Image segmentation with NVIDIA Omniverse and UNet

Federico Dittaro
DMIF, Università degli studi di Udine, Italia

27 settembre 2024

Indice

Capitolo 1

Introduzione

La segmentazione delle immagini è uno degli argomenti portanti della Computer Vision le cui applicazioni spaziano dalla guida autonoma all'analisi medica. Questo processo consiste nel dividere un'immagine in più parti o regioni, ciascuna delle quali rappresenta un oggetto o un'area di interesse specifica.

Tra le reti neurali convoluzionali, l'architettura UNet si è affermata come una delle più efficaci per la segmentazione semantica, grazie alla sua capacità di catturare dettagli sia globali che locali.

In questo progetto è stata affrontata la problematica della segmentazione delle immagini utilizzando un approccio basato su deep learning. È stato creato un dataset personalizzato di immagini e le rispettive maschere di segmentazione, sfruttando l'ambiente di simulazione NVIDIA Omniverse. La piattaforma Omniverse ha permesso di generare un dataset sintetico di 2000 immagini, accompagnate dalle relative maschere segmentate, garantendo un controllo accurato sui dati e una vasta varietà di scenari, oggetti ed illuminazione.

Il dataset così ottenuto è stato utilizzato per addestrare una rete UNet, con un backbone ResNet34 per consentire una migliore estrazione delle caratteristiche. Dopo l'addestramento, la rete è stata testata su nuove immagini non presenti nel dataset di addestramento, per valutare la capacità di generalizzazione del modello.

Questo report descrive il processo di generazione del dataset, l'implementazione del modello UNet ed i risultati ottenuti.

Capitolo 2

Stato dell'arte

2.1 Computer vision

La visione artificiale (o computer vision) è un campo dell'intelligenza artificiale che permette ai computer e ai sistemi di ricavare informazioni significative da immagini digitali, video e altri input visivi e di intraprendere azioni o formulare delle segnalazioni sulla base di esse. Se l'IA permette ai computer di pensare, la computer vision permette loro di vedere, osservare e capire.

La computer vision può essere paragonata alla vista umana eccetto che quest'ultima ha il vantaggio di disporre di anni e anni di esperienza in cui si è allenata a distinguere gli oggetti, valutare la loro distanza, movimento e rilevare eventuali errori. La computer vision permette alle macchine di svolgere queste funzioni attraverso l'utlizzo di telecamere, dati e algoritmi. Un sistema sviluppato per ispezionare i prodotti o guardare un asset di produzione può analizzare migliaia di prodotti o processi al minuto, notando difetti o problemi impercettibili, superando rapidamente le capacità umane.

La computer vision ha bisogno di molti dati. Esegue delle analisi più e più volte fino a quando non distingue e riconosce le immagini. Per realizzare ciò vengono utilizzate essenzialmente due tecnologie: il deep learning e una rete neurale convoluzionale (CNN - convolutional neural network).

Le principali aree della computer vision sono le seguenti:

- Classificazione delle immagini: permette di stabilire con precisione la classe di appartenenza di una certa immagine;
- Rilevamento di oggetti: usa la classificazione delle immagini per identificare una certa classe per poi rilevare e tabulare la loro apparizione in un'immagine o in un video;
- Object tracking: permette di seguire o tracciare un oggetto una volta che è stato rilevato.

2.2 Image segmentation

La segmentazione delle immagini è una tecnica di computer vision che suddivide un'immagine digitale in gruppi di pixel (segmenti di immagine) per informare il rilevamento degli oggetti e le attività correlate. Gli algoritmi convenzionali di segmentazione delle immagini elaborano funzioni visive di alto livello di ciascun pixel, come il colore o la luminosità, per identificare i confini degli oggetti e le regioni di sfondo. Il machine learning, che sfrutta i dataset annotati, viene utilizzato per addestrare i modelli a classificare con precisione i tipi specifici di oggetti e regioni contenuti in un'immagine.

