parametri in meno, riesca ad avvicinarsi ai risultati dello stato dell'arte. Ovviamente XiNet non ha lo scopo di essere migliore in termini di *accuracy*, ma di minimizzare l'impatto energetico del modello, cercando di avere performance quanto più vicine ai modelli con le valutazioni migliori.

I risultati sono quindi soddisfanceti al fine di provare ad esplorare, con questo modulo, eventuali soluzioni alternative per trovare una soluzione migliore di PDV1 e PDV2 nel campo del MDE.

4

Attention

Le reti neurali convoluzionali, sebbene funzionino particolarmente bene per essere addestrate su immagini e video, hanno un problema alla base, ovvero non sono in grado di considerare delle dipendenze a lungo raggio tra i vari *pixel* o *patch* del contenuto in analisi. Alcune delle conseguenze che questo comporta sono:

- Un campo ricettivo limitato alla dimensione del kernel;
- Il campo ricettivo cresce linearmente con la profondità della rete, e quindi potrebbero essere necessarie reti estremamente profonde, perchè il modello riesca ad analizzare un contesto globale;
- I pesi dopo l'addestramento del modello rimangono fissi e non possono quindi adattarsi dinamicamente al contesto globale dell'immagine.

Tuttavia, negli anni precedenti una particolare tecnica (originariamente pensata per il natural language processing) è emersa anche nel campo della *computer vision*, l'attenzione. I meccanismi di attenzione cercano di captare dipendenze a lungo raggio al fine di migliorare, spesso significativamente, la qualità della predizione.

Il contenuto delle seguenti sezioni è riassumibile come segue:

- Sezione 4.1: descrive un potente ma computazionalmente costoso meccanismo di attenzione, il quale però permette miglioramenti significativi nelle predizioni, andando a trovare correlazioni tra ogni *pixel* o *patch* del contenuto analizzato;
- Sezione 4.2: descrive un meccanismo di attenzione più leggero, creato apposta per essere utilizzato in reti a collo di bottiglia, cercando di estrarre informazioni significative, prima della riduzione di dimensionalità, analizzando prima la dimensione dei canali, e poi la dimensione spaziale (altezza e larghezza).

4.1 *Self attention*

La *self attention* è sostanzialmente un meccanismo originariamente creato per riuscire a trovare correlazioni tra le parole di una frase, di modo da assegnare dei pesi a ciascuna di esse per fornire una migliore predizione [10].

Tuttavia nel lavoro discusso in [2], viene descritta un'architettura che permette l'uso di principi simili, al fine di trovare correlazioni ad ampio raggio tra i vari *pixel* dei contenuti analizzati. Viene quindi enunciato il concetto di *operatore non locale* ovvero il blocco presentato in [2] che

calcola la risposta in una posizione come una media ponderata delle caratteristiche di tutte le posizioni dell'*input*.

L'architettura del *Non-local block* è la seguente:

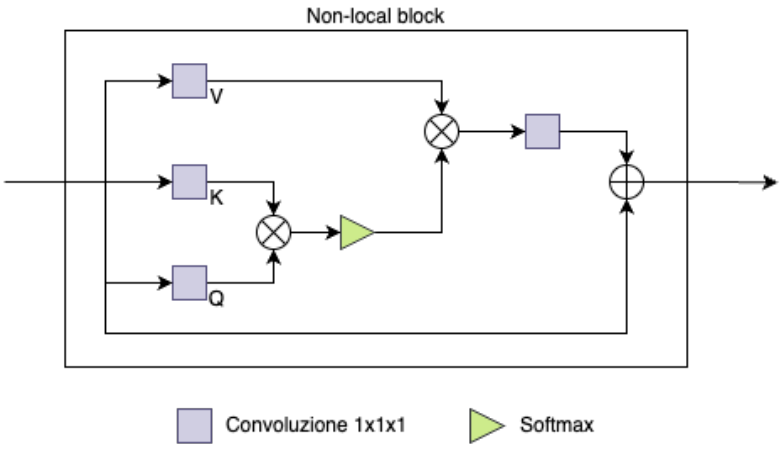


Figura 11: Il *Non-local block*, presentato in [2]

In questo caso "⊗" rappresenta la moltiplicazione tra matrici.

Come in [10], anche in questo blocco si identificano tre entità per riuscire ad ottenere l'attenzione:

- **Query** (Q): rappresenta la posizione per la quale stiamo calcolando l'*output* non locale;
- **Key** (K): rappresenta le posizioni con cui la query viene confrontata per determinare la somiglianza;
- **Value** (V): rappresenta le informazioni che vengono aggregate per formare l'*output* finale.

Si vuole far notare che in questo caso le tre matrici sono matrici della dimensione di $\text{numCanali} \times (\text{altezza} \cdot \text{larghezza})$, questo per far si che le moltiplicazioni matriciali combinino il valore di un *pixel* con tutti gli altri.

La prima moltiplicazione tra matrici $Q \cdot K$ rappresenta il calcolo dell'affinità, ovvero calcolare i pesi di attenzione che determinano l'influenza di ogni *pixel* (la colonna di K presa in considerazione) sul *pixel* in analisi (la riga di Q presa in considerazione). Successivamente le affinità calcolate vengono normalizzate tramite una funzione *softmax* per ottenere i veri pesi di attenzione A.

Il prodotto matriciale $A \cdot V$ invece va a creare un'aggregazione ponderata delle informazioni, dando quindi luogo alla matrice di attenzione Y.

Questa viene poi moltiplicata per un peso apprendibile (equivalente ad applicare una convoluzione $1 \times 1 \times 1$), anche chiamato γ, che serve per apprendere quanto applicare dell'attenzione calcolata.

Essendo in seguito fatta una somma tensoriale tra l'*input* (anche detto *residual*, come menzionato in [11]) e il risultato del calcolo dell'attenzione, il γ può anche partire da 0, andando

quindi inizialmente a non applicare attenzione, questo per stabilizzare inizialmente il *training* del modello.

4.2 *Convolutional Block Attention Module*

Il *Convolutional Block Attention Module* (CBAM), presentato in [3] è un blocco fortemente basato sulle convoluzioni, che combina la *channel* e la *spacial attention*, per migliorare le predizioni del modello.

Elementi di particolare interesse per questo modello sono:

- La capacità di riuscire comunque, anche se non come per la *self attention*, nel catturare relazioni globali;
- Aumentare l'efficienza computazionale poichè aggiunge, rispetto ad altri meccanismi di attenzione, solo un discreto numero di parametri alla rete e nessuna moltiplicazione matriciale.

Il blocco CBAM è principalmente composto da due sotto-moduli, uno per calcolare l'attenzione sui canali, che una volta combinato mediante un **prodotto di Hadamard** con il *residual* dell'*input*, viene passato al modulo di attenzione spaziale. L'*output* di attenzione spaziale viene poi combinato con il *residual* dell'*output* del modulo di attenzione sui canali mediante un **prodotto di Hadamard**, generando quindi l'*output* con l'attenzione applicata.

L'architettura del blocco CBAM è quindi la seguente:

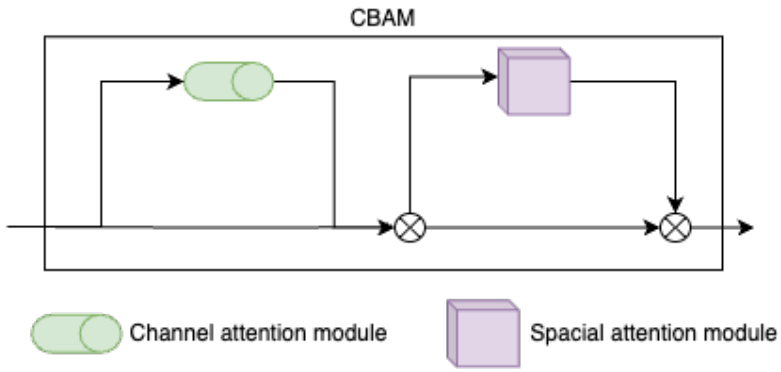


Figura 12: Il *Convolutional Block Attention Module*, presentato in [3]

Il modulo di attenzione sui canali, opera sulla spazialità del tensore (altezza e larghezza) mediante *pooling* massimo e *pooling* medio, andando quindi a creare due vettori, i quali rappresenteranno il valore massimo e la media di ciascun canale del tensore di *input*.

