Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

Neural Radiance Field per la ricostruzione e la segmentazione di scene 3D da immagini 2D

Elaborato in:
Computer Graphics

Relatore:
Prof.
Damiana Lazzaro

Presentata da:
Marco Pesic

Sessione Unica
Anno Accademico 2022-2023

Per i miei genitori e mia sorella
che mi hanno sempre sostenuto...

Introduzione

Oggigiorno con lo sviluppo di un nmero sempre maggiore di tecnologie che permettono l'esplorazione autonoma di vari ambienti sia reali che virtuali, riuscire a comprendere lo spazio che circonda questi agenti intelligenti, diventa un'abilità necessaria. Lo *scene understanding* prova ad analizzare oggetti rispetto allo spazio 3D in cui si trovano, il loro posizionamento ed la loro relazione semantica e funzionale. Per svolgere questo compito di *scene understanding* il metodo più utilizzato negli ultimi anni è di sfruttare le reti neurali. Modelli di segmentazione semantica su immagini presenti in [1], [2] [3] [4] [5] permettono di distinguere i vari oggetti/entità presenti nell'immagine di input molto abilmente, attraverso la supervisione di etichette/label semantiche. Il problema risiede nella generazione di queste label semantiche che è un processo costoso e lungo, che in determinati campi può richiedere anche l'ausilio di esperti e.g.(campo medico). In alternativa si potrebbero anche utilizzare modelli di segmentazione semantica 3D in cui l'input è un volume tridimensionale, però allora la difficoltà risiede nel ottenere questi volumi tridimensionali, oltre all'aumento del costo computazionale. Quindi per ovviare a molti di questi problemi, alcuni lavori [6] [7] [8] sfruttano la capacità di modelli Neural Radiance Field (NeRF) [9] di imparare la rappresentazione volumetrica 3D della scena attraverso varie immagini della scena stessa. Attraverso questa rappresentazione volumetrica e con l'assistenza di label semantiche , il modello NeRF riesce a segmentare semanticamente l'intera scena, da notare che il numero di immagini e anche di label semantiche e di vari ordini di grandezza minori rispetto ad ordinari dataset adoperati per

la segmentazione semantica. Una volta imparata la segmentazione semantica della scena è possibile sfruttare questo modello come generatore di nuove label semantiche, per allenare modelli di segmentazione più complessi.
L'obiettivo di questa tesi è di implementare un Semantic-Nerf[9] con analogo modello di segmentazione semantica Unet[1] seguendo la linea di ricerca data da [8] e [10]. I risultati raggiunti usando questo metodo, poi mostrati dalla tabella 4.1, dimostrano la utilità di questi metodi. Il seguito della tesi è così organizzato:

1. Il primo capitolo, introduce conoscenze di background necessarie per capire il modello NeRF[9]: Funzione plenottica e Volume Rendering. Inoltre formula il task di segmentazione semantica in modo formale e introduce le metriche adoperate.

2. Il secondo capitolo è dedicato alla spiegazione del modello NeRF originale [9] , ma sono descritti anche altri lavori che apportano modifiche alla architettura originale introducendo varie migliorie.

3. Il terzo capitolo introduce Semantic-Nerf[6], [7] e le loro proprietà . Infine viene descritto un metodo molto simile a [8] per allenare una rete Unet[1].

4. Il quarto e ultimo capitolo, si occupa della parte tecnica del progetto, ovvero dell'implementazione pratica , descrivendo snippet di codice più importanti e anche dei dettagli tecnici di allenamento.

Indice

Elenco delle figure

Capitolo 1

Radiance Field e Volume Rendering

La rappresentazione 3d di scene e modelli, oggigiorno torna molto utile per molti ambiti e.g. robotica, digital twins, video games, autonomous driving. Esistono vari tipi di rappresentazione 3d Mesh, point cloud, voxels ognuno con i suoi vantaggi e svantaggi.
I Neural Radiance Field(NeRF) rappresentano la scena 3d come uno spazio volumetrico continuo (diversamente dai voxel che è discreto). In questo capitolo inizieremo a introdurre concetti necessari per comprendere i NeRF [9] come il volume rendering e radiance field.Inoltre sarà spiegato in cosa consiste il task di segmentazione semantica, che sarà realizzato mediante tecnologia NeRF.

1.1 Funzione Plenottica e Radiance Field

Per introdurre il radiance field che è definito come una funzione $L : \mathbb{R}^5 \rightarrow \mathbb{R}^3$, si introduce prima la funzione plenottica [12]. Quest'ultima è definita come una funzione

$$P : \mathbb{R}^7 \rightarrow \mathbb{R}^3$$

dove i 7 parametri di input sono

$$P(\theta, \phi, \lambda, t, Vx, Vy, Vz)$$

i primi 2 parametri θ, ϕ indicano le coordinate polari di entrata del raggio di luce nel punto di vista .λ indica lunghezza d'onda del raggio necessaria per esprimere il colore del raggio. t che raffigura il tempo , permette di rappresentare la scena in vari snapshot temporali. Gli ultimi 3 parametri Vx, Vy, Vz servono per cambiare la posizione del punto di vista. Tutte insieme questi 7 parametri permettono la rappresentazione di una scena 3d da ogni angolazione e snapshot temporale. Naturalmente implementare tale funzione è impossibile , quindi per ottenere il radiance field vengono eliminati 2 parametri il tempo t e la lunghezza d'onda λ. Di solito la funzione di radiance field viene rappresentata anche come $(L : \mathbb{R}^3 \times \mathbb{S}^2 -> \mathbb{R}^3)$.

1.2 Volume Rendering

Il volume rendering è una collezione di metodi usati in computer graphics per creare proiezioni 2D , campionando punti in uno spazio 3D. Per prima cosa bisogna capire come la luce interagisce e si propaga attraverso un volume. Si immagina la presenza di una camera (spettatore) che osserva da una particolare direzione ω. Esistono quattro fenomeni fondamentali (Fig. 1.1):

- Assorbimento: la radianza viene diminuita quindi la luce viene assorbita.

- Out-scattering: alcuni dei fotoni che attraversano il volume subiscono un cambio di direzione, non raggiungendo quindi lo spettatore, questo provoca una perdita di radianza.

- In-scattering: i fotoni proveniente da direzione diverse da ω vengono reindirizzati verso lo spettatore, questo provoca un aumento della radianza.

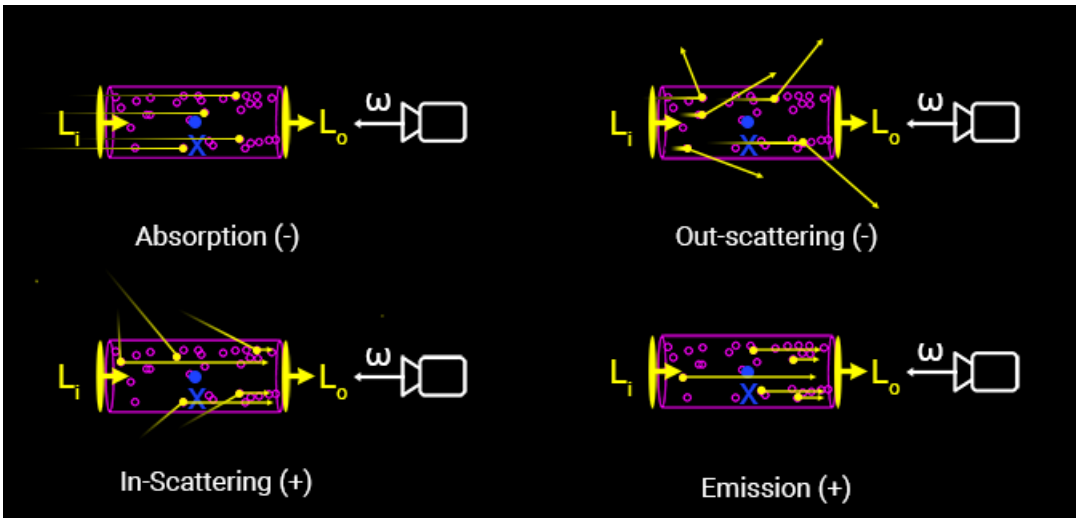


Figura 1.1: figura presa da www.scratchapixel.com. Questa figura illustra diversi fenomeni che si possono verificare con l'interazione della luce ed un volume. In alto a sinistra abbiamo l'assorbimento, in alto a destra l'out-scattering, in basso a sinistra l'in-scattering e in basso a destra abbiamo l'emissione

- Emissione: in presenza di materiali particolari e.g. gas, essi tendono ad emettere fotoni eventualmente nella direzione ω provocando un aumento di radianza.

Dati questi 4 fattori e già possibile approssimare una sorta di equazione che andrà ad descrivere il fenomeno del volume rendering.

$$dL(x,\omega) = emissione + in\_scattering - out\_scattering - assorbimento$$

Naturalmente per descrivere questi tipi di fenomeni esistono 2 coefficienti : il coefficiente di assorbimento σ_a e il coefficiente di scattering σ_s. σ_a e σ_s non sono altro che densità di probabilità , che tengono conto dell'assorbimento/-scattering di luce per unità di distanza, che è indipendente dall'intensità della luce. Questi due vengono riuniti in un altro coefficiente chiamato coefficiente di estinzione $\sigma_t = \sigma_a + \sigma_s$.
Di seguito viene introdotta la differenza tra la Trasmittanza e la Radianza, utile per derivare la legge di Beer-Lambert. La Trasmittanza indica quanta

luce attraverso il volume, in altre parole indica quant'è opaco un oggetto. Mentre la radianza indica quant'è luminoso l'oggetto volumetrico.Per ottenere la legge di Beer-Lambert si deriva la radianza rispetto una direzione ω e l'inizio alla posizione x

$$dL(x,\omega) = -\sigma_a L(x,\omega)$$

Per il momento si tiene conto soltanto il contributo del coefficiente di assorbimento. Si fa un cambiamento di variabile da x a s dove s rappresenta la distanza totale di percorrenza

$$\frac{dL(s)}{ds} = -\sigma_a L(s)$$

Per ottenere la legge di Beer-Lambert bisogna risolvere la seguente equazione differenziale di primo ordine lineare

$$\frac{dy}{dx} = cy \tag{1.1}$$

$$\frac{1}{c}\frac{1}{dx} = \frac{y}{dy} \tag{1.2}$$

$$cdx = \frac{dy}{y} \tag{1.3}$$

$$\int \frac{1}{y}\,dy = \int c\,dx \tag{1.4}$$

$$ln(y) = cx \tag{1.5}$$

$$e^{ln(y)} = e^{cx} \tag{1.6}$$

$$y = e^{cx} \tag{1.7}$$

$$L(s) = e^{-\sigma_a s} \tag{1.8}$$

Tenendo anche conto del out-scattering l'equazione 1.8 si può riscrivere come

$$L(s) = e^{-(\sigma_a+\sigma_s)s} = e^{-\sigma_t s}$$

Data L_i radianza entrante ed L_o radianza di uscita, la trasmittanza si può scrivere come:

$$T = \frac{L_o}{L_i}$$

Questa equazione però funziona esclusivamente per medium omogenei, per far funzionare la funzione anche con volumi eterogenei la legge di Beer-Lambert diventa:

$$T(d) = exp(-\int_{s=0}^{d} \sigma_t(x_s)d_s)$$

Per includere il contributo dell'in-scattering si deve introdurre la funzione di fase. Quando i fotoni interagiscono con le particelle che costituiscono il medium , i fotoni possono essere dispersi e non solamente assorbiti. La dispersione si può verificare in varie direzioni anche nella direzione $-\omega$. Per questo motivo la luce nella direzione $-\omega$ riceve più energia. Si deve usare la funzione di fase dato $-\omega$ la direzione dell'osservatore, ω' la direzione della luce e θ l'angolo tra le due direzioni. La funzione di fase è definita come $f_p(x, \omega, \omega')$ e possiede alcune proprietà.

$$f_p(x, \omega, \omega') = f_p(x, \omega', \omega)$$

, infatti può essere riscritta come $f_p(x, \theta)$ dove θ è l'angolo tra ω' ω. Un'altra proprietà

$$\int_{S^2} f_p(x, \omega, \omega')d\theta = 1$$

dove S^2 è sono tutte le direzioni possibili in una sfera. La funzione di fase in termini matematici si definisce come una distribuzione angolare della luce(radianza) dispersa. La proprietà di normalizzazione è necessaria perché ci sia una conservazione della luce(ovvero della radianza). Un'altra cosa da considerare e che ogni tipo di medium può esibire differenti tipi di dispersioni: isotropica, anisotropica. La prima disperde i fotoni in tutte le direzioni(il fotone ha la stessa probabilità di finire in qualsiasi direzione). Mentre la seconda la distribuzione non è uniforme rispetto tutte le direzioni, infatti ci sono direzioni che non possono essere ottenute, mentre altre che sono più probabili. L'esempio di una funzione isotropica (tutte calcolate in steradianti) è

$$f_p(x, \theta) = \frac{1}{4\pi}$$

, tenendo conto che l'area di una sfera in steradianti è 4π. Un altro esempio di funzione di fase anisotropica è la funzione di *Henyey-Greenstein*:

$$f_p(x,\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2gcos\theta)^{\frac{3}{2}}}$$

originariamente sviluppata per simulare la dispersione della luce attraverso la polvere intergalattica.

Di seguito viene definita *Radiative Transfer Equation* (RTE) questa funzione tiene conto sia del fenomeno di assorbimento che di out-scattering e in-scattering. Questa equazione definisce il cambio di radianza lungo una direzione (si deriva rispetto ω)

$$\frac{L(x,\omega)}{d\omega} = -\sigma_t L(x,\omega) + \sigma_s \int_{S^2} f_p(x,\omega,\omega')L(x,\omega')d\omega'$$

Il primo elemento rappresenta l'assorbimento e l'out-scattering mentre il secondo elemento rappresenta l'in-scattering. Anche questa è una equazione differenziale di primo grado, risolvendola si ottiene :

$$L(x,\omega) = \int_{t=0}^{s} exp(-\int_{q=0}^{t}\sigma_t(x_q)dq)[\sigma_s(x_t)L_s(x_t)]dt + L(0)exp(-\int_{t=0}^{s}\sigma_t(x_t)dt) \tag{1.9}$$

dove

$$L_s(x) = \int_{S^2} f_p(x,\omega,\omega')L(x,\omega')d\omega'$$

Il termine $L(0)$ in 1.9 si riferisce alla radianza che può provenire da un oggetto che sta dietro al volume. Se si aggiunge anche L'emissione l'equazione diventa:

$$L(x,\omega) = \int_{t=0}^{s} exp(-\int_{q=0}^{t}\sigma_t(x_q)dq)[\sigma_s(x_t)L_s(x_t) + \sigma_a(x_t)L_e(x_t)]dt + L(0)exp(-\int_{t=0}^{s}\sigma_t(x_t)dt) \tag{1.10}$$

Per riscriverla in modo definitivo tendendo conto della legge di Beer per i medium eterogenei $T(s) = exp(-\int_{t=0}^{s}\sigma_t(x_t)dt)$

$$L(x,\omega) = \int_{t=0}^{s} T(t)[\sigma_s(x_t)L_s(x_t) + \sigma_a(x_t)L_e(x_t)]dt + T(s)L(0)$$