Essendo un metodo altamente versatile, la segmentazione delle immagini ha un'ampia varietà di casi d'uso dell'intelligenza artificiale, dall'aiuto alla diagnosi nell'imaging medico alle auto a guida autonoma fino all'identificazione di oggetti di interesse nelle immagini satellitari.

2.3 NVIDIA

NVIDIA Corporation è un'azienda statunitense fondata nel 1993 conosciuta a livello globale per il suo ruolo fondamentale nello sviluppo di unità di elaborazione grafica (GPU). La missione di NVIDIA è quella di reinventare il computing visivo, offrendo soluzioni che spaziano dall'intrattenimento alla ricerca scientifica, fino alle applicazioni nel campo dell'intelligenza artificiale e del machine learning.

Le sue GPU, in particolare l'architettura CUDA, sono diventate fondamentali nell'ambito del deep learning, rendendo NVIDIA un attore chiave nello sviluppo di AI e tecnologie avanzate per data center, automotive, sanità e ricerca scientifica.

NVIDIA è inoltre leader nel campo della simulazione avanzata e della realtà virtuale da cui deriva la realizzazione di strumenti come NVIDIA Omniverse.

2.3.1 NVIDIA Omniverse

NVIDIA Omniverse è una piattaforma di simulazione e collaborazione virtuale sviluppata per offrire uno spazio condiviso in cui designer, ingegneri e creatori possono collaborare in tempo reale utilizzando una gamma di strumenti e applicazioni 3D. Lanciato ufficialmente nel 2021, Omniverse sfrutta le tecnologie di NVIDIA come le GPU RTX, l'intelligenza artificiale e la simulazione fisica in tempo reale per creare mondi virtuali estremamente dettagliati e dinamici.

Una delle caratteristiche principali di NVIDIA Omniverse è l'interoperabilità tra diversi software di progettazione e creazione, grazie all'Universal Scene Description (USD), un formato sviluppato da Pixar che permette la condivisione e il collegamento di dati 3D tra varie piattaforme.

Omniverse è pensato per una vasta gamma di applicazioni: dalla progettazione e ingegneria industriale alla produzione cinematografica e animazione, fino all'intelligenza artificiale e alla simulazione di robotica e veicoli autonomi.

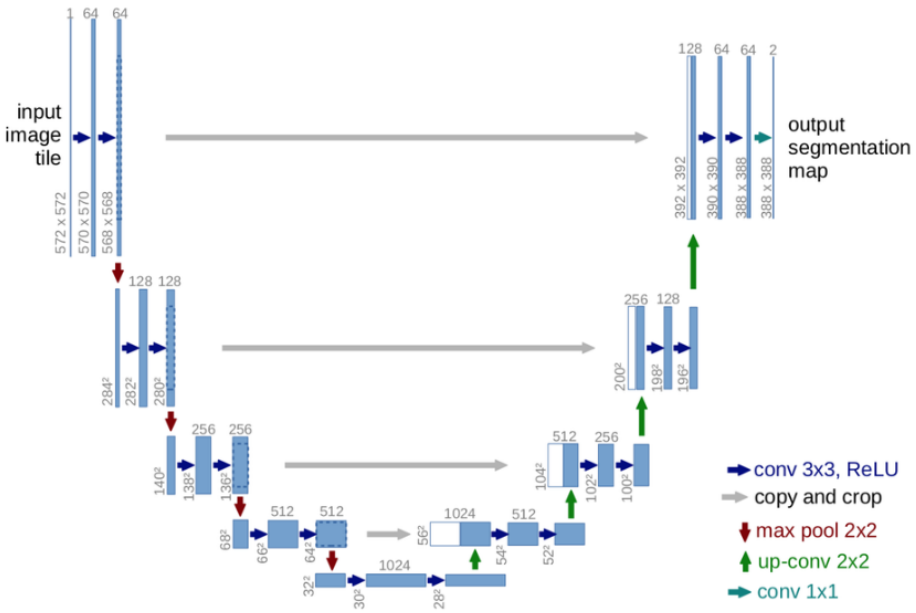


Figura 2.1: Architettura rete UNet

Offre strumenti avanzati per la simulazione fisica e la creazione di mondi virtuali interattivi, consentendo di simulare e testare modelli 3D.