Successivamente entrambi questi vettori vengono passati attraverso un *multi layer perceptron* da tre strati, con il primo e l'ultimo dalle medesime dimensioni dell'*input*, mentre quello centrale, essendo più piccolo, applica un fattore di compressione.

Entrambi i vettori risultanti vengono poi sommati tra loro e passati attraverso una sigmoide, la quale genererà il vettore di attenzione sui canali.

L'architettura è la seguente:

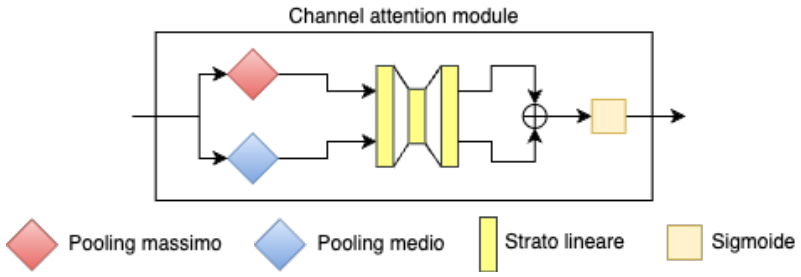


Figura 13: Il modulo di attenzione sui canali, presentato in [3]

Il modulo di attenzione spaziale invece, opera sulla profondità del tensore (i canali) mediante *pooling* massimo e *pooling* medio, andando quindi a generare due matrici, le quali rappresenteranno i valori massimi e i valori medi sui canali, per ciascun elemento nelle coordinate dell'altezza e della larghezza.

Queste matrici vengono concatenate sulla dimensione dei canali e successivamente passate attraverso una convoluzione con kernel di dimensione 3×3. Infine l'*output* di questa operazione viene passato attraverso una sigmoide che genererà quindi la matrice di attenzione sulla spazialità.

Questa è quindi l'architettura:

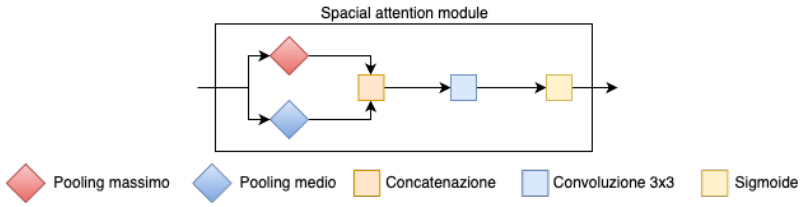


Figura 14: Il modulo di attenzione sulla spazialità, presentato in [3]

Come spesso viene fatto, in [3] viene anche citata la possibilità di aggiungere uno step dove all'attenzione calcolata viene poi sommato il *residual* dell'*input*, al fine di stabilizzare l'apprendimento. Questo lo rende particolarmente interessante per la famiglia delle reti convoluzionali ResNet [11], in quanto fortemente basata su questo meccanismo.

Si può quindi implementare un approccio ResNet, ponendo il blocco CBAM subito dopo una convoluzione, per poi sommare il suo *output* all'*output* della convoluzione iniziale.

L'architectura proposta è quindi la seguente:

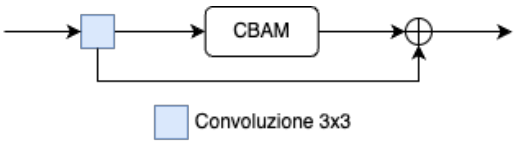


Figura 15: Applicazione del blocco CBAM con approccio *ResNet*

5

PyXiNet

PyXiNet (**Py**ramidal **Xi Net**work) è una famiglia di modelli che tenta di combinare le diverse soluzioni trattate fino ad ora per massimizzare l'efficacia e l'efficienza nel portare a termine il compito di MDE.

Il nome suggerisce che sia una rete fortemente basata sia su PyDNet che su XiNet, infatti il primo modello, in particolare la seconda versione viste le sue ottime *performance* di base, darà una direzione sullo stile dell'architettura generale, mentre il secondo cercherà di migliorare l'encoder della rete.

Il seguente capitolo andrà quindi a mostrare l'approccio sperimentale e esplorativo condotto, nel testare ipotesi e progressivamente migliorare e raffinare le architetture proposte.

5.1 PyXiNet α

PyXiNet α rappresenta il primo approccio all'uso di XiNet come encoder. In particolare sono stati realizzati due modelli, chiamati α I e α II.

Le architetture create sono le seguenti:

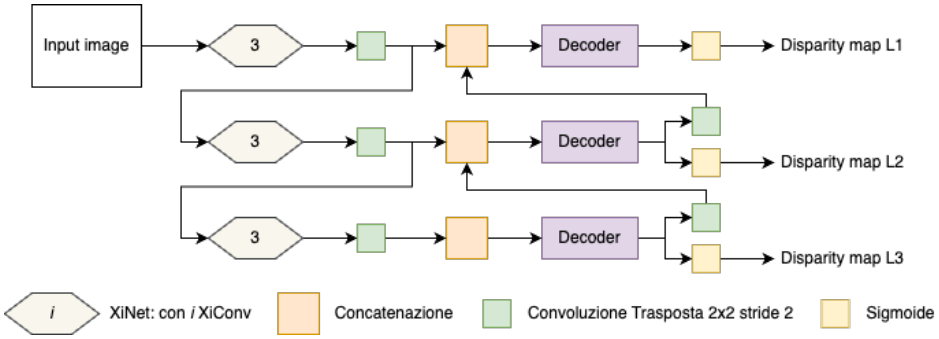


Figura 16: Architettura di PyXiNet α I

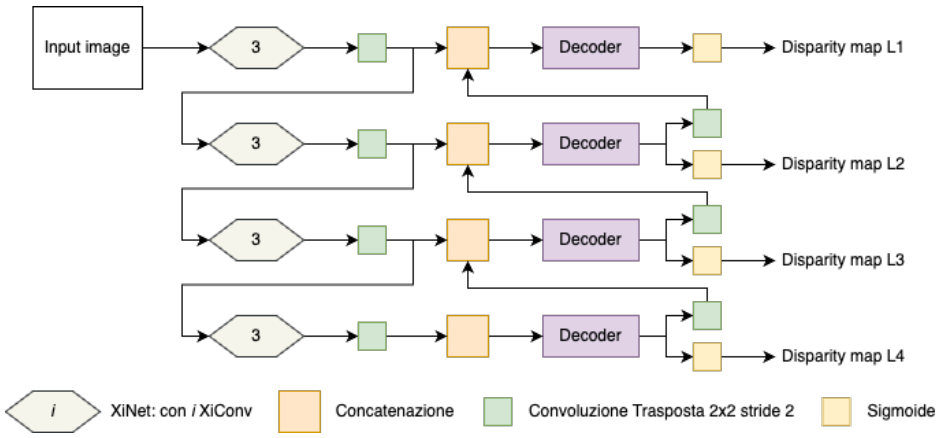


Figura 17: Architettura di PyXiNet α II

Si può notare che dopo l'uso di ogni XiNet, è presente una convoluzione trasposta, questo perchè come già discusso nel capitolo **XiNet**, l'uso di tale rete effettua una riduzione della dimensionalità spaziale (altezza e larghezza) all'inizio, e poi ne fa una per ogni coppia all'interno della rete.

La convoluzione trasposta serve per far si che la dimensionalità spaziale sia la medesima che avrebbe prodotto l'**encoder** originale di PDV1. Anche in questo caso, come per PDV1, la convoluzione trasposta è seguita da una funzione di attivazione *ReLU*, con coefficiente di crescita di 0,2 per la parte negativa.

Essendo state allenate assieme, con ciascuna si voleva fornire una risposta a diverse domande:

- XiNet se usato come **encoder** porta a buoni risultati?
- A parità di livelli nella piramide, si riesce ad avere una *performance* migliore o uguale a quella di PDV2?
- Se si rimuove un livello alla piramide, come vengono impattate le *performance*?
- L'uso di XiNet come impatta il numero di parametri e il tempo di inferenza?
  - Si vuole far notare che per rispondere a questa domanda, rispetto alle tabelle di valutazione precedenti, sono stati introdotte due nuove metriche di valutazione: numero di parametri (#p), e tempo di inferenza in secondi (Inf. (s)) (il quale viene calcolato facendo una media del tempo di inferenza su 10 immagini, passate in successione al modello e non in batch, con una elaborazione su *CPU* Intel i7-7700).