1.3 Segmentazione Semantica

Esistono varie applicazioni di segmentazione semantica e.g. nell'ambito della guida autonoma , nel scene understanding per agenti intelligenti, nelle immagini mediche, etc.In questa tesi ci interesseremo nella segmentazione semantica per ambienti chiusi per agenti intelligenti. Data una immagine $\mathbb{R}^{C\times H\times W}$ ogni singolo pixel p_{ij} dovrà essere assegnato ad una classe c_k dove $k \in |\mathcal{K}|$. Durante gli anni sono stati sviluppati vari metodi [2] [3] [13] che cercano di migliorare la performance della segmentazione semantica cercando di far imparare ad una rete neurale di non concentrarsi solamente su zone locali, ma ad utilizzare il contesto globale dell'immagine. Altri metodi [4][5][14] sopratutto nell'ultimo periodo usano l'ausilio del meccanismo di attenzione [15] , che può essere classificato come un algoritmo di non *local-mean* [16]. Infatti questi metodi sfruttano implicitamente il fatto che l'attenzione attende da subito tutti i t*token*(in questo caso pixel), quindi può sfruttare tutte le informazioni globali dell'immagine. Proprio per questo motivo molti modelli che si basano sull'attenzione diversamente da quelli che sfruttano i CNN tendono a essere più costosi computazionalmente, oltre il fatto a non possedere l'inductive bias [17], che le reti convoluzionali hanno a priori. La segmentazione esiste anche per elementi nello spazio 3D, i i metodi adoperati sono molti costosi a livello computazionale. Nel caso di studio di questa tesi, pur andando a costruire una scena 3D, l'input della rete considerata non è un modello 3D o *cloud points*, bensì saranno posizioni nel spazio 3D e le direzioni di vista, che dimensionalmente sono di gran lunga minori rispetto a modelli 3D composti da vari vertici.
Per valutare la performance di un modello di segmentazione semantica esistono varie metriche, ma quelle principalmente utilizzate sono *mIoU*, e *dice score*. Il *Dice score* è definito come:

$$\frac{2TP(A,B)}{2TP(A,B)+FP(A,B)+FN(A,B)}$$

dove A, B sono rispettivamente l'insieme dei *ground truth pixel* e quelli predetti dal modello, mentre la notazione $TP(A,B), FP(A,B), FN(A,B)$ corri-

sponde all'intersezione tra questi due insieme A, B. In particolare $TP(A, B)$ rappresenta l'insieme dei *true positive*($a_i = 1$ e $b_i = 1$), $FP(A, B)$ corrisponde ai *false positive* ($a_i = 0$ e $b_i = 1$) ed $FN(A, B)$ corrisponde ai *false negative* ($a_i = 1$ e $b_i = 1$). Ognuna di queste funzioni ritorna il numero di elementi che si intersecano. Prima di introdurre $mIoU$, introduciamo il coefficiente di Jaccard o *Intersection over Union*(Iou) che è definito come :

$$JAC(A, B) = \frac{TP(A, B)}{TP(A, B) + FP(A, B) + FN(A, B)}$$

una volta definito il coefficiente di *Jaccard* definiamo $mIoU$ come :

$$mIoU = \frac{1}{N} \sum_{i}^{N} JAC(A_i, B_i)$$

dove N è il numero di oggetti o classi totali presenti nella scena, ed A_i, B_i si riferiscono ai pixel assegnati alla classe i.

Capitolo 2

Neural Radiance Field

In questo capitolo verrà descritta la tecnologia dei Neural Radiance Field e successivamente gli avanzamenti ed i miglioramenti che sono stati apportati all'idea iniziale che ne è alla base.

2.1 NeRF:Neural Radiance Field

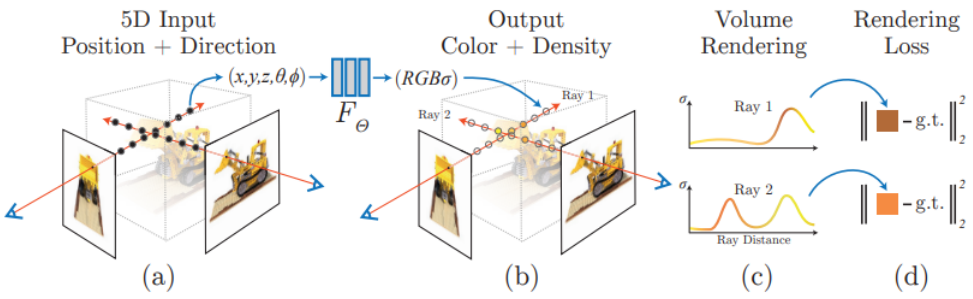


Figura 2.1: immagine presa da [9]. In questa immagine viene illustrato tutto il processo di training di un modello nerf. (a) vengono passate in input le coordinate $xyz\theta\phi$.(b) l'output del modello che corrisponde ad C colore e la densità σ.(c) viene mostrato il processo di *Hierarchical sampling* dove i picchi della distribuzione corrispondono ad una particella incontrata.(d)la funzione di loss applicata all'output dei rispettivi coarse e fine network

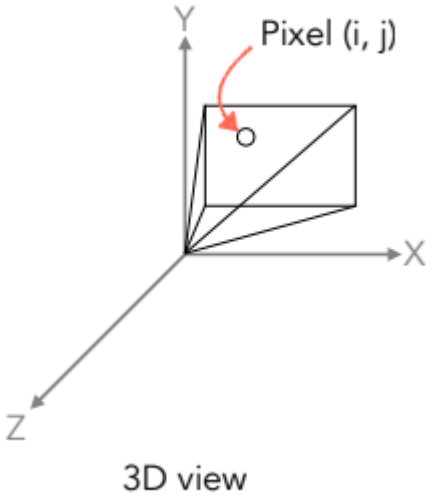


Figura 2.2: immagine presa da [18].

NeRF, o Neural Radiance Fields, è un approccio innovativo nell'ambito di computer graphics e visione artificiale che mira a ricostruire scene 3D partendo da una collezione di immagini 2D della scena. L'idea di base di Nerf è descrivibile in 3 fasi Fig.2.1.

1. si fanno partire raggi dalla telecamera per campionare un insieme di punti 3D 2.1.1.

2. i punti campionati e le direzioni di vista vengono date in input ad una rete neurale MLP che produce in output n insieme di colori e di densità.

3. vengono utilizzate tecniche classiche di volume rendering per accumulare questi colori e densità in un'immagine 2D.

L'input di nerf è una scena continua rappresentata da un vettore $\mathbb{R}^5$, dove l'input $\mathbb{R}^3$ corrisponde ad $x = (x, y, z)$ ovvero la posizione. Mentre il $\mathbb{R}^2$ corrisponde ad (θ, ϕ) la direzione di uscita del raggio dalla camera. In pratica la direzione è rappresentata come un vettore unitario $\mathbb{R}^3$. Per quanto riguarda

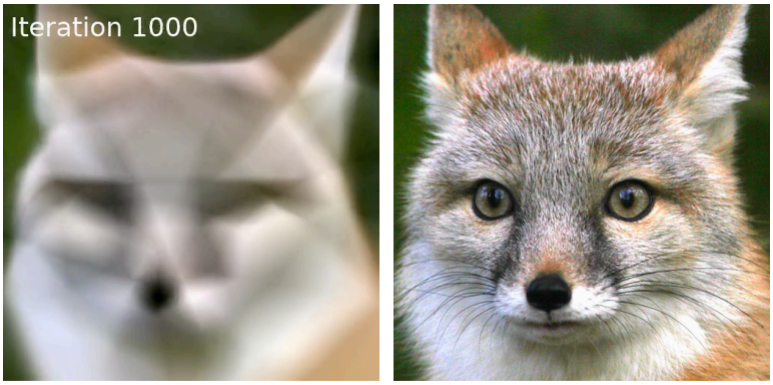


Figura 2.3: immagine presa da [18]. L'immagine mostra l'allenamento di un rete neurale composta da MLP e ReLU passando come input le coordinate normalizzate di questa immagini e.g.$(x, y) \rightarrow [0, 1]$ ed in output si aspetta il colore del pixel selezionato.L'immagine a sinistra mostra il risultato senza utilizzare *positional encoding* , infatti questa porta la rete neurale ad non imparare funzione ad alta frequenza[19].Mentre l'immagine a destra è l'immagine da imparare attraverso le coordinate dei suoi pixel.

l'output abbiamo il colore $c = (r, g, b)$ e la densità σ. Quindi la rete neurale è rappresentabile come

$$F_\Theta : (x, d) \rightarrow (c, \sigma)$$

Poiché questo processo è differenziabile è possibile usare la discesa del gradiente per ottimizzare il modello minimizzando l'errore tra ogni immagine osservata e la corrispondente generata dalla rappresentazione tramite rete neurale. Il modello è incoraggiato ad imparare una scena da molteplici angolazioni, costringendolo a dipendere solamente dalla posizione x per la densità σ e facendo dipendere il colore c sia dalla posizione x che dal punto di vista d. Viene passato in input la posizione x ad un MultiLayer Perceptron (MLP) formato da 8 layer ognuno seguito da una attivazione non lineare ReLU, questo restituisce come output la densità σ vettore di dimensionalità $\mathbb{R}^{256}$. Una volta ottenuta la densità σ questa viene concatenata alla direzione e passata in input ad un ultimo layer MLP 2.4 , che darà come output il colore c. Per effettuare il rendering di un colore di un raggio che attraversa la scena ven-

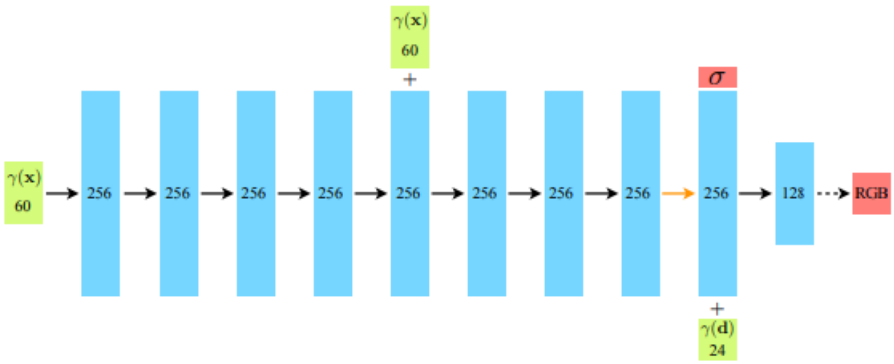


Figura 2.4: Architettura NeRF[9].

gono utilizzati i processi di volume rendering trattati nella scorsa sezione. La densità $\sigma(x)dt$ può essere interpretata come la probabilità che un un raggio colpisca una particella alla locazione x attorno un intervallo molto piccolo dt(differenziale)

$$P(raggio\_colpisce\_particella\_in\_x) = \sigma(x)dt$$

Si ricorda dalla sezione precedente la legge di Beer-Lambert per medium eterogenei

$$T(t) = exp(-\int \sigma(t)dt)$$

La funzione $T(t)$ che denota la trasmittanza può essere interpretata come la probabilità che il raggio che parte dal *near bound* t_n ed arriva a t senza colpire nessuna particella. Quindi la probabilità che il raggio colpisca una particella al punto t sarà:

$$P(raggio\_colpisce\_a\_t) = exp(-\int \sigma(t)dt)\sigma(t)dt$$

Calcolare il valore atteso (il colore) lungo il raggio corrisponde ad :

$$\int_{t_0}^{t_1} T(t)\sigma(t)c(t)dt$$

Per semplificare il calcolo analitico, si approssima l'integrale più esterno con la somma di integrali su intervalli più piccoli. Si suddivide il raggio in n

segmenti $t_1, t_2, \ldots, t_n + 1$ dove si definisce con $\delta_i = t_{i+1} - t_i$, è possibile ipotizzare che questi n segmenti sono abbastanza piccoli che il colore ed la densità siano costanti all'interno di ogni singolo intervallo, però questo non implica trasmittanza costante.

$$\int T(t)\sigma(t)c(t)dt \approx \sum_{i=1}^{n} \int_{t_i}^{t_{i+1}} T(t)\sigma_i c_i dt$$

Infatti è necessario valutare la trasmittanza su valori continui che possono stare anche in mezzo ad un intervallo. Dati $t \in [t_i, t_{i+1}]$

$$T(t) = exp(-\int_{t_1}^{t_i} \sigma_i ds) exp(-\int_{t_i}^{t} \sigma_i (t - t_i)))$$

$$T(t) = exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j) exp(-\int_{t_i}^{t} \sigma_i (t - t_i)))$$

$$\sum_{i=1}^{n} \int_{t_i}^{t_{i+1}} T(t)\sigma_i c_i dt = \sum_{i=1}^{n} T_i \sigma_i c_i \int_{t_i}^{t_{i+1}} exp(-\sigma_i (t - t_i))dt) \tag{2.1}$$

$$= \sum_{i=1}^{n} T_i \sigma_i c_i \frac{exp(-\sigma_i (t_{i+1} - t_i)) - 1}{-\sigma_i} \tag{2.2}$$

$$= \sum_{i=1}^{n} T_i c_i (1 - exp(-\sigma_i \delta_i)) \tag{2.3}$$

$$= \sum_{i=1}^{n} T_i c_i \alpha_i \tag{2.4}$$

dove abbiamo che $T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$ e $\alpha_j = (1 - exp(-\sigma_i \delta_i))$ da queste equazioni si ottiene che $T_i \alpha_i$ non sono altro che *pesi di rendering*, ovvero distribuzioni di probabilità lungo il raggio. Questi pesi ci permettono di calcolare il valore atteso per altre quantità e.g. la profondità attesa

$$\bar{t} = \sum_{i} T_i \alpha_i t_i$$

In generale questa idea è applicabile a qualsiasi quantità che vogliamo renderizzare in una immagine. Finché v vive in uno spazio $\mathbb{R}^3$ e.g. *features*

semantiche , vettori normali, etc...

$$\sum_i T_i \alpha_i v_i$$

NeRF tiene conto soltanto dei processi assorbimento ed emissione , ma non di scattering(in e out). La rete neurale F_Θ pur essendo un approssimatore universale di funzioni, se opera direttamente sulle coordinate $xyz\theta\phi$ per ricostruire la scena, produce un risultato di scarsa qualità. Questo è dovuto al fatto che una MLP riesce a imparare features a bassa frequenza, ma fatica ad imparare *features* ad alta frequenza[19]Fig.2.3. Per risolvere questo problema si utilizzano dei *positional embeddings* $\gamma : \mathbb{R} \to \mathbb{R}^{2L}$ dove

$$\gamma(p) = [sin(2^0\pi p), cos(2^0\pi p), \ldots, sin(2^{L-1}\pi p), cos(2^{L-1}\pi p)]$$

La funzione γ è applicata ad ogni coordinata della posizione x dove il range dei valori che può assumere è di $[-1, 1]$. Naturalmente vengono anche applicate alle coordinate di direzione, sperimentalmente si utilizza $L = 10$ per x ed $L = 4$ per d. Queste funzioni permettono di mappare un input continuo in una dimensione maggiore, così la rete può approssimare le funzioni ad alta frequenza. Per renderizzare la scena si potrebbe pensare che basti semplicemente campionare da ogni raggio N_c posizioni, come ad esempio il centro di ogni intervallo generato precedentemente, ma questo potrebbe introdurre dei problemi. Infatti se l'intervallo è troppo grande potremmo non catturare oggetti molto piccoli o dettagli molto piccoli 2.5 .Per ovviare a questo problema si utilizza *Hierarchical Sampling*, che consiste nell'eseguire prima un campionamento grossolano (coarse) ed utilizzare i pesi trovati $T_i\alpha_i$ per localizzare i punti più influenti

$$\hat{C}_c(r) = \sum_{j=1}^{N_c} \omega_i c_i \quad \omega_i = T_i(1 - exp(-\sigma_i\delta_i))$$

Normalizzare i pesi $\hat{\omega} = \dfrac{\omega_i}{\sum_{j=1}^{N_c} \omega_j}$ produce una funzione di densità probabilità, attraverso la quale si può calcolare la funzione cumulativa F_X e la sua inversa F_X^{-1}. Successivamente vengono campionati un secondo insieme

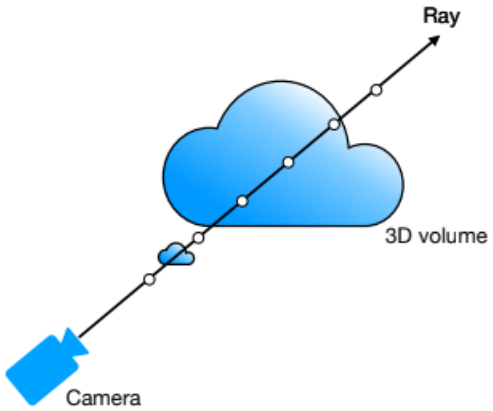


Figura 2.5: immagine presa da [18]. Se il campionamento si verifica su intervalli troppo grandi potremmo andare a perdere dettagli importanti come in questo caso la nuvola più piccola.