2.4 UNet

U-Net è una Convolutional Neural Network (CNN) sviluppata inizialmente per applicazioni mediche ma tutt'oggi utilizzata principalmente per applicazioni di Image Segmentation. L'architettura di una U-Net è costiuita da:

- **Encoder**: riduce l'immagine in ingresso in una feature map attraverso pooling layers, estraendone gli elementi chiave;
- **Decoder**: amplifica la feature map in un'immagine, usando i livelli di deconvoluzione, impiegando cioè i pooling layers appresi per permettere la localizzazione degli elementi;
- **Skip connections**: copiano le feature map dall'encoder e le concatenano alle mappe del decoder, consentendo alla rete di mantenere le informazioni di dettaglio che potrebbero essere altrimenti perse durante il processo di downsampling.

U-Net è molto efficace anche in scenari con dataset limitati grazie alla sua architettura basata su connessioni skip.

Capitolo 3

Generazione del dataset

Il dataset è stato generato sfruttando le potenzialità offerte dalla piattaforma NVIDIA Omniverse. Tutto il codice relativo a questa fase è contenuto all'interno del file *generator.py*. La scena scelta per la segmentazione riguarda le scrivanie da ufficio in cui verranno segmentati gli oggetti posti su di essa, la rispettiva scrivania/tavolo e il backgorund.
Tutti gli elementi che costituiscono la scena sono contenuti nei seguenti file usd:

- La cartella *elem* contiene i seguenti file:
  - *desktop.usd*: contiene un monitor ed un mouse;
  - *dualmonitors.usd*: contiene un doppio monitor;
  - *hedphones.usd*: contiene delle cuffie;
  - *keyboard.usd*: contiene una tastiera;
- *table_X.usd*: questi file rappresentano diverse forme e colori di scrivanie/tavoli (6 in totale);
- *clean_cloudy_sky_and_floor.usd*: rappresenta l'ambiente di backgorund.

I file per poter essere utilizzati dallo script devono essere salvati all'interno del **Nucleus** nel rispettivo ambiente Omniverse. Si tratta nello specifico di un sistema di gestione dei dati che permette la condivisione e la sincronizzazione di asset, file e scene 3D (file usd). È costituito da diversi servizi interconnessi, alcuni di questi sono ad esempio il Nucleus Server che ospita gli asset e gestisce la comunicazione tra i client e il Nucleus Database che memorizza e organizza le risorse e le informazioni relative al progetto.

3.1 Script

In questa sezione non verrà presentato l'intero codice ma solamnete le sue parti più importanti, il codice completo è rappresentato dal file *generator.py*. Il codice affinchè funzioni deve essere eseguito nello *Script Editor* all'interno dell'ambiente Omniverse.

```
cfg = {
    "output": 'C:/Users/student1/output',
    "colors": True,
    "frames": 2000,
    "subframes": 6,
    "format": "png"
}
```

Figura 3.1: Impostazioni globali dello script

3.1.1 Configurazioni iniziali

Questa parte di codice definisce le impostazioni globali del progetto. In particolare oltre alla cartella di output e al tipo di formato delle immagini sono stati definiti i frame, che rappresentano il numero totale di immagini che verranno generate e i subframe necessari per dare qualità alle immagini (la loro assenza comporta la generazione di immagini con componenti totalmente bianche).

3.1.2 Oggetti e ambiente

L'import degli oggetti (desktop, mouse, tastiera, cuffie) dei tavoli e del backgorund è stato fatto assegnando una variabile ad ogni relativo percorso.