L'allenamento dei due modelli ha portato ai seguenti risultati:

	Minore è meglio						Maggiore è meglio		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	**0.253**	0.782	0.916	**0.964**
PDV2	716680	**0.1**	**0.157**	**1.487**	**6.167**	0.254	**0.783**	**0.917**	**0.964**
PyXiNet α I	**429661**	0.14	0.17	1.632	6.412	0.269	0.757	0.903	0.958
PyXiNet α II	709885	0.12	0.168	1.684	6.243	0.259	0.777	0.913	0.96

Tabella 11: PDV1 e PDV2 vs. PyXiNet α

Da come si può osservare, sebbbene il numero di parametri sia stato intorno a quelli di PDV2, se non minore, tutte le altre metriche sono fortemente peggiorate, andando a suggerire che forse non è quello il migliore uso di XiNet come encoder.

5.2 PyXiNet β

Dato l'insuccesso dato della tipologia α, la tipologia β cerca di utilizzare al meglio XiNet, al fine di migliorare almeno una metrica di valutazione.

Per quanto scritto in XiNet, le reti proposte all'interno del paper sono composte da almeno cinque XiConv. Questo suggerisce come la profondità sia un elemento essenziale per il successo nel suo utilizzo.

La famiglia β è composta da quattro varianti, due da tre livelli e due da quattro livelli, tutte con una leggera variazione rispetto all'architettura a piramide tradizionale, questo per riuscire a rispondere alle seguenti domande:

- La profondità aiuta XiNet nel migliorare le *performance* del modello?
- XiNet è un encoder efficace?
- Parallelizzare gli encoder può rendere più veloce il tempo di inferenza rispetto ad averli in serie?

Le architetture proposte sono quindi le seguenti:

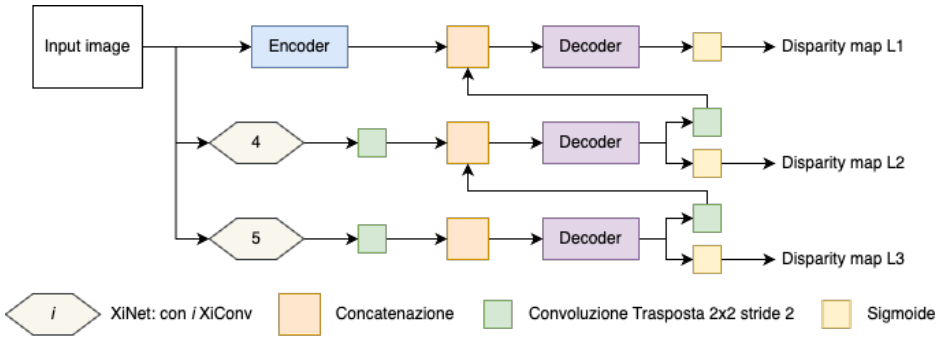


Figura 18: Architettura di PyXiNet β I

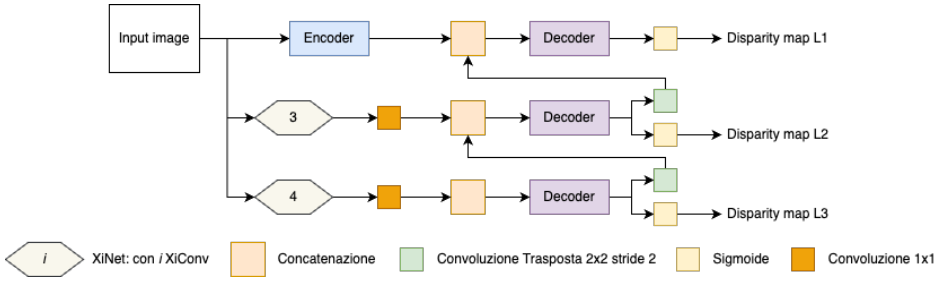


Figura 19: Architettura di PyXiNet β II

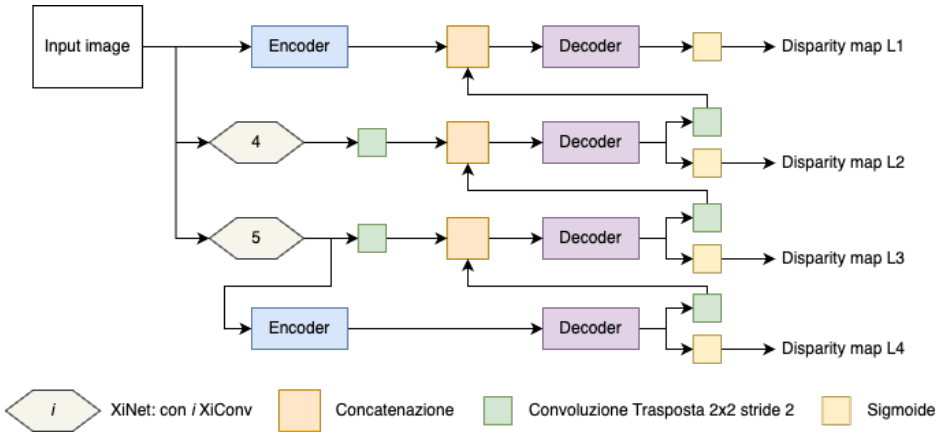


Figura 20: Architettura di PyXiNet β III

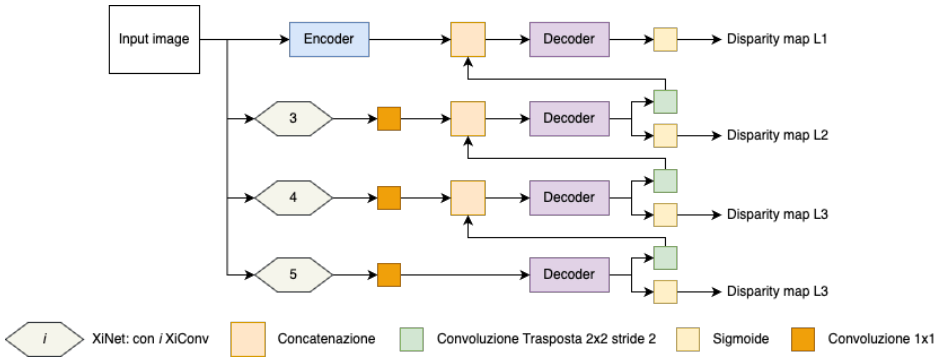


Figura 21: Architettura di PyXiNet β IV

Come si può notare, tutte le architetture della famiglia β hanno gli encoder in parallelo. Questo è stato fatto per due motivi:

1. Vedere se la parallelizzazione del grafo della rete neurale migliorava il tempo di inferenza, potendo eseguire i vari livelli in parallelo (invece di fare aspettare ad ogni livello i livelli superiori);
2. Permettere agli encoder composti da XiNet, di diventare più profondi. Questo perchè come spiegato nel capitolo XiNet, per ogni coppia successiva di blocchi XiConv viene dimezzata la dimensionalità spaziale (altezza e larghezza). Non dipendendo ogni livello da quello precedente, possiamo sfruttare questa caratteristica per far si che ogni nuovo encoder si occupi del ridimensionamento della risoluzione del proprio livello.

Il modello β I si ispira ad α I, andando però semplicemente a parallelizzare gli encoder per quindi permettere delle XiNet più lunghe successivamente.

Il modello β II rispetto a β I cerca di verificare se un eventuale problema potrebbe essere l'uso della convoluzione trasposta, come metodo per aggiustare la risoluzione del tensore. Per verificare ciò le XiNet sono di un blocco più corte rispetto a β I, e per far si che si ottenga lo stesso numero di canali di prima, si usa una convoluzione con kernel di dimensione 1×1.