di punti N_f usando il metodo dell'inversione sfruttando F_X^{-1}, poi si uniscono tutti i punti $N_c + N_f$ per calcolare il colore finale utilizzando la funzione

$$\hat{C} = \sum_{i=1}^{N_c+N_f} T_i c_i (1 - exp(-\sigma_i \delta_i))$$

Infine per allenare il modello finale si adopera la seguente funzione obiettivo

$$\mathcal{L} = \sum_{r \in \mathcal{R}} [\|\hat{C}_c(r) - C(r)\|_2^2 + \|\hat{C}_f(r) - C(r)\|_2^2]$$

dove $\mathcal{R}$ è l'insieme dei raggi all'interno della batch e $C(r)$ è il colore ground truth. Una cosa da tenere in considerazione che i dati utilizzati in input se sintetici (generati su blender o altri software 3D) di norma posseggono già tutti i dati sulla camera (parametri intrinseci ed estrinseci), però per scene reali si adopera il sfm Structure From Motion[20]. Dove il compito principale di Structure From Motion è ricavare i parametri intrinseci ed estrinseci della camera

2.1.1 Campionamento dei punti 3D

Date diverse immagini prese da diversi punti di vista della scena da ricostruire ed i parametri intrinseci ed estrinseci della camera , per ogni punto di vista. È possibile usare queste informazioni per calcolare la direzione che parte dall'origine della telecamera e arriva ad un particolare pixel (i, j)Fig.2.2

$$(i, j) \rightarrow (\frac{i - w/2}{f_x}, \frac{j - h/2}{f_y}, -1)$$

dove w è la larghezza dell'immagine h è lunghezza dell'immagine ed fx e fy sono le rispettive lunghezze focali per l'asse x e l'asse y. Una volta ottenute la direzione basta moltiplicarla per la posizione della camera(matrice $W_{c2w} \in \mathbb{R}^{4\times4}$ di cui si utilizza la sotto-matrice $\mathbb{R}^{3\times3}$) $d = direction \cdot W_{c2w}$. L'equazione del raggio sarà quindi:

$$r = o + td$$

dove o è l'origine della camera $\mathbb{R}^3$ ed $t \in \mathbb{R}$ è uno scalare che controlla la distanza del punto campionato lungo il raggio(possiamo prendere più valori di t in modo tale da ottenere uniformemente tutti i punti tra il *near* e *far plane*)

2.2 Related Work

Negli ultimi anni NeRF [9] ha superato le 3500 citazione nell'arco di appena 3 anni, portando miglioramenti ed applicazioni in diversi campi. In questa sezione andremo a discutere di alcune ricerche che hanno apportato diversi miglioramenti alla sua architettura rendendolo sia più veloce che accurato dal punto di vista geometrico.

2.2.1 Mip-Nerf[11]

Un problema del Nerf Originale è il campionamento a punti lungo i raggi proiettati dalla camera. Infatti proiettare un raggio infinitesimamente stretto per pixel dà luogo ad un fenomeno conosciuto come *Aliasing*. Questo

può essere visualizzato anche nell'ambito del campionamento delle immagini, infatti se campionassimo dei pixel partendo da una figura non filtrata ,è pos-

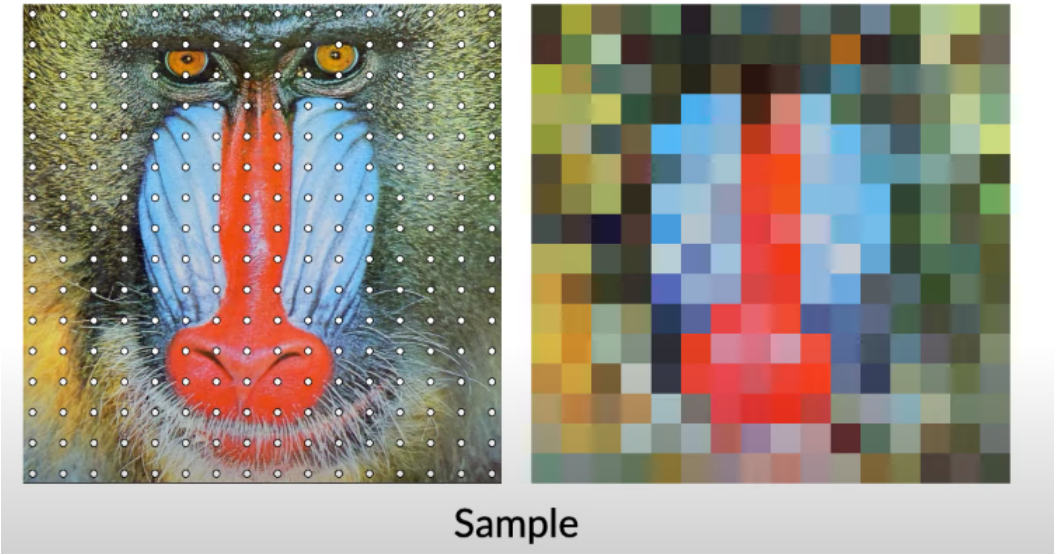


Figura 2.6: immagine presa da [18].Campionamento di una immagine senza applicare un filtro di blurring.

sibile vedere che il risultato non è molto soddisfacente Fig.2.6. Una soluzione per questo problema è fare il pre-filtering con un kernel gaussiano Fig.2.7 . Per ovviare a questo problema nell'ambito dei NeRF , invece di usare un raggio per campionare i punti per ogni pixel, si utilizza un tronco di cono Fig.2.8. Invece di costruire dei *Positional Encoding* (PE) partendo da un un punto nello spazio, si costruisce un *Integrated Positional Encoding*(IPE) che rappresenta il volume ricoperto da ogni tronco di cono. Questo cambiamento permette al MLP di "ragionare" meglio sulla grandezza e la forma di ogni singola sezione di tronco di cono, rispetto ad un singolo punto. La rappresentazione del volume catturato dal tronco di cono dovrebbe essere simile a quella dei *positional encoding* di [9] . La soluzione più semplice è di calcolare il valore atteso rispetto il *positional encoding* di tutte le coordinate che risiedono all'interno del tronco di cono:

$$\gamma^*(o, d, \dot{r}, t_0, t_1) = \frac{\int \gamma(x) F(x, o, d, \dot{r}, t_0, t_1) dx}{\int F(x, o, d, \dot{r}, t_0, t_1) dx}$$

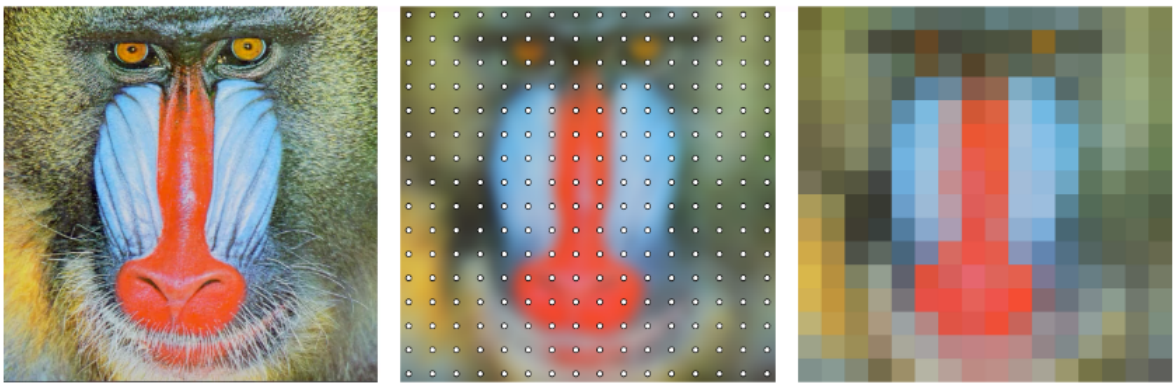


Figura 2.7: immagine presa da [18].campionamento di una immagine prima applicando un filtro gaussiano.

il problema risiede nel numeratore che non è chiaro come calcolarlo. Quindi per calcolare il valore atteso del *positional encoding* di tutte le coordinate che risiedono all'interno del tronco di cono, le varie sezioni del cono vengono approssimate con una Gaussiana multivariata. La gaussiana necessita del calcolo della media e della covarianza di $F(x, \cdot)$ (la media lungo il raggio μ_t, la varianza lungo il raggio σ_t^2 e la varianza perpendicolare al raggio σ_r^2). Per derivare i vari momenti della distribuzione uniforme rispetto al tronco di cono, si parte dal cono $(x, y, z) = \phi(r, t, \theta) = (rt\ cos\theta, rt\ sin\theta, t)$ dove $\theta \in [0, 2\pi), t \geq 0 |r| \leq \dot{r}$. Prima di calcolare il volume del tronco di cono bisogna eseguire il cambio di variabile dallo spazio cartesiano:

$$dxdydz = |det(D\phi)(r, t, \theta)|drdtd\theta = rt^2 drdtd\theta$$

che servirà come costante di normalizzazione di una distribuzione uniforme

$$V = \int_0^{2\pi} \int_{t_0}^{t_1} \int_0^{\dot{r}} rt^2 drdtd\theta \tag{2.5}$$

$$= \frac{\dot{r}^2}{2} \cdot \frac{t_1^3 - t_0^3}{3} \cdot 2\pi \tag{2.6}$$

$$= \pi \dot{r}^2 \frac{t_1^3 - t_0^3}{3} \tag{2.7}$$

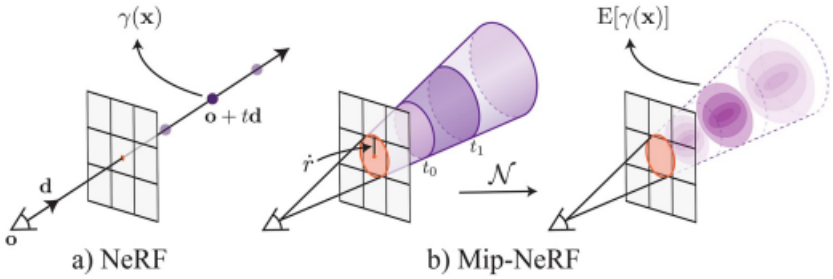


Figura 2.8: immagine presa da [11].Mostra partendo da sinistra a destra il sampling classico di [9] poi il tronco di cono ed infine l'approssimazione delle sezioni del cono con gaussiane.

Quindi la funzione di densità probabilità per punti campionati uniformemente in un tronco di cono è rt^2/V. Il primo momento di t :

$$E[t] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} t \cdot rt^2 dr dt d\theta \tag{2.8}$$

$$= \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} rt^3 dr dt d\theta \tag{2.9}$$

$$= \frac{1}{V} \cdot \pi\dot{r}^2 \frac{t_1^4 - t_0^4}{4} \tag{2.10}$$

$$= \frac{3(t_1^4 - t_0^4)}{4(t_1^3 - t_0^3)} \tag{2.11}$$

Mentre i momenti di x e y sono entrambi 0 per simmetria. Il secondo momento di t è:

$$E[t^2] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} t^2 \cdot rt^2 dr dt d\theta \tag{2.12}$$

$$= \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}} rt^4 dr dt d\theta \tag{2.13}$$

$$= \frac{1}{V} \cdot \pi\dot{r}^2 \frac{t_1^5 - t_0^5}{5} \tag{2.14}$$

$$= \frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)} \tag{2.15}$$

Il secondo momento di x e y invece è

$$E[x^2] = \frac{1}{V}\int_0^{2\pi}\int_{t_0}^{t_1}\int_0^{\dot{r}}(rtcos\theta)^2 \cdot rt^2 drdtd\theta \tag{2.16}$$

$$= \frac{1}{V}\int_{t_0}^{t_1}\int_0^{\dot{r}} r^3t^4 \int_0^{2\pi} cos^2\theta d\theta drdt \tag{2.17}$$

$$= \frac{1}{V}\cdot\frac{\dot{r}^4}{4}\frac{t_1^5 - t_0^5}{5}\cdot\pi \tag{2.18}$$

$$= \frac{\dot{r}^2}{4}\cdot\frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)} \tag{2.19}$$

Con tutti questi momenti definiti,si costruisce la media e la covarianza per un punto a caso all'interno del tronco di cono. La media lungo il raggio μ_t è semplicemente il primo momento rispetto a t:

$$\mu_t = \frac{3(t_1^4 - t_0^4)}{4(t_1^3 - t_0^3)}$$

La varianza del tronco conico rispetto a t è ricavabile attraverso la formula della varianza $Var(t) = E[t^2] - E[t]^2$:

$$\sigma_t^2 = \frac{3(t_1^5 - t_0^5)}{5(t_1^3 - t_0^3)} - \mu_t^2$$

La varianza del tronco conico rispetto al raggio r è uguale alla varianza del tronco rispetto a x o y per simmetria. Siccome il primo momento rispetto a x è zero, la varianza è uguale al secondo momento:

$$\sigma_r^2 = \dot{r}^2(\frac{3(t_1^5 - t_0^5)}{20(t_1^3 - t_0^3)})$$

Calcolare tutte e tre le quantità nella forma data è numericamente instabile, questo è dovuto al rapporto tra le differenza t_1 e t_0 elevato a una potenza molto alta, quando i due valori sono ravvicinati(cosa che occorre frequentemente durante l'allenamento). Quindi per risolvere questo problema si riscrivono t_0 e t_1 come $t_\mu = (t_0 + t_1)/2$ e $t_\delta = (t_1 - t_0)/2$. Ora si ha:

$$\mu_t = t_\mu + \frac{2t_\mu t_\delta^2}{3t_\mu^2 + t_\delta^2}$$

$$\sigma_t^2 = \frac{t_\delta^2}{3} - \frac{4t_\delta^4(12t_\mu^2 - t_\delta^2)}{15(3t_\mu^2 - t_\delta^2)^2}$$

$$\sigma_r^2 = \dot{r}^2(\frac{t_\mu^2}{4} + \frac{5t_\delta^2}{12} - \frac{4t_\delta^4}{15(3t_\mu^2 + t_\delta^2)})$$

Trasformare questa gaussiana dalle coordinate del tronco conico nelle coordinate del mondo:

$$\mu = o + \mu_t d \quad \Sigma = \sigma_t^2(dd^T) + \sigma_r^2(I - \frac{dd^T}{\|d\|_2^2})$$

Per calcolare IPE è necessario calcolare il valore atteso rispetto la gaussiana multivariata rispetto la base adoperata anche in PE con media e covarianza $\mu_\gamma = P\mu$ e $\Sigma = P\Sigma P^T$.

$$\gamma(\mu, \Sigma) = \sum_{x \sim \mathcal{N}(\mu_\gamma, \Sigma_\gamma)} [\gamma(x)] \tag{2.20}$$

$$= \begin{bmatrix} sin(\mu_\gamma) * exp(-0.5diag(\Sigma_\gamma)) \\ cos(\mu_\gamma) * exp(-0.5diag(\Sigma_\gamma)) \end{bmatrix} \tag{2.21}$$

dove il simbolo $*$ rappresenta la moltiplicazione elemento per elemento delle matrici. Si adopera la $diag(\Sigma_\gamma)$ perché calcolare l'intera matrice è molto costoso dovuto soprattutto alla sua dimensione.

$$diag(\Sigma_\gamma) = [diag(\Sigma), 4diag(\Sigma), \ldots, 4^{L-1}diag(\Sigma)]$$

dove

$$diag(\Sigma) = \sigma_t^2(d * d) + \sigma_r^2(1 - \frac{d * d}{\|d\|_2^2})$$

IPE funziona in questo modo, se una particolare frequenza ha un periodo che è più grande della larghezza dell'intervallo con cui è stato costruito il *positional encoding*, allora la codifica dell'input a quella frequenza è inalterata. Invece se il periodo è più piccolo dell'intervallo, allora la codifica dell'input a quella frequenza viene azzerata Fig.2.9.