3.1.3 Writer personalizzato

All'interno dell'ambiente NVIDIA Omniverse, i Writer possono essere visti come moduli di output, responsabili della creazione e scrittura di contenuti e dati in un formato specifico. In particolare permettono di esportare scene, asset o modelli da Omniverse verso formati compatibili con altri strumenti 3D (es. Blender) e/o di convertire i dati in un formato leggibile o utilizzabile da altre applicazioni o sistemi. Nel codice originale il Writer implementa le seguenti funzioni:

1. **Costruttore della classe (_init_)**: questo costruttore della classe permette di definire la directory in cui verranno salvati i file di output (immagini e segmentazioni), le etichette su cui effettuare la segmentazione ed il formato delle immagini. Inizializza inoltre un contatore dei frame utilizzato per denominare l'immagine di output ed il backend per la scrittura dei file nella directory specificata;

2. **Scrittura dei dati (write)**: questa funzione riceve i dati come input e chiama le funzioni *_write_img* e *_write_segmentation* per scrivere rispettivamente l'immagine e la segmentazione semantica. Queste due funzioni definsicono i path di salvataggio sulla base del numero di frame corrente e sul formato specificato, le immagini vengono poi effettivamente salvate utilizzando la funzione *self.backend*;

3. **Conversione dei dati di segmentazione in etichette personalizzate**: questa funzione converte i dati della segmentazione in una mappa

di etichette numeriche basate su un mapping personalizzato in cui i pixel sono etichettati con valori numerici;

4. **Registrazione del writer**: questo permette di utilizzare il writer personalizzato all'interno dell'ambiente NVIDIA Omniverse.

3.1.4 Funzioni di randomizzazione

Le due funzioni di randomizzazione presenti sono:

- **randomize_obj()**: utilizzata per generare e distribuire oggetti casualmente all'interno della scena. Recupera i file usd e li istanzia garantendo che gli oggetti non vengano sostituiti o ripetuti, successivamente applica una rotazione casuale sull'asse Y di 180 gradi, assegna ad oguno di loro la classe semantica obj ed evita le sovrapposizioni utilizzando la funzione *scatter_2d*;

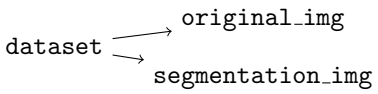
- **rep.trigger.on_frame**: gestisce la logica di randomizzazione e modifica della scena su ogni fotogramma. Attraverso la matrice *viz_matrix* garantisce la presenza di solo un tavolo e ne modifica la visibilità per ogni frame. Similmente viene randomicamente modificata la posizione della telecamera utilizzando l'intervallo specificato e il parametro *look_at* garantisce che vengano sempre inquadrati gli oggetti posti sulla scrivania. Richiama infine la funzione *randomize_obj* e *randomizer.sphere_lights* per randomizzare gli oggetti presenti e l'illuminazione.

```
def randomize_obj():
        obj = rep.randomizer.instantiate(
            rep.utils.get_usd_files(OBJ),
            size=3,
            with_replacements=False
            )

        with obj:
            rep.modify.pose(rotation=(0, 180, 0))
            rep.modify.semantics([("class", 'obj')])
            rep.randomizer.scatter_2d(plane_samp)
        return obj.node

with rep.trigger.on_frame(num_frames=cfg["frames"], rt_subframes=cfg["subframes"]):
        for idx, table in enumerate(tables):
            with table:
                rep.modify.visibility(rep.distribution.sequence(viz_matrix[idx]))

        with camera:
            rep.modify.pose(position=rep.distribution.uniform((0, 300, -600), (1000, 1000, -200)), look_at=plane_samp)

        rep.randomizer.randomize_obj()
        rep.randomizer.sphere_lights()
```

Figura 3.2: implementazione delle due funzioni di randomizzazione

3.2 Suddivisione dataset

Per la suddivisione del dataset si è utilizzata la regola del 80/10 (training e test) in quanto consente di bilanciare l'uso dei dati, assicurando che ce ne siano abbastanza per addestrare efficacemente il modello e per testarlo su un set indipendente. Lo script descritto nella sezione 3.1 crea due cartelle: *original_img* per salvare le immagini originali e *segmentation_img* per salvare le relative segmentazioni.
Nello specifico la suddivisione del dataset in training e test avviene direttamente durante l'addestramento di UNet, così coome descritto nel sezione 4.1.2. La struttura del dataset originale può essere quindi riassunta in questa figura:



Capitolo 4

UNet

L'implementazione dell'architettura UNet (descritta nella sezione 2.4) con backbone ResNet34 è stata fatta intermente utilizzando l'ambiente Google Colab, una piattaforma gratuita di Google che permette di scrivere applicazioni in Python via browser appoggiandosi alle risorse in cloud di Mountain View. È basata su Jupyter Notebooks, che supporta GPU (Graphics Processing Units) libere permettendo di utilizzare librerie come PyTorch, TensorFlow, Keras e OpenCV.