I modelli β III e β IV vanno semplicemente ad aggiungere un livello in più nella piramide, rispetto ai corrispondenti β I e β II, così da verificare che eventualmente a parità di numero di livelli si riescano ad ottenere *performance* simili, se non migliori, a quelle di PDV2.

Le due reti però utilizzano approcci differenti per aggiungere un nuovo livello:

- β III aggiunge un encoder tradizionale di PDV1 che è posto in serie con l'encoder del livello superiore, questo per riuscire ad avere un risparmio sul numero di parametri (poichè avere una XiNet profonda sei blocchi XiConv, aumenta significatamente la grandezza della rete);
- β IV invece aggiunge un'ulteriore XiNet di un blocco più lunga rispetto al suo livello precedente, seguita da una convoluzione con kernel di dimensioni 1×1 per il ridimensionamento dei canali.

Con le architetture precedentemente discusse ho ottenuto i seguenti risultati:

	Minore è meglio						**Maggiore è meglio**		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV2	716680	**0.1**	0.157	1.487	6.167	0.254	0.783	0.917	0.964
PyXiNet β I	941638	0.16	0.156	1.546	6.259	0.251	0.791	0.921	0.965
PyXiNet β II	**481654**	0.14	0.168	1.558	6.327	0.259	0.762	0.91	0.963
PyXiNet β III	1246422	0.16	0.148	1.442	**6.093**	**0.241**	**0.803**	**0.926**	**0.967**
PyXiNet β IV	1446014	0.18	**0.146**	**1.433**	6.161	**0.241**	0.802	**0.926**	**0.967**

Tabella 12: PDV1 e PDV2 vs. PyXiNet β

In questo caso i risultati migliorano su tutte le metriche per i modelli β III e β IV, tranne per il tempo di inferenza e per il numero di parametri. Tuttavia notiamo che sebbene il numero di parametri di β III non è minore di quello di PDV2, è di un $\sim 37\%$ inferiore a quello di PDV1 e il tempo di inferenza di questi due modelli è molto simile. Questo è quindi un buon punto di partenza per poter applicare meccanismi di attenzione, in grado di migliorare ulteriormente le prestazioni del modello.

5.3 PyXiNet 𝕄

Essendo coscienti che la *self attention* è parecchio costosa in termini di tempo e quindi un'opzione non praticabile in contesti dove il tempo di inferenza deve essere corto e la potenza computazionale limitata, la famiglia 𝕄 va in realtà semplicemente a provare e verificare quale valore aggiunto questo meccanismo di attenzione può portare nel miglioramento delle metriche di valutazione, senza avere alcune pretese sul poter essere applicata come soluzione per il caso d'uso MDE *embedded*.

Nel tenativo di implementazione di una *self attention* meno impattante computazionalmente, rispetto al normale modulo di attenzione presentato in [2] e affrontato nella Sezione 4.1, ho realizzato un blocco chiamato *Light Self Attetion Module* (LSAM) che rispetto al *Self Attention Module* (SAM) discusso precedentemente, applica dell'interpolazione ad *area* per ridimensionare la risoluzione del tensore in ingresso.

Ne sono state realizzate due versioni per riuscire a capire dove utilizzare l'interpolazione ad *area*, al fine di ottenere i migliori risultati.

Le architetture delle due versioni di *LSAM* sono le seguenti:

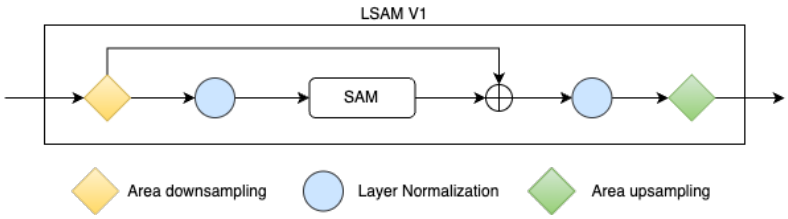


Figura 22: Architettura di *LSAM V1*

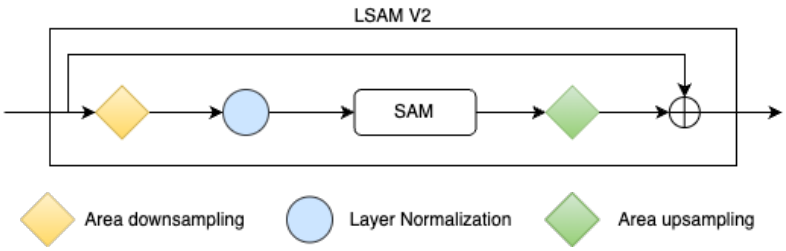


Figura 23: Architettura di *LSAM V2*

Nel primo caso l'applicazione dell'attenzione avviene nell'ambiente a risoluzione ridotta, nel secondo caso invece si vuole sfruttare la maggiore informazione presente nell'input, andando quindi ad applicarla nell'ambiente a risoluzione piena. È stata poi aggiunta la normalizzazione dei tensori, come consigliato in [12].

Come modello di partenza per la famiglia 𝕄 è stato scelto il modello β IV, per vedere appunto come peggiora il suo tempo di inferenza e quanto migliorano le sue metriche di valutazione.

Le architetture della famiglia $\mathbb{M}$ sono le seguenti:

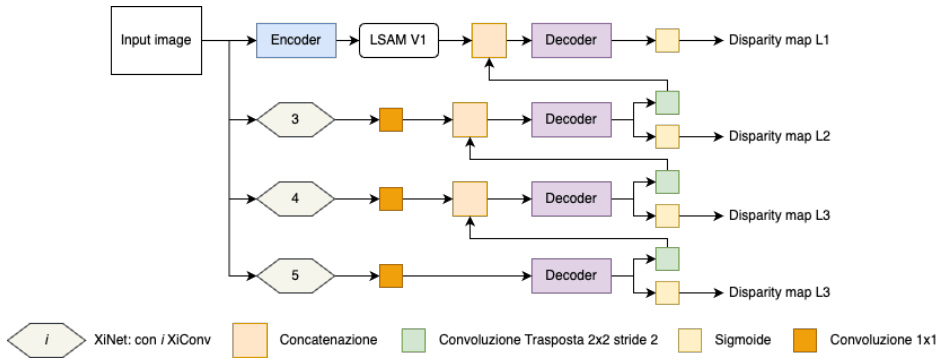


Figura 24: Architettura di PyXiNet $\mathbb{M}$ I

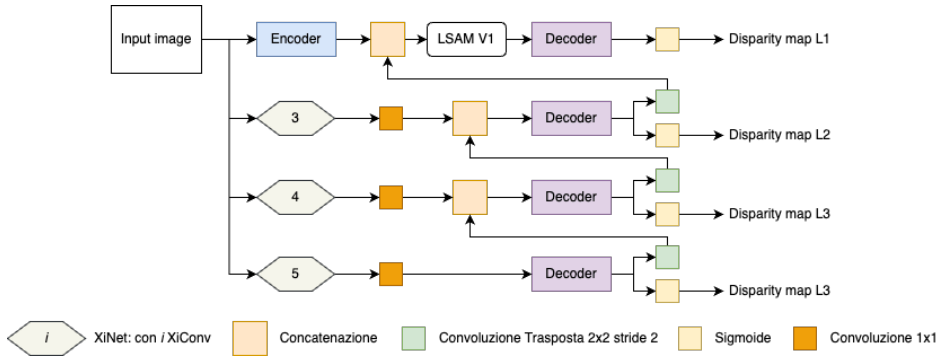


Figura 25: Architettura di PyXiNet $\mathbb{M}$ II

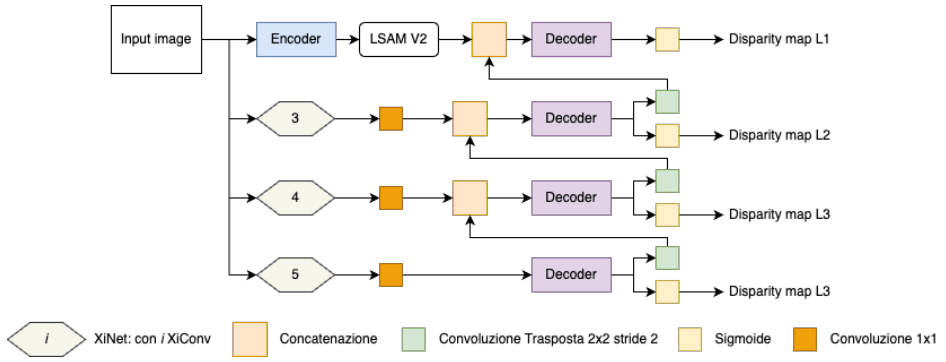


Figura 26: Architettura di PyXiNet $\mathbb{M}$ III

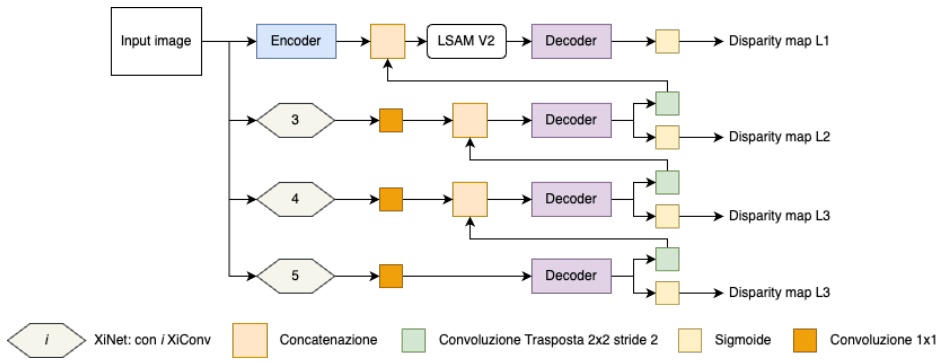


Figura 27: Architettura di PyXiNet $\mathbb{M}$ IV

Da quanto osservabile, si può notare come il blocco di attenzione sia stato posizionato nell'encoder del primo livello, questo in quanto si tratta del livello che opera alla risoluzione massima e quindi il livello che opera su quanta più informazione vicina all'input originale del modello. Entrambe le versioni di *LSAM* sono state posizionate sia prima che dopo la concatenazione, per riuscire a trovare la posizione migliore tra le due.

I risultati ottenuti per la famiglia $\mathbb{M}$ sono quanto viene riportato in seguito:

	Minore è meglio						**Maggiore è meglio**		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV2	**716680**	**0.1**	0.157	1.487	6.167	0.254	0.783	0.917	0.964
PyXiNet β IV	1446014	0.18	0.146	1.433	6.161	0.241	0.802	0.926	0.967
PyXiNet $\mathbb{M}$ I	1970643	0.36	0.147	1.351	5.98	0.244	0.8	0.926	0.967
PyXiNet $\mathbb{M}$ II	2233197	0.38	**0.14**	1.289	**5.771**	**0.234**	**0.814**	**0.933**	**0.969**
PyXiNet $\mathbb{M}$ III	1708499	0.35	0.141	1.279	5.851	0.239	0.808	0.927	0.968
PyXiNet $\mathbb{M}$ IV	1839981	0.36	0.145	**1.25**	5.885	0.242	0.798	0.926	0.967

Tabella 13: PDV1, PDV2 e PyXiNet β IV vs. PyXiNet $\mathbb{M}$

Da quanto si può notare nei risultati, le *performance* di tutta la famiglia $\mathbb{M}$, superano significativamente quelle di β IV. Si può altresì osservare che il tempo di inferenza è tuttavia almeno raddoppiato rispetto a β IV per tutti i modelli.

Sempre dai risultati ottenuti si può anche capire che solo nella coppia di modelli $\mathbb{M}$ I - $\mathbb{M}$ II, mettere il blocco *LSAM* dopo la concatenazione ha portato a risultati migliori, infatti nella coppia $\mathbb{M}$ III - $\mathbb{M}$ IV è successo l'esatto opposto.

5.4 PyXiNet βCBAM

Come discusso nel capitolo ***Attention*** nella Sezione 4.2, uno dei vantaggi principali del blocco CBAM è il suo efficiente uso di risorse garantendo comunque un buon miglioramento delle *performance*.

Sappiamo tuttavia che, per quanto citato in [3], è particolarmente adatto nell'implementazione all'interno di reti di tipo *ResNet*. Questa informazione ha quindi guidato il design di un nuovo decoder, chiamato per l'appunto *CBAM Decoder* (CBAMD). È stato scelto il decoder come posto dove applicare più volte il blocco, poichè rispecchia l'architettura di una rete neurale convoluzionale a collo di bottiglia.

L'architettura del nuovo decoder è come segue:

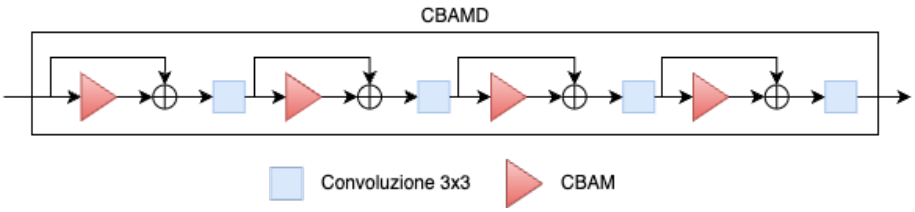


Figura 28: Architettura del decoder CBAMD

È osservabile l'approccio residuale dell'*input*, tra due convoluzioni successive.

La famiglia di modelli βCBAM è basata, da come si può evincere dal nome, sui modelli β, in particolare β III e β IV, i quali sono stati i modelli con i migliori risultati ottenuti (senza considerare i risultati ottenuti mediante l'applicazione della *self attention*).

Le architetture dei due modelli derivanti dalle appena discusse scelte sono i seguenti:

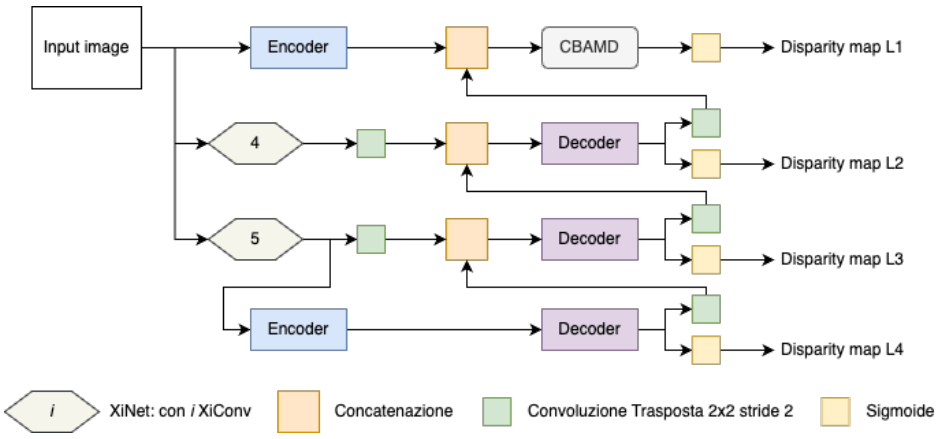


Figura 29: Architettura di PyXiNet βCBAM I

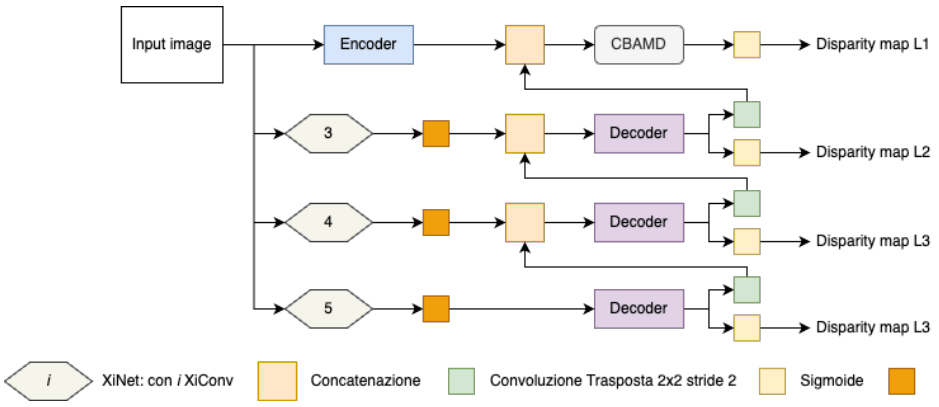


Figura 30: Architettura di PyXiNet βCBAM II

Con la creazione di questi modelli si vuole verificare l'effettivo beneficio che questo modulo di attenzione può apportare in rapporto a quanto viene degradato il tempo di inferenza e a quanto aumenta il numero totale del parametri.