Altri cambiamenti di questo modello rispetto al Nerf originale riguardano il campionamento gerarchico. Infatti grazie al IPE che riesce a codificare

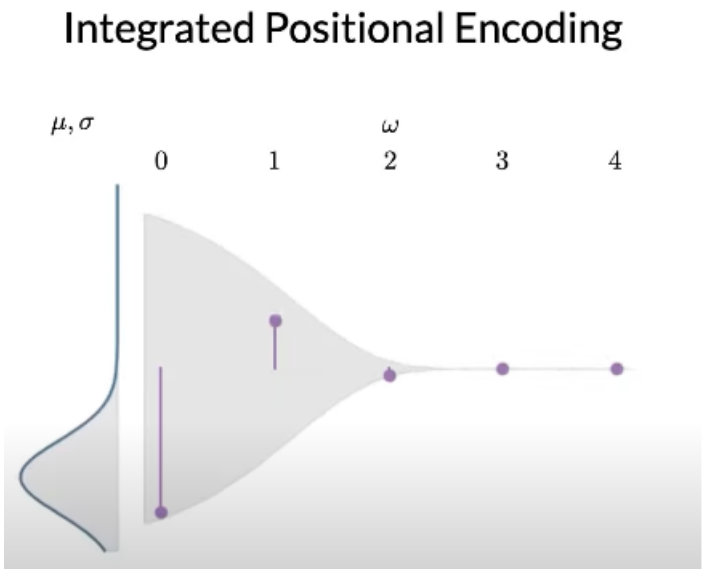


Figura 2.9: immagine presa da [11].Integrated positional encodings.

la grandezza dell'oggetto all'interno del tronco di cono, il campionamento gerarchico non è più strettamente necessario. La funzione obiettivo diventa:

$$min_{\Theta} \sum_{r \in \mathcal{R}} (\lambda \| C(r) - \hat{C}(r; \Theta, t^c) \|_2^2 + \| C(r) - \hat{C}(r; \Theta, t^f) \|_2^2)$$

dove $\lambda = 0.1$

2.2.2 NeRF-W

Un altro problema principale di NeRF è la sua incapacità di modellare la scena con presenza di oggetti non-statici e cambiamenti di illuminazioni. Questi fenomeni sono molto comuni, soprattutto con foto di monumenti o posti pubblici, che vengono prese durante gli anni. Per gestire tali scenari per prima cosa vengono modellate in uno spazio latente le variazioni di apparenza per ogni singola immagine come: esposizione , illuminazione e post-processing. Per ogni singola immagine viene ottimizzato un *apperance embedding*, permettendo a Nerf-W di distinguere variazioni fotometriche e ambientali tra immagini, imparando una rappresentazione condivisa da

tutta la collezione di immagini utilizzate nell'allenamento. Come seconda cosa la scena viene gestita come unione di elementi condivisi ed unici (appartenenti alle singole immagini), questo permette una decomposizione non supervisionata della scena in componenti statici e transienti. Quindi l'ipotesi di consistenza di punti di vista 3d viene violata da 2 fenomeni: variazione fotometrica, oggetti transienti. La prima influisce sulla radianza degli oggetti che di solito viene intaccata da fenomeni atmosferici, invece i secondi solitamente corrispondono a persone, cantieri, etc... Per risolvere il primo fenomeno viene modellato lo spazio latente. Per adattare il modello NeRF a cambiamenti di illuminazione, bisogna assegnare ad ogni immagine I_i un corrispondente vettore l_i^a di lunghezza n^a. Quindi il calcolo della radianza che è indipendente dall'immagine utilizzata $c(t)$, viene rimpiazzato da $c_i(t)$, che a sua volta introduce una dipendenza dall'indice dell'immagine usata per il calcolo del colore atteso $\hat{C}_i$

$$\hat{C}_i(r) = \mathcal{R}(r, c_i, \sigma)$$

$$c_i(t) = MLP_\theta(z(t), \gamma_d, l_i^a)$$

Gli *embedding* $\{l_i^a\}_{i=1}^N$ vengono ottimizzati insieme a θ. Usare questi *apperance embeddings* come input solo alla rete responsabile all'output del colore, permette di variare la radianza emessa dalla scena, lasciando invariata la geometria. Invece per gestire il problema degli oggetti transienti, viene aggiunto un altro insieme di MLP per gli oggetti transienti Fig.2.10 che fornisce in output sia il colore che la densità. La densità può variare in base all'immagine, e permette a Nerf-w di ricostruire immagini senza introdurre artefatti nella rappresentazione statica della scena. Quindi si ha $\sigma_i^\tau(t)$ ed $C_i^\tau(t)$

$$\hat{C}(r) = \sum_{k=1}^{K} T_i(t_k)(\alpha(\sigma(t_k)\delta_k)c_i(t_k) + \alpha(\sigma_i^\tau(t_k)\delta_k)c_i^\tau(t_k))$$

$$T_i(t_k) = exp(-\sum_{k'=1}^{k-1}(\sigma(t_{k'}) + \sigma_i^\tau(t_{k'}))\delta_{k'})$$

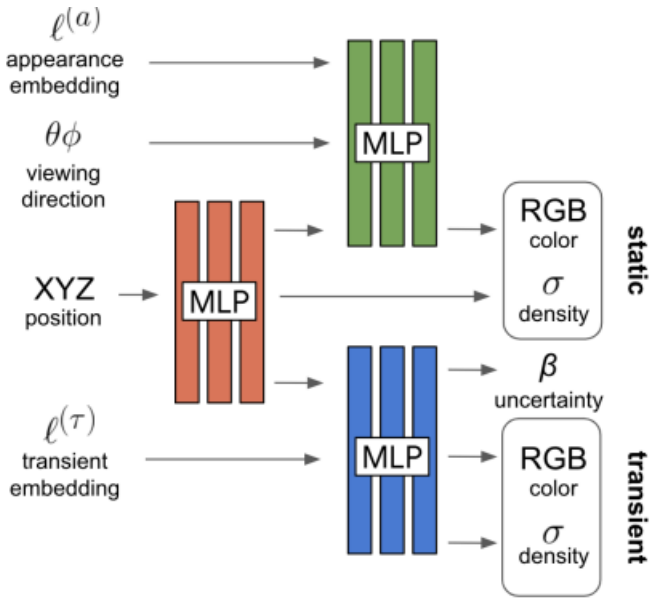


Figura 2.10: il modello di Nerf-w[21].

2.2.3 PlenOctrees

Uno dei problemi principali di NeRF è sempre stato il tempo di training e di inferenza. Per velocizzarne i tempi si sono sviluppate 2 categorie: *baked*, *non-baked*. Nella prima categoria i modelli vengono allenati o valutati su strutture dati più accessibili. Il miglioramento principale è nell'inferenza del modello, anche di molto. Nella seconda categoria dei modelli *non-baked* sono incluse diverse innovazioni e migliorie. I modelli *non-baked* solitamente provano a imparare delle *feature* della scena in modo diverso rispetto ai parametri di un MLP(*Sparse Voxel grids*). Permettono di utilizzare MLP più piccoli (dimensionalmente) per produrre densità e colore, velocizzando il tempo di allenamento ed inferenza, al costo di un aumento della memoria. In determinati casi gli stessi MLP vengono sostituti con altre componenti non neurali [23].

Plenoctrees ricade nella categoria dei metodi *baked*, quindi il principale miglioramento è nel tempo di inferenza. Per esempio per un modello NeRF [9] ci vogliono circa 30 secondi per renderizzare una immagine 800x800 su

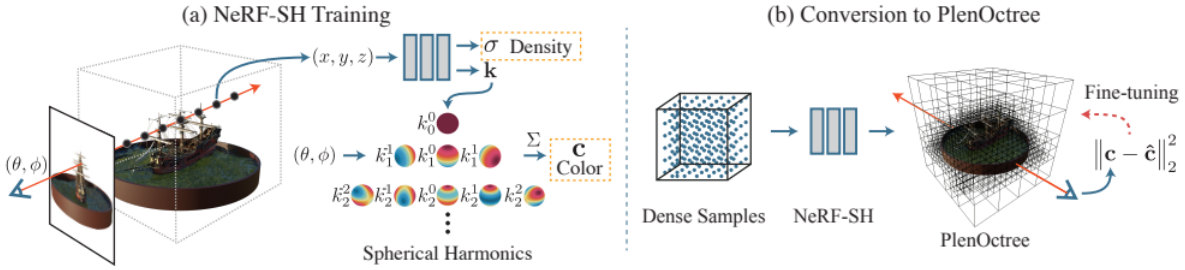


Figura 2.11: immagine presa da [22].A sinistra abbiamo il modello NeRF-SH mentre a destra abbiamo il processo di conversione da MLP a Octrees

una scheda grafica Nvidia A100 rendendolo non molto pratico per applicazioni real time. La lentezza del modello NeRF deriva dal fatto che dobbiamo campionare densamente l'intera scena attraverso la MLP che la rappresenta. Inoltre non è possibile conservare semplicemente la densità e il colore in una struttura dati classica non organizzata perché richiederebbe molta memoria. Quindi per risolvere questo problema viene utilizzato uno *sparse voxel octree* [24] in cui ogni foglia dell'albero memorizza la densità e il colore che serve per modellare la radianza di ogni punto della scena. Per tenere conto del comportamento non-Lambertiano di certi materiali, viene proposto di rappresentare il colore RGB attraverso le armoniche sferiche (SH). Le armoniche sferiche possono essere richieste da qualsiasi direzione di vista per recuperare il colore dipendente dal punto di vista.Inoltre sono sempre state molto popolari come rappresentazioni a bassa dimensione, per modellare superfici Lambertiane o anche superfici lucide. Questo nuovo modello viene definito come NeRF-SH, dove l'output del colore RGB viene sostituito con k coefficienti .

$$f(x) = (k, \sigma) \qquad k = (k_l^m)_{0 \le l \le l_{max}}^{m:-l \le m \le l}$$

dove $k_l^m \in \mathcal{R}^3$ è un insieme di 3 coefficienti che corrispondono alle rispettive componenti RGB. Il colore c al punto x e all'angolo d può essere determinato dalla funzione

$$Y_l^m : \mathbb{S}^2 \to \mathbb{R}$$

$$Y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}(-1)^m Im[Y_l^{|m|}] \\ Y_m^0 \\ \sqrt{2}(-1)^m Re[Y_l^{|m|}] \end{cases} \tag{2.22}$$

$$c(d;k) = S(\sum_{l=0}^{l_{max}} \sum_{m=-l}^{l} k_l^m Y_l^m(d))$$

dove $S : x \to (1 + exp(-x))^{-1}$ è la funzione sigmoide. Un'alternativa rispetto le armoniche sferiche sono le superfici Gaussiane sferiche (SG) o anche conosciute come *distribuzione di von Mises–Fisher*

$$G(d;p,\lambda) = e^{\lambda(d\cdot p-1)}$$

$$L(d) = \sum_{l=0}^{n} k_l G_l(d;p,\lambda)$$

dove $p \in \mathbb{R}^2$ e $\lambda \in \mathbb{R}$. Una volta allenato il modello possiamo passare alla conversione a Octrees. Il processo di conversione si può suddividere in tre fasi: *Evaluation, Filtering, Sampling.*

1. La prima fase *Evaluation* corrisponde a valutare NeRF-SH in ogni posizione per ottenere tutti i valori diσ distribuiti uniformemente su una griglia 3D.

2. La seconda fase*Filtering* corrisponde a filtrare ogni signolo voxels in modo tale da ottenere la rappresentazione più piccola per rappresentare la scena. In specifico per filtrare si utilizzano i valori α, si eliminano i voxel aventi dei pesi minori ad un *threshold* τ_w. I voxel rimanenti vengono salvati nell'*octree*.

3. L'ultima fase di *Sampling* corrisponde invece al campionamento di 256 punti per ogni voxel rimanente e definire il valore delle foglie dell'octree. Ogni foglia contiene la densità σ e un vettore di coefficienti armoniche sferiche per ogni canale RGB.

Capitolo 3

Segmentazione Semantica attraverso Neural Radiance Field

Permettere ad agenti intelligenti, come robot mobili, di esplorare ambienti chiusi e programmare azioni in base all'ambiente, richiede la conoscenza della geometria e della semantica della scena. Questo obbliga allenare tutti questi agenti con label semantici, il che a sua volta richiede un grande investimento di tempo e di denaro per etichettare immagini in modo corretto con la presenza di molte classi.

3.1 Semantic Neural Radiance Field

Il task di discernere tra diversi elementi semantici è strettamente correlato con la geometria della scena, come dimostrato da molti metodi *multi-task learning* [25]. È possibile per questo task sfruttare la capacità di NeRF di imparare la geometria della scena senza nessuna supervisione esplicita, tranne le immagini presenti. Però rimane un problema fondamentale, che pur imparando la geometria della scena in modo non supervisionato, le label semantiche utilizzate sono concetti definiti dagli essere umani e non possono

essere imparati senza una forma di supervisione umana. NeRF non facilita soltanto il processo di etichettatura, ma facilita anche il processo di generazione di etichette più precise e consistenti. NeRF viene modificato in [6], in modo tale da aggiungere un altro layer MLP di output per la label semantica Fig.3.1

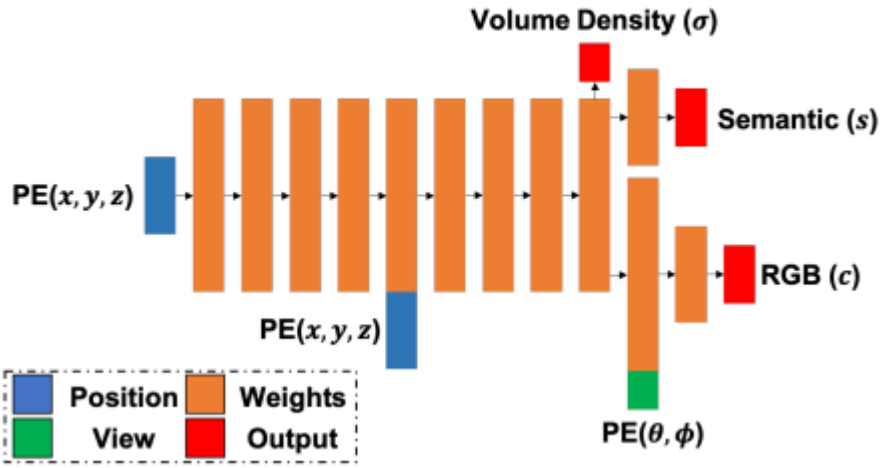


Figura 3.1: il modello di Semantic-NeRF[6]

La label semantica viene fatta dipendere solo dalla posizione e non dal punto di vista

$$c = F_\Theta(x, d) \qquad s = F_\Theta(x)$$

$$\hat{S}(r) = \sum_{k=1}^{K} T(t_k)(1 - exp(-\sigma(t_k)\delta_k))s(t_k) \tag{3.1}$$

$$\hat{S}(r) = \sum_{k=1}^{K} T(t_k)\alpha_k s(t_k) \tag{3.2}$$

$\hat{S}$ prima di essere passata alla funzione obiettivo viene trasformata in una distribuzione multi-classe attraverso un *softmax-layer*. Per l'allenamento della

rete vengono utilizziate due funzioni obiettivo:

$$L_p = \sum_{r \in \mathcal{R}} [\|\hat{C}_c(r) - C(r)\|_2^2 + \|\hat{C}_f(r) - C(r)\|_2^2]$$

$$L_s = -\sum_{r \in \mathcal{R}} [\sum_{l=1}^{L} p^l(r) log(\hat{p}_c^l(r)) + \sum_{l=1}^{L} p^l(r) log(\hat{p}_f^l(r))]$$

dove $\mathcal{R}$ sono i vari raggi campionati dal batch per l'allenamento e $C(r), \hat{C}_c(e), \hat{C}_f(r)$ sono rispettivamente il colore ground truth, il colore acquisito dal campionamento grossolano e il colore acquisito dal campionamento fine. Similarmente $p^l, \hat{p}_c^l, \hat{p}_f^l$ sono le probabilità semantiche multi-classe , della classe l del ground truth, del volume grossolano e del volume fine. La funzione obiettivo finale sarà:

$$\mathcal{L} = L_p + \lambda L_p$$

dove $\lambda \in \mathbb{R}$ uno scalare che gestisce quanta influenza ha la funzione obiettivo semantica.