4.1 Addestramento

In questa sezione verranno descritte le parti più importanti del codice. La versione completa si torva nel file *UNet.ipynb*

4.1.1 Import e dipendenze iniziali

L'intero dataset è stato precedentemente caricato su Google Drive per poi essere importato attraverso l'operazione di mount, un comando che permette di connettersi temporaneamente a Drive per poter poi successivamente leggere i dati. Altre dipendenze necessarie sono la versione di tensorflow (2.10.0), di keras (2.10.0), efficientnet (1.0.0) e image-classifiers (1.0.0). Il tutto può essere scaricato attraverso i seguenti comandi:

```
!pip install tensorflow==2.10.0 keras==2.10.0
!pip install segmentation-models --no-deps
!pip install efficientnet==1.0.0 image-classifiers==1.0.0
```

Successivamente si può proseguire con l'import delle librerie.

4.1.2 Loading dei dati e preparazione del dataset

In questa fase sono stati definiti i percorsi contenenti le immagini e le rispettive maschere su cui vengono successivamente fatte delle manipolazioni per

permettere l'addestramento della rete neurale. Le due funzioni fondamentali sono:

1. **data_loader**: viene utilizzata per caricare e preprocessare le immagini e le segmentazioni. Prende in input il percorso di una cartella per poi successivamente iterare su tutti i file al suo interno. Per ognuno di essi viene caricata e letta l'immagine corrispondente, ridimensionata e trasformata in un array numpy;

2. **rgb_to_labels**: converte un'immagine con una maschera di segmentazione in un array di etichette numeriche, restituendo in output una versione dell'immagine in cui i pixel hanno valori numerici interi che corrispondono alle etichette delle classi. Questo è utile per l'addestramento di UNet.

La seconda funzione utilizza in particolare *mask_labels*, ottenuto leggendo un file csv, necessario per mappare i colori RGB presenti nelle immagini delle maschere di segmentazione alle rispettive classi numeriche. Ogni riga del CSV associa un colore specifico a una classe, consentendo di tradurre le informazioni visive (colori) in etichette numeriche. Il contenuto del file corrisponde a quanto riportato nella seguente tabella.

	name	**r**	**g**	**b**
0	background	0	0	0
1	table	140	25	255
2	obj	140	255	25

Viene infine calcolato il numero di classi distinte presenti nelle maschere, convertite le etichette da numeri interi a vettori con one-hot encoding e divisi i dati in set di addestramento e test per allenare il modello e valutarne le prestazioni.

4.1.3 Import del modello e addestramento

Questa parte svolge le seguenti attività:

1. **Selezione del modello pre-addestrato**: viene selezionato il backbone ResNet34 e restituita una funzione di preprocessamento compatibile attraverso il comando *sm.get_preprocessing*

2. **Definizione del modello**: crea un modello UNet con ResNet34 come encoder nel quale viene specificato che i pesi sono quelli pre-addstrati su ImageNet e che l'attivazione finale sarà una softmax (utilizzata per classificare ogni pixel in una delle n_classes);

3. **Compilazione del modello**: il modello viene compilato con l'ottimizzatore Adam e viene definita la funzione di perdita *categorical_crossentropy* (adatta per problemi di calssificazione multiclasse).

4. **Compilazione del modello**: il modello viene addestrato per tre epoche sui dati preprocessati con una dimensione del batch pari a 16.