Si può notare che il blocco CBAMD è stato utilizzato come decoder, solo del primo livello. Questa scelta è stata fatta per non aumentare di troppo i parametri e il tempo di inferenza, facendolo concentrare solamente sulla risoluzione maggiore (la risoluzione che poi verrà effettivamente utilizzata in un contesto reale).

I risultati di questi due modelli sono riportati nella seguente tabella:

	Minore è meglio						**Maggiore è meglio**		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV2	**716680**	**0.1**	0.157	1.487	6.167	0.254	0.783	0.917	0.964
PyXiNet βCBAM I	1250797	0.19	**0.143**	**1.296**	**5.91**	**0.239**	0.805	**0.928**	**0.968**
PyXiNet βCBAM II	1450389	0.23	0.147	1.379	5.974	**0.239**	**0.806**	0.927	**0.968**

Tabella 14: PDV1 e PDV2 vs. PyXiNet βCBAM

I risultati di questa famiglia di modelli come si può notare sono molto promettenti, riuscendo ad incrementare notevolmente le *performance* del modello di partenza, senza impattare troppo fortemente sul tempo di inferenza.

Il modello migliore tra i due (βCBAM I) riesce addirittura a stare sotto i 0.20s come tempo di inferenza medio (permettendo un *framerate* di circa 5 *fps*) e con un numero di parametri inferiore di un $\sim 37\%$ rispetto a PDV1.

Un'osservazione interessante che può essere fatta è che rispetto alle controparti della famiglia β, l'aggiunta dei vari blocchi CBAM ha contribuito al massimo ad un incremento del $\sim 0,4\%$ sul numero dei parametri.

5.4.1 *CBAM PyDNet*

Visti gli enormi progessi che il blocco CBAM ci permette di ottenere, sorge spontanea la messa in questione dell'utilità dell'encoder composto dal blocco XiNet.

Per verificare questa ipotesi è stata presa l'archiettura originale di PDV2, al fine di verificare se posizionando un blocco CBAM prima di ogni convoluzione della rete, con passaggio dei residuali dall'*output* della convoluzione precedente, si sarebbero potute ottenere *performance* interessanti.

Il decoder quindi rimande il blocco CBAMD, che però verrà usato ora su tutti i livelli, mentre l'encoder (*CBAME*) sarà come segue:

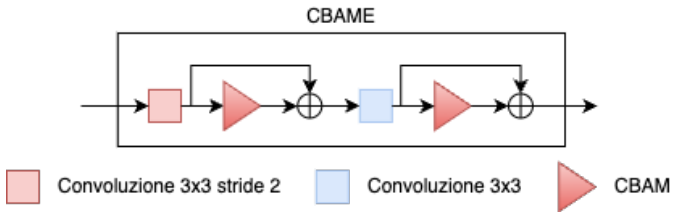


Figura 31: Architettura dell'encoder *CBAME*

L'architettura del modello risultante da quanto appena descritto è quindi quanto segue:

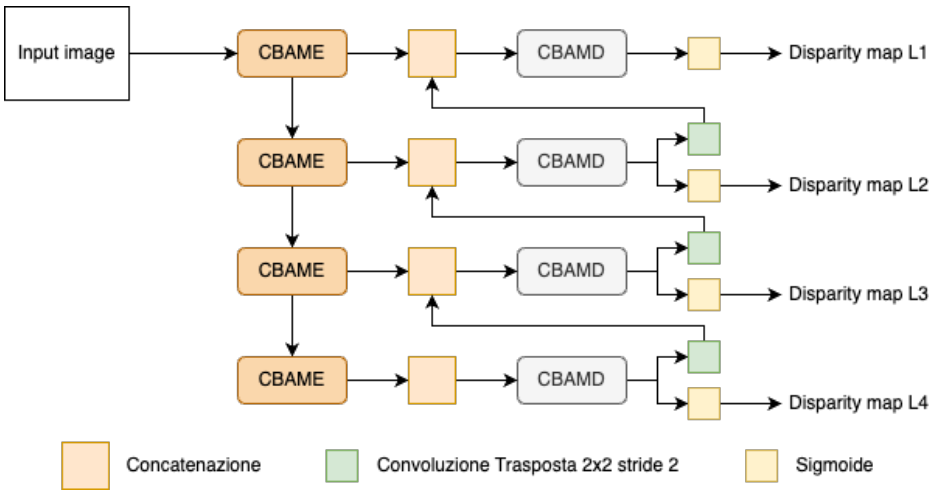


Figura 32: Architettura di *CBAM PyDNet*

I risultati di questo modello sono riportati in seguito:

	Minore è meglio						Maggiore è meglio		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV2	**716680**	**0.1**	**0.157**	**1.487**	**6.167**	0.254	**0.783**	**0.917**	0.964
CBAM PyDNet	746673	0.28	0.167	1.722	6.509	**0.251**	0.776	0.916	**0.965**

Tabella 15: PDV1 e PDV2 vs. *CBAM PyDNet*

Per quanto si può notare, posizionare il blocco CBAM dopo ogni convoluzione non fa altro che degradare pesantemente le prestazioni del modello sulla maggior parte delle metriche di valutazione. Si deduce quindi che l'uso di XiNet è sicuramente responsabile per parte del miglioramento delle predizioni dei modelli dove è stato usato.

6

Risultati

Nel seguente capitolo si farà un breve riepilogo sui risultati quantitativi ottenuti usando le metriche di valutazione usate in [8] e le due metriche in più introdotte nella valutazione dei modelli PyXiNet, soffermandosi ad analizzare casistiche interessanti. Successivamente vengono esposti come risultati qualitativi, le mappe di profondità prodotte rispettivamente da PDV1, PDV2 (riscritti in PyTorch) e i dai migliori modelli sperimentali $\mathbb{M}$ II e β CBAM I.

6.1 Risultati quantitativi

I seguenti sono tutti i risultati ottenuti dai due PyDNet e dai tredici esperimenti effettuati:

	Minore è meglio						**Maggiore è meglio**		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	0.253	0.782	0.916	0.964
PDV2	716680	**0.1**	0.157	1.487	6.167	0.254	0.783	0.917	0.964
PyXiNet α I	**429661**	0.14	0.17	1.632	6.412	0.269	0.757	0.903	0.958
PyXiNet α II	709885	0.12	0.168	1.684	6.243	0.259	0.777	0.913	0.96
PyXiNet β I	941638	0.16	0.156	1.546	6.259	0.251	0.791	0.921	0.965
PyXiNet β II	481654	0.14	0.168	1.558	6.327	0.259	0.762	0.91	0.963
PyXiNet β III	1246422	0.16	0.148	1.442	6.093	0.241	0.803	0.926	0.967
PyXiNet β IV	1446014	0.18	0.146	1.433	6.161	0.241	0.802	0.926	0.967
PyXiNet $\mathbb{M}$ I	1970643	0.36	0.147	1.351	5.98	0.244	0.8	0.926	0.967
PyXiNet $\mathbb{M}$ II	2233197	0.38	**0.14**	1.289	**5.771**	**0.234**	**0.814**	**0.933**	**0.969**
PyXiNet $\mathbb{M}$ III	1708499	0.35	0.141	1.279	5.851	0.239	0.808	0.927	0.968
PyXiNet $\mathbb{M}$ IV	1839981	0.36	0.145	**1.25**	5.885	0.242	0.798	0.926	0.967
PyXiNet βCBAM I	1250797	0.19	0.143	1.296	5.91	0.239	0.805	0.928	0.968
PyXiNet βCBAM II	1450389	0.23	0.147	1.379	5.974	0.239	0.806	0.927	0.968
CBAM PyDNet	746673	0.28	0.167	1.722	6.509	0.251	0.776	0.916	0.965

Tabella 16: Risultati di tutti gli esperimenti a confronto

Come già espresso nei capitoli precedenti, i risultati ottenuti con $\mathbb{M}$ II sono i migliori. Essendo però troppo pesante e lento come modello per essere eseguito, in un contesto embedded sicuramente βCBAM I sarebbe preferibile.