3.1.1 Sparse Labels

In molti dataset reali ottenere label esatte di tutte le immagini è molto difficile. Quindi in [6] si sfrutta soprattutto la ripetitività dovuta al campionamento di frame molto ravvicinati (tecnica adoperata anche in tecnologie come SLAM) prendendo come key-frame un sottoinsieme di immagini per allenare il modello. La riduzione di label semantic va da 0% a 95% e in qualsiasi caso la riduzione di performance non è drastica, nel test-dataset come è possibile vedere da Fig.3.2 e da Fig.3.3

3.1.2 Semantic Fusion

Semantic-Nerf non è solo in grado di imparare rappresentazione semantiche con l'uso di poche annotazioni, grazie alla ripetitività presente nelle label semantiche, ma un'altra proprietà del modello è la *multi-view consistency* tra

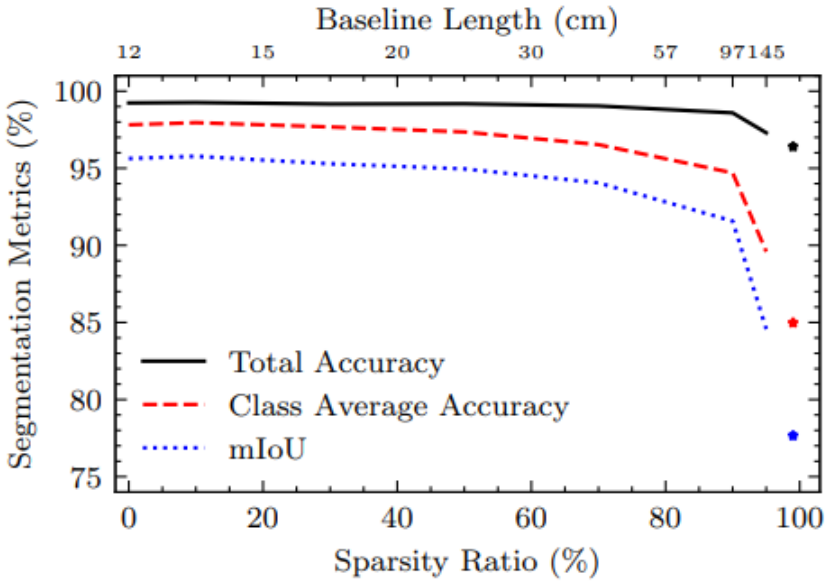


Figura 3.2: immagine presa da [6]. Performance quantitativa di Semantic-Nerf allenato su Replica con sparse semantic labels. Il *sparsity ratio* è la percentuale di frame che viene tralasciata rispetto alla sequenza per intero. Vengono adoperate tre metriche(più e alto il valore , migliore è la performance) per valutare la qualità della segmentazione semantica su determinati punti di vista presi dal *test-set*. La performance peggiora utilizzando meno label, motivo probabile è dovuto a regioni della scena che non vengono esplorate oppure occluse da determinati punti di vista. Possiamo notare che se pure diminuiamo il numero di label fino a un certo *threshold* circa 90% questo deteriora leggermente. Le $\star$ rappresentano la performance del modello usando soltanto due immagini con label. Le due immagini sono state scelte in modo tale da riprendere circa tutta la scena.

le varie label semantiche. Infatti la *multi-view consistency* permette di utilizzare label semantiche parzialmente etichettate o (regioni che non vengono etichettate, oppure gli viene assegnata la classe vuota) con presenza di rumore(singoli pixel vengono invertiti con altre classi) ed il modello riesce a fondere queste label in uno spazio 3D che permette di estrarli etichettati correttamente. Questa proprietà torna utile anche in mancanza di label semantiche, permette di utilizzare altri modelli segmentazione semantica e.g. *Deeplabv3* anche non molto preformanti usando i loro output come pseudo-label per il modello.

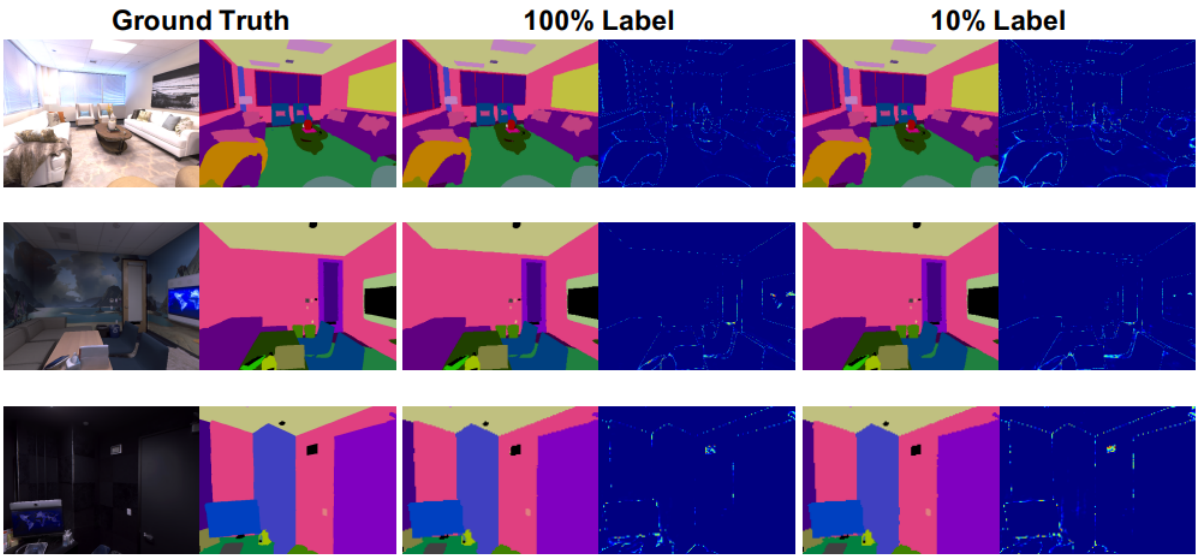


Figura 3.3: Da sinistra a destra mostra le immagini con i label semantici *ground truth* associati, al centro mostra i risultati utilizzando il 100% dei label dati dal dataset, mentre a destra utilizzando soltanto il 10%

3.1.3 SuperResolution

Un'altra applicazione molto utile è la super-risoluzione delle label semantiche (dato una label passare da bassa risoluzione con pochi dettagli a una maggiore $(16x16) \rightarrow (256, 256)$). Ad esempio in un *real-time semantic mapping system*, dove il modello semantico con pochi parametri deve predire ad una risoluzione più bassa le label semantiche. Semantic-NeRF è in grado di essere allenato con label a bassa risoluzione e dopo dare in output una versione più accurata con più dettagli. Vengono testate due strategie differenti. Nella prima le label semantiche vengono direttamente interpolate ad una risoluzione minore. Nel secondo caso la geometria dell'immagine viene preservata (la risoluzione rimane invariata), una serie di pixel viene classificata con le rispettive classi, mentre gli altri vengono tutti classificati con la classe nulla Fig.3.5 Questo mostra uno dei principali vantaggi nel rappresentare insieme la geometria della scena e la sua segmentazione semantica, permette di recuperare o correggere le label semantiche mancanti o corrotte

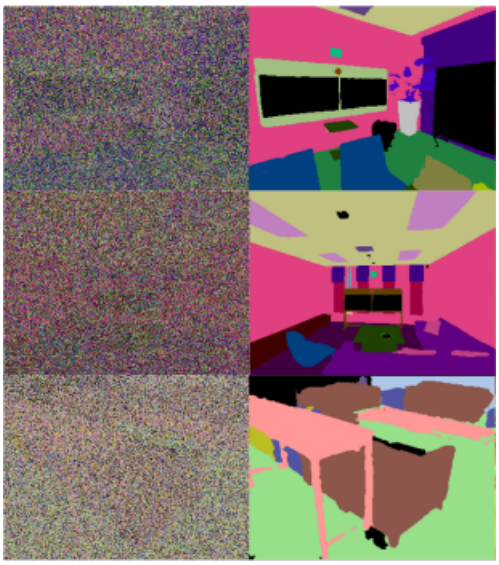


Figura 3.4: Da sinistra a destra mostra i label semantici con rumore usati come input, mentre a destra l'output del modello allenato dallo stesso punto di vista. Come è possibile vedere, il modello riesce a recuperare completamente il label semantico

in un frame attraverso la fusione di altri frame.

3.1.4 SS-NeRF

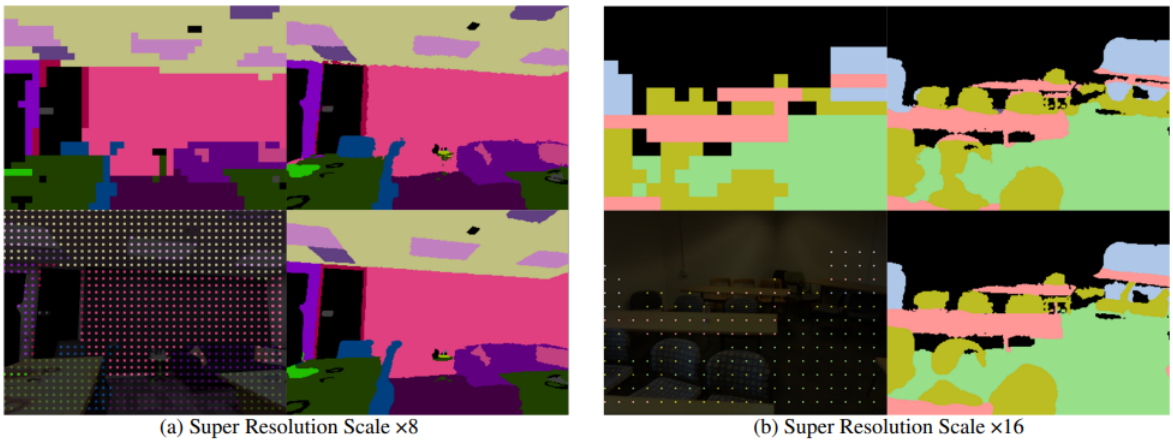
Un altro lavoro simile a Semantic-Nerf[6] è SS-Nerf[7] che oltre a fornire come output le label semantiche, aggiunge anche le normali alla superficie, il riconoscimento dei contorni, Shading e il riconoscimento di Keypoints. Il modello viene modificato per avere N(numero di proprietà) layer MLP dove ciascuno di loro produce una proprietà in output. Il modello Fig.3.6viene diviso in un encoder condiviso che prende in input la posizione x

$$e_{\mathbf{x}} = F_{enc}(x, y, z)$$

e in due decoder

$$\hat{p}_i^v = F_{dec}^v(e_{\mathbf{x}}, \theta, \phi) \qquad \hat{p}_i^{nv} = F_{dec}^{nv}(e_{\mathbf{x}})$$

la divisione è necessaria perché certe proprietà non hanno bisogno del punto di vista e.g. segmentazione semantica, normali alle superfici, mentre il restan-



(a) Super Resolution Scale ×8 (b) Super Resolution Scale ×16

Figura 3.5: In (a) vengono mostrati entrambi le strategie dove i label sono *downsampled* di un ottavo rispetto l'originale. L'immagine sopra corrisponde con la strategia di Interpolazione , mentre quella sotto con il campionamento che cerca di preservare la geometria dell'immagine. A destra di queste immagini possiamo vedere la predizione del modello .In (b) il valore di *downsampling* viene raddoppiato

te adopera questa informazione. Per l'ottimizzazione del modello vengono adoperate diverse funzioni obiettivo

$$\mathcal{L}_{rgb} = \sum_{r\in\mathcal{R}}[\|\hat{C}_c(r) - C(r)\|_2^2 + \|\hat{C}_f(r) - C(r)\|_2^2]$$

$$\mathcal{L}_{seg} = -\sum_{r\in\mathcal{R}}[\sum_{l=1}^{L} p^l(r) log(\hat{p}_c^l(r)) + \sum_{l=1}^{L} p^l(r) log(\hat{p}_f^l(r))]$$

$$\mathcal{L}_r = \sum_{r\in\mathcal{R}}[\|\hat{p}_c(r) - p(r)\|_1 + \|\hat{p}_f(r) - p(r)\|_1]$$

$$\mathcal{L} = \mathcal{L}_{rgb} + \lambda_1\mathcal{L}_{seg} + \sum_i \lambda_i \mathcal{L}_r^{(i)}$$

Le prime due funzioni di *loss* sono state introdotte in [9] ed in [6] mentre la terza *loss* è usata per le rimanenti proprietà da predire. Infine si sommano tutte le *loss* dove però ognuna, fatta eccezione $\mathcal{L}_{rgb}$, vengono moltiplicate per uno scalare λ_j.

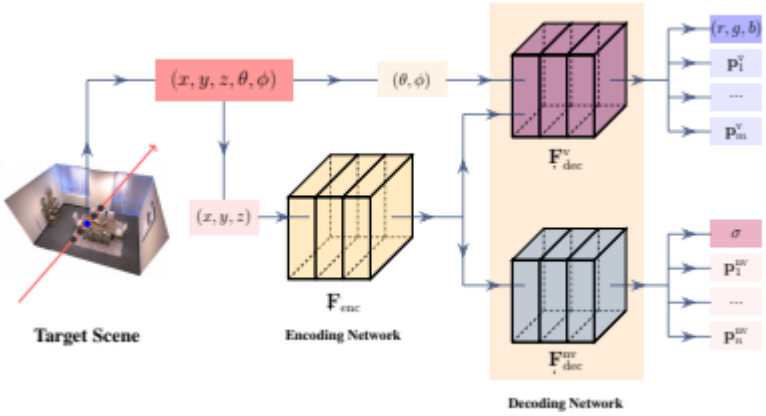


Figura 3.6: immagine presa da [7].il modello di SS-NeRF

3.1.5 Segmentazione semantica attraverso Label Propagation

Come si è visto prima Semantic-NeRF[6] è in grado di produrre label semantiche accurate partendo a sua volta da label sparse o inaccurate, attraverso la rappresentazione implicita dello spazio 3D. Quindi è possibile utilizzare un modello di segmentazione semantica f_ϕ allenato su una scena iniziale S_0 che produrrà delle label iniziali per allenare il semantic-Nerf e dopo utilizzare le sue label di output per rifinire il primo modello di segmentazione semantica. Questo tipo di setup non è molto lontano da quello che si può trovare nell'applicazione di ambienti virtuali o reali [10][8], dove si possono esplorare diverse scene, ed il dominio delle classi non cambia (il numero di classi non aumenta). Il modello iniziale f_ϕ allenato su $S_0 = \{I_0, Y_0\}$ dove S_i è la scena che comprende le immagini I_i e i rispettivi Y_0 label semantici, definiamo $\hat{Y}_i = \{f_\phi^{(i)}(x_0), \ldots, f_\phi^{(i)}(x_n)\}$ l'insieme di predizioni di f_ϕ per la scena i. L'obiettivo è quello di ottenere una performance migliore per ogni nuova scena S_i

$$metrica(f_\phi^{(i)}) > metrica(f_\phi^{(j)}) \qquad i > j$$

dove però per la scena S_i non è disponibile nessun label semantico (finché $i \neq 0$). Per quanto riguarda il modello NeRF $F_\Theta^{(i)}$ prenderà in input le po-

sizioni e i punti di vista ricavati dalle immagini della scena S_i ed in output restituirà la densità, il colore e il label semantico $\{\sigma, \hat{C}, \tilde{y}\}$. Una volta allenato il modello NeRF può potenzialmente generare infiniti nuovi *viewpoint* $\{\tilde{I}_i, \tilde{Y}_i\}$ che possono essere utilizzati per allenare f_ϕ. La motivazione principale è un aumento di performance da parte del modello f_ϕ.
Un altro dettaglio preso sempre da [8] è il regime di allenamento, infatti è possibile assumere due strategie. La prima consiste nell'allenare prima il modello F_Θ NeRF (utilizzando le label parziali ricavate da f_ϕ), per poi utilizzare le sue immagini e le sue label per supervisionare f_θ. Invece come seconda strategia allenare parzialmente il modello Nerf e poi iniziare simultaneamente ad allenare f_ϕ e F_Θ. Quest'ultima strategia mostra un miglioramento nelle performance rispetto alla prima, dovuta a una supervisione mutuale di entrambi i modelli. Inoltre per velocizzare a tempo di inferenza il modello NeRF possiamo trasformarlo in un octree[24] come trattato in precedenza su [22].Il modello che sarà utilizzato in questa tesi per la segmentazione semantica sarà UNet [**?**] con *backend* una rete *Efficient-Net0*[26].