L'output della fase di addestramento è quello riportato in fig. 4.1

```
Epoch 1/3
100/100 [==============================] - 1231s 12s/step - loss: 0.2087 - accuracy: 0.9402 - val_loss: 0.8120 - val_accuracy: 0.6977
Epoch 2/3
100/100 [==============================] - 1228s 12s/step - loss: 0.0343 - accuracy: 0.9909 - val_loss: 0.1006 - val_accuracy: 0.9755
Epoch 3/3
100/100 [==============================] - 1222s 12s/step - loss: 0.0339 - accuracy: 0.9895 - val_loss: 0.2149 - val_accuracy: 0.9267
```

Figura 4.1: Output della fase di addestramento

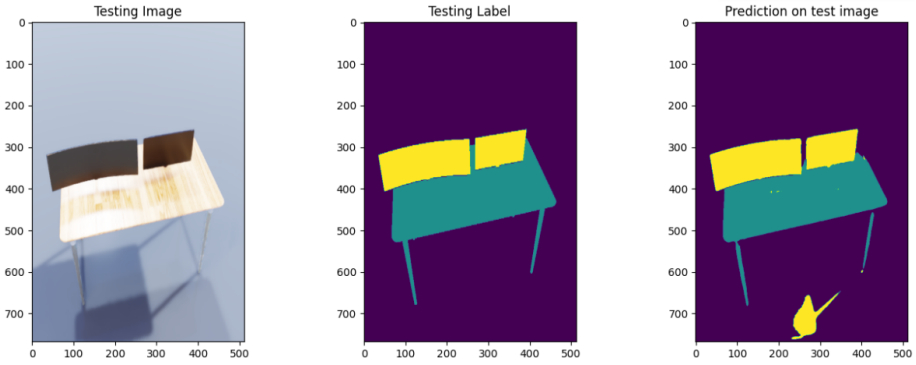


Figura 4.2: Output della fase di predizione

4.1.4 Risultati

Il grafico prodotto da questa parte del codice, permette di visualizzare come l'accuratezza sul set di addestramento e di validazione evolve durante le epoche:

- **history.history['accuracy']**: estrae i valori di accuratezza sul set di allenamento dopo ogni epoca di addestramento;
- **history.history['val_accuracy']**: estrae i valori di accuratezza sul set di validazione per ciascuna epoca.

4.2 Predizioni

Il modello ottenuto al termine dell'addestramento viene poi salvato con estensione **hdf5** (abbreviazione di Formato dati gerarchico versione 5). Si tratta di un formato di file open source per l'archiviazione e l'organizzazione di grandi quantità di dati.
Il modello viene poi caricato attraverso la funzione *load_model* e successivamente usato per fare previsioni sulle immagini di test. Il risultato della previsione (y_pred) è una matrice che contiene la probabilità di ogni classe per ogni pixel, per selezionare la classe con probabilità più alta viene utilizzato *np.argmax*.
Ogni previsione viene poi salvata su Drive utilizzando un ciclo for.
In fondo alla pagina si trovano due sezioni, una per scegliere randomicamente una delle immagini predette e l'altra per stampare l'immagine originale, quella segmentata e la predizione. Un'esempio è la fig. 4.2

Capitolo 5

Conclusioni

In questo progetto è stata affrontata la problematica della segmentazione di immagini utilizzando un approccio basato su deep learning. Si è inoltre dimostrato come la piattaforma NVIDIA Omniverse sia uno strumento versatile per la generazione di dataset sintetici in quanto garantisce una grande varietà di scenari e condizioni.
L'uso di un dataset sintetico di 2000 immagini ha permesso di addestrare una rete UNet con backbone ResNet34. I risultati ottenuti indicano che la rete è stata in grado di generalizzare su immagini non presenti nel dataset di addestramento, mostrando una buona capacità di segmentazione su nuovi esempi. Ci sono tuttavia ancora alcune aree di miglioramento. Si può infatti notare che in alcune immagini le ombre degli oggetti spesso tendono a ingannare la rete portandola a classificare queste porzioni come parte degli oggetti o del tavolo. Un incremento e una modifca del dataset potrebbe migliorare ulteriormente le prestazioni del modello, specialmente in contesti più complessi.