Possiamo notare però che il divario tra le *performance* dei due modelli appena menzionati non è eccessivo, soprattutto considerando il divario nei tempi di inferenza e nel numero di parametri:

	Minore è meglio						Maggiore è meglio		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PyXiNet $\mathbb{M}$ II	2233197	0.38	**0.14**	**1.289**	**5.771**	**0.234**	**0.814**	**0.933**	**0.969**
PyXiNet βCBAM I	**1250797**	**0.19**	0.143	1.296	5.91	0.239	0.805	0.928	0.968

Tabella 17: PyXiNet $\mathbb{M}$ II e PyXiNet βCBAM I a confronto

Si vuole inoltre far notare come, sebbene il modulo CBAM sia più semplice come meccanismo di attenzione rispetto alla *self attention*, non tutti gli esperimenti utilizzanti quest'ultima tecnica hanno portato a prestazioni migliori risetto alla prima:

	Minore è meglio						Maggiore è meglio		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PyXiNet $\mathbb{M}$ I	1970643	0.36	0.147	1.351	5.98	0.244	0.8	0.926	0.967
PyXiNet $\mathbb{M}$ IV	1839981	0.36	0.145	**1.25**	**5.885**	0.242	**0.98**	0.926	0.967
PyXiNet βCBAM I	**1250797**	**0.19**	**0.143**	1.296	5.91	**0.239**	0.805	**0.928**	**0.968**

Tabella 18: PyXiNet $\mathbb{M}$ I e IV vs. PyXiNet βCBAM I

Se si va invece a prendere in considerazione l'uso di XiNet come encoder, si può facilmente dedurre che è il diretto responsabile per parte dell'aumento del tempo di inferenza. Questo lo si può notare nel confronto tra i modelli α e i modelli PDV1 e PDV2:

	Minore è meglio						Maggiore è meglio		
Fonte	**#p**	**Inf. (s)**	**Abs Rel**	**Sq Rel**	**RMSE**	**RMSE log**	**d1**	**d2**	**d3**
PDV1	1971624	0.15	0.16	1.52	6.229	**0.253**	0.782	0.916	**0.964**
PDV2	716680	**0.1**	**0.157**	**1.487**	**6.167**	0.254	**0.783**	**0.917**	**0.964**
PyXiNet α I	**429661**	0.14	0.17	1.632	6.412	0.269	0.757	0.903	0.958
PyXiNet α II	709885	0.12	0.168	1.684	6.243	0.259	0.777	0.913	0.96

Tabella 19: PyXiNet $\mathbb{M}$ I e IV vs. PyXiNet βCBAM I

La famiglia α infatti anche possedendo in tutti i suoi esperimenti un numero di parametri inferiore a PDV2, ha comunque un tempo di inferenza maggiore, rispetto a quest'ultimo, di almeno il 20%. Questo è dovuto al gran numero di somme tensoriali *element wise* che le XiNet eseguono. Questo tipo di operazioni infatti, anche se non contribuiscono direttamente al far crescere il numero dei parametri, aumentano il numero di calcoli da eseguire. Di conseguenza a meno di un uso radicalmente diverso di XiNet all'interno delle architetture, rispetto a quanto provato, non sarà possibile scendere sotto il tempo di inferenza di PDV2.

6.2 Risultati qualitativi

In seguito vengono elencati quattro risultati qualitativi dei modelli precedentemente citati.

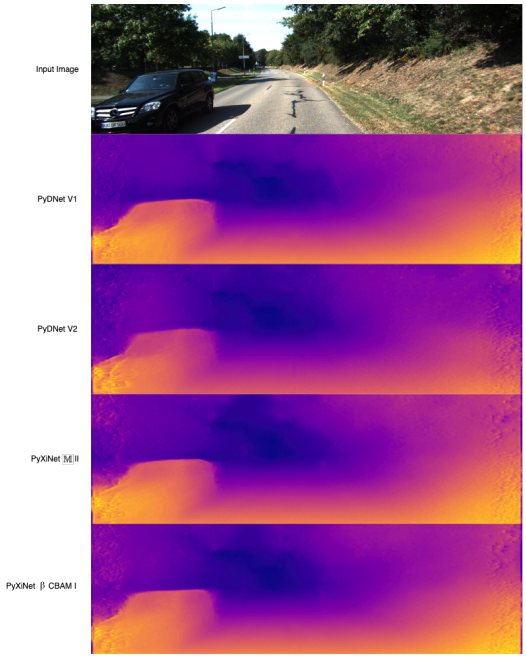


Figura 33: Inferenza sulla prima immagine.

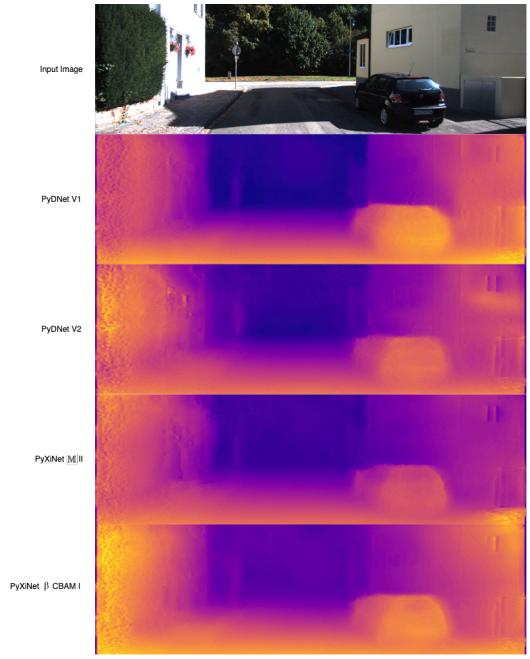


Figura 34: Inferenza sulla seconda immagine.

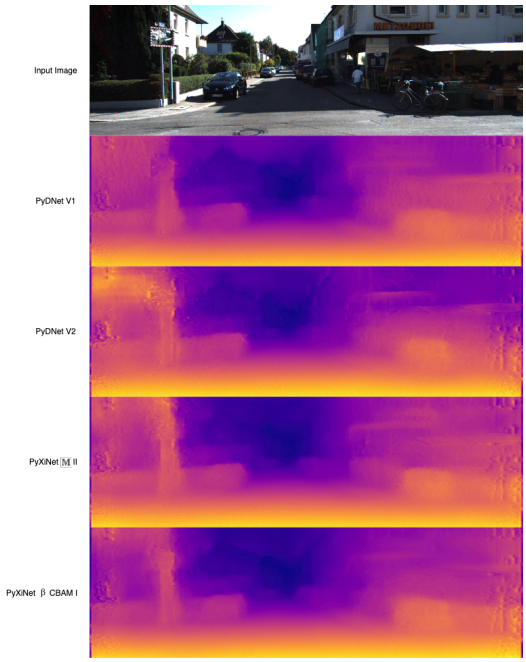


Figura 35: Inferenza sulla terza immagine.

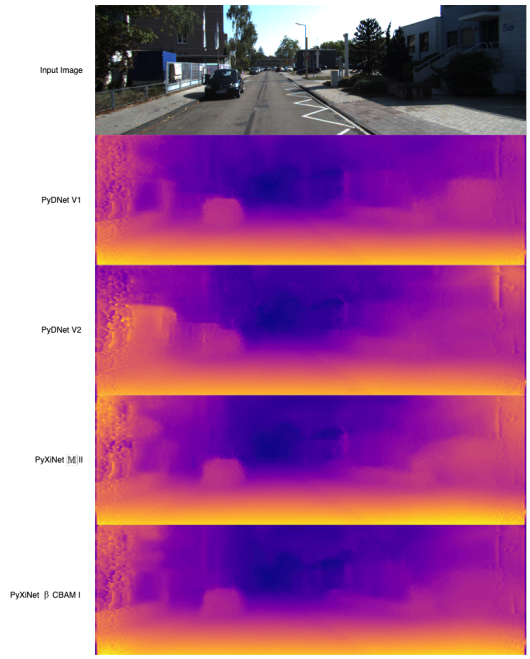


Figura 36: Inferenza sulla quarta immagine.