Capitolo 4

Tecnologie ed Esperimenti

In questo capitolo verrà discusso l'implementazione pratica di tutti i modelli. Verranno introdotti il modello NeRF ed il modello Unet, con i dettagli del loro allenamento. Per implementare il codice di Unet viene usata la libreria implementata su Pytorch [27]. Per NeRF invece parte del codice sia per l'allenamento che per l'estrazione dei dati da Replica viene da nerf-pytorch nerf-w, semantic-nerf e ss-nerf

4.1 Replica dataset

Replica [28] è un dataset 3D con 112 classi differenti che rappresenta con grande accuratezza 18 scene con geometria, texture HDR ed label semantici. Il dataset utilizzato per questi esperimenti , viene realizzato usando una camera virtuale che esplora le differenti scene, cercando di riprodurre il movimento causato da una camera tenuta da un essere umano. Vengono create 900 immagini ad una risoluzione di 640×480. Per creare le immagini per l'allenamento, viene campionata un'immagine ogni cinque frame e poi alcuni frame intermedi vengono utilizzati per generare il test dataset.Per gli esperimenti vengono utilizzate le seguenti scene:

- office_0: 180 immagini di training e testing con le rispettive label semantiche e le posizioni della camera.

- office_2: 180 immagini di training e le posizioni della camera, nel test set vengono incluse anche le label semantiche.
- office_4: lo stesso set-up di *office_2*.

4.2 NeRF

Il modello NeRF è stato completamente implementato in Pytorch con l'ausilio di numpy. Il codice che si occupa della costruzione dei raggi e del campionamento dei punti.

4.2.1 Creazione raggi

```
def get_rays_camera(B, H, W, fx, fy,  cx, cy, depth_type,
    convention="opencv"):

     assert depth_type == "z" or depth_type == "euclidean"
     i, j = torch.meshgrid(torch.arange(W), torch.arange(H))
    we transpose to "xy" moode

     i = i.t().float()
     j = j.t().float()

     size = [B, H, W]

     i_batch = torch.empty(size)
     j_batch = torch.empty(size)
     i_batch[:, :, :] = i[None, :, :]
     j_batch[:, :, :] = j[None, :, :]

     if convention == "opencv":
         x = (i_batch - cx) / fx
         y = (j_batch - cy) / fy
         z = torch.ones(size)
     elif convention == "opengl":
         x = (i_batch - cx) / fx
```

```
        y = -(j_batch - cy) / fy
        z = -torch.ones(size)
    else:
        assert False

    dirs = torch.stack((x, y, z), dim=3)  # shape of [B, H, W, 3]

    if depth_type == 'euclidean':
        norm = torch.norm(dirs, dim=3, keepdim=True)
        dirs = dirs * (1. / norm)

    return dirs


def get_rays_world(T_WC, dirs_C):
    R_WC = T_WC[:, :3, :3]  # Bx3x3
    dirs_W = torch.matmul(R_WC[:, None, ...], dirs_C[..., None]).squeeze(-1)
    origins = T_WC[:, :3, -1]  # Bx3
    origins = torch.broadcast_tensors(origins[:, None, :], dirs_W)[0]
    return origins, dirs_W
```

La funzione **get_rays_camera** prende in input: la grandezza del batch B, la lunghezza H, la larghezza W, la lunghezza focale lungo l'asse x e l'asse y fx, fy, il centro cx lungo l'asse x ed il centro cy rispetto l'asse y(di solito $cx = cy$ però dipendono entrambi dalla risoluzione dell'immagine), *depth_type* l'asse di profondità (z) ed infine la convenzione usata dalla camera.OpenCV definisce x, y, z come destra, giù, davanti, mentre OpenGl definisce x, y, z come destra, su, dietro (da notare la camera guarda sempre verso l'avanti $-z$).L'importante è scegliere la stessa convenzione della posizione della camera.L'output della funzione **get_rays_camera** corrisponde alle direzioni dalla camera verso ogni pixel dell'immagine. L'altra funzione **get_rays_world** prende in input la matrice della camera(ricordiamo essere una $\mathbb{R}^{4\times4}$) e le direzioni generate precedentemente. In output quest'ultima

funzione restituisce l'origine della camera e dei raggi passante per ogni pixel.

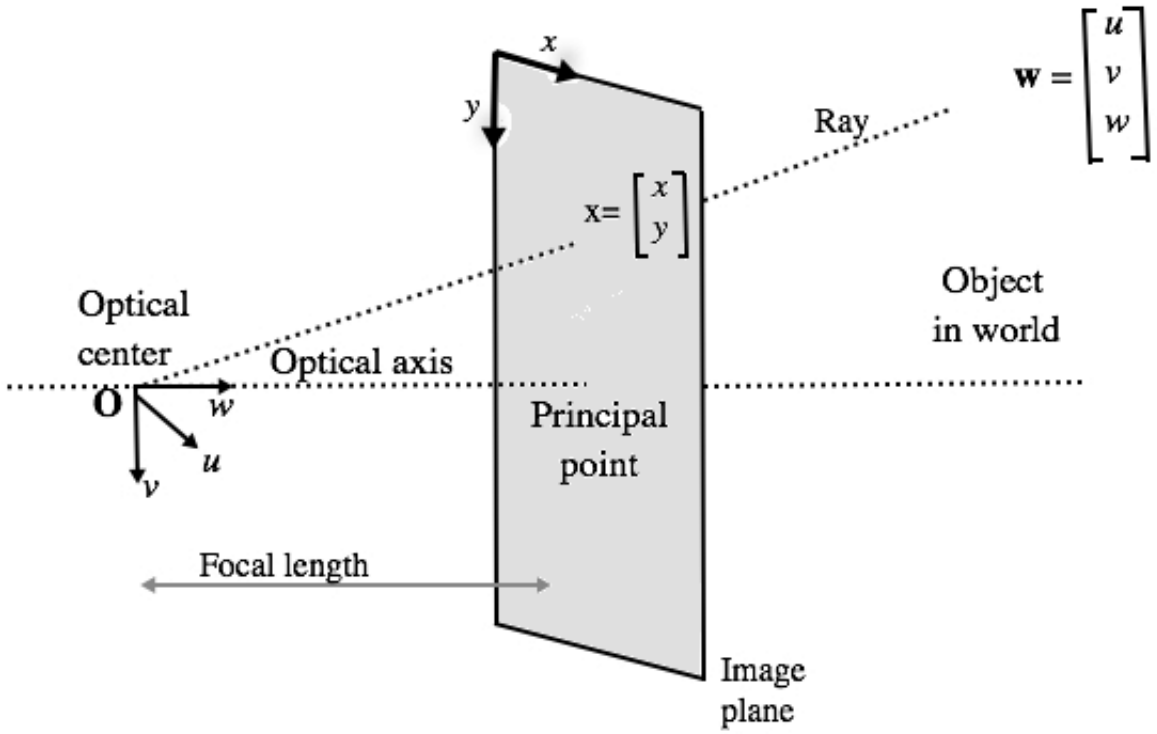


Figura 4.1: immagine presa da [9].

```
    def sample_stratified(
  rays_o: torch.Tensor,
  rays_d: torch.Tensor,
  near: float,
  far: float,
  n_samples: int,
  perturb: bool = True,
```

```
    inverse_depth: bool = False
):

    # Grab samples for space integration along ray
    t_vals = torch.linspace(0., 1., n_samples, device=rays_o.
     device)
    if not inverse_depth:
        # Sample linearly between `near` and `far`
        z_vals = near * (1.-t_vals) + far * (t_vals)
    else:
        # Sample linearly in inverse depth (disparity)
        z_vals = 1./(1./near * (1.-t_vals) + 1./far * (t_vals))

    # Draw uniform samples from bins along ray
    if perturb:
        mids = .5 * (z_vals[1:] + z_vals[:-1])
        upper = torch.concat([mids, z_vals[-1:]], dim=-1)
        lower = torch.concat([z_vals[:1], mids], dim=-1)
        t_rand = torch.rand([n_samples], device=z_vals.device)
        z_vals = lower + (upper - lower) * t_rand
    z_vals = z_vals.expand(list(rays_o.shape[:-1]) + [n_samples
     ])

    # Apply scale from `rays_d` and offset from `rays_o` to
     samples
    # pts: (width, height, n_samples, 3)
    pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals
     [..., :, None]
    return pts, z_vals
```

La funzione **sample_stratified** ha il compito di campionare i vari punti lungo i raggi generati. In input prende l'origine della camera, il raggio , *near plane, far plane*, il numero di campionamenti e due variabili flag. Nel progetto di tesi il *near plane* e il *far plane* corrispondono a un 0.1 metri e 10 metri. Il valore Ture del flag *perturb* perturba il campionamento (il punto viene spostato leggermente rispetto la posizione originale).Il valore False del flag *inverse_depth* dà origine ad n campionamento lineare tra i *near* ed *far*

plane.

4.2.2 NeRF modello

```
    class PositionalEncoder(nn.Module):
  r"""
  Sine-cosine positional encoder for input points.
  """
  def __init__(
    self,
    d_input: int,
    n_freqs: int,
    log_space: bool = False
  ):
    super().__init__()
    self.d_input = d_input
    self.n_freqs = n_freqs
    self.log_space = log_space
    self.d_output = d_input * (1 + 2 * self.n_freqs)
    self.embed_fns = [lambda x: x]

    # Define frequencies in either linear or log scale
    if self.log_space:
      freq_bands = 2.**torch.linspace(0., self.n_freqs - 1,
   self.n_freqs)
    else:
      freq_bands = torch.linspace(2.**0., 2.**(self.n_freqs -
    1), self.n_freqs)

    # Alternate sin and cos
    for freq in freq_bands:
      self.embed_fns.append(lambda x, freq=freq: torch.sin(x
   * freq))
      self.embed_fns.append(lambda x, freq=freq: torch.cos(x
   * freq))

  def forward(
```

```
30      self,
31      x
32    ) -> torch.Tensor:
33      r"""
34      Apply positional encoding to input.
35      """
36      return torch.concat([fn(x) for fn in self.embed_fns], dim
      =-1)
37
38  class Semantic_NeRF(nn.Module):
39      """
40      Compared to the NeRF class wich also predicts semantic
      logits from MLPs, here we make the semantic label only a
      function of 3D position
41      instead of both positon and viewing directions.
42      """
43      def __init__(self, enable_semantic, num_semantic_classes,
      D=8, W=256, input_ch=3, input_ch_views=3, output_ch=4,
      skips=[4], use_viewdirs=False,
44                   ):
45          super(Semantic_NeRF, self).__init__()
46          """
47                  D: number of layers for density (sigma)
      encoder
48                  W: number of hidden units in each layer
49                  input_ch: number of input channels for xyz
      (3+3*10*2=63 by default)
50                  in_channels_dir: number of input channels for
       direction (3+3*4*2=27 by default)
51                  skips: layer index to add skip connection in
      the Dth layer
52          """
53          self.D = D
54          self.W = W
55          self.input_ch = input_ch
56          self.input_ch_views = input_ch_views
57          self.skips = skips
58          self.use_viewdirs = use_viewdirs
```

```
59          self.enable_semantic = enable_semantic
60
61          # build the encoder
62          self.pts_linears = nn.ModuleList(
63              [nn.Linear(input_ch, W)] + [nn.Linear(W, W) if i
    not in self.skips else nn.Linear(W + input_ch, W) for i in
64                                          range(D - 1)])
65
66          ### Implementation according to the official code
    release (https://github.com/bmild/nerf/blob/master/
    run_nerf_helpers.py#L104-L105)
67
68          # Another layer is used to
69          self.views_linears = nn.ModuleList([nn.Linear(
    input_ch_views + W, W // 2)])
70          if use_viewdirs:
71              self.feature_linear = nn.Linear(W, W)
72              self.alpha_linear = nn.Linear(W, 1)
73              if enable_semantic:
74                  self.semantic_linear = nn.Sequential(nn.
    Linear(W, W//2),nn.ReLU(W // 2), nn.Linear(W // 2,
    num_semantic_classes))
75              self.rgb_linear = nn.Linear(W // 2, 3)
76          else:
77              self.output_linear = nn.Linear(W, output_ch)
78
79      def forward(self, x, show_endpoint=False):
80          """
81          Encodes input (xyz+dir) to rgb+sigma+semantics raw
    output
82          Inputs:
83              x: (B, self.in_channels_xyz(+self.in_channels_dir
    ))
84                  the embedded vector of 3D xyz position and
    viewing direction
85          """
86          input_pts, input_views = torch.split(x, [self.
    input_ch, self.input_ch_views], dim=-1)
```

```
        h = input_pts
        for i, l in enumerate(self.pts_linears):
            h = self.pts_linears[i](h)
            h = F.relu(h)
            if i in self.skips:
                h = torch.cat([input_pts, h], -1)

        if self.use_viewdirs:
            # if using view-dirs, output occupancy alpha as well as features for concatenation
            alpha = self.alpha_linear(h)
            if self.enable_semantic:
                sem_logits = self.semantic_linear(h)
            feature = self.feature_linear(h)

            h = torch.cat([feature, input_views], -1)

            for i, l in enumerate(self.views_linears):
                h = self.views_linears[i](h)
                h = F.relu(h)

            if show_endpoint:
                endpoint_feat = h
            rgb = self.rgb_linear(h)

            if self.enable_semantic:
                outputs = torch.cat([rgb, alpha, sem_logits], -1)
            else:
                outputs = torch.cat([rgb, alpha], -1)
        else:
            outputs = self.output_linear(h)

        if show_endpoint is False:
            return outputs
        else:
            return torch.cat([outputs, endpoint_feat], -1)
```

la classe **PositionalEncoder** serve per codificare la posizione e la direzione di vista con dei *positional encoding*. Il parametro *n_freqs* gestisce la dimensionalità del *positional encodeding* (*n_freqs*=10 per la posizione e *n_freqs* = 4).

4.2.3 Volume Rendering

La funzione **volumetric_rendering** in input prende tutti i raggi insieme alle direzioni di vista, mentre in output restituisce il colore estratto dal campionamento grossolano e quello più accurato insieme anche al label semantico. Per calcolare $\hat{C}, \{\alpha\}, \hat{S}$ viene utilizzata la funzione **raw2outputs**. La funzione **run_network** semplicemente codifica la posizione e la direzione di vista e poi la passa alla rete neurale. **sample_pdf** serve per calcolare il metodo dell'inversione.

```
def raw2outputs(raw, z_vals, rays_d, raw_noise_std=0,
    white_bkgd=False, enable_semantic=True,
                num_sem_class=0, endpoint_feat=False):
    """Transforms model's predictions to semantically
    meaningful values.
    Args:
        raw: [num_rays, num_samples along ray, 4]. Prediction
    from model.
        z_vals: [num_rays, num_samples along ray].
    Integration time.
        rays_d: [num_rays, 3]. Direction of each ray.
        raw_noise_std: random perturbations added to ray
    samples

    Returns:
        rgb_map: [num_rays, 3]. Estimated RGB color of a ray.
        disp_map: [num_rays]. Disparity map. Inverse of depth
    map.
        acc_map: [num_rays]. Sum of weights along each ray.
        weights: [num_rays, num_samples]. Weights assigned to
    each sampled color.
```

```
15          depth_map: [num_rays]. Estimated distance to object.
16      """
17      raw2alpha = lambda raw, dists, act_fn=F.relu: 1.-torch.
    exp(-act_fn(raw)*dists)
18
19      dists = z_vals[..., 1:] - z_vals[..., :-1]  # # (N_rays,
    N_samples_-1)
20      dists = torch.cat([dists, torch.Tensor([1e10]).expand(
    dists[..., :1].shape).cuda()], -1)  # [N_rays, N_samples]
21
22      # Multiply each distance by the norm of its corresponding
     direction ray
23      # to convert to real world distance (accounts for non-
    unit directions).
24      dists = dists * torch.norm(rays_d[..., None, :], dim=-1)
25
26      rgb = torch.sigmoid(raw[..., :3])  # [N_rays, N_samples,
    3]
27
28      if raw_noise_std > 0.:
29          noise = torch.randn(raw[..., 3].shape) *
    raw_noise_std
30          noise = noise.cuda()
31      else:
32          noise = 0.
33
34      alpha = raw2alpha(raw[..., 3] + noise, dists)  # [N_rays,
     N_samples]
35
36
37      # weights = alpha * tf.math.cumprod(1.-alpha + 1e-10, -1,
     exclusive=True)
38      weights = alpha * torch.cumprod(torch.cat([torch.ones((
    alpha.shape[0], 1)).cuda(), 1.-alpha + 1e-10], -1), -1)[:,
     :-1]
39      # [1, 1-a1, 1-a2, ...]
40      # [N_rays, N_samples+1] sliced by [:, :-1] to [N_rays,
    N_samples]
```

```
41
42      rgb_map = torch.sum(weights[..., None] * rgb, -2)  # [
    N_rays, 3]
43      # [N_rays, 3], the accumulated opacity along the rays,
    equals "1 - (1-a1)(1-a2)...(1-an)" mathematically
44
45      if enable_semantic:
46          assert num_sem_class>0
47          # https://discuss.pytorch.org/t/multi-class-cross-
    entropy-loss-and-softmax-in-pytorch/24920/2
48          sem_logits = raw[..., 4:4+num_sem_class]  # [N_rays,
    N_samples, num_class]
49          sem_map = torch.sum(weights[..., None] * sem_logits,
    -2)  # [N_rays, num_class]
50      else:
51          sem_map = torch.tensor(0)
52
53
54      if endpoint_feat:
55          feat = raw[..., -128:] # [N_rays, N_samples, feat_dim
    ] take the last 128 dim from predictions
56          feat_map = torch.sum(weights[..., None] * feat, -2)
    # [N_rays, feat_dim]
57      else:
58          feat_map = torch.tensor(0)
59
60      depth_map = torch.sum(weights * z_vals, -1)  # (N_rays,)
61      disp_map = 1./torch.max(1e-10 * torch.ones_like(depth_map
    ), depth_map / torch.sum(weights, -1))
62      acc_map = torch.sum(weights, -1)
63
64      if white_bkgd:
65          rgb_map = rgb_map + (1.-acc_map[..., None])
66          if enable_semantic:
67              sem_map = sem_map + (1.-acc_map[..., None])
68
69      return rgb_map, disp_map, acc_map, weights, depth_map,
    sem_map, feat_map
```

```
70
71
72 def sample_pdf(bins, weights, N_samples, det=False):
73     """ Sample @N_importance samples from @bins with
    distribution defined by @weights.
74
75     Inputs:
76         bins: N_rays x (N_samples_coarse - 1)
77         weights: N_rays x (N_samples_coarse - 2)
78         N_samples: N_samples_fine
79         det: deterministic or not
80     """
81     # Get pdf
82     weights = weights + 1e-5  # prevent nans, prevent
    division by zero (don't do inplace op!)
83     pdf = weights / torch.sum(weights, -1, keepdim=True)
84     cdf = torch.cumsum(pdf, -1)  # N_rays x (N_samples - 2)
85     cdf = torch.cat([torch.zeros_like(cdf[..., :1]), cdf],
    -1)  # N_rays x (N_samples_coarse - 1)
86     # padded to 0~1 inclusive, (N_rays, N_samples-1)
87
88     # Take uniform samples
89     if det:  # generate deterministic samples
90         u = torch.linspace(0., 1., steps=N_samples,  device=
    bins.device)
91         u = u.expand(list(cdf.shape[:-1]) + [N_samples])
92     else:
93         u = torch.rand(list(cdf.shape[:-1]) + [N_samples],
    device=bins.device)
94         # (N_rays, N_samples_fine)
95
96     # Invert CDF
97     u = u.contiguous()
98     inds = torch.searchsorted(cdf.detach(), u, right=True)  #
     N_rays x N_samples_fine
99     below = torch.max(torch.zeros_like(inds-1), inds-1)
100    above = torch.min((cdf.shape[-1]-1) * torch.ones_like(
    inds), inds)
```

```
101     inds_g = torch.stack([below, above], -1)  # (N_rays,
    N_samples_fine, 2)
102
103     matched_shape = [inds_g.shape[0], inds_g.shape[1], cdf.
    shape[-1]]  # (N_rays, N_samples_fine, N_samples_coarse -
    1)
104
105     cdf_g = torch.gather(cdf.unsqueeze(1).expand(
    matched_shape), 2, inds_g)  # N_rays, N_samples_fine, 2
106     bins_g = torch.gather(bins.unsqueeze(1).expand(
    matched_shape), 2, inds_g)  # N_rays, N_samples_fine, 2
107
108     denom = (cdf_g[..., 1]-cdf_g[..., 0])  # # N_rays,
    N_samples_fine
109     denom = torch.where(denom < 1e-5, torch.ones_like(denom),
     denom)
110     # denom equals 0 means a bin has weight 0, in which case
    it will not be sampled
111     # anyway, therefore any value for it is fine (set to 1
    here)
112
113     t = (u-cdf_g[..., 0])/denom
114     samples = bins_g[..., 0] + t * (bins_g[...,1]-bins_g
    [...,0])
115
116     return samples
117
118     def volumetric_rendering(self, ray_batch):
119         """
120         Volumetric Rendering
121         """
122         N_rays = ray_batch.shape[0]
123
124         rays_o, rays_d = ray_batch[:, 0:3], ray_batch[:, 3:6]
      # [N_rays, 3] each
125         viewdirs = ray_batch[:, -3:] if ray_batch.shape[-1] >
     8 else None
126
```

```
127
128         bounds = torch.reshape(ray_batch[..., 6:8], [-1, 1,
    2])
129         near, far = bounds[..., 0], bounds[..., 1]  # [N_rays
    , 1], [N_rays, 1]
130
131         #pts_coarse_sampled = sample_stratified(rays_o,rays_d
    ,near,far,self.N_samples,self.perturb > 0. and self.
    training,False)
132
133         t_vals = torch.linspace(0., 1., steps=self.N_samples)
    .cuda()
134
135         z_vals = near * (1. - t_vals) + far * (t_vals) # use
    linear sampling in depth space
136
137
138         z_vals = z_vals.expand([N_rays, self.N_samples])
139
140         if self.perturb > 0. and self.training:  # perturb
    sampling depths (z_vals)
141             if self.training is True:  # only add
    perturbation during training intead of testing
142                 # get intervals between samples
143                 mids = .5 * (z_vals[..., 1:] + z_vals[...,
    :-1])
144                 upper = torch.cat([mids, z_vals[..., -1:]],
    -1)
145                 lower = torch.cat([z_vals[..., :1], mids],
    -1)
146                 # stratified samples in those intervals
147                 t_rand = torch.rand(z_vals.shape).cuda()
148
149                 z_vals = lower + (upper - lower) * t_rand
150
151         pts_coarse_sampled = rays_o[..., None, :] + rays_d
    [..., None, :] * z_vals[..., :, None]  # [N_rays,
    N_samples, 3]
```

```
152
153         raw_noise_std = self.raw_noise_std if self.training
    else 0
154         raw_coarse = run_network(pts_coarse_sampled, viewdirs
    , self.ssr_net_coarse,
155                                  self.embed_fn, self.
    embeddirs_fn, netchunk=self.netchunk)
156         rgb_coarse, disp_coarse, acc_coarse, weights_coarse,
    depth_coarse, sem_logits_coarse, feat_map_coarse = \
157             raw2outputs(raw_coarse, z_vals, rays_d,
    raw_noise_std, self.white_bkgd, enable_semantic = self.
    enable_semantic,
158             num_sem_class = self.num_valid_semantic_class,
    endpoint_feat = False)
159
160
161         if self.N_importance > 0:
162             z_vals_mid = .5 * (z_vals[..., 1:] + z_vals[...,
    :-1])  # (N_rays, N_samples-1) interval mid points
163             z_samples = sample_pdf(z_vals_mid, weights_coarse
    [..., 1:-1], self.N_importance,
164                                    det=(self.perturb == 0.)
    or (not self.training))
165             z_samples = z_samples.detach()
166             # detach so that grad doesn't propogate to
    weights_coarse from here
167             # values are interleaved actually, so maybe can
    do better than sort?
168
169             z_vals, _ = torch.sort(torch.cat([z_vals,
    z_samples], -1), -1)
170             pts_fine_sampled = rays_o[..., None, :] + rays_d
    [..., None, :] * z_vals[..., :, None]  # [N_rays,
    N_samples + N_importance, 3]
171
172
173             raw_fine = run_network(pts_fine_sampled, viewdirs
    , lambda x: self.ssr_net_fine(x, self.endpoint_feat),
```

```
174                         self.embed_fn, self.embeddirs_fn,
    netchunk=self.netchunk)
175
176             rgb_fine, disp_fine, acc_fine, weights_fine,
    depth_fine, sem_logits_fine, feat_map_fine = \
177                 raw2outputs(raw_fine, z_vals, rays_d,
    raw_noise_std, self.white_bkgd, enable_semantic = self.
    enable_semantic,
178                 num_sem_class = self.num_valid_semantic_class
    , endpoint_feat = self.endpoint_feat)
179
180         ret = {}
181         ret['raw_coarse'] = raw_coarse
182         ret['rgb_coarse'] = rgb_coarse
183         ret['disp_coarse'] = disp_coarse
184         ret['acc_coarse'] = acc_coarse
185         ret['depth_coarse'] = depth_coarse
186         if self.enable_semantic:
187             ret['sem_logits_coarse'] = sem_logits_coarse
188
189         if self.N_importance > 0:
190             ret['rgb_fine'] = rgb_fine
191             ret['disp_fine'] = disp_fine
192             ret['acc_fine'] = acc_fine
193             ret['depth_fine'] = depth_fine
194             if self.enable_semantic:
195                 ret['sem_logits_fine'] = sem_logits_fine
196             ret['z_std'] = torch.std(z_samples, dim=-1,
    unbiased=False)  # [N_rays]
197             ret['raw_fine'] = raw_fine  # model's raw,
    unprocessed predictions.
198             if self.endpoint_feat:
199                 ret['feat_map_fine'] = feat_map_fine
200         for k in ret:
201             # if (torch.isnan(ret[k]).any() or torch.isinf(
    ret[k]).any()) and self.config["experiment"]["debug"]:
202             if (torch.isnan(ret[k]).any() or torch.isinf(ret[
    k]).any()):
```

```
                print(f"! [Numerical Error] {k} contains nan or inf.")

        return ret
```

4.2.4 Training

```
    def step(
        self,
        global_step
    ):
        # Misc
        img2mse = lambda x, y: torch.mean((x - y) ** 2)
        mse2psnr = lambda x: -10. * torch.log(x) / torch.log(torch.Tensor([10.]).cuda())
        CrossEntropyLoss = nn.CrossEntropyLoss(ignore_index=self.ignore_label)
        KLDLoss = nn.KLDivLoss(reduction="none")
        kl_loss = lambda input_log_prob,target_prob: KLDLoss(input_log_prob, target_prob)
        # this function assume input is already in log-probabilities

        crossentropy_loss = lambda logit, label: CrossEntropyLoss(logit, label-1)  # replica has void class of ID==0, label-1 to shift void class to -1

        logits_2_label = lambda x: torch.argmax(torch.nn.functional.softmax(x, dim=-1),dim=-1)
        logits_2_prob = lambda x: F.softmax(x, dim=-1)
        to8b_np = lambda x: (255 * np.clip(x, 0, 1)).astype(np.uint8)
        to8b = lambda x: (255 * torch.clamp(x, 0, 1)).type(torch.uint8)


        # sample rays to query and optimise
```

```
22          sampled_data = self.sample_data(global_step, self.
    rays, self.H, self.W, no_batching=True, mode="train")
23          if self.enable_semantic:
24              sampled_rays, sampled_gt_rgb, sampled_gt_depth,
    sampled_gt_semantic, sematic_available = sampled_data
25          else:
26              sampled_rays, sampled_gt_rgb = sampled_data
27
28          output_dict = self.render_rays(sampled_rays)
29
30          rgb_coarse = output_dict["rgb_coarse"]  # N_rays x 3
31          disp_coarse = output_dict["disp_coarse"] # N_rays
32          depth_coarse = output_dict["depth_coarse"] # N_rays
33          acc_coarse = output_dict["acc_coarse"] # N_rays
34          if self.enable_semantic:
35              sem_logits_coarse = output_dict["
    sem_logits_coarse"] # N_rays x num_classes
36              sem_label_coarse = logits_2_label(
    sem_logits_coarse)  # N_rays
37
38          if self.N_importance > 0:
39              rgb_fine = output_dict["rgb_fine"]
40              disp_fine = output_dict["disp_fine"]
41              depth_fine = output_dict["depth_fine"]
42              acc_fine = output_dict["acc_fine"]
43              z_std = output_dict["z_std"] # N_rays
44              if self.enable_semantic:
45                  sem_logits_fine = output_dict["
    sem_logits_fine"]
46                  sem_label_fine = logits_2_label(
    sem_logits_fine)
47
48          self.optimizer.zero_grad()
49
50          img_loss_coarse = img2mse(rgb_coarse, sampled_gt_rgb)
51
52          if self.enable_semantic and sematic_available:
```

```
            semantic_loss_coarse = crossentropy_loss(sem_logits_coarse, sampled_gt_semantic)
        else:
            semantic_loss_coarse = torch.tensor(0)


        with torch.no_grad():
            psnr_coarse = mse2psnr(img_loss_coarse)

        if self.N_importance > 0:
            img_loss_fine = img2mse(rgb_fine, sampled_gt_rgb)
            if self.enable_semantic and sematic_available:
                semantic_loss_fine = crossentropy_loss(sem_logits_fine, sampled_gt_semantic)
            else:
                semantic_loss_fine = torch.tensor(0)
            with torch.no_grad():
                psnr_fine = mse2psnr(img_loss_fine)
        else:
            img_loss_fine = torch.tensor(0)
            psnr_fine = torch.tensor(0)

        total_img_loss = img_loss_coarse + img_loss_fine
        total_sem_loss = semantic_loss_coarse + semantic_loss_fine


        wgt_sem_loss = float(self.config["train"]["wgt_sem"])
        total_loss = total_img_loss + total_sem_loss* wgt_sem_loss

        total_loss.backward()
        self.optimizer.step()

        ###   update learning rate   ###
        decay_rate = 0.1
        decay_steps = self.lrate_decay
```

```
86          new_lrate = self.lrate * (decay_rate ** (global_step
    / decay_steps))
87          for param_group in self.optimizer.param_groups:
88              param_group['lr'] = new_lrate
89          ################################
```

NeRF utilizza come *optimizer* Adam con *learning rate* $5e-4$, inoltre utilizziamo uno scheduler che corrisponde ad :

$$lr = lr * (0.1^{\frac{global\_step}{250k}})$$

dove *global_step* corrisponde all'iterazione di allenamento, il numero di iterazioni massime sono $200k$ nel nostro caso.