Sebbene i risultati siano molto simili tra loro a livello qualitativo, si può notare con occhio più attento che i modelli βCBAM I e $\mathbb{M}$ II riescono a carpire meglio le forme, al contempo riducendo artefatti e distorsioni presenti nell'immagine.

7

Conclusioni

7.1 Raggiungimento obbiettivi

Si riassumono in seguito gli obbiettivi raggiunti durante il tirocinio:

- Il modello PyDNet V1 è stato verificato nei risultati enunciati dal proprio *paper* [4];
- È stata migrata l'intera *codebase* di PyDNet V1 e il modello di PyDNet V2 dal framework TensorFlow al framework PyTorch con successo:
  - Sono stati ottenuti gli stessi risultati nell'allenamento a 50 epoche sul *dataset* KITTI;
  - Sono stati ottenuti gli stessi risultati nell'allenamento a 200 epoche sul *dataset* KITTI;
  - Sono stati ottenuti gli stessi risultati nell'allenamento a 50 epoche sul *dataset* CityScapes, con successivo fine tuning di 50 epoche sul *dataset* KITTI;
  - È stata di conseguenza acquisita conoscenza e abilità nell'uso della libreria PyTorch e del linguaggio Python, specificatamente per la creazione di modelli di *machine learning* e per la creazione di ambienti d'allenamento;
  - È stata acquisita abilità nell'uso di *cluster* di calcolo per l'allenamento dei modelli;
- È stata acquisita conoscenza relativamente a meccanismi di attenzione quali la *self attention* e i *convolutional block attention module*, e ne è stata fatta successivamente l'implementazione in codice;
- È stata acquisita conoscenza nell'implementazione di moduli convoluzionali efficienti, specialmente lo *XiConv* presentato in [1];
- È stata condotta dell'attività di ricerca e sviluppo con l'obbiettivo di implementare valide alternative ai modelli PyDNet nel contesto embedded:
  - È stato studiato come il cambiamento di vari iperparametri può portare a cambiamenti nelle prestazioni del modello;
  - Sono stati eseguiti un totale di tredici esperimenti, per tredici modelli diversi;
  - Tra i tredici modelli è stato trovato un modello particolarmente interessante (β CBAM I), che con il 37% di parametri in meno rispetto a PyDNet V1, riesce ad essere migliore rispetto a quest'ultimo, nelle metriche di valutazione proposte in [8], di un minimo di $\sim 1\%$ e di un massimo del $\sim 15\%$.

7.2 Considerazioni

In questa tesi, sono stati esplorati e sviluppati diversi modelli di rete neurale per la predizione della profondità da immagini monoculari, con l'obiettivo di migliorare la precisione e l'efficienza di tali predizioni. Le principali conclusioni raggiunte sono le seguenti:

1. **Migrazione di PyDNet**: La migrazione del modello PyDNet da TensorFlow a PyTorch ha dimostrato non solo la fattibilità di tale processo, ma ha anche permesso una più facile integrazione con modelli e blocchi più moderni;
1. **Esplorazione di XiNet**: L'introduzione di XiNet, un modello con un'architettura più efficiente, ha portato a miglioramenti significativi nella stima della profondità. Inoltre XiNet ha dimostrato di essere un modello più leggero e performante, quindi adatto per applicazioni su dispositivi con risorse computazionali limitate;
2. **Esplorazione dei Moduli di Attenzione (*CBAM* e *self attention*)**: l'integrazione di moduli di attenzione come il *Convolutional Block Attention Module* (*CBAM*) e i meccanismi di *self attention* si è verificata essere una scelta vincente al fine di migliorare le performance del modello. Si è tuttavia notato come l'utilizzo della seconda non sia ideale per contesti di tipo *embedded*, mentre l'ultilizzo della prima deve essere fatto con criterio per ottenere buoni risultati;
3. **Ricerca e sviluppo di PyXiNet**: Le varianti sperimentali PyXiNet, che combinano le architetture di PyDNet e XiNet con moduli di attenzione, hanno dimostrato progressivi miglioramenti. In particolare PyXiNet βCBAM I ha evidenziato come l'uso combinato di queste tecnologie possa portare a predizioni molto più accurate, con un piccolo compromesso sul tempo di inferenza e sul numero di parametri.

I risultati appena discussi fanno quindi capire che l'uso di tecniche di *deep learning* per la stima delle profondità da immagini monoculari si è rivelato essere molto promettente, anche in ambienti con un potenziale di risorse molto limitato come il contesto *embedded*.

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. 9., 10., 11., 15.

CUDA: **C**ompute **U**nified **D**evice **A**rchitecture è un'architettura hardware per l'elaborazione parallela creata da NVIDIA. 10., 15.

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., 2.

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. 2., 7., 34.

PyTorch: Libreria *open source* per l'apprendimento automatico sviluppata da Meta AI. vii, 2., 3., 5., 12., 13., 14., 16., 17., 18., 19., 41., 45., 46.

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*. 9., 13., 14.

TensorFlow: Libreria *open source* per l'apprendimento automatico sviluppata da Google Brain. 2., 3., 5., 9., 45., 46.

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*. 9., 14.

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. 9.

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. 10.

decoder: Rete neurale che ha lo scopo di analizzare un input compresso da un encoder, per generare la predizione. v, 6., 7., 13., 19., 36., 37.

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. 6., 7., 8., 11., 14.

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. 2., 5., 41., 45.

encoder: Rete neurale che comprime un input in una rappresentazione di dimensioni ridotte, estraendo le caratteristiche essenziali. v, 6., 7., 13., 19., 29., 30., 31., 32., 42.

fine tuning: Processo di adattamento di un modello pre-addestrato su un nuovo dataset specifico per migliorare le sue prestazioni su un compito particolare. 16., 45.

feature map: Il risultato delle operazioni di una convoluzione sull'immagine, rappresentando le caratteristiche rilevate come bordi e texture. 7.

GitHub: Piattaforma di hosting per il controllo di versione e la collaborazione, basata su Git. 23.

prodotto di Hadamard: Date due matrici dalle stesse dimensioni, la matrice risultato dell'operazione avrà le medesime dimensioni dell'input, e il valore di ogni sua cella corrisponderà al prodotto tra i valori delle celle corrispondenti nelle due matrici di input. 22., 27.

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. 6., 7., 21., 25., 28., 33., 48.

linter: Strumento che analizza il codice sorgente per individuare errori, bug, stile non conforme e altri problemi di qualità. 13.

natural language processing: Il **N**atural **L**anguage **P**rocessing è un campo dell'intelligenza artificiale che si occupa dell'integrazione tra computer e linguaggio umano. 25.

pip: *Package-management system* scritto in *Python* e usato per installare e gestire pacchetti software. 9.

stride: Il passo con cui il kernel si sposta sull'immagine, determinando la distanza tra le posizioni successive del kernel in una convoluzione. 6., 7., 23.

Bibliografia

[1] A. Alberto, P. Francesco, e F. Elisabetta, «XiNet: Efficient Neural Networks for tinyML». ICCV, 2023.

[2] Xiaolong Wang, Ross Girshick, Abhinav Gupta, e Kaiming He, «Non-local Neural Networks». CVPR, 2018.

[3] Sanghyun Woo, Jongchan Park, Joon-Young Lee, e In So Kweon, «CBAM: Convolutional Block Attention Module». ECCV, 2018.

[4] 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.

[5] 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.

[6] 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.

[7] Clément Godard, Oisin Mac Aodha, Michael Firman, e Gabriel J. Brostow, «Unsupervised Monocular Depth Estimation with Left-Right Consistency». CVPR, 2017.

[8] 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.

[9] B. Antonio, M. Davide, e M. Massimo, «Efficient Adaptive Ensembling for Image Classification». Expert Systems, Wiley, 2023.

[10] V. Ashish *et al.*, «Attention Is All You Need». NeurIPS, 2017.

[11] H. Kaiming, Z. Xiangyu, R. Shaoqing, e S. Jian, «Deep Residual Learning for Image Recognition». CVPR, 2016.

[12] D. Alexey *et al.*, «An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale». ICLR, 2021.