4.2.5 Novel viewpoints

Per generare la scena da nuovi *viewpoint* ci basta utilizzare la funzione **generate_camera_poses** che prende in input il centro(verso cui punta sempre la camera), il raggio unitario, ed il numero di *viewpoint* da generare. Questa funzione campiona posizioni prese da una semisfera con raggio uno ed il centro corrisponde all'origine della semisfera. **new_viewpoints_rays** semplicemente inizializza tutti i raggi con le nuove posizioni della camera. Infine **new_paths** prende il modello più recente e restituisce colore e label semantico per ogni nuovo raggio creato in precedenza.

```

2     def generate_camera_poses(center, radius, num_poses=180):
3   camera_poses = []
4   for _ in range(num_poses):
5     # Sample random angles within the hemisphere
6     theta = np.random.uniform(0, np.pi / 2)  # Polar angle
7     phi = np.random.uniform(0, 2 * np.pi)    # Azimuthal
     angle
8
9     # Convert spherical coordinates to Cartesian coordinates
10    x = radius * np.sin(theta) * np.cos(phi)
```

```
11      y = radius * np.sin(theta) * np.sin(phi)
12      z = radius * np.cos(theta)
13
14      # Create a camera pose matrix (4x4) using look-at
    transformation
15      # You may choose a target point or origin as the point to
     look at
16      target_point = np.array(center)  # Change this as needed
17      camera_position = np.array([x, y, z])
18      forward = target_point - camera_position
19      forward /= np.linalg.norm(forward)
20      right = np.cross([0, 1, 0], forward)
21      right /= np.linalg.norm(right)
22      up = np.cross(forward, right)
23      up /= np.linalg.norm(up)
24
25      pose_matrix = np.eye(4)
26      pose_matrix[:3, :3] = np.vstack((right, up, -forward)).T
27      pose_matrix[:3, 3] = camera_position
28
29      camera_poses.append(pose_matrix)
30  return camera_poses
31
32      def new_viewpoints_rays(self, n_rays=180):
33          camera_poses = generate_camera_poses([0,0,0], 1,
     num_poses=n_rays)
34          rays = create_rays(n_rays, torch.tensor(camera_poses).
     float(), self.H_scaled, self.W_scaled, self.fx_scaled,
     self.fy_scaled,
35                                  self.cx_scaled, self.
     cy_scaled, self.near, self.far, use_viewdirs=self.
     use_viewdir, convention=self.convention)
36          self.new_rays = rays.cuda()
37
38      def new_paths(self, path):
39          to8b_np = lambda x: (255 * np.clip(x, 0, 1)).astype(np.
     uint8)
40          self.load_models(path)
```

```
41
42      self.training = False  # enable testing mode before
    rendering results, need to set back during training!
43      self.ssr_net_coarse.eval()
44      self.ssr_net_fine.eval()
45      testsavedirRGB = os.path.join(self.config["experiment"
    ]["save_dir"], 'training_cnn', 'rgb')
46      testsavedirLabel = os.path.join(self.config["experiment
    "]["save_dir"], 'training_cnn', 'label')
47      os.makedirs(testsavedirRGB, exist_ok=True)
48      os.makedirs(testsavedirLabel, exist_ok=True)
49      with torch.no_grad():
50          rgbs, vis_sems, sems = self.
    render_path_for_training(self.new_rays, save_dir_rgbs=
    testsavedirRGB, save_dir_labels = testsavedirLabel)
51          imageio.mimwrite(os.path.join(testsavedirRGB, 'rgb.
    mp4'), to8b_np(rgbs), fps=30, quality=8)
52          imageio.mimwrite(os.path.join(testsavedirLabel, '
    vis_sems.mp4'), vis_sems, fps=30, quality=8)
```

4.3 Unet

Per implementare il modello Unet è stata utilizzata , in qesta tesi la libreria **smp** con *backend* efficientNet-0 Inoltre per allenare il modello diversamente da come fatto con il modello NeRF , utilizziamo una *Dice Loss*:

$$\mathcal{L}_{DSC} = 1 - DSC(\hat{y}, y)$$

```
1   class UnetModel(nn.Module):
2   def __init__(self, cfg):
3       super(UnetModel, self).__init__()
4
5       self.cfg = cfg
6       self.training = True
7
8       self.model = smp.Unet(
```

```
            encoder_name=cfg.encoder,
            encoder_weights=cfg.weights,
            classes=num_sem_classes,
            activation=cfg.activation,
        )

        self.loss_fn = smp.losses.DiceLoss(mode='multiclass',
    classes = [i for i in range(num_sem_class)],ignore_index
    = -1)

    def forward(self, imgs, targets=None, train=False):

        x = imgs
        y = targets

        logits = self.model(x)
        #print(y.view(16, -1))
        if train and targets!=None:
            loss = self.loss_fn(logits, y.long())
            return {"loss": loss, "logits": logits, "
    logits_raw": logits, "target": y}
        return {"logits_raw": logits, "logits": logits}
```

4.3.1 Training Unet

```
    def train(model, train_dataloader, test_dataloader,
    optimizer, epochs, device, scheduler_plateau=None,
    image_sizes = [384, 512]):
    results = {'train_loss': [],
               'val_loss': [np.inf],
               'val_dice': [0.0]}
    for epoch in range(epochs):

        set_seed(Config.seed + epoch)
        print("EPOCH:", epoch)
        train_loss = train_step(model,
                                train_dataloader,
```

```
                                optimizer,
                                device)
        val_loss, val_dice = test_step(model,
                               test_dataloader,
                               device)

        train_loss = train_loss / len(train_dt)
        val_loss = val_loss / len(valid_dt)

        if scheduler_plateau != None:
          scheduler_plateau.step(val_loss)

        print(f'Train Loss: {train_loss:.4f} | Val Loss: {
    val_loss:.4f} | Val Dice: {val_dice:.4f}')
        print(f"Learning rate: {optimizer.param_groups[0]['lr
    ']}")

        if(np.min(results['val_loss']) >= val_loss and np.max
    (results['val_dice']) <= val_dice):
            PATH = "model.ckpt"
            torch.save(model.state_dict(), PATH)
            print(f"Saving new model {epoch}")
        results['train_loss'].append(train_loss)
        results['val_loss'].append(val_loss)
        results['val_dice'].append(val_dice)

    return results
```

Il training di Unet è avvenuto in 30 epoche con *batch size* 16, come *optimizer* per Unet viene utilizzato Adam con *learning rate* $3e-4$ e come *scheduler* *CosineAnneling* con 100 iterazione di *warmup* (ovvero il *learning rate* parte da 0 per arrivare ad $3e-4$ in 100 iterazione prima di iniziare la discesa).Il *backend* del modello *EfficientNet-0* è *pretrained* su *imagenet*, perché adoperare reti neurali *pretrained* anche su dataset con un dominio diverso aiuta nell'accorciare i tempi di training e migliora la performance più velocemente[3][4]. È possibile salvare il modello NeRF in una memoria a lungo termine sic-

come il peso per ogni scena corrisponde soltanto a 14.5MB. La strategie di allenamento complessiva è la seguente:

1. All'inizio viene allenato il modello Semantic-NeRF su 180 immagini prese dalla scena *office_0* con le rispettive label semantiche. Una volta finito l'allenamento il modello viene utilizzato per generare nuove immagini e label semantiche. Vengono generate 10000 immagini e etichette e usate come training set per il modello Unet.

2. viene utilizzata la strategia di allenamento suggerita nella sezione 3.1.5. I label generati da Semantic-Nerf servono per supervisionare l'allenamento della rete Unet. Naturalmente per diminuire il fenomeno di *overfitting* con così poche immagini, vengono applicate varie forme di regolarizzazione al modello e al dataset. Innanzitutto viene utilizzato il weight-decay $1e-4$ e *RandAugment*[29](Trasformazioni applicate sulle immagini di training e.g. *Gaussian noise, brightness contrast, ...*) per aumentare il numero di immagini di training, attraverso la libreria Albumentations[30].

3. Data una nuova scena (*office_2*) dove abbiamo soltanto le immagini di input I_{office_1}, le label semantiche $\hat{Y}_{office\_2}$ vengono generate attraverso il modello Unet già allenato sulla scena precedente.

4. Dato $\{I_{office\_2}, \hat{Y}_{office\_2}\}$ possiamo utilizzarli per allenare un nuovo Semantic-Nerf e generare come al primo passo nuove immagini per il miglioramento di performance del modello di segmentazione semantica.

4.3.2 Note sui esperimenti

Per tutti gli esperimenti si è adoperata una GPU Nvidia T4 con 15GB di VRAM. La ricostruzione della scena 3d data da NeRF su punti di vista non usati duranti il training, genera scene con artefatti geometrici 4.2.

L'origine di questi artefatti geometrici è dovuta ad un problema intrinseco al *Multilayer Perceptron* che soffre di [19], le soluzioni a questo problema

mIoU↑		
scenes	Unet w/o NeRF	Unet w/ NeRF
office_0	—	0.805
office_2	0.643	0.837
office_4	0.667	0.851

Tabella 4.1: A sinistra il risultato senza l'ausilio dei dati generati da NeRF, mentre a destra Unet allenato con i dati generati da NeRF. Nella prima riga colonna a sinistra il risultato è vuoto senza l'ausilio di NeRF perché possediamo soltanto 180 immagini. Ad ogni riga riutilizziamo il modello Unet allenato nella scena precedente , è possibile vedere il miglioramento delle performance sia nella colonna a destra che a sinistra. Il miglioramento di Unet è dovuto al fatto che le scene adoperate sono tutte *in-domain* (la distribuzione delle classi tra scena e scena varia di poco).Anche se

possono essere adoperare metodi più complessi che non utilizzano componenti neurali [23] [31] , ovvero non rappresentano la scena 3d implicitamente attraverso la rete neurale.

Figura 4.2: Esempio di artefatto geometrico (scena *office_2*)

Figura 4.3: Esempio di artefatto geometrico (scena *office_0*)

Figura 4.4: Esempio di segmentazione data un'immagine di *office_4*

Conclusioni

Come visto dai risultati Table.4.1 l'utilità di questo metodo, può portare miglioramenti nel task di *scene understanding* mediante la generazione di label per la segmentazione semantica di una scena. Anche se i risultati sono promettenti, ci sono possibili miglioramenti sull'architettura che potrebbero essere provati.

- Gli artefatti geometrici presenti nella scena 3D , quando nuovi punti di vista vengono campionati.Questi artefatti influiscono sia sulla performance della segmentazione che sull'accuratezza della scena. Un possibile miglioramento è trovare un modo per combinare [31] con un modulo di segmentazione semantica.

- Un altro possibile miglioramento è l'utilizzo di architetture più complesse nella componente di segmentazione del Semantic-NeRF. Infatti viene adoperato un semplice blocco MLP (2 MLP layer con in mezzo un'activation function). Si potrebbero adoperare una forma di decoder più complesso che può essere riutilizzato con altri Semantic-Nerf.

Bibliografia

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL `http://arxiv.org/abs/1505.04597`.

[2] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *CoRR*, abs/1606.00915, 2016. URL `http://arxiv.org/abs/1606.00915`.

[3] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017. URL `http://arxiv.org/abs/1706.05587`.

[4] Jun Fu, Jing Liu, Haijie Tian, Zhiwei Fang, and Hanqing Lu. Dual attention network for scene segmentation. *CoRR*, abs/1809.02983, 2018. URL `http://arxiv.org/abs/1809.02983`.

[5] Zilong Huang, Xinggang Wang, Yunchao Wei, Lichao Huang, Humphrey Shi, Wenyu Liu, and Thomas S. Huang. Ccnet: Criss-cross attention for semantic segmentation. *CoRR*, abs/1811.11721, 2018. URL `http://arxiv.org/abs/1811.11721`.

[6] Shuaifeng Zhi, Tristan Laidlow, Stefan Leutenegger, and Andrew J. Davison. In-place scene labelling and understanding with implicit scene

representation. *CoRR*, abs/2103.15875, 2021. URL https://arxiv.org/abs/2103.15875.

[7] Mingtong Zhang, Shuhong Zheng, Zhipeng Bao, Martial Hebert, and Yu-Xiong Wang. Beyond rgb: Scene-property synthesis with neural radiance fields, 2022.

[8] Zhizheng Liu, Francesco Milano, Jonas Frey, Roland Siegwart, Hermann Blum, and Cesar Cadena. Unsupervised continual semantic adaptation through neural rendering, 2023.

[9] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *CoRR*, abs/2003.08934, 2020. URL https://arxiv.org/abs/2003.08934.

[10] Jonas Frey, Hermann Blum, Francesco Milano, Roland Siegwart, and Cesar Cadena. Continual adaptation of semantic segmentation using complementary 2d-3d data representations. *IEEE Robotics and Automation Letters*, 7(4):11665–11672, 2022. doi: 10.1109/LRA.2022.3203812.

[11] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. *CoRR*, abs/2103.13415, 2021. URL https://arxiv.org/abs/2103.13415.

[12] Edward Adelson and James Bergen. The plenoptic function and the elements of early vision. 08 1997.

[13] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *CoRR*, abs/1612.01105, 2016. URL http://arxiv.org/abs/1612.01105.

[14] Hengshuang Zhao, Yi Zhang, Shu Liu, Jianping Shi, Chen Change Loy, Dahua Lin, and Jiaya Jia. Psanet: Point-wise spatial attention network

for scene parsing. In *European Conference on Computer Vision*, 2018. URL `https://api.semanticscholar.org/CorpusID:52958802`.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

[16] Xiaolong Wang, Ross B. Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. *CoRR*, abs/1711.07971, 2017. URL `http://arxiv.org/abs/1711.07971`.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL `https://arxiv.org/abs/2010.11929`.

[18] Pratul Srinivasan Jon Barron Angjoo Kanazawa Matt Tancik, Ben Mildenhall. Eccv 2022 tutorial neural volumetric rendering for computer vision. `https://sites.google.com/berkeley.edu/nerf-tutorial/home`, 2022.

[19] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A. Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks, 2019.

[20] Onur Özyesil, Vladislav Voroninski, Ronen Basri, and Amit Singer. A survey on structure from motion. *CoRR*, abs/1701.08493, 2017. URL `http://arxiv.org/abs/1701.08493`.

[21] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the

wild: Neural radiance fields for unconstrained photo collections. *CoRR*, abs/2008.02268, 2020. URL `https://arxiv.org/abs/2008.02268`.

[22] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. Plenoctrees for real-time rendering of neural radiance fields. *CoRR*, abs/2103.14024, 2021. URL `https://arxiv.org/abs/2103.14024`.

[23] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. *CoRR*, abs/2112.05131, 2021. URL `https://arxiv.org/abs/2112.05131`.

[24] Samuli Laine and Tero Karras. Efficient sparse voxel octrees – analysis, extensions, and implementation. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, February 2010.

[25] Shikun Liu, Edward Johns, and Andrew J. Davison. End-to-end multi-task learning with attention. *CoRR*, abs/1803.10704, 2018. URL `http://arxiv.org/abs/1803.10704`.

[26] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019. URL `http://arxiv.org/abs/1905.11946`.

[27] Pavel Iakubovskii. Segmentation models pytorch. `https://github.com/qubvel/segmentation_models.pytorch`, 2019.

[28] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J. Engel, Raul Mur-Artal, Carl Yuheng Ren, Shobhit Verma, Anton Clarkson, Mingfei Yan, Brian Budge, Yajie Yan, Xiaqing Pan, June Yon, Yuyang Zou, Kimberly Leon, Nigel Carter, Jesus Briales, Tyler Gillingham, Elias Mueggler, Luis Pesqueira, Manolis Savva, Dhruv Batra, Hauke M. Strasdat, Renzo De Nardi, Michael Goesele, Steven Lovegrove, and Richard A. Newcombe. The replica dataset:

A digital replica of indoor spaces. *CoRR*, abs/1906.05797, 2019. URL http://arxiv.org/abs/1906.05797.

[29] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical data augmentation with no separate search. *CoRR*, abs/1909.13719, 2019. URL http://arxiv.org/abs/1909.13719.

[30] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020. ISSN 2078-2489. doi: 10.3390/info11020125. URL https://www.mdpi.com/2078-2489/11/2/125.

[31] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023. URL https